Document Title	AUTOSAR BSW & RTE Conformance Test Specification Part 4: Execution Constraints
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	284
Document Classification	Auxiliary

Document Version	1.2.0
Document Status	Final
Part of Release	4.0
Revision	2

Document Change History			
Date	**Version**	**Changed by**	**Change Description**
14.04.2011	1.2.0	AUTOSAR Administration	• Deletion/update of outdated chapters • Add chapter about CT process • Add chapter about test CT functionalities & TTCN-3 organization. • Update process with RTE and OS specificities • CTA accreditation replaced by CTA self assessment.
30.11.2009	1.1.0	AUTOSAR Administration	• Typos in section 3.2.1.2 corrected • Unnecessary information in section 1.1.1 deleted • Legal disclaimer revised
23.06.2008	1.0.1	AUTOSAR Administration	Legal disclaimer revised
14.11.2007	1.0.0	AUTOSAR Administration	Initial Release

Disclaimer

Advice for users

AUTOSAR specifications may contain exemplary items (exemplary reference models, "use cases", and/or references to exemplary technical solutions, devices, processes or software).

Any such exemplary items are contained in the specifications for illustration purposes only, and they themselves are not part of the AUTOSAR Standard. Neither their presence in such specifications, nor any later documentation of AUTOSAR conformance of products actually implementing such exemplary items, imply that intellectual property rights covering such exemplary items are licensed under the same rules as applicable to the AUTOSAR Standard.

Table of Contents

1 Overview of the document

This document is part of the process specifications for AUTOSAR BSW conformance testing. It describes the execution environment for the AUTOSAR BSW Conformance Test Specifications (CTSpecs) and its constraints. The objective is to provide the technical foundation and all information that allows the development and realization of a test infrastructure that is suitable for executing the AUTOSAR CTSpecs.

The overall CTSpec creation process is described in [4] and the methodology for its realization in [3].

The first part of this document (Chapters 2 and 3.2) describes the generic TTCN-3 test system architecture and its application to AUTOSAR. It further classifies the two possible test setups for AUTOSAR BSW conformance testing.

The second part (Chapters 3.3 and 3.4) contains the design overview for the Test Adapters and in particular the functional specification of the “target adapter”. The latter one makes the BSW module under test accessible for conformance testing. Within this functional specification, generic API functions to be provided by the target adapter are defined as well.

1.1 Purpose, Scope and Target Audience

1.1.1 Purpose and Scope

The main focus of this document is to mention all aspects that are related to enable the execution of the AUTOSAR conformance tests

- In a simulation environment (“Class A”) or
- Against a real embedded system running the BSW module implementation (“Class B”)

For a detailed definition of these classes of test setups, see Chapter 3.2.2.

It is not within the scope of this document to describe the process of properly executing the AUTOSAR conformance tests with the objective of officially certifying conformance of the BSW module under test. This certification process requires definitions of work steps and artifacts that go beyond the definitions in this document.

1.1.2 Target Audience

This document set is intended to be used by any and all companies, groups and individuals engaged in the implementation of the execution environment for the AUTOSAR conformance tests. This document is the basic specification of this implementation. The content is applicable to both, the validation execution environment and the execution environment for conformance testing real BSW module implementations.

In order to completely understand this document, the documents [3] and [4] should have been read first.

1.2 Acronyms and Abbreviations

Abbreviation	*Description*
API	Application Program Interface
ATS	Abstract Test Suite
BSW	Basic Software
BSWM	BSW module
BSWMD	BSWM Description
BSWMDT	BSWMD Template
CC	Conformance Class
CD	Coder/Decoder (TTCN-3 – see Part 4)
CH	Component Handling (TTCN-3 – see Part 4)
CTA	Conformance Test Agency
CTSpec	Conformance Test Specification
CTS	Conformance Test Suite
CTSystem	Conformance Test System
ECU	Electronic Control Unit
ETS	Executable Test Suite
FCC	Functional Conformance Class
ICC	Implementation Cluster Conformance Class
ICS	Implementation Conformance Statement
IP	Intellectual Property
PA	Platform Adapter (TTCN-3 – see Part 4)
PS	Product Supplier
RTE	Run Time Environment
SA	System Adapter (TTCN-3 – see Part 4)
SID	Service IDentifier
SUT	System Under Test
SW-C	Software Component
SWS	Software Specification
TE	TTCN-3 Executable (TTCN-3 – see Part 4)
TM	Test Manager (TTCN-3 – see Part 4)
TRI	TTCN-3 Runtime Interface (TTCN-3 – see Part 4)
TTCN-3	Testing and Test Control Notation, version 3

1.3 Referenced Documents

[1] TTCN-3 specifications:
http://www.ttcn-3.org/StandardSuite.htm (accessed 20th of June, 2007)

[2] AUTOSAR BSW & RTE Conformance Test Specification Part 1: Background
AUTOSAR_PD_BSWCTSpecBackground.pdf

[3] AUTOSAR BSW & RTE Conformance Test Specification Part 2: Process Overview
AUTOSAR_PD_BSWCTSpecProcessOverview.pdf

[4] AUTOSAR BSW & RTE Conformance Test Specification Part 3: Creation & Validation
AUTOSAR_PD_BSWCTSpecCreationValidation.pdf

[5] Specification of BSW Module Description Template
AUTOSAR_TPS_BSWModuleDescriptionTemplate.pdf

[6] General configuration and test parameters used for validating conformance tests
AUTOSAR_CTCF_General.pdf

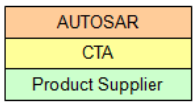
[7] Conformance Test Process Definition Path A-C
AUTOSAR_PD_CTProcessDefinitionPathAToC.pdf

2 CT process

The following process is provided as hint, to understand how to get a conformance test result from a BSW module to test and its associated CTSpec.
Responsibilities can be adapted regarding contract between Product Supplier and CTA.
Due to RTE specificities, a RTE CT Process is provided for better understanding.

Legend for next processes

Responsibilities:



2.1 Process overview

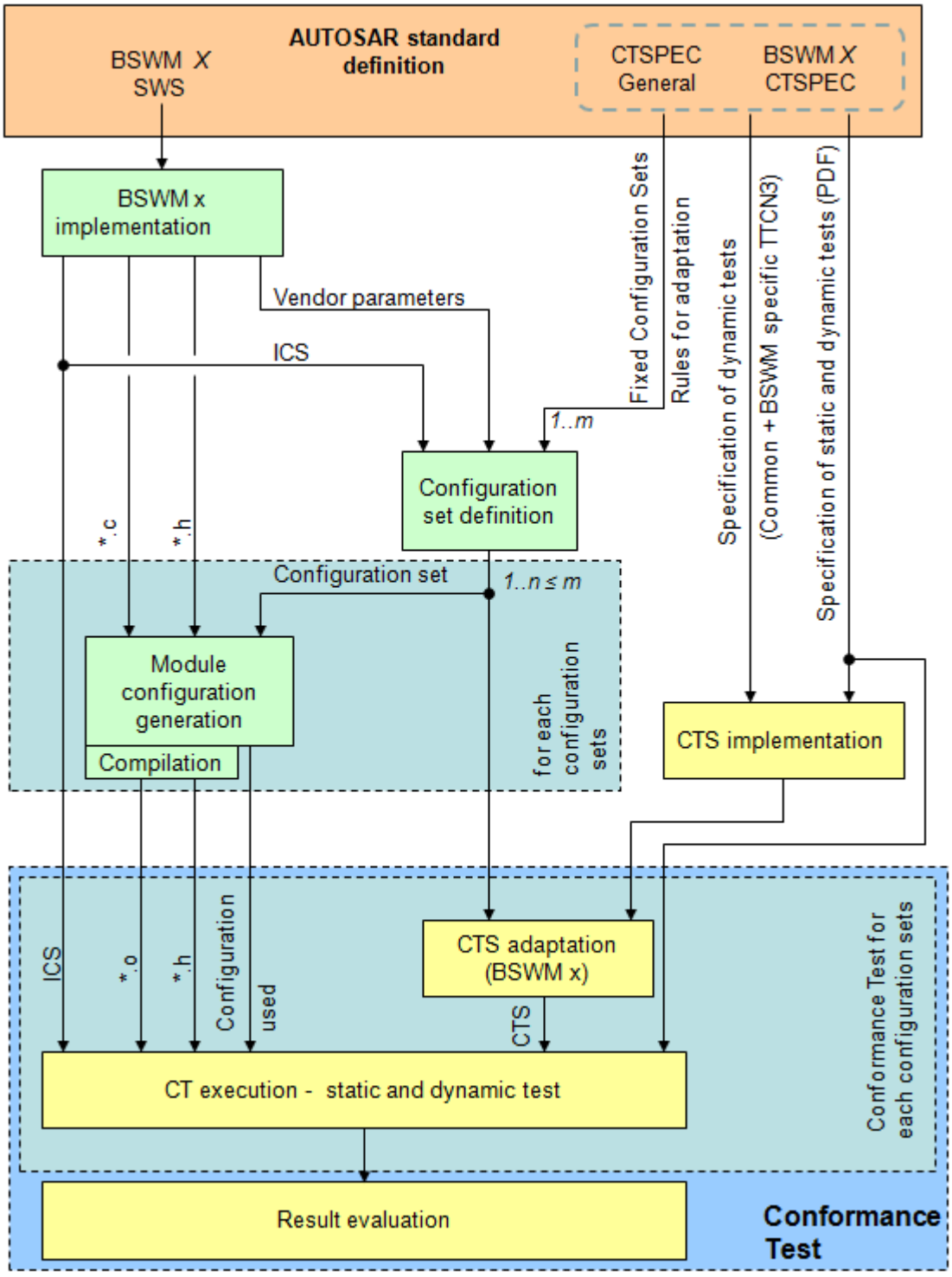


Figure 1 – CT process overview – except RTE

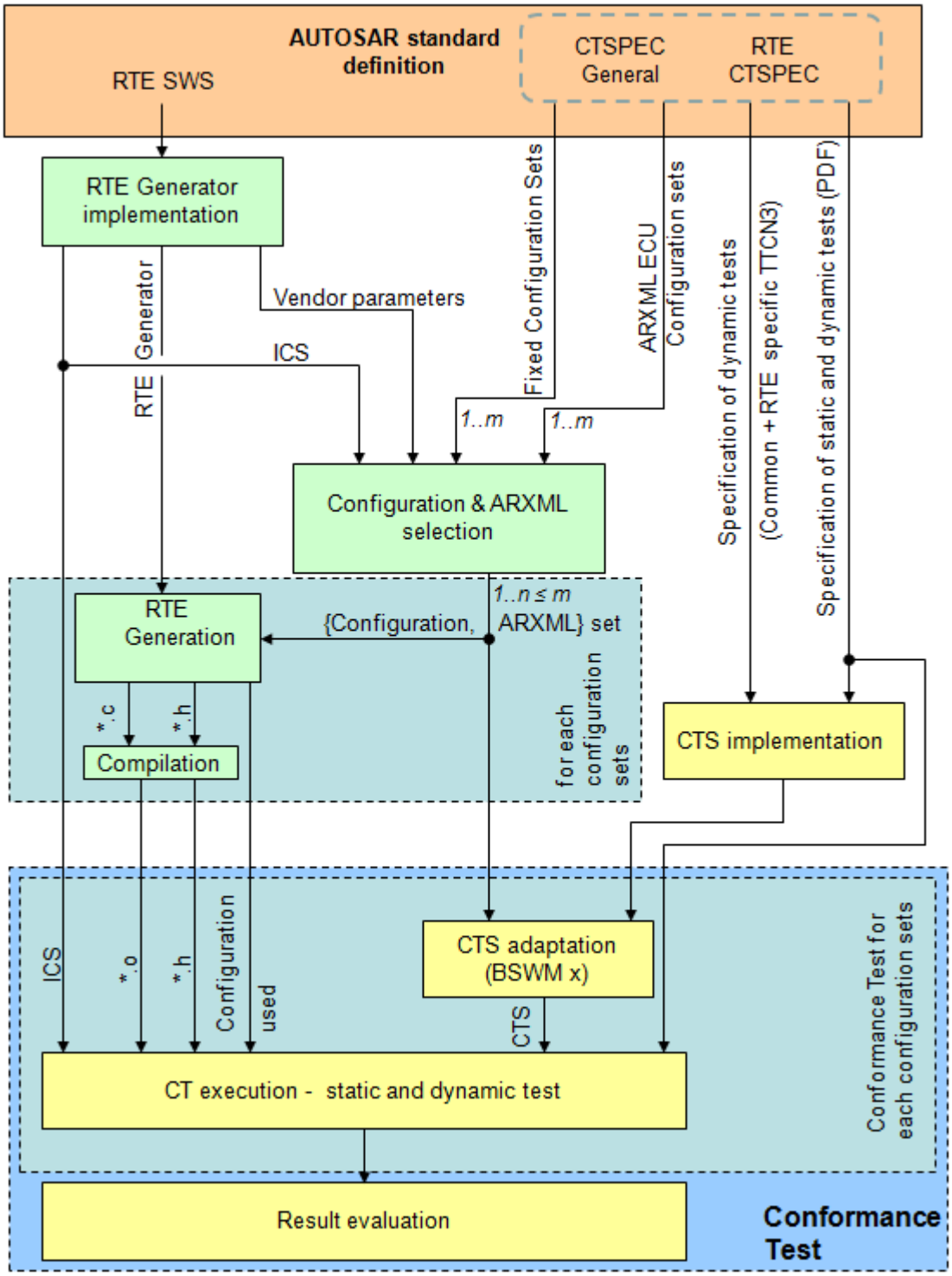


Figure 2 – RTE specific CT process overview

2.2 CT process artifacts

2.2.1 CTSpec

CTSpec are delivered in different standard ZIP files:

- AUTOSAR_CTSP_*<BSWM>*

  Contains the *Specification of static and dynamic tests (PDF)*, the *Specification of dynamic tests (TTCN-3)* specific to the module and the *Fixed configuration set* (adding *ARXML ECU Configuration* for RTE).

- AUTOSAR_CTSP_General

  Contains the *Specification of dynamic tests (TTCN-3)* generic to all modules.

Auxiliary documents are available for information:

- AUTOSAR_CTCF_*<BSWM>*

  Contains an overview of the configuration parameters (values and constraints) used to define *Fixed configuration set*.

- AUTOSAR_CTCF_General

  Contains the *CTCF General* document [6] defining rules for configuration adaptation.

2.2.2 Specification of static and dynamic tests (PDF)

Test cases are organized in two main categories:

- **static test cases**: configuration inspection, source code inspection and API check;
- **dynamic test cases**: tests on specified behaviors. Are also specified in TTCN-3 formal language (see *Specification of dynamic tests (TTCN-3)* artifact).

2.2.3 Specification of dynamic tests (TTCN-3)

Implementation in TTCN-3 language of the dynamic tests specifications.
These are Abstract Test Suites to be made executable.

2.2.4 Fixed configuration set

Fixed configuration set have been defined to test every BSWM with a correct coverage.
These files provide values to BSWM's parameters –defined in the SWS document- as well as *CT parameters*, which are specified by the test suites.

They are used to configure both the module under test and the test suite and are provided in a XML format.
Only parameters relevant to CT process appear in these files and are given a value. Other mandatory parameters should be added during *Configuration set definition* or *Configuration & ARXML Selection* action to get complete configurations.

For RTE, each *Fixed configuration set* is delivered along with an *ARXML ECU Configuration*.

2.2.5 ARXML ECU Configuration

These artifacts are specific to RTE CT process.

RTE ARXML ECU Configuration files describe AUTOSAR software level configuration (SW-Components).
Along with the *Fixed configuration set* for RTE, they are used to generate the “RTE under test” form the RTE Generator.
Values should not be modified, as conformance test cases for RTE require specific configurations to be executed.
Information such as OS task mapping that is important for CT can be found in the *Fixed configuration set* of the RTE.

2.2.6 RTE Generator

This artifact is specific to RTE CT process.

The CTSpec of RTE does not check conformance of the RTE Generator, but of generated RTEs.
Due to RTE specificities, RTE Generator will be used to generate “RTEs under test” as expected by RTE CTSpec.

2.2.7 ICS

The *ICS* restricts the specifications covered by the module under test, like minimized parameter’s range or not implemented API.
The format is a subset of the AUTOSAR BSWMDT (see [5]).
It is used to adapt the *Fixed configuration set*, and so to adapt the generic conformance tests (ATS), to the specific implementation under test.

2.2.8 Vendor specific parameters

List of parameters defined by the Product Supplier to configure his own BSWM. These parameters are not specified by AUTOSAR and so are not taken into account by the CTSpec, but they are required by *Configuration set definition* and *Configuration & ARXML Selection* actions to correctly configure/generate the module under test.

2.2.9 BSWM *.c / *.h

The module under test, as implemented by the Product Supplier or generated by the RTE Generator implemented by the Product Supplier.

2.3 Actions

Actions 2.3.1 to 2.3.4 are applicable to all modules.
Actions 2.3.5 to 2.3.6 are applicable to all modules but RTE.
Actions 2.3.7 to 2.3.8 are applicable to RTE only.

2.3.1 CTS implementation

The objective here is to build an executable test suite from the abstract test suite provided by AUTOSAR.
Two different activities are related to this action:

- Automation of static test cases: it is up to the CTA to automate a maximum of static test cases;
- Dynamic tests specifications are provided in TTCN-3 language. Implementation of standard TTCN-3 environment has to be designed and build by the CTA.
  Details are provided in chapter 3. Information about abstract test suite structure is given in chapter 4. Information about Functionality of all stubs defined by CT process is given in chapter 5.

2.3.2 CTS adaptation

Test suite has to be adapted to the configuration sets, output of *Configuration set definition* action.
CTA has to derive from each configuration sets a TTCN-3 configuration file which will be used to configure the executable dynamic test cases. The format of such configuration files is out of AUTOSAR scope (specific to each TTCN-3 tool).
CTA also has to adapt static test cases consequently, like inhibiting test cases covering requirements out of module under test scope.

Output for this action is one configured CTS for each configuration set.

2.3.3 CT Execution

With respect to AUTOSAR criteria and to the commitments of the CTA self assessment [7], CTA has to execute the configured CTS on its corresponding configured BSWM.

- **Static tests**
  - **Source Code Inspection**: executed on BSWM's header files (generic and configuration specific).

  - **Configuration inspection**: executed on ICS to ensure it is a subset of AUTOSAR complete BSWMD.
  - **Operation signature tests**: executed on BSW module, to check prototypes of APIs it provides and to check prototypes of APIs it requires.
- **Dynamic tests**: executed in class A or class B on the configured BSWM.

2.3.4 Result evaluation

With respect to AUTOSAR criteria and to the commitments of the CTA self assessment [7], the compliance of the BSWM under test is deduced from results of *CT Execution* action.
Conformance declaration or attestation can be established during this action.

2.3.5 Configuration set definition

This action does not apply to RTE CT process.

Fixed configuration set provided within a CTSpec have been defined taking into account the whole SWS. As Product Supplier can limit the scope of its module (see *ICS* artifact), *Fixed configuration set* have to be adapted to the specific module under test regarding ICS by applying adapting rules [6]. Missing parameters, like *Vendor specific parameters*, have to be provided within the configuration.

For example, in the EEPROM Driver specification, *EepDefaultMode*'s range is {MEMIF_MODE_FAST ; MEMIF_MODE_SLOW}.
If the ICS states that the module under test only supports the mode MEMIF_MODE_SLOW, *Fixed configuration set* in which *EepDefaultMode* is set to MEMIF_MODE_FAST have to be adapted consequently.

There are 2 outputs for this action:

- ARXML files to configure the BSW (input for *Select applicable Fixed configuration set* regarding *ICS;*
- Add missing parameters (AUTOSAR parameters that are not required by test suite and vendor specific parameters) to selected configurations.
- Module configuration generation action) including BSWM parameters and vendor specific parameter (not CT parameters);
- Files to configure the test suite, including BSW parameters and CT parameters (not vendor specific parameters).

Exception

Operating System is a special case as test cases require specific configuration to be executed.

Values of OS *Fixed configuration set* should not be modified. For OS specifically, in this *Configuration set definition* action you have to

- Select applicable *Fixed configuration set* regarding *ICS;*
- Add missing parameters (AUTOSAR parameters that are not required by test suite and vendor specific parameters) to selected configurations.

2.3.6 Module configuration generation

This action does not apply to RTE CT process.

The output from this action is a set of configuration files for each configuration sets created by *Configuration set definition* action.
They will be compiled and then linked with the BSWM's "generic" source files to get one configured BSWM for each configuration set.

2.3.7 Configuration & ARXML Selection

This action is specific to RTE CT process.

RTE is a special case as test cases require specific configuration to be executed.

Values of RTE *Fixed configuration set* and *ARXML ECU Configuration* should not be modified. In this *Configuration & ARXML Selection* action you have to

- Select applicable *Fixed configuration set* and *ARXML ECU Configuration* regarding *ICS;*
- Add missing parameters (AUTOSAR parameters that are not required by test suite and vendor specific parameters) to selected configurations.

2.3.8 RTE Generation

This action is specific to RTE CT process.

Using configuration and ARXML files selected for CT during the *Configuration & ARXML Selection* action, RTEs will be generated to be test objects.

3 Dynamic tests

3.1 Generic TTCN-3 Test System Architecture

Figure 3 shows the generic architecture of a TTCN-3 based test system as defined in the TTCN-3 standard [1]. A series of TTCN-3 test cases is commonly referred to as a *TTCN-3 test suite*.

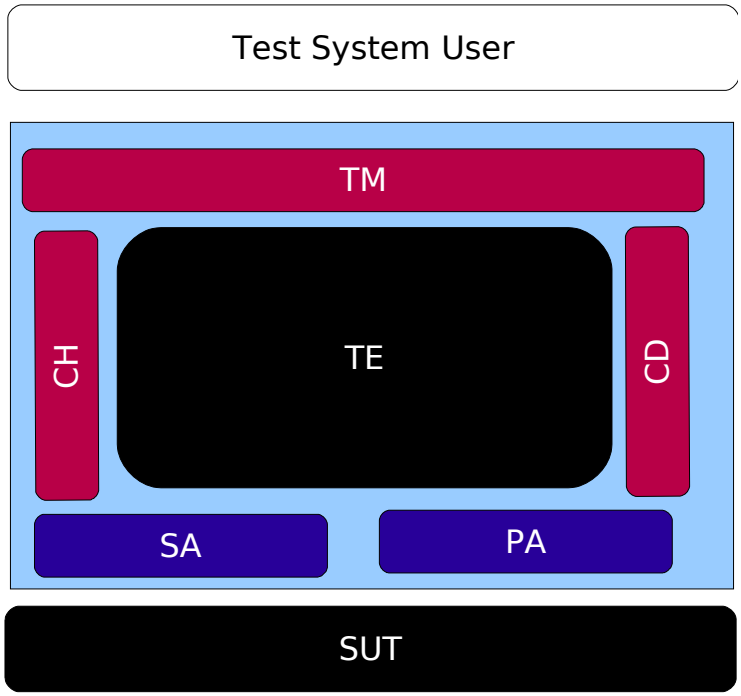


Figure 3 – TTCN-3 test system overview

An executable TTCN-3 test suite is thus made up of the following parts[1] (see Figure 3 and [1] for more details):

- **SUT** (System Under Test): The system that is tested.
- **TE** (TTCN-3 Executable): Contains the compiled TTCN-3 test cases and handles the execution of the test cases' statements.
- **SA** (SUT Adapter): Implements the interface between the TE and the SUT. It handles the sending and reception of all messages or procedure/function invocations between the TE and the SUT.
- **PA** (Platform Adapter): Realizes platform dependent functions, such as timers and TTCN-3 external functions.
- **CD** (Coder/Decoder): Transforms between TTCN-3 abstract data types used in the TE and concrete data types of the SUT.

[1] The abbreviations used for these parts are the same as in the TTCN-3 standardization documents.

- **TM** (Test Management): Controls the execution order of test cases in a test suite, logs all events from the TE, and implements the user interface of the test system.
- **CH** (Component Handling): Handles the creation, distribution and termination of TTCN-3 test components used in a test case.

The parts **TM** and **CH** are in general implemented and provided by the TTCN-3 tooling. When designing a TTCN-3 test system for a specific test job (such as AUTOSAR BSW conformance testing), the parts **SA**, **PA** and **CD** need to be specified. For AUTOSAR BSW conformance testing, the **TE** part is automatically generated by compiling the TTCN-3 test cases from the CTSpecs, and the **SUT** part is identical to the BSW module under test.

3.2 Specification of the Test System Architecture

3.2.1 The TTCN-3 Test Architecture for AUTOSAR Conformance Tests

3.2.1.1 Test Management

When executing conformance tests against an AUTOSAR BSW module under test, the user interacts with the TM as it implements the user interface. For execution of the CTSpecs, the user requires TM functionality on selecting and loading TTCN-3 files, and on selecting and executing the “control part” of TTCN-3 modules.

The logging and reporting functionality is of course also a crucial part of the TM as it delivers the results of the test execution and provides further information that is required when errors have occurred and need to be found.

However, TM functionality is generally implemented by the various TTCN-3 tools. Thus, the actual way of user interaction with the TM is tool specific.

3.2.1.2 Component Handling

A TTCN-3 test case contains at least one *test component* which resembles a self-contained thread that executes test steps. Complex test cases usually employ multiple test components that can potentially run on different software environments or hardware devices.

For AUTOSAR BSW conformance testing, test cases make use of multiple test components but they all run on the same software and hardware environment: the Tester PC.

Component Handling is a functionality provided by the TTCN-3 tool. Since for AUTOSAR BSW conformance testing, test components do not need to be distributed onto several software and hardware environments, the basic CH functionality pro-

vided by all TTCN-3 tools is sufficient. Since this basic CH functionality is usually hidden from the user, no further specification on CH is required.

3.2.1.3 TTCN-3 Executable

The TE is the compiled and ready-to-be-executed form of TTCN-3 test cases. Building the TE from the TTCN-3 test cases is usually a core functionality of TTCN-3 test systems. The handling of this build process is tool specific.

Since the execution behavior of the CTSpec is highly dependent on the configuration parameters, the TE has to be specifically generated for a set of configuration parameters. That is, for a new set of configuration parameters, the *TTCN-3 Config Module* has to be generated (see [3]) and added to the CTSpec which is then re-compiled to generate a new TE.

3.2.1.4 System Under Test

For AUTOSAR BSW conformance testing, the SUT refers to the BSW module under test. Since the SUT is also highly configured by configuration parameters, a generation/compilation process is usually required to obtain the SUT for a specific configuration set.

The SUT is executed within an environment provided by the target platform. Prior to execution of the SUT, the BSW module has to be integrated into the software environment of the target platform. This includes in particular the

- integration of the SUT with the “target adapter” (see Chapter 3.3.1);
- correct linking with called functions as well as callback functions.

3.2.1.5 Platform Adapter

The PA provides platform specific services to the TE with regard to timers and external functions.

For conformance testing of AUTOSAR BSW modules with no real-time behavior, the accuracy of the standard real-time clock of the Tester PC can be regarded as sufficient. Thus, the standard timer functionality of the PA commonly provided by TTCN-3 tools is used.

TTCN-3 external functions are functions that can be invoked from the test cases but are implemented in a programming language other than TTCN-3 (e.g. Java, C). The PA realizes the invocation and return of *TTCN-3 external functions*. They can be helpful in case when special functionality is required by a test case (e.g. fast calculation of a CRC). If an AUTOSAR BSW CTSpec makes use of *TTCN-3 external functions*, the external function has to be carefully specified and should be easily portable.

3.2.1.6 Coder/Decoder

The CD is responsible for converting data between their abstract representation in the test cases and their concrete representation in the SUT.

In AUTOSAR BSW, basic data types (e.g. signed und unsigned integers of various bit width), enumeration types, pointers and structures based on the aforementioned types are used.

Conversion of the basic data types is simple and might involve a conversion between little-endian and big-endian representation of data.

Within the C source of BSW modules, enumeration values are mapped to unsigned integer values. For TTCN-3 conformance test cases, AUTOSAR enumeration types are also mapped to unsigned integer. This means, that the CD can handle enumeration types as unsigned integer resulting in the simple conversion described above.

Pointers are in general handled transparently by the test cases. That means, the whole pointer value (i.e. the memory address), regardless of its bit width, is handled by a conformance test cases when invoking or receiving APIs of a BSW module containing pointer parameters or pointer return values. However, the test case cannot directly evaluate these pointer values. It has to use specific target adapter functions to process and evaluate pointer values when necessary.

3.2.1.7 SUT Adapter

The SA connects the TE with the SUT. It transforms the communication operations (i.e. stimulation and observation activities) from within a TTCN-3 test case into appropriate operations towards the SUT.

3.2.2 Possible Test Setups

For conformance testing of AUTOSAR BSW modules, two different basic test setups are possible which have been identified and described in Chapter 5 of [2] and are referred to as “Class A” test setup and “Class B” test setup. The CTSpec is implemented and applicable to both test setups.

3.2.2.1 Class A Test Setup

With the Class A test setup, the BSW module under test (i.e. the SUT) is executed within a simulation environment on a PC which is (for the sake of simplicity) in general the same PC that executes the conformance test cases. With this type of test setup, hardware related functionality can often not be tested unless the simulation environment also provides a simulation of the hardware that is interfacing with the SUT.

However, providing a simulation of a hardware component and correctly interfacing it with the SUT usually involves high efforts and might often be abandoned in favor for the Class B test setup with real hardware.

Therefore, Class A test setups are primarily expected to be used for BSW modules that have interfaces to other software modules only and do not interface with hardware components. For those types of BSW modules, Class A tests provide the possibility to execute conformance tests without the hassle of setting up and handling an embedded target system with its required hardware components.

3.2.2.2 Class B Test Setup

Class B test setups require the execution of the BSW module under test on the embedded target system. In general, this type of test setup promises more reliable results on conformance of a BSW module since the BSW module is executed under conditions that are potentially very close to those of the later production use.

Figure 4 shows the Class B test setup consisting of the test system made up of its different parts, the target platform with the target adapter and the communication means between test system and target platform.

However, the separation between test system and target platform (i.e. an embedded system) introduces additional complexity:

- A communication means and scheme between test system and target platform is required.
- A “SUT adapter” (or “upper test adapter”) on the Test PC side is required. It is used to give the test case execution access to the communication means.
- A “target adapter” (or “lower test adapter”) that provides this communication functionality on the target system side and interacts with the BSW module is required.
- Both “SUT adapter” and “target adapter” need to be properly configured with respect to the properties of the communication means and the SUT.

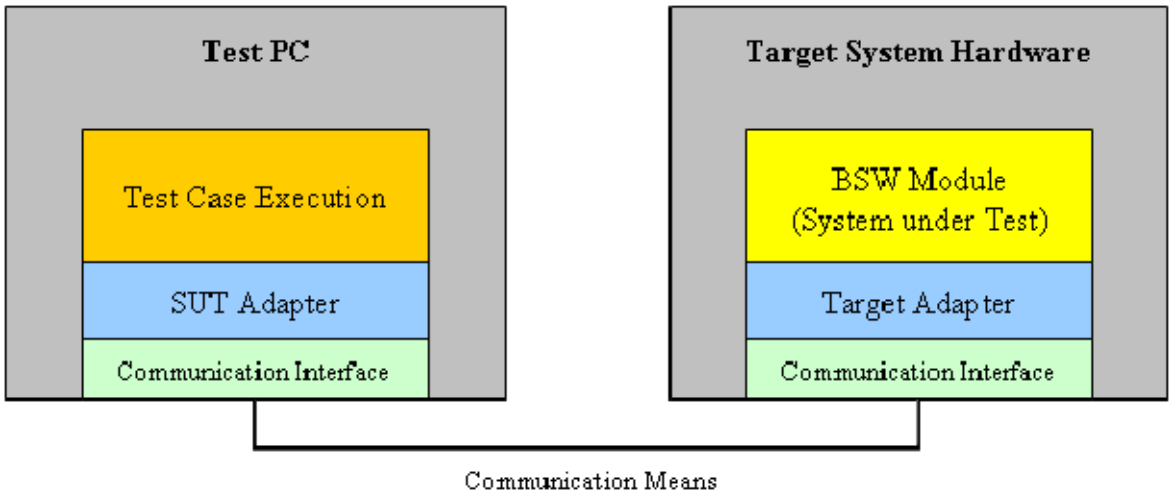


Figure 4 – Class B test setup

3.2.2.3 Class A vs. Class B from the view point of the CTSpec

The conformance test cases should be as much as possible independent from the type of test setup that is chosen for execution of the tests. In this way, the CTSpecs can be used at a wide range of the development process (e.g. Class A test setup during an early development stage when the module is developed within a PC based simulation environment).

Meeting this objective is very well supported by the TTCN-3 concept of “test adapters” which can be regarded as an “intermediate layer” between the test case execution and the SUT. The purpose of the test adapter is to provide the test cases with an “abstract” test interface. This concept is intended to be used for abstraction issues as, in this case, to abstract as much as possible from the type of test setup being used (Class A or Class B).

The following sections discuss the most relevant aspects of the CTSpec with regard to their handling and realization by a Class A or Class B test setup

3.2.2.3.1 Software APIs

Within the conformance test cases, the APIs of the BSW modules that interface with other software components are handled in the same way for Class A and Class B test setups.

For Class A test setups, it is the responsibility of the test adapter integrated with the simulated execution environment for the BSW module to correctly forward API calls, callbacks and their parameters to the BSW module.

For Class B test setups, the test adapter together with the target adapter has to realize the forwarding of API calls, callbacks and their parameters. This functional requirement on the target adapter is described in more detail in Chapter 3.4.1.

3.2.2.3.2 Hardware interfaces

With a Class B test setup, the BSW module interfaces with hardware (mostly hardware driver modules), for example, by directly accessing specific memory registers and with the help of interrupt routines. These mechanisms are hardware specific and internal to the BSW implementation.

When the same BSW module implementation is to be executed in a simulation environment for Class A tests, the same mechanisms for hardware interaction need to be provided by the simulation environment.

Since currently, hardware interfaces of BSW modules are not directly relevant for conformance testing, the issues associated with handling these interfaces for Class A and Class B test setups are not further analyzed within this document.

3.2.2.3.3 Test Adapters

For both Class A and Class B test setups, the test adapters need to be implemented specifically for the components of the test environment. In Chapter 3.3, functional requirements on the test adapters are given. However, it is not in scope of this document to present a detailed design or an implementation specification for these adapters.

- For Class A test setups, the design and implementation of the test adapters mainly depends on the interfaces and properties of the simulation environment.

- For Class B test setups, the design and implementation of the SUT adapter and the target adapter mainly depend on the communication means

3.3 Design Overview of the Test Adapter

As already described in Chapter 3.2.2 and depicted in Figure 4, the test adapters consist of both the SUT adapter and the target adapter.

3.3.1 Integration with Execution Environment

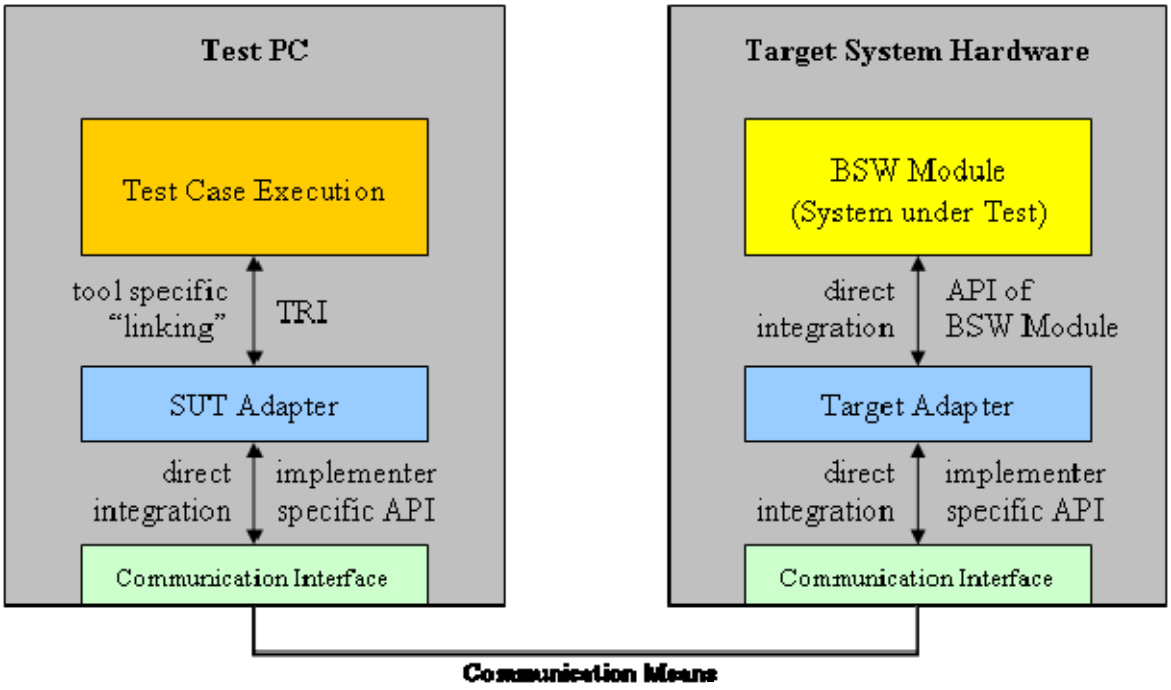


Figure 5 – Integration interfaces for SUT adapter and target adapter

As shown in Figure 5, the SUT adapter and target adapter need both to be integrated with their execution environments:

- SUT adapter:
  - The integration with the test execution is realized by implementing the "TTCN-3 Runtime Interface" (TRI). The TRI is specified in [1]. Implementing the TRI involves implementing TRI functions to be used by the test execution and making use of TRI functions that are provided by the test execution. How the SUT adapter and the test execution are actually "linked" together is tool specific.
  - The integration with the communication interface needs to be done in a direct way (i.e. compiled and linked together). In general, the SUT adapter implements directly the communication functions (e.g. socket

functions for TCP/IP communication). Alternatively, an implementer specific communication API can be used here.

- Target adapter:
  - The integration with the BSW module needs to be done directly using the API specified for the BSW module. That means, the software structure of the target adapter must implement direct calls to API functions of the BSW module and must provide function stubs to be called as API call-backs by the BSW module.
  - The integration with the communication interface is the same as for the SUT adapter.

3.3.2 Communication Means

The communication means consists of both the hardware realizing the data transfer (i.e. Ethernet adapters and cabling) and the software/protocols that provides the communication services (e.g. Ethernet driver, TCP/IP protocol).

With respect to AUTOSAR BSW conformance testing, the main purpose of the communication means that interconnects the SUT adapter with the target adapter is to efficiently transmit

- function calls and their parameters (if defined),
- function returns with return value (if defined)

in both directions.

The following “quality of service” for this communication means is required:

- Communication events (i.e. function calls and function returns) must not get lost during transmission.
- The content of communication events must not be altered.
- Reordering of communication events must not happen.
- Delay and jitter for transmission of communication events must be magnitudes of order smaller than the smallest time-out value in the BSW module or in the CTSpec.

How the communication means is realized is implementer specific and out of scope of AUTOSAR standards.

3.3.3 SUT Adapter

The main functionality of the SUT adapter is to forward communication events (i.e. function calls and function returns) between test execution and target adapter by making use of the communication means. In order to realize an efficient communication, this forwarding functionality usually involves encoding and decoding of the function names, their parameters and the possible return value. This main functionality is illustrated in Figure 6.

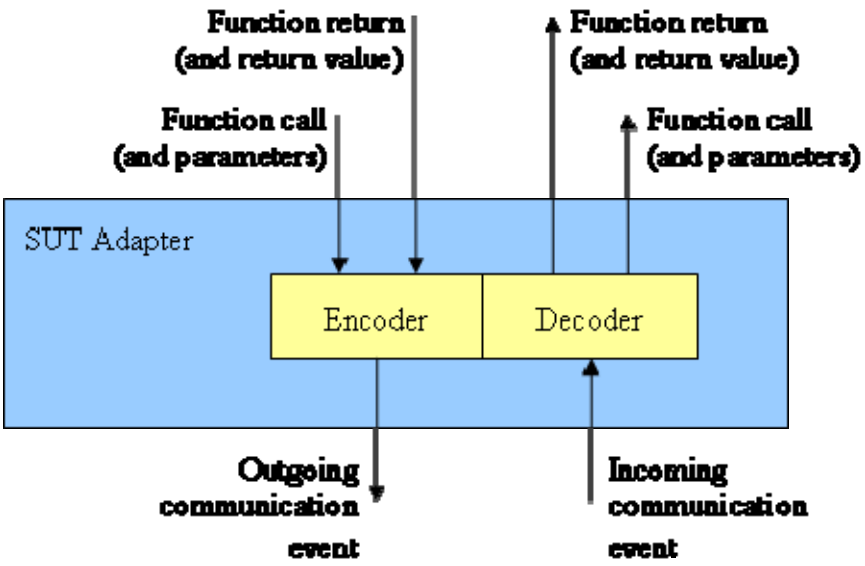


Figure 6 – Overview on the main functionality of the SUT adapter

The encoding/decoding strategy and scheme is implementer specific and out of scope of AUTOSAR standards.

3.3.4 Target Adapter

The target adapter must be able to apply the encoding/decoding scheme that the SUT adapter uses to correctly convert communication events received from the SUT adapter into appropriate function calls or function returns towards the BSW module and vice versa.

In addition to that, the target adapter must implement the following active functionality:

- Custom test functions (“target adapter functions”) that the test cases might use (e.g. for interacting directly with the memory of the target system).
- Automatic cyclic invocation of the main function of the BSW module (can be activated/deactivated by target adapter functions)

Figure 7 gives an overview on the main functionality of the target adapter. The target adapter functionality is specified in more detail in the following chapter.

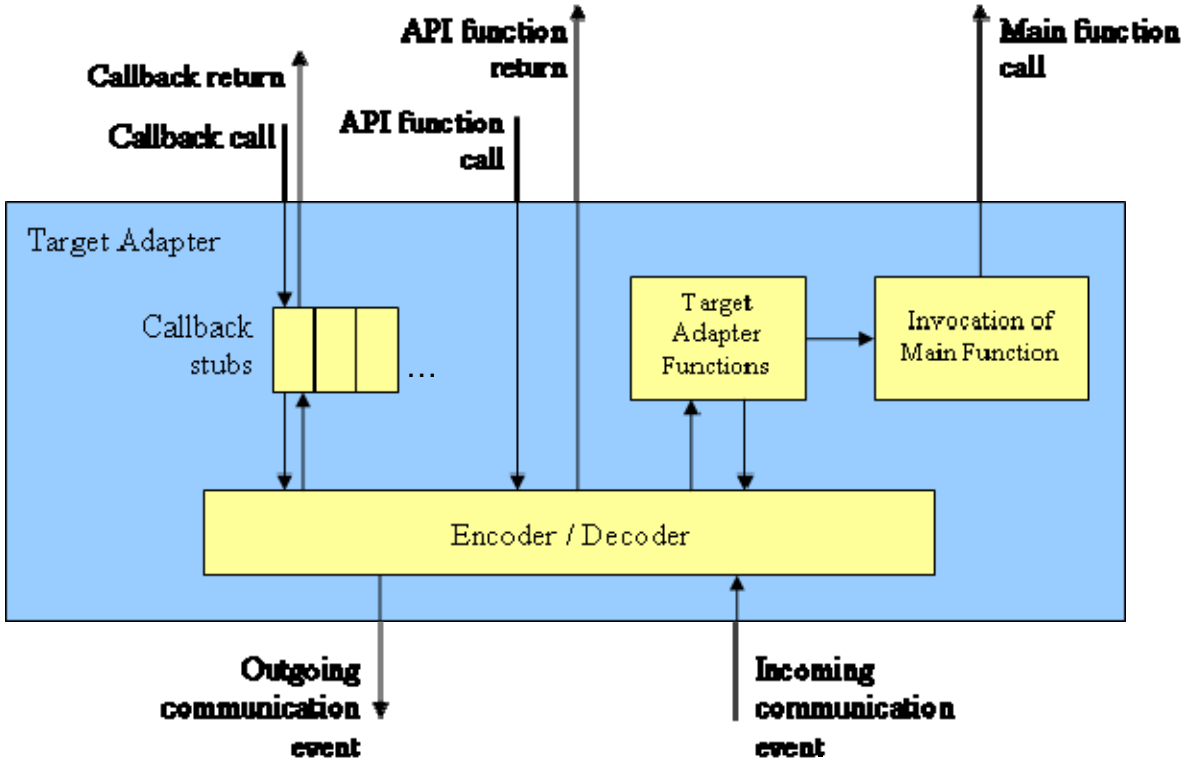


Figure 7 – Overview on the main functionality of the target adapter

3.4 Functional Specification of the Target Adapter

The target adapter (TA) refers to the lower part of the test adapter and runs on the target platform (Class B tests only). Therefore, it is usually specifically implemented for the target hardware.

It also may contain custom functions for testing certain BSW module functionalities.

Prior to the execution of the BSW conformance tests, the test personnel have to integrate the TA on the target platform. An overview on the integration issues has been given in Chapter 3.3.1.

3.4.1 Generic Functional Requirements

The target adapter has to provide the basic functionality described in the following sections.

3.4.1.1 Invocation of API Calls at the SUT

During test case execution, the SUT is stimulated by invocation of its APIs.

The test executable sends an API invocation together with optional arguments to the target adapter which then calls the API with the optional arguments at the BSW module. After execution of the API call, it returns with an optional return value (depending on the specification of the API). When the API call returns, the target adapter sends the appropriate notification to the test executable together with the optional return value.

3.4.1.2 Reception of API Calls from the SUT

During test integration, the test integrator has to make sure that the invocation of APIs belonging to other modules and called by the SUT is redirected to the target adapter.

When the test case is executed, the SUT may invoke these APIs belonging to other modules but provided by the target adapter. Upon reception of an API invocation from the SUT, the target adapter forwards the API invocation together with optional arguments to the test executable. The test case receives this API invocation and reacts to it according to the implemented test steps. This reaction usually involves sending of a reply notification to the target adapter indicating the return of the API invocation with optional return values.

3.4.1.3 Invocation of Callbacks towards the SUT

The BSW module under test may provide callbacks to other modules, e.g. for notification purposes.

For conformance testing, these callbacks are issued by the test case. Since callbacks towards the SUT behave exactly like APIs provided by the SUT, the same mechanism as described in Section 3.4.1.1 applies for callbacks towards the SUT.

3.4.1.4 Reception of Callbacks from the SUT

The BSW module under test may invoke callbacks towards other modules.

Again, the callbacks provided by other modules behave exactly like APIs provided by other modules and invoked by the SUT. Therefore, the same mechanism as described in Section 3.4.1.2 applies for callbacks invoked by the SUT towards other modules.

3.4.1.5 Reception of Error Reports

During test case execution, development errors and/or diagnostic events may occur within the BSW module under test. These errors and events are received by the target adapter and forwarded to the test executable where they are logged and a proper action (e.g., the setting of test verdict “fail”) is concluded from them.

The mechanisms for error and event reporting by the SUT make use of the APIs `Det_ReportError()` and `Dem_ReportErrorStatus()` provided by the modules "Development Error Tracer (DET)" and "Diagnostic Event Manager (DEM)", respectively.

For conformance testing, the target adapter provides these two APIs to the SUT instead of the real DET and DEM. When the SUT invokes one of these two APIs together with the arguments describing the error or event, the target adapter forwards the API invocation to the test executable. The test executable is then able to log the error or diagnostic event and to react appropriately on it.

3.4.2 Specific Functional Requirements

3.4.2.1 Pattern Form

This document describes solutions to problems that recur during the specification and implementation of AUTOSAR conformance tests for various BSW modules. Because of the recurring nature of the problems and solutions this document describes them in the form of patterns.

The patterns are described using a consistent format. Each pattern is divided into sections according to the following template:

- *Pattern name (required):*
  The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of the vocabulary.
- *Abstraction level (required):*
  The range of scale and abstraction of pattern:
  - *Test Architecture*
  - *Test Implementation*
- *Intent (required):*
  A short problem statement followed by a sketch and a summary.
- *Also known as (optional):*
  Other well-known names for the pattern, if any.
- *How it works (required):*
  This section describes the essence of how the pattern is structured (the solution to the problem) and what it is about.
- *When to use it (required):*
  This section describes the circumstances (the problem, the driving forces, and the context) which must be considered using the pattern.
- *Implementation notes (required):*
  This section describes the nuts and bolts of *how to implement* the pattern for
  - validation runs on simulating modules and running on development machines only during the CTSpec creation Action
  - test runs on production modules implemented in C and running on development machines or ECUs during the CTSpec application action

and *how to use* the pattern in test specifications (TTCN-3).

- *Consequences (required):*
  What are the trade-offs and results of using the pattern? What aspect of test system's structure does it let one vary independently?
- *Examples (required):*
  The test systems of which modules apply this pattern.
- *Related patterns (optional):*
  What design patterns are closely related to this one? What are the important differences?
- *References (optional):*
  References to related documents, e.g. to *AUTOSAR CTSpec Execution Constraints*.

Structures and functions are illustrated with UML diagrams, like component diagrams, sequence diagrams and deployment diagrams, and code snippets (TTCN-3 or C).

3.4.2.2 Patterns List

Pattern name	Intent
Network Stub - 5.1	How to control and observe the network side of SUTs with a network interface?
Memory accessor - 5.2	How to control and observe the effect of operations of SUTs on the main memory?
Average Neighbor - 5.3	How to replace Bsw modules and distribute test functionality that drives or stubs the SUT?

Note:

Apart from the above listed patterns, this document also contains a section "Important Notes". This section contains the important points that need to be considered while using the conformance test specifications.

4 Dynamic test organization (TTCN-3 files dependencies)

4.1 Common TTCN-3 files

Custom Tooling generated common TTCN-3 files

- <Msn>_config_types.ttcn

  This file contains configuration types for a BSW module <Msn>. This is a tool generated file. This file is generated from the AUTOSAR CTSPEC Meta model.
- <Msn>_config_parameters_default.ttcn

  This file contains configuration parameters default value for a BSW module <Msn>. This is a tool generated file. This file is generated from the AUTOSAR CTSPEC Meta model.
- <Msn>_api_types.ttcn

  This file contains API types for a BSW module <Msn>. This is a tool generated file. This file is generated from the AUTOSAR BSW UML model.
- <Msn>_api.ttcn

  This file contains declaration of APIs for a BSW module <Msn>. This shall be a tool generated file. This file is generated from the AUTOSAR BSW CTSPEC UML model.
- All_modules_generic_types.ttcn

  This file contains Standard and ComStack API Types.

Manually prepared common TTCN-3 file

- Common_base_functions.ttcn

  This file contains the API definitions of the commonly used functions across the modules.
- Dem_stub.ttcn

  This file provides the functionality of the DEM stub.
- Det_stub.ttcn

  This file provides the functionality of the DET stub.
- EcuM_stub.ttcn

  This file provides the functionality of the EcuM stub.
- Log_module.ttcn

  This file provides the functions for log messages.
- Test_strategy.ttcn

This file provides the test strategy for the module. The verdict of a test component (PTC and, at the end of a test case i.e. MTC) is automatically calculated and logged when a test component terminates.

- Validation_module.ttcn

  This file provides the validation for parameters and sets the verdict of a test case.

Note:

1. Tool generated and manually prepared common TTCN-3 files contain the information (configurations, types, etc.) for all AUTOSAR BSW modules (which are present in the CTSpec Meta model).
2. For a particular BSW module <Msn>_api_types.ttcn and <Msn>_api.ttcn implies the specific module and neighboring module API and API types. Example: If Flash Driver is the module under test, then <Msn>_api_types.ttcn implies Fls_api_types.ttcn, Fee_api_types.ttcn, Det_api_types.ttcn and <Msn>_api.ttcn implies Fls_api.ttcn, Fee_api.ttcn, Det_api.ttcn.

4.2 Module specific TTCN-3 files

Manually prepared module specific TTCN-3 files

- <Msn>_api_functions.ttcn
- <Msn>_test_suite.ttcn
- <Msn>_*_test_cases.ttcn
- <Msn>_*_base_functions.ttcn
- <Msn>_test_architecture.ttcn
- <Msn>_stub_signatures.ttcn
- <Msn>_*_stub.ttcn
- <Msn>_config_functions.ttcn
- <Msn>_pc_*_functions.ttcn
- <Msn>_test_api_functions.ttcn
- <Msn>_iterator_*_functions.ttcn

Tool generated module specific TTCN-3 files
AUTOSAR_<Msn>_Ecuc_<n>.par

4.3 TTCN-3 Files Import Dependencies

Definitions of one TTCN-3 module (file) can be used in other TTCN-3 module (file) by using “import” statements.

CTSpecs of each AUTOSAR BSW module is implemented in more than one file, hence each TTCN-3 file may import other module (s) (i.e. definition from other file)

e.g. If a test case in the file “<msn>_*_test_cases.ttcn” uses a utility function defined in the file “<msn>_*_base_functions.ttcn”, then the TTCN-3 module in the file “<msn>_*_test_cases.ttcn” will import the module defined in the file “<msn>_*_base_functions.ttcn”.

One sample of import is given below:

```
module IcuNotificationTestCases
{
  …
  import from IcuBaseFunctions all;
  …
}
```

Dependencies of each TTCN-3 file with other TTCN-3 files i.e. for every TTCN-3 file, the other modules (files) that are imported are provided in Figure 8 to Figure 25.

Note: File names are used instead of module names in diagrams for easy understanding. The difference between file name and module name is, in module name title case is used instead of underscores (as in file names).

e.g. File name: Icu_notification_test_cases.ttcn ;
TTCN-3 module name inside the file will be: IcuNotiifcationTestCases

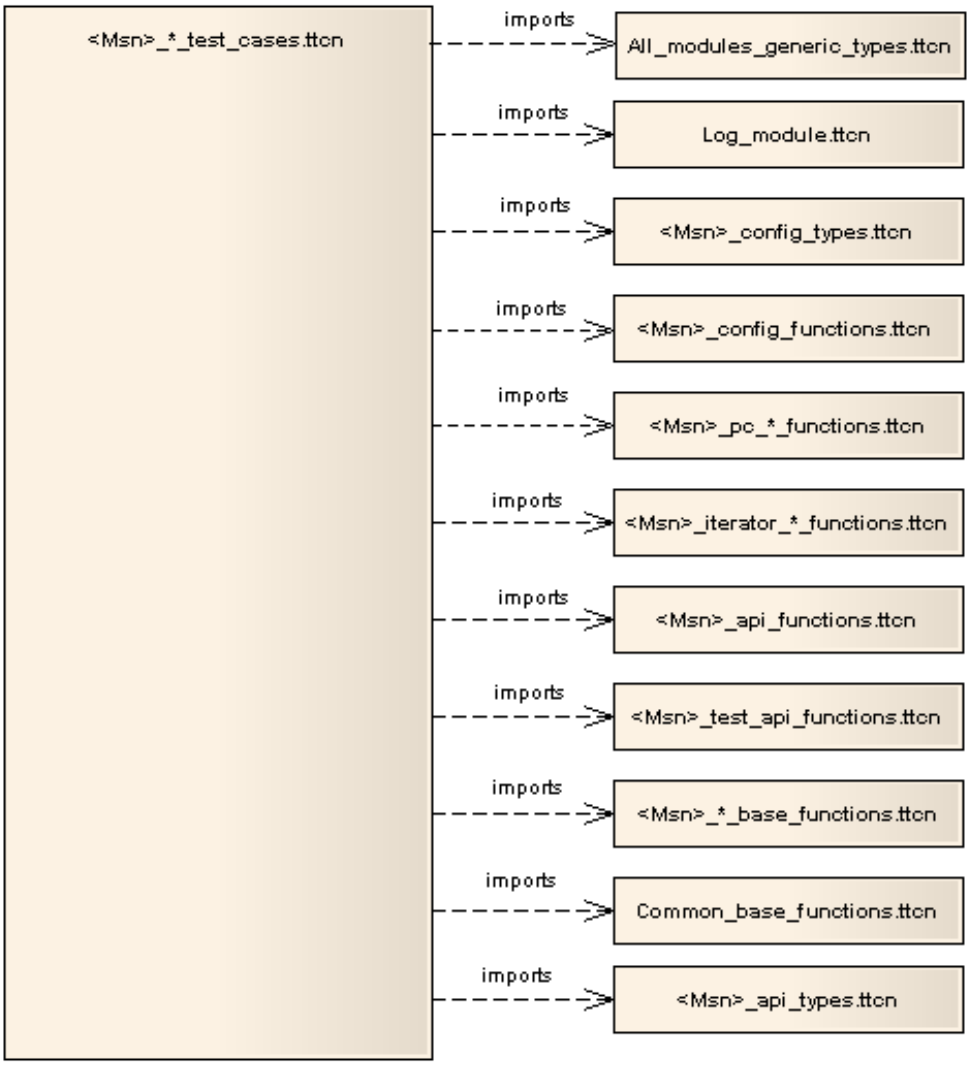


Figure 8: <Msn>_*_test_cases.ttcn

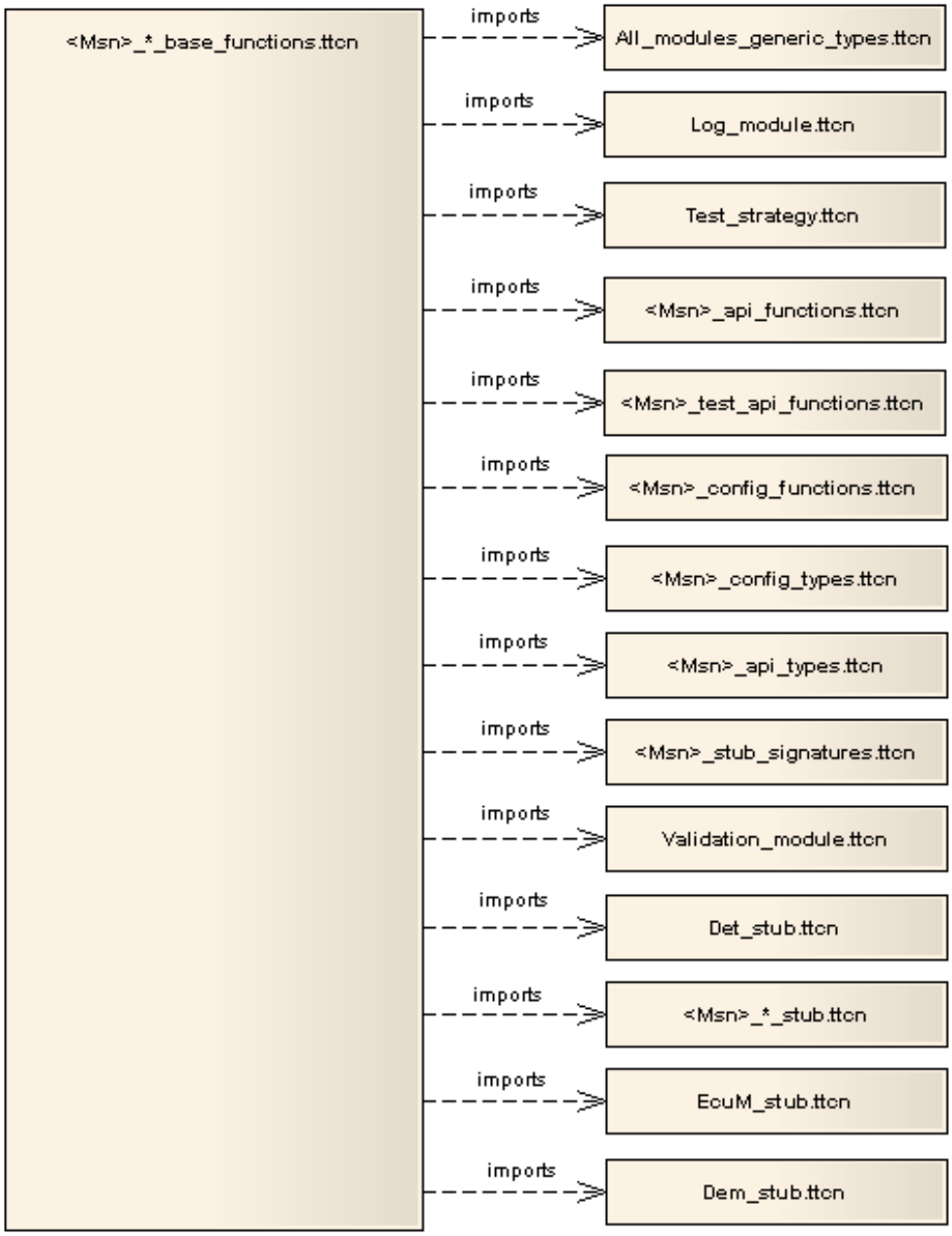


Figure 9: <Msn>_*_base_functions.ttcn

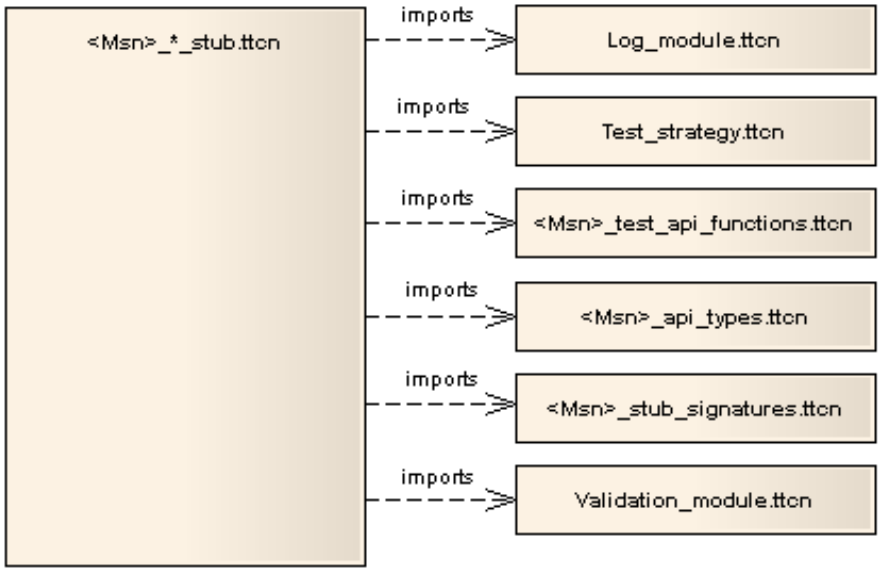


Figure 10: <Msn>_*_stub.ttcn

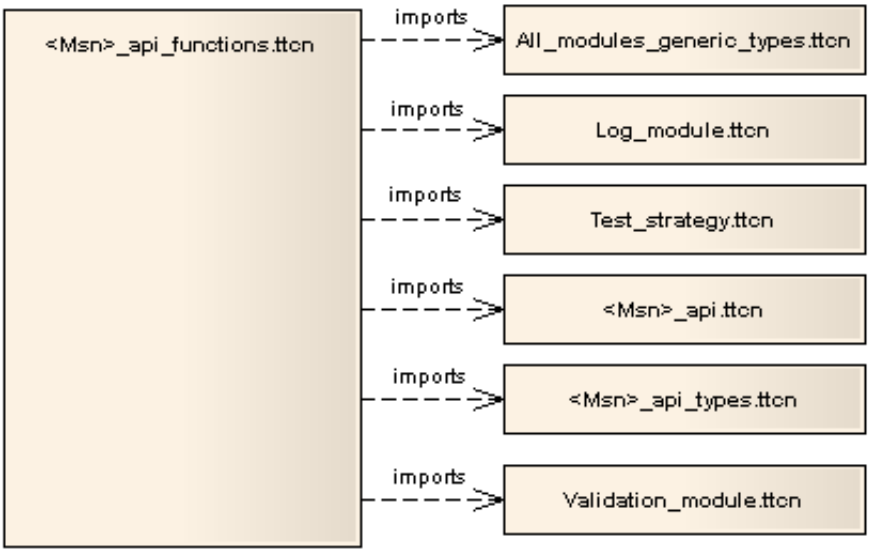


Figure 11: <Msn>_api_functions.ttcn

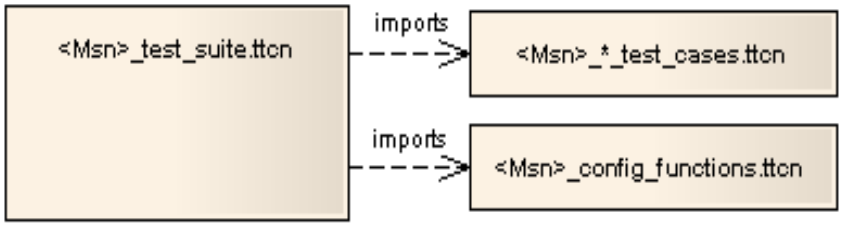


Figure 12: <Msn>_test_suite.ttcn

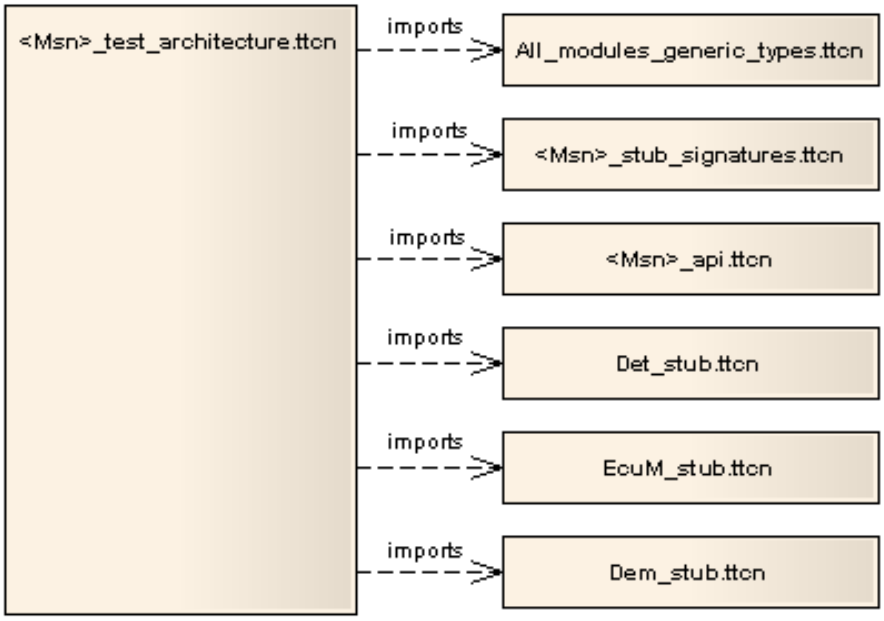


Figure 13: <Msn>_test_architecture.ttcn

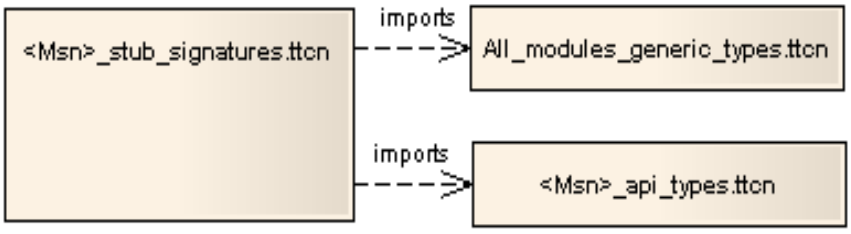


Figure 14: <Msn>_stub_signatures.ttcn

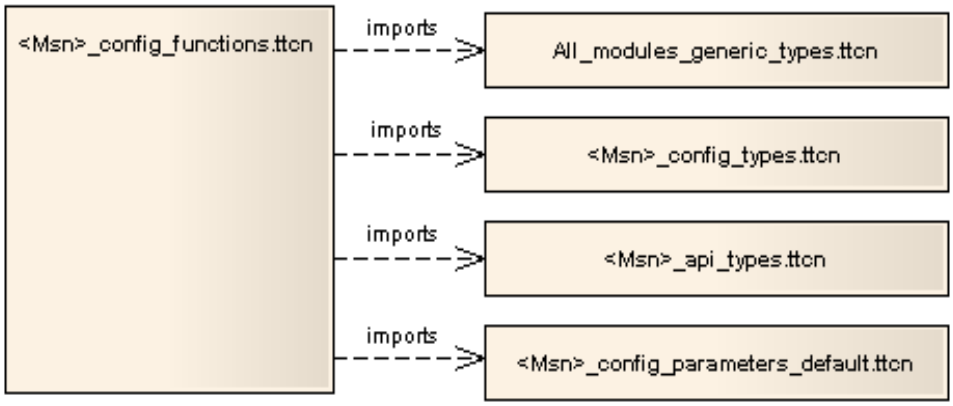


Figure 15: <Msn>_config_functions.ttcn

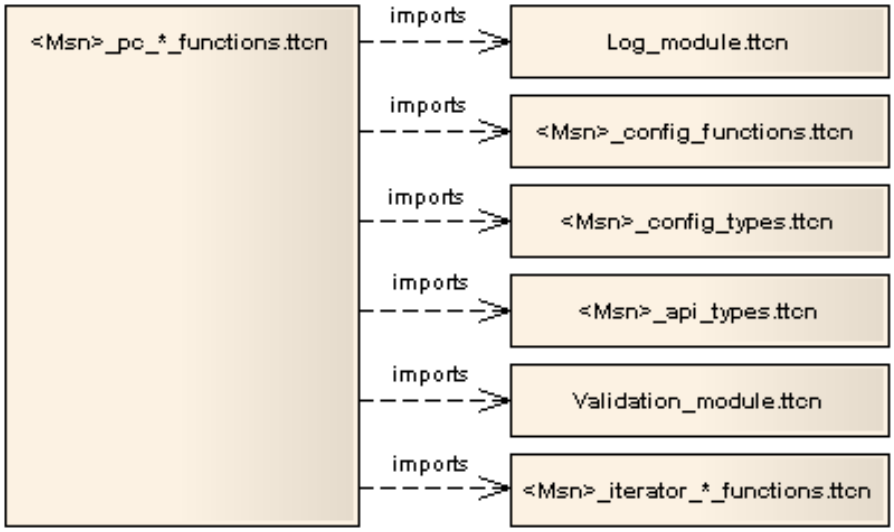


Figure 16: <Msn>_pc_*_functions.ttcn

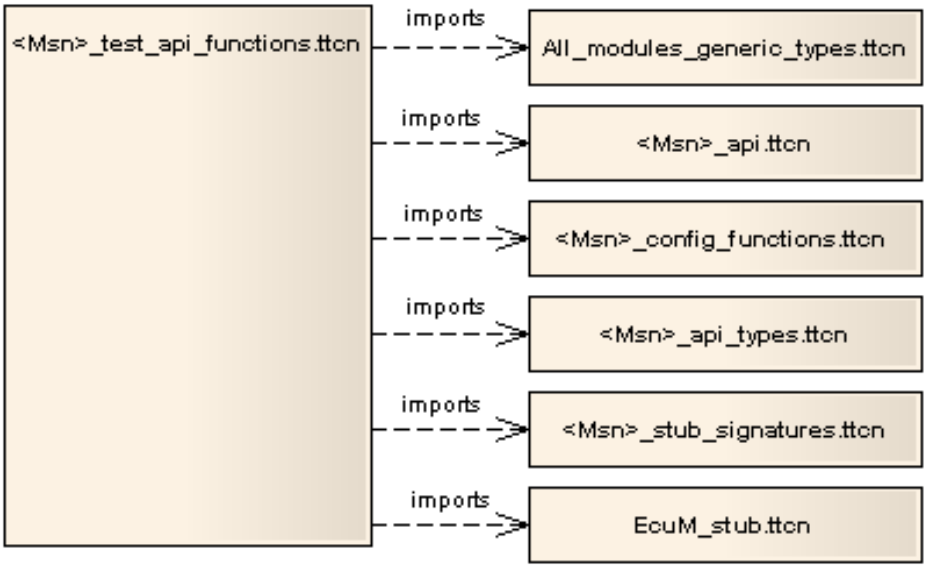


Figure 17: <Msn>_test_api_functions.ttcn

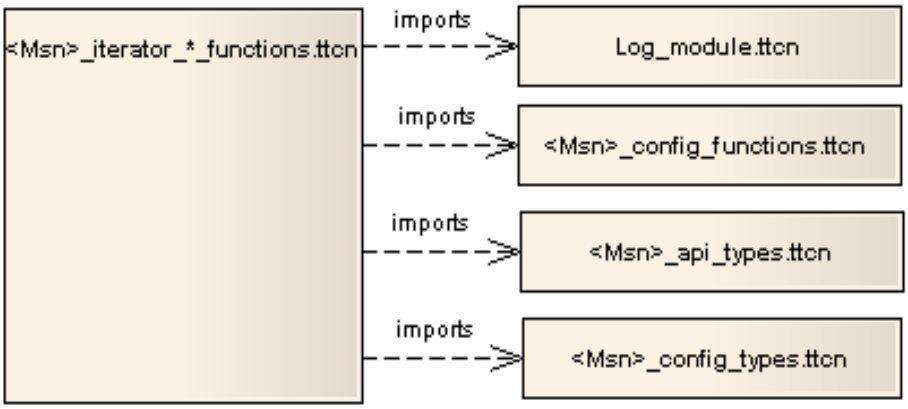


Figure 18: <Msn>_iterator_*_functions.ttcn

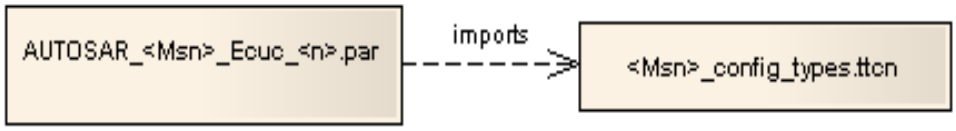


Figure 19: AUTOSAR_<Msn>_Ecuc_<n>.par

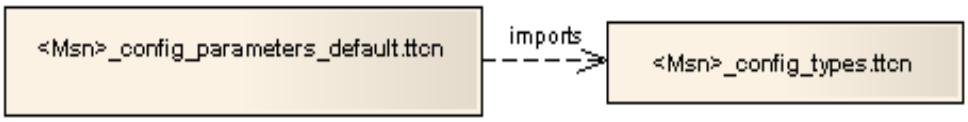


Figure 20: <Msn>_config_parameters_default.ttcn

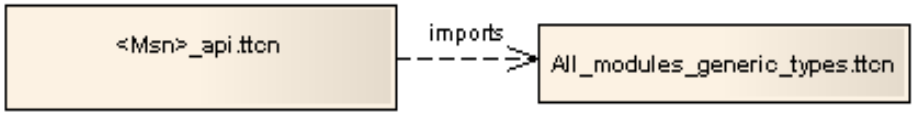


Figure 21: <Msn>_api.ttcn

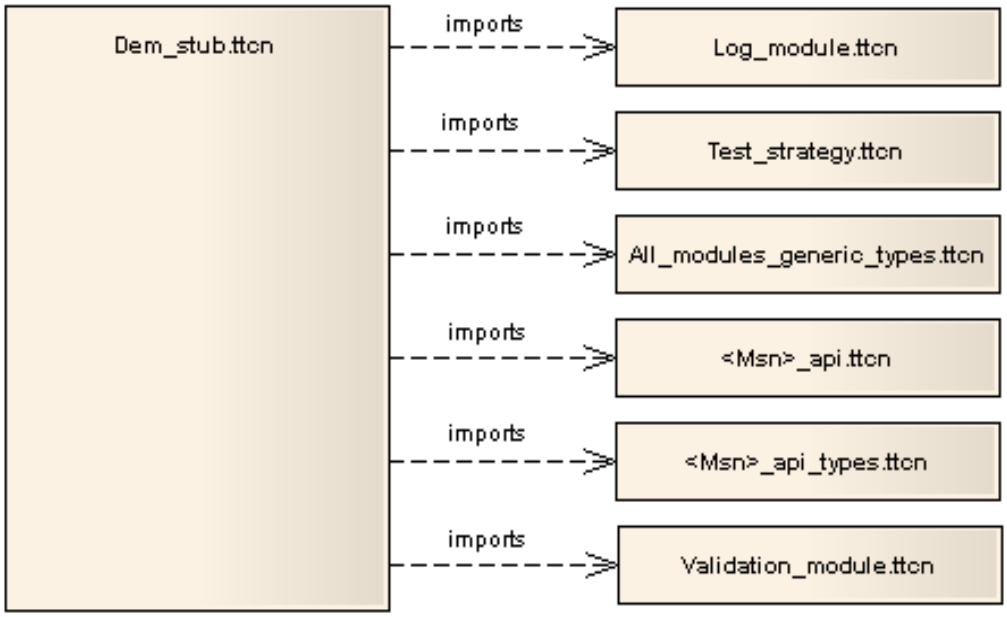


Figure 22: Dem_stub.ttcn

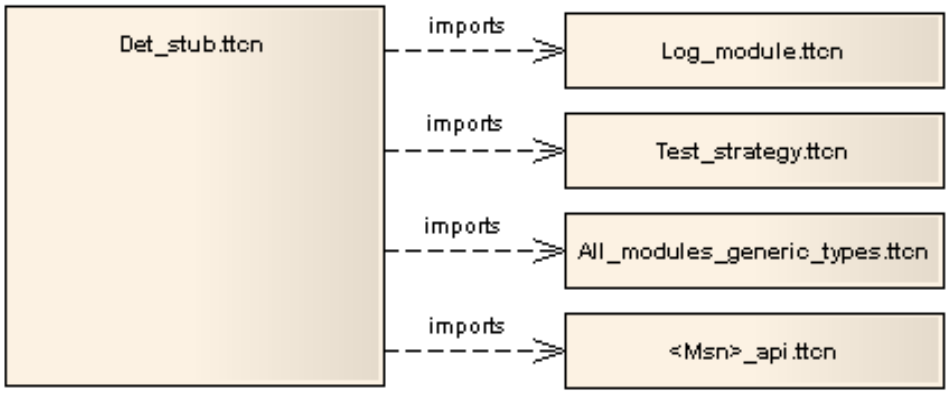


Figure 23: Det_stub.ttcn

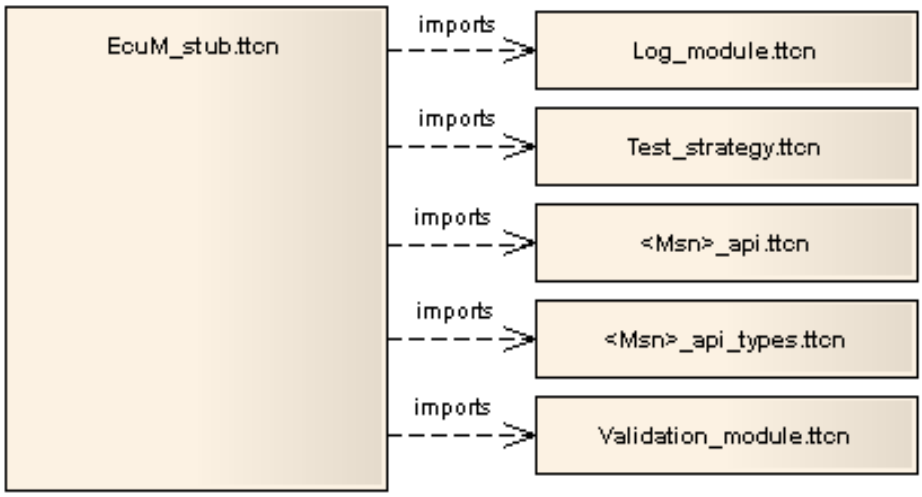


Figure 24: EcuM_stub.ttcn

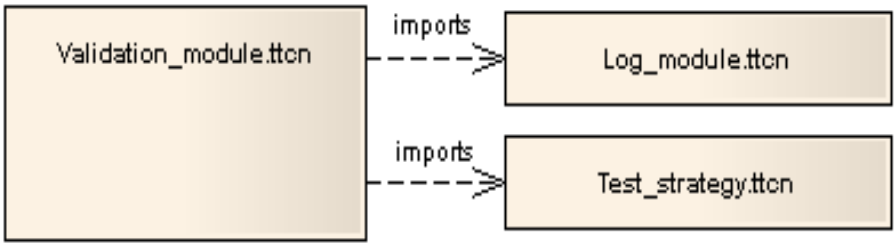


Figure 25: Validation_module.ttcn

Notes:

- The files All_modules_generic_types.ttcn and <Msn>_config_types.ttcn do not have any dependencies with other files
- The files <Msn>_config_types.ttcn, <Msn>_config_parameters_default.ttcn and AUTOSAR_<Msn>_Ecuc_<n>.par are not implemented (used) in any of the AUTOSAR BSW modules
- In files <Msn>_*_test_cases.ttcn and <Msn>_*_base_functions.ttcn importing of <Msn>_*_stub.ttcn and <Msn>_stub_signatures.ttcn is optional. Stubs and stub signatures will be imported by other file only if other file uses stub functions

In files <Msn>_*_test_cases.ttcn and <Msn>_pc_*_functions.ttcn importing of <Msn>_iterator_*.ttcn is optional. Iterator will be imported by other file only if other file uses iterator functions

4.4 Interaction between the TTCN-3 Scripts

- Test Suite will invoke the test cases implemented in the file "<msn>_*_test_cases.ttcn"
- Test case may invoke base functions defined in the file <msn>_*_base_functions.ttcn and/or the stub APIs defined in the file <msn>_*_stub.ttcn
- Mapping and Unmapping of the ports are defined in the base functions
- Default Altstep behavior is defined in the base functions for catching unexpected messages and as guard against waiting infinitely for responses from the SUT
- The neighboring modules APIs/callbacks are implemented as Parallel Test Components (PTCs) in the file(s) <msn>_*_stub.ttcn
- Within the test cases or base functions (that are invoked from the test cases), the behavior of the PTCs will be started and stopped as required by the test cases
- Test cases and base functions are part of Main Test Component (MTC)

To check whether SUT has invoked neighboring module's APIs/callbacks, MTC will use user defined stub functions that are implemented in PTCs

5 Dynamic test patterns

5.1 Network Stub

How to control and observe the network sid of a module with a network interface?

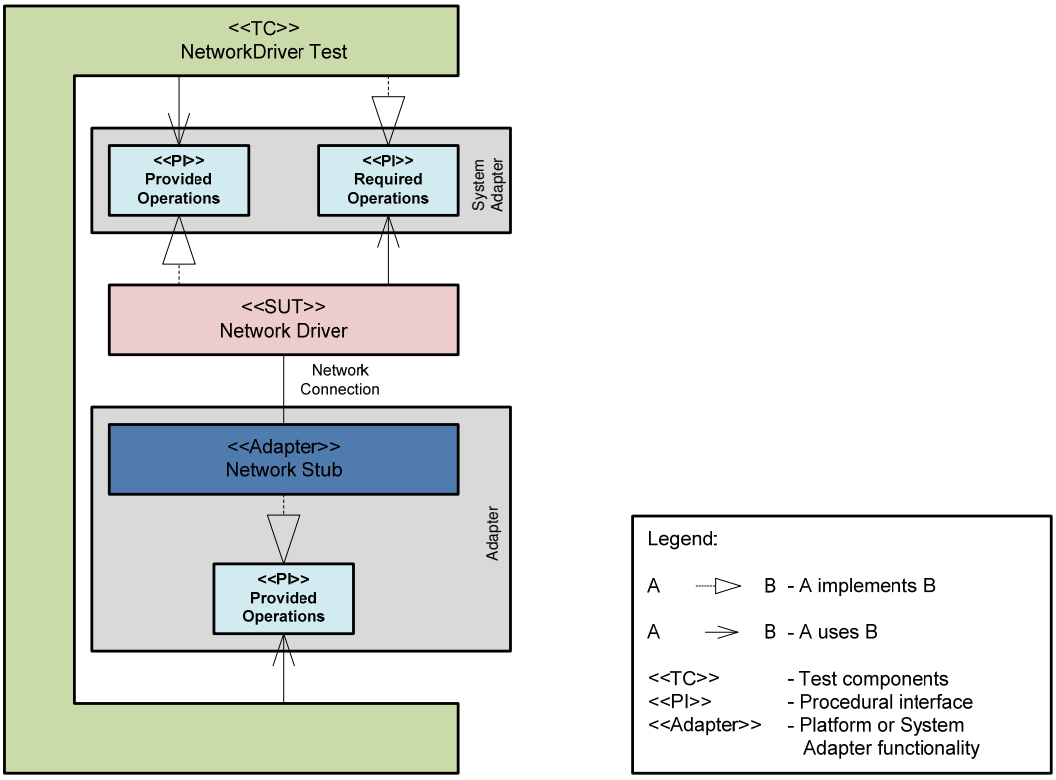


Figure 26: Network Stub overview

5.1.1 How it works

When a module is a network driver it can be tested when the behavior of the network hardware which the module is controlling can be observed and stimuli can be induced. Hence the test components need to be put into the position to send and receive messages on the same network that the module under test is connected to.

The network Stub will provide this functionality to the test components. Its interface is published to the test components in a similar way the SUTs interfaces are published (see Figure 26). This interface is specific to the network technology which is tested but not specific to a specific hardware or vendor. The test components will use this interface provided by the network Stub to induce message, read messages from the

network or trigger or observe other technology specific effects (e.g. bus off events). It can thus be considered to be a part of the System Adapter or Platform Adapter.

5.1.2 When to use it

Whenever the module under test is controlling hardware that can be observed using a well defined and standardized hardware interface this pattern should be applied. This well standardized interface is the connection between the network driver and the network Stub in Figure 26. Hardware related functionality is in general not supported in the conformance test. Hence, this pattern is only to be applied for network technologies which are an exception from that rule.

5.1.3 Implementation notes

The interface that the network Stub provides to the test components needs to be defined in a formal way (lower box provided interface in **Figure 26**). The TTCN-3 API is standardized along with the tests themselves. The system or platform adapter needs to implement this interface in a proprietary way and provide the functionality to the test components. The network Stub establishes an independent procedural synchronous interface to access the network. The implementation of this interface can be accomplished for example using third party libraries or proprietary software modules.

The validation environment will implement this behavior as part of the simulation. The network technology will be simulated internally to the simulation; therefore no real network exists during validation. In a real test setup, the same functionality needs to be provided using a real network for testing.

The test case can use the stub to send messages and will expect to receive the identical message using the AUTOSAR API of the network module. Coarsely a test case could look like this:

```
testcase TC_NETWORKXYZ_0001() runs on MTCXYZ {

     setverdict(none);

     int param1a = 1;
     octetstring param2a = "0xFFFAAA444111";

     // send data to SUT on network using the Stub
     NetworkStubName_sendMsg(param1a, param2a);

     // receive data on the SUT
      NetworkAPI_receive(param1b, param2b);

     // compare data
     if (param1a == param1b && param2a == param2b) {
          // received data was matching sent data
```

```
            setverdict(pass);
        }
        else {
            // data was not transmitted correctly
            setverdict(fail);
        }
}
```

This listing also illustrates why it is sensible to have the Network Stub use procedural interfaces. Once the call to the stub returns the MTC can be certain that the message is actually transmitted and on the network.

For the TTCN-3 side the signatures provided should enable the test to send and receive messages on the network. Hence they should be similar to

```
signature NetworkStubName_sendMsg ( in type1 param1,
  in type2 param2) exception (Std_ExceptionType_);

signature NetworkStubName_getLastReceivedMsg ( out type1 pa-
ram1,
  out type2 param2) exception (Std_ExceptionType_);
```

Where param1 and param2 are describing the message itself.

The implementation of the Network Stub which would need to be done as part of the implementation of the test system and could look similar to this pseudo code:

```
void NetworkStubName_sendMsg (type1* param1, type2* param2) {
    Message msg = processMessageParameters(param1, param2);
    sendMessageOntoNetwork(msg);
}
```

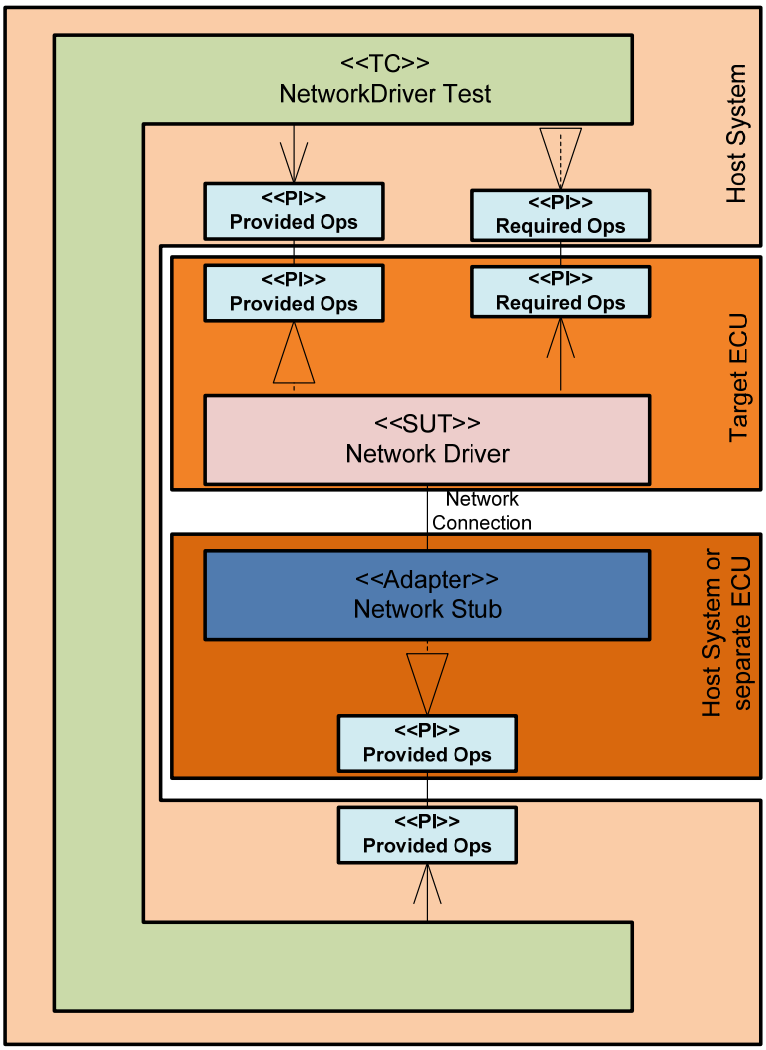


Figure 27: Class-B scenario deployment view of the Network Stub

The above figure shows a possible deployment of the network Stub in a Class-B test scenario. The host system itself will execute the test components and contain one part of the system adapter which ensures the interfacing of SUT and test components. The system adapter then connects the test system to the target ECU where the SUT itself is running.

On this ECU the network driver module interfaces with the network hardware and thus allows the SUT to transmit and receive messages on the network. The network itself ensures the connection to the network Stub. Here a real physical network connection will be used.

The network Stub itself can be deployed either on the host system or on a separate ECU. There the network Stub receives and transmits information to and from the network that the SUT is connected to.

The interface of the network Stub and the test system itself is very similar to the interface between test system and SUT. The system adapter allows the test components to access the functionality of the network Stub. If the network Stub is collocated with the test components on the host system the system adapter will not need to cross a machine boundary which would be the case where the network Stub is located on a separate ECU.

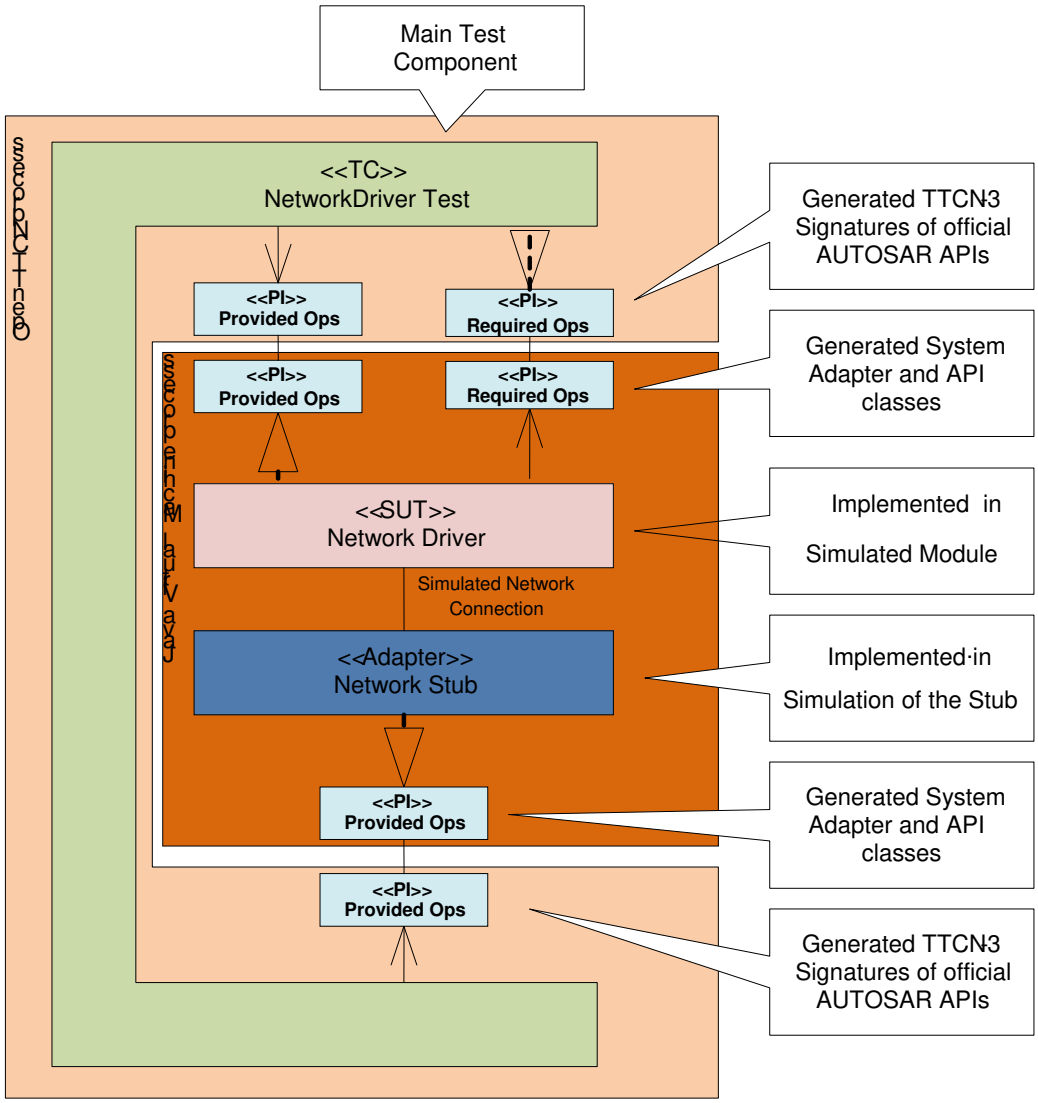


Figure 28: Validation scenario deployment diagram of the Network Stub

The figure above shows the same setup but in the validation scenario. Here the module under test consists of a simulation of a BSW module that runs in the same virtual machine with the Network Stub. The connection of the Network Stub and the BSW Module here is not the real network technology but this connection itself is simulated.

To the test system there is no difference as the system adapter along with the generated interfaces in TTCN-3 handles the connection to the simulation.

5.1.4 Consequences

The network Stub pattern allows testing hardware specific modules without requiring test components that are specific to one particular hardware implementation. The pattern harnesses the fact that network technologies specify a standardized interface on the network level which is also relatively easy to access. This interface is accessed in an arbitrary and proprietary way but the communicated information is standardized. This is then used by the test component for testing.

5.1.5 CAN Network Stub

CAN Network stub (CanStub) uses the following interfaces for the CanStub (representation in TTCN-3) which allows the test components to access a CAN network that is connected to SUT which itself interacts with this CAN network

```
a) signature TestCanNet_SendCanDataFrame(
     in integer CanControllerId,
     in Can_IdType_ CanId, in integer SduLength,
     in octetstring Sdu)
     exception (Std_ExceptionType_);

b) signature TestCanNet_GetLastReceivedCanDataFrame(
     in integer CanControllerId,  in Can_IdType_ CanId,
     out integer SduLength, out octetstring Sdu) return boo-
lean
     exception (Std_ExceptionType_);

c) signature TestCanNet_SendRemoteFrame(
     in integer CanControllerId,
     in Can_IdType_ CanId, in integer SduLength)
     exception (Std_ExceptionType_);

d) signature TestCanNet_Clear(in integer ControllerId)
     exception (Std_ExceptionType_);
```

5.1.5.1 TestCanNet_SendCanDataFrame

Function Name	TestCanNet_SendCanDataFrame
Syntax	TestCanNet_SendCanDataFrame(in integer CanControllerId, in Can_IdType_ CanId, in integer SduLength, in octetstring Sdu)

Parameters In	CanControllerId – ID of the controller to which the CAN frame should be sent CanId – CAN frame ID SduLength – Length of the CAN frame Sdu – CAN frame payload data bytes
Parameters Out	-
Return Value	-

Description:

The method TestCanNet_SendCanDataFrame is used by the test case to send a data frame onto a CAN bus to which the SUT is connected. The test component will provide all the necessary information to the module, like the ID of the controller on which the SUT should receive the message. Also all necessary information to construct the message like Can_Id, PduLength and the actual payload are provided to the test component by the CanStub. The CanStub will use that information to transmit the CAN frame and only after the transmission of the CAN frame return to the test synchronously.

Example:

```
var octetstring v_Data := '1122334455667788'O;
TestCanNet_SendCanDataFrame(0, 100, 8, v_Data);
```

This will send a CAN frame from the network stub to the controller "0" in the SUT, with CAN ID = 100, CAN frame data length = 8 and CAN frame data bytes = 0x1122334455667788, where CAN frame data byte[0] = 0x11.

5.1.5.2 TestCanNet_GetLastReceivedCanDataFrame

Function Name	TestCanNet_GetLastReceivedCanDataFrame
Syntax	TestCanNet_GetLastReceivedCanDataFrame(in integer CanControllerId, in Can_IdType_ CanId, out integer SduLength, out octetstring Sdu)
Parameters In	CanControllerId – ID of the controller from which the CAN frame should be received CanId – CAN frame ID
Parameters Out	SduLength – Length of the CAN frame Sdu – CAN frame payload data bytes
Return Value	boolean True – if frame is received successfully by the CanStub False – if frame is not received or if the frame is re-

	ceived with errors by the CanStub

Description:

The method TestCanNet_GetLastReceivedCanDataFrame is used by the test case to get the previously received CAN frame by the CanStub.

The CanStub continuously reads messages from each CAN bus to which it is connected and stores the last received message. It allows the test component to read the last message that was received on any of the available CAN busses.

The CAN network stub shall update the out parameters for the latest received CAN frame from the controller “CanControllerId” and with CAN ID “CanId“:

Sdu - received CAN frame data bytes
SduLength – Number of data bytes received

The return value indicates whether a message has been received without any errors.

Example:

```
var octetstring v_Data := 'AABBCCDD'O;
TestCanNet_GetLastReceivedCanDataFrame(1, 200, 4, v_Data);
```

This means that the latest CAN frame received (by the CanStub) for the CAN ID “200” on the controller “1” is having “4” bytes length with data bytes = 0xAABBCCDD, where CAN frame data byte[0] = 0xAA.

5.1.5.3 TestCanNet_SendRemoteFrame

Function Name	TestCanNet_SendRemoteFrame
Syntax	TestCanNet_SendRemoteFrame (in integer CanControllerId, in Can_IdType_ CanId, in integer SduLength)
Parameters In	CanControllerId – ID of the controller for which the remote CAN frame should be initiated CanId – CAN frame ID SduLength – Length of the CAN frame
Parameters Out	-
Return Value	-

Description:

The method TestCanNet_SendRemoteFrame is used by the test case to send a remote CAN frame from CanStub to SUT for the CAN ID “CanId”.

After receiving this call, the CanStub should send a remote CAN frame on to the controller = “CanControllerId” on the SUT, with CAN ID = “CanId” and DLC = “SduLength”

Example:

```
TestCanNet_SendRemoteFrame(1, 300, 6);
```

This will send a remote CAN frame from the network stub to the controller “1” in the SUT, with CAN ID = 300, CAN frame data length = 6.

5.1.5.4 TestCanNet_Clear

Function Name	TestCanNet_Clear
Syntax	TestCanNet_Clear(in integer CanControllerId)
Parameters In	CanControllerId – ID of the controller for which the previously stored information to be cleared.
Parameters Out	-
Return Value	-

Description:

The method TestCanNet_Clear is for administrative purposes only and allows the test to clear all messages that might be stored in the CanStub for a particular controller (CanControllerId).

After receiving this call, the CanStub should clear all the messages that are stored in the CanStub for the controller “CanControllerId“

Example:

```
TestCanNet_Clear(0);
```

This will clear the stored messages (in the CanStub) for the controller “0”.

e.g. The call TestCanNet_GetLastReceivedCanDataFrame will return “False” immediately after this call.

5.1.6 CAN Transceiver Network Stub

CAN Transceiver Network stub (CanTrcvStub) uses the following interface for the CanTrcvStub (representation in TTCN-3) which allows the test components to send a wakeup pattern on a required channel by CanTrcvStub:

```
signature TestCanTrcvNet_SendWakeUpPattern(in integer CanNet-
work)
  exception (Std_ExceptionType_);
```

Function Name	TestCanTrcvNet_SendWakeUpPattern
Syntax	TestCanTrcvNet_SendWakeUpPattern(in integer Can-Network)
Parameters In	CanNetwork – CAN Transceiver channel on which the wakeup pattern should be sent
Parameters Out	-
Return Value	-

Description:

The method TestCanTrcvNet_SendWakeUpPattern is used by the test case to send a wakeup pattern on a requested CAN Transceiver network (CanNetwork) by the CanTrcvStub.

After receiving this call CanTrcvStub shall send a wakeup pattern on the requested network. E.g. to trigger a wakeup the CanTrcvStub can initiate a CAN frame transmission from the CanTrcvStub.

Example:

```
TestCanTrcvNet_SendWakeUpPattern(1);
```

This will send a wakeup pattern from CanTrcvStub to the Transceiver network ”1” on SUT

5.1.7 LIN Network Stub

LIN Network stub (LinStub) uses the following interfaces for the LinStub (representation in TTCN-3) which allows the test components to access a LIN network that is connected to SUT which itself interacts with this LIN network :

```
a) signature TestLinNet_SetUpResponse(in integer Channel,
     in Lin_PduType_ pduInfoPtr, in octetstring Sdu)
     exception (Std_ExceptionType_);

b) signature TestLinNet_GetLastFrame(in integer Channel,
     out Lin_PduType_ pduInfoPtr, out octetstring Sdu)
    return boolean
     exception (Std_ExceptionType_);
```

```
c) signature TestLinNet_SendWakeupPulse(in integer Channel)
     exception (Std_ExceptionType_);
```

Note: In the above signatures a) and b), parameter pduInfoPtr.SduPtr is not used. Instead "Sdu" of type octetstring is used.

By using octetstring, LinStub need not use the memory accessor stub to read the LIN frame data, since the LIN frame data is directly available in the function call through the parameter "Sdu".

5.1.7.1 TestLinNet_SetUpResponse

Function Name	TestLinNet_SetUpResponse
Syntax	TestLinNet_SetUpResponse (in integer Channel, in Lin_PduType_ pduInfoPtr, in octetstring Sdu)
Parameters In	Channel – LIN channel on which the LIN frame response should be sent pduInfoPtr – PDU containing the PID, checksum model, response type, Data Length Sdu – LIN frame data
Parameters Out	-
Return Value	-

Description:

The method TestLinNet_SetUpResponse is used by the test case to setup a response for particular LIN frame (specified through the parameters pduInfoPtr and Sdu).).

The test case provides the necessary information for which the LinStub should be respond to a LIN frame header. By receiving this call LinStub shall store the information locally and respond to the LIN header (received from SUT) for the PID = pduInfoPtr.Pid with response data provided Sdu and appropriate check sum byte.

Example:

Test case (TTCN-3):

```
var Lin_PduType_ v_PduInfoPtr;

/* Set Data */
var octectstring v_Sdu := '1122334455667788'O;

/* Set PID = 128 */
v_PduInfoPtr.Pid := 128;
```

```
/* Set Data length = 8 */
v_PduInfoPtr.DI := 8;

/* Set check sum model is classic checksum */
v_PduInfoPtr.Cs := LIN_CLASSIC_CS;

/* Set frame direction is slave response frame */
v_PduInfoPtr.Drc := LIN_SLAVE_RESPONSE;

/* Initiate a call to LinStub to set up the response */
pt_TestLinNet.call(TestLinNet_SetUpResponse:{0, v_PduInfoPtr,
v_Sdu},
  nowait);
```

LinStub:

```
Setp1: Read the frame details: Pid, DI, Cs and Drc from pduIn-
foPtr and store them locally
Step2: Read the data bytes from the Sdu and store the data lo-
cally
Step3: Return to test system
Step4: LinStub receives a LIN frame header on channel "0" with
Pid = 128.
Step5: LinStub responds to the header with 0x1122334455667788
and check sum byte = the check sum calculated based on the
checksum model pduInfoPtr.Cs
```

5.1.7.2 TestLinNet_GetLastFrame

Function Name	TestLinNet_GetLastFrame
Syntax	TestLinNet_GetLastFrame (in integer Channel, out Lin_PduType_ pduInfoPtr, out octetstring Sdu)
Parameters In	Channel – LIN channel on which the LIN frame response was received
Parameters Out	pduInfoPtr – PDU containing the PID, checksum model, response type, Data Length Sdu - LIN frame data
Return Value	boolean True – if frame is received successfully by the LinStub False – if frame is not received or if the frame is received with errors by the LinStub

Description:

The method TestLinNet_GetLastFrame is used by the test case to get the previously received LIN frame by the LinStub.

The LinStub continuously reads messages from each LIN bus to which it is connected and stores the last received message. It allows the test component to read the last message that was received on any of the available LIN busses.

The LinStub shall update the out parameters for the latest received LIN frame from the channel “Channel”

The return value indicates whether a message has been received without any errors.

Example:

```
/* The below variables are initialized by LinStub (representation in “C” language) */
pduInfoPtr.Pid = 3;
pduInfoPtr.Drc = LIN_MASTER_RESPONSE;
pduInfoPtr.DI = 8;
pduInfoPtr.Cs = LIN_CLASSIC_CS;
Sdu = 0x1122334455667788;
TestLinNet_GetLastFrame(0, pduInfoPtr, Sdu);
```

This means LinStub was received a LIN frame with PID =3, 8 data bytes with 0x1122334455667788 and checksum byte that was calculated using classic check sum model

LinStub:

Setp1: Receive the LIN frame from the bus and store the received frame information (Pid, data bytes , length, checksum model, etc.)
Step2: If a call to TestLinNet_GetLastFrame is received from the test case, then do the following:

a. Prepare pduInfoPtr using the information from Step1.
   pduInfoPtr.Pid = The PID received
   pduInfoPtr.DI = number of data bytes received
   Sdu = received data bytes on the network
   ...
b: Return to test system

5.1.7.3 TestLinNet_SendWakeupPulse

Function Name	TestLinNet_SendWakeupPulse
Syntax	TestLinNet_SendWakeupPulse (in integer Channel)
Parameters In	Channel – LIN channel on which the wakeup pulse should be sent
Parameters Out	-
Return Value	-

Description:

The method TestLinNet_SendWakeupPulse is used by the test case to send a wakeup pulse on a requested LIN channel (Channel) by the LinStub.

After receiving this call LinStub shall send a wakeup pulse on the requested channel. The wakeup pulse should be according to the LIN2.x protocol specifications (i.e. a dominant pulse of > 250us and <5ms).

Example:

```
TestLinNet_SendWakeupPulse(1);
```

This will send a wakeup pulse from LinStub to the LIN channel ”1” on SUT

(Hint: Sync break can be used as wakeup pulse, i.e. LinStub can send a sync break for the call TestLinNet_SendWakeupPulse)

5.1.8 LIN Transceiver Network Stub

LIN Transceiver Network stub (LinTrcvStub) uses the following interface for the LinTrcvStub (representation in TTCN-3) which allows the test components to send a wakeup pattern on a required channel by LinTrcvStub:

```
signature TestLinTrcvNet_SendWakeUpPattern(in integer LinNet-
work)
  exception (Std_ExceptionType_);
```

Function Name	TestLinTrcvNet_SendWakeUpPattern
Syntax	TestLinTrcvNet_SendWakeUpPattern(in integer Lin-Network)
Parameters In	LinNetwork – LIN Transceiver channel on which the wakeup pattern should be sent
Parameters Out	-
Return Value	-

Description:

The method TestLinTrcvNet_SendWakeUpPattern is used by the test case to send a wakeup pulse on a requested LIN Transceiver network (LinNetwork) by the LinTrcvStub.

After receiving this call LinTrcvStub shall send a wakeup pulse on the requested network. The wakeup pulse should be according to the LIN2.x protocol specifications (i.e. a dominant pulse of > 250us and <5ms).

Example:

```
TestLinTrcvNet_SendWakeUpPattern(1);
```

This will send a wakeup pulse from LinTrcvStub to the Transceiver network ”1” on SUT
(Hint: Sync break can be used as wakeup pulse, i.e. LinTrcvStub can send a sync break for the call TestLinNet_SendWakeupPulse)

5.1.9 FlexRay Network Stub

FlexRay Network stub (FrStub) uses the following interfaces for the FrStub (representation in TTCN-3) which allows the test components to access a FlexRay network that is connected to SUT which itself interacts with this FlexRay network

```
a) signature TestFrNet_TransmitFrame(in integer FrCtrlIdx,
      in Fr_ChannelType_ FrChnlIdx,
      in integer FrLpduIdx,
      in integer FrLSduLength,
      in octetstring Sdu)
      exception (Std_ExceptionType_);

b) signature TestFrNet_ReceiveWakeup(in integer FrCtrlIdx,
      in Fr_ChannelType_ FrChnlIdx,
      out CTFr_WakeupStatusType_ FrWakeupStatus)
      exception (Std_ExceptionType_);

c) signature TestFrNet_GetTransmittedData(in integer
   FrCtrlIdx,
      in Fr_ChannelType_ FrChnlIdx,
      in integer FrLpduIdx,
      out integer FrSduLength,
      out octetstring Sdu)
      exception (Std_ExceptionType_);

d) signature TestFrIH_CheckDisableAbsoluteTimerIRQ(
     in charstring VP,
      in integer FrCtrlIdx,
      in integer FrAbsTimerIdx,
      in integer FrCycle,
      in integer FrOffset)
      return Std_ReturnType_
      exception (Std_ExceptionType_);

e) signature TestFrIH_CheckCancelAbsoluteTimer(in charstring
   VP,
      in integer FrCtrlIdx,
      in integer FrAbsTimerIdx,
      in integer FrCycle,
      in integer FrOffset)
      return Std_ReturnType_
```

```
      exception (Std_ExceptionType_);

f) signature TestFrIH_CheckAckAbsoluteTimerIRQ(in charstring
   VP,
      in integer FrCtrlIdx,
      in integer FrAbsTimerIdx,
      in integer FrCycle,
      in integer FrOffset,
      out boolean FrStatusPtr)
      return Std_ReturnType_
      exception (Std_ExceptionType_);

g) signature TestFrIH_CheckTransmitTxLpdu(in charstring VP,
      in integer FrCtrlIdx,
      in integer FrAbsTimerIdx,
      in integer FrCycle,
      in integer FrOffset,
      in integer FrLpduIdx,
      in integer FrSduLength,
      in octetstring Sdu)
      return Std_ReturnType_
      exception (Std_ExceptionType_);

h) signature TestFrIH_CheckSetAbsoluteTimer(in charstring
   VP,
      in integer FrCtrlIdx,
      in integer FrAbsTimerIdx,
      in integer FrCycle,
      in integer FrOffset,
      out integer FrCyclePtr,
      out integer FrOffsetPtr)
      return Std_ReturnType_
      exception (Std_ExceptionType_);

i) signature TestFrIH_CheckReceiveRxLpdu(in charstring VP,
      in integer FrCtrlIdx,
      in integer FrAbsTimerIdx,
      in integer FrCycle,
      in integer FrOffset,
      in integer FrLpduIdx,
      in octetstring Sdu,
      out Fr_RxLPduStatusType_ FrLPduStatusPtr,
      out integer FrSduLengthPtr)
      return Std_ReturnType_
      exception (Std_ExceptionType_);

j) signature TestFrIH_CheckPrepareLpdu(in charstring VP,
      in integer FrCtrlIdx,
      in integer FrAbsTimerIdx,
      in integer FrCycle,
```

```
      in integer FrOffset,
      in integer FrLpduIdx,
      in integer FrSduLength,
      in octetstring Sdu,
      out Fr_RxLPduStatusType_ FrLPduStatusPtr)
      return Std_ReturnType_
      exception (Std_ExceptionType_);

k) signature TestFrIH_CheckGetGlobalTime(in charstring VP,
      in integer FrCtrlIdx,
      out integer FrCyclePtr,
      out integer FrOffsetPtr)
      return Std_ReturnType_
      exception (Std_ExceptionType_);

l) signature TestFrIH_CheckGetAbsoluteTimerIRQStatus(
     in charstring VP,
      in integer FrCtrlIdx,
      in integer FrAbsTimerIdx,
      in integer FrCycle,
      in integer FrOffset,
      out boolean FrStatusPtr)
      return Std_ReturnType_
      exception (Std_ExceptionType_);

m) signature TestFrIH_CheckEnableAbsoluteTimerIRQ(
     in charstring VP,
      in integer FrCtrlIdx,
      in integer FrAbsTimerIdx,
      in integer FrCycle,
      in integer FrOffset)
      return Std_ReturnType_
      exception (Std_ExceptionType_);
```

5.1.9.1 TestFrNet_TransmitFrame

Function Name	TestFrNet_TransmitFrame
Syntax	TestFrNet_TransmitFrame (in integer FrCtrlIdx, in Fr_ChannelType_ FrChnlIdx, in integer FrLpduIdx, in integer FrLSduLength, in octetstring Sdu)
Parameters In	FrCtrlIdx – The Controller on which the FlexRay frame to be transmitted by FrStub FrChnlIdx – FlexRay Channel (A or B or AB) FrLpduIdx – <<>> FrLSduLength – Number data bytes in the FlexRay frame

	Sdu – FlexRay frame payload data
Parameters Out	-
Return Value	-

Description:

The method TestFrNet_TransmitFrame is used by the test case to send a FlexRay frame from FrStub to the SUT

The test case provides the necessary information for which the FrStub should send a FlexRay frame. By receiving this call FrStub shall send a FlexRay frame with the data in “Sdu” on the channel “FrChnlIdx” of the controller “FrCtrlIdx” of the SUT

Example:

Test case (TTCN-3):

```
/* Allocate 10 bytes memory */
var PointerAddr_ v_SduPtr:=
pt_TestMemoryAccess.call(TestMemoryAccess_Allocate:{8},
nowait);

/* Write 10 bytes data into the memory v_SduPtr */
pt_TestMemoryAccess.call(TestMemoryAccess_Write:{ v_SduPtr, 8,
  '11223344556677889AA'O}, nowait);


pt_TestFrNet.call(TestFrNet_TransmitFrame:{0, FR_CHANNEL_A, 1,
10, v_SduPtr}, nowait);
```

FrStub:

```
Step1: Read the data from the pointer v_SduPtr using memory
accessor stub (i.e. Data[] = TestMemoryAccess_Read(v_SduPt,
10))
Step2: Send a FlexRay frame on the Channel "A" of the control-
ler "0" with the data read in Step1
Step3: return to test system
```

5.1.9.2 TestFrNet_ReceiveWakeup

Function Name	TestFrNet_ReceiveWakeup

Syntax	signature TestFrNet_ReceiveWakeup(in integer FrCtrlIdx, in Fr_ChannelType_ FrChnlIdx, out CTFr_WakeupStatusType_ FrWakeupStatus)
Parameters In	FrCtrlIdx – The Controller on which the FlexRay frame to be transmitted by FrStub FrChnlIdx – FlexRay Channel (A or B or AB)
Parameters Out	FrWakeupStatus WAKEUP_TRANSMITTED – If a wakeup was received by FrStub WAKEUP_NOT_TRANSMITTED – If a wakeup was not received by FrStub
Return Value	-

Description:

The method TestFrNet_ReceiveWakeup is used by the test case to know whether a wakeup was received by FrStub from the SUT

The FrStub continuously checks if a wakeup was received from SUT. And stores the information on which controller and on which channel the wakeup was received.

Upon reception of the call TestFrNet_ReceiveWakeup, the FrStub looks into the stored wakeup information and updates the out parameter accordingly.

Example:

```
TestFrNet_ReceiveWakeup(0, FR_CHANNEL_B, WAKEUP_TRANSMITTED);
```

This means FrStub was received a wakeup from Channel “B” of Controller “0”

5.1.9.3 TestFrNet_GetTransmittedData

Function Name	TestFrNet_GetTransmittedData
Syntax	TestFrNet_GetTransmittedData(in integer FrCtrlIdx, in Fr_ChannelType_ FrChnlIdx, in integer FrLpduIdx, out integer FrSduLength, out octectstring Sdu)
Parameters In	FrCtrlIdx – The Controller on which the FlexRay frame to be transmitted by FrStub FrChnlIdx – FlexRay Channel (A or B or AB) FrLpduIdx – <<>> FrLSduPtr – Pointer points to FlexRay frame payload data that was received by FrStub (this parameter acts as a out parameter in “C” language environment. Here

	the representation is based on the TTCN-3 language)
Parameters Out	FrSduLength – Number of data bytes received in the FlexRay frame Sdu – FlexRay frame payload data that was received by FrStub
Return Value	-

Description:

The method TestFrNet_GetTransmittedData is used by the test case to get the last received FlexRay frame by the FrStub from the SUT on a requested channel and controller

FlexRay stub receives the FlexRay frames from the SUT and stores the received frame information in the local buffers. When the test case asks for the received frames (through TestFrNet_GetTransmittedData) FrStub provides the stored information

The FrStub shall update the out parameters for the latest received FlexRay frame from the controller “FrCtrlIdx” and the channel “FrChnlIdx“:

Sdu – The received FlexRay frame data bytes

FrSduLength – Number of data bytes received

Example:

```
/* The below variables are initialized by FrStub upon recep-
tion of a FlexRay frame */
var octetstring v_Sdu := '11223344556677889AA'O;
var integer FrSduLength := 10;

/* v_Sdu contains the data and FrSduLength contains the length
of the data */
TestFrNet_GetTransmittedData(0, FR_CHANNEL_B, 0, FrSduLength,
v_Sdu);
```

This means FrStub was received a FlexRay frame of 10 bytes with 0x112233445566778899AA, on the controller “0”, and the channel “B”

FrStub:

```
Setp1: Receive the FlexRay frame from the bus and store the
received frame information (data bytes, length, controller,
channel,...)

Step2: If a call to TestFrNet_GetTransmittedData is received
from the test case, then do the following:
```

```
a. Copy the data bytes from the local storage (in Step1) to
the "Sdu".
b. Set FrSduLength equal to the length stored in Step1.
c. Return to test system
```

5.1.9.4 FlexRay Interrupt Handler services

FlexRay driver uses the interrupt handlers (signature d to m in the section 2.7) to test the timer related functionality. The details of the Interrupt handles are given in detail in the Appendix of FlexRay driver conformance test specification.

5.1.10 Also known as

The network Stub implementations are also known as TestCanNet, TestLinNet, TestFrNet, TestCanTrcvNet, TestLinTrcvNet and TestFrTrcvNet.

5.1.11 Related patterns

As usually data to be transmitted over the network is provided to the SUT using pointers, the memory accessor pattern in most cases needs to be used together with the network Stub pattern.

5.1.12 FlexRay Transceiver Network Stub

FlexRay Transceiver Network stub (FrTrcvStub) uses the following interface for the FrTrcvStub (representation in TTCN-3) which allows the test components to send a wakeup pattern on a required channel by FrTrcvStub:

```
signature TestFrTrcvNet_SendWakeUpPattern(in integer
FrTrcvIdx)
  exception (Std_ExceptionType_);
```

Function Name	TestFrTrcvNet_SendWakeUpPattern
Syntax	TestFrTrcvNet_SendWakeUpPattern(in integer FrTrcvIdx)
Parameters In	FrTrcvIdx – FlexRay Transceiver channel on which the wakeup pattern should be sent
Parameters Out	-
Return Value	-

Description:

The method TestFrTrcvNet_SendWakeUpPattern is used by the test case to send a wakeup pattern on a requested FlexRay Transceiver (FrTrcvIdx) by the FrTrcvStub.

After receiving this call FrTrcvStub shall send a wakeup pattern on the requested transceiver. The wakeup pattern is the FlexRay wakeup frame according to the FlexRay protocol specifications.

Example:

```
TestFrTrcvNet_SendWakeUpPattern(1);
```

This will send a wakeup pattern from FrTrcvStub to the Transceiver ”1” on SUT

5.2 Memory accessor

How to read and write memory cells when the interface of the module under test require pointer to memory areas as a parameter or when a module behavior can be observed only by observing changes in memory areas controlled by the module?

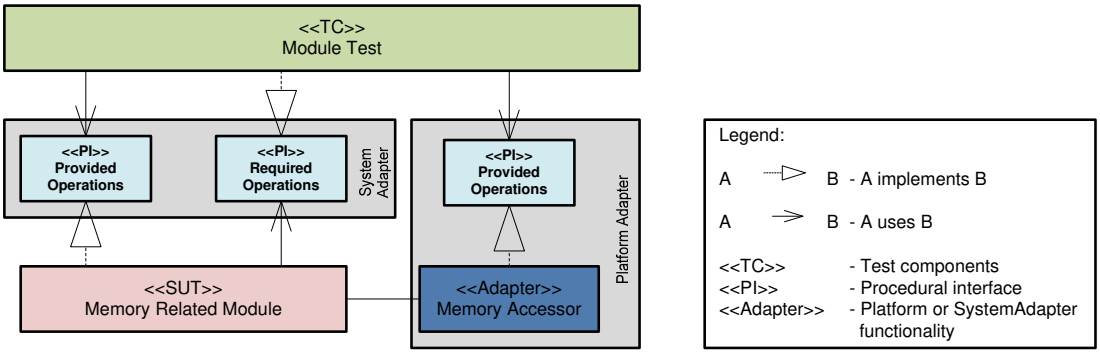


Figure 29: Memory accessor overview

5.2.1 How it works

The memory accessor provides functionality to the test components to allocate, read, modify and compare memory on the target. By calling the signatures of the memory access, represented in **Figure 29** by the right-most box called “Provided Operations”, there is a standardized and defined way of transferring binary data between the test and the target’s memory. How the data is stored may depend on the specific target in use.

5.2.2 When to use it

The pattern should be applied when the module under test has an API (in **Figure 29** represented by the left two light-blue boxed called “Provided Operations” and “Required Operations”) that requires the use of a pointer. Such a pointer needs to point

to a memory location that is valid and that contains the anticipated content. The AUTOSAR API defines whether the allocation of such memory might either a task of the module under test or might be required by the user which during conformance testing would be the test component.

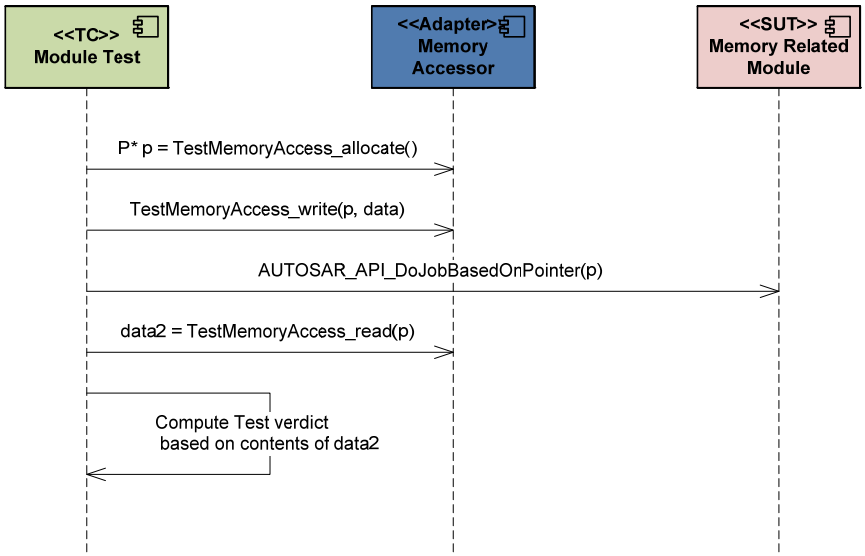


Figure 30: Sample application of the memory accessor

Test components use the memory accessor to allocate valid memory areas, to prepare these memory areas with content that serves the purpose of the test so that it finally is passed to the module under test. The compare and read methods can be used to check whether the module under test has carried out anticipated changes in memory areas.

5.2.3 Implementation notes

The interface that the memory accessor provides to the tests (right-most light-blue box marked “Provided Operations” in **Figure 29** above) needs to be defined in a formal way. The TTCN-3 API is standardized along with the tests themselves. The system adapter needs to implement this interface in a platform specific way and provide the functionality to the test components.

The validation environment will implement this behavior as part of the simulation. Memory itself will be simulated internally to the simulation. In a real test setup, the same functionality needs to be provided that access the memory of the specific target in use. Most likely the memory access will therefore need to be reimplemented for each different target platform.

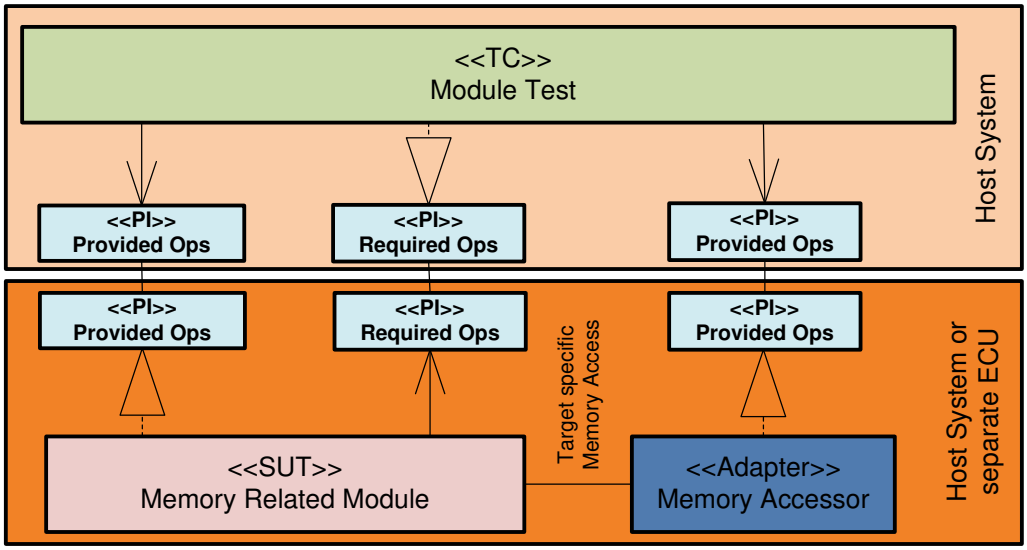


Figure 31: Deployment diagram of the memory accessor in a class-B scenario

5.2.4 Consequences

The memory accessor pattern allows the tests to fulfill the preconditions which are necessary to access AUTOSAR APIs that require the use of pointers. The test can prepare memory locations to which the pointers are pointing and can inspect them. This allows the test components to test memory related behavior.

5.2.5 Example

Since there is only one application of this pattern that serves all modules, the one example provided here will be the only application of this pattern.

The major methods of the memory access are the TestMemoryAccess_Allocate, TestMemoryAccess_Read and TestMemoryAccess_Write methods. Using these APIs the tests can allocate memory on the target. Additionally it can read from memory on the target and can write to memory on the target that might either be allocated by the test component itself or by the module under test. It can be seen that the binary data is mapped to the TTCN-3 binary data type octetstring.

```
signature TestMemoryAccess_Allocate(in integer size)
  return PointerAddr_
  exception (Std_ExceptionType_);

signature TestMemoryAccess_Write(in PointerAddr_ destination,
  in integer size, in octetstring Value)
  exception (Std_ExceptionType_);

signature TestMemoryAccess_Read(in PointerAddr_ source,
```

```
  in integer size) return octetstring
  exception (Std_ExceptionType_);
```

To make the tests more efficient the following two methods allow the test component to copy or compare memory on the target directly. While both functionalities can be carried out internally in the test component, the test of some modules might require the interaction with the module by using large memory sections, for example large network messages in the case of network modules or large chunks of memory in the case of memory drivers. By using the below methods, the test component can copy large amounts of data between two memory locations on the target. Additionally the comparison of two areas in memory can be carried out.

```
signature TestMemoryAccess_Copy ( in PointerAddr_ Destination,
  in PointerAddr_ Source, in integer Size )
  exception (Std_ExceptionType_);

signature TestMemoryAccess_Compare ( in PointerAddr_ Loca-
tion1,
  in PointerAddr_ Location2, in integer Size ) return boolean
  exception (Std_ExceptionType_);
```

To free the previously allocated memory the function TestMemoryAccess_Deallocate is used. This function will free all the previously allocated memory

```
signature TestMemoryAccess_Deallocate()
  exception (Std_ExceptionType_);
```

For illustration, one method of the validation simulation implementation is shown:

```
/* uint8 - unsignedshortint */
/* Let us assume target usable RAM locations: 0x100000 to
0x10FFFF */
#define RAM_START_ADDRESS       (volatile uint8 *)0x100000
#define RAM_END_ADDRESS       (volatile uint8 *)0x10FFFF
#define NULL_PTR       (volatile uint8 *)0x000000

uint8 *Ptr; /* Global variable */

/* During start up */
Ptr = RAM_START_ADDRESS;

uint8 * TestMemoryAccess_Allocate(int numberOfBytes)
{
  Ptr + = numberOfBytes;
  if(Ptr > RAM_END_ADDRESS)
  {
    return NULL_PTR;
  }
  else
  {
    return Ptr;
  }
}
```

```
/* Free the previously allocated memory */
void TestMemoryAccess_Deallocate()
{
  Ptr = RAM_START_ADDRESS;
}
```

5.2.6 Specification of methods

5.2.6.1 Allocate target memory

Function Name	TestMemoryAccess_Allocate
Syntax	TestMemoryAccess_Allocate (in integer size) returns uint8*
Parameters In	size – Target memory block size in bytes
Parameters Out	-
Return Value	PointerAddr_ - Handle to allocated target memory block If the allocation fails, NULL is returned.

Description:

The function TestMemoryAccess_Allocate allocates a chunk of memory on the target. The return value is a handle to be used with APIs.
No assumptions should be made about target addresses.

Example:

```
/* allocate chunk of 10 bytes */
PointerAddr_ hMem1 = TestMemoryAccess_Allocate(10);
if( hMem1 != NULL) { /* use other TestMemoryAccess APIs */ }
```

5.2.6.2 Allocate target memory at a specific location

Function Name	TestMemoryAccess_AllocateSpecfic
Syntax	TestMemoryAccess_AllocateSpecfic(in PointerAddr_ location, in integer size)
Parameters In	size – Target memory block size in bytes location – The exact location of the memory where pointer has to be created

Parameters Out	-
Return Value	-

Description:

The function TestMemoryAccess_AllocateSpecfic allocates a chunk of memory on the target at a specified memory location.

Example:

```
/* Allocate chunk of 10 bytes with start address 1000 */
PointerAddr_ hMem1 = TestMemoryAccess_AllocateSpecfic(1000,
10);
```

This will allocate a memory location with start address “1000” and size “10” bytes (i.e. memory location from 1000 to 1010)

5.2.6.3 Compare target memory

Function Name	TestMemoryAccess_Compare
Syntax	TestMemoryAccess_Compare (in PointerAddr_ location1, in PointerAddr_ location2, in integer size) returns boolean
Parameters In	PointerAddr_ location1 – Handle to memory location 1 PointerAddr_ location2 – Handle to memory location 2 size – Size of chunk to be compared in bytes
Parameters Out	-
Return Value	boolean – True if memory blocks match, False if at least one byte mismatch.

Description:

The function TestMemoryAccess_Compare compares two target memory chunks up to given size. The user of the API must ensure that compare size does not exceed bounds of allocated memory chunks, represented by location 1 and 2.
The return value is true if the memory chunks match for at least compare size bytes.

Example:

```
/* allocate chunk 1 with 10 bytes */
PointerAddr_ hMem1 = TestMemoryAccess_Allocate(10);
/* fill chunk 1 with pattern */

/* allocate chunk 2 with 10 bytes */
PointerAddr_ hMem2 = TestMemoryAccess_Allocate(10);
```

```
/* fill chunk 2 with pattern */

/* compare 10 bytes of chunks */
if( TestMemoryAccess_Compare(hMem1, hMem2, 10)) { /* match */ }
```

5.2.6.4 Copy target memory

Function Name	TestMemoryAccess_Copy
Syntax	TestMemoryAccess_Copy (in PointerAddr_ destination, in PointerAddr_ source, in integer size) returns void
Parameters In	PointerAddr_ destination – Handle to destination memory location PointerAddr_ source - Handle to source memory location size – Size of chunk to be copied in bytes
Parameters Out	-
Return Value	-

Description:

The function TestMemoryAccess_Copy copies a target memory chunk to another target memory chunk up to given size. The user must ensure that size does not exceed bounds of allocated memory chunks, represented by source and destination.

Example:

```
/* allocate source chunk with 10 bytes */
PointerAddr_ hSrc = TestMemoryAccess_Allocate(10);
/* fill source chunk with pattern */

/* allocate destination chunk with 10 bytes */
PointerAddr_ hDst = TestMemoryAccess_Allocate(10);

/* copy 10 bytes from hSrc to hDst */
TestMemoryAccess_Copy(hDst, hSrc, 10);
```

5.2.6.5 Read target memory

Function Name	TestMemoryAccess_Read
Syntax	TestMemoryAccess_Read (in PointerAddr_ source, in integer size) returns octetstring

Parameters In	PointerAddr_ source – Handle to source memory location size – Size of chunk to be read in bytes
Parameters Out	-
Return Value	octetstring - Pointer to memory block which contains the bytes. If the read fails due to invalid parameters, NULL is returned.

Description:

The function TestMemoryAccess_Read copies the contents of a target memory chunk to local memory buffer. The user must ensure that size does not exceed bounds of allocated memory chunk, represented by source.
It's on behalf of the implementer, how the returned local memory/representation is to be managed.

Example:

```
/* allocate source chunk with 10 bytes */
PointerAddr_* hSrc = TestMemoryAccess_Allocate(10);
/* fill source chunk with pattern */

/* read 10 bytes */
PointerAddr_ ptr = TestMemoryAccess_Read(hSrc, 10);
if( ptr != NULL) { /* access the bytes */ }
```

5.2.6.6 Write target memory

Function Name	TestMemoryAccess_Write
Syntax	TestMemoryAccess_Write (in PointerAddr_ destination, in uint32 size, in octetstring byteblock) returns void
Parameters In	PointerAddr_ source – Handle to destination memory location size – Size of chunk to be write in bytes octetstring byteblock – pointer to byte block to read from
Parameters Out	-
Return Value	-

Description:

The function TestMemoryAccess_Write copies the contents of a local memory buffer into a target memory chunk. The user must ensure that size does not exceed bounds of allocated memory chunk, represented by destination and byteblock.
It's on behalf of the implementer, how the returned local memory/representation is to be managed.

Example:

```
/* allocate destination chunk with 10 bytes */
PointerAddr_ hDst = TestMemoryAccess_Allocate(10);

/* write 10 bytes pattern into destination block */
TestMemoryAccess_Write(hDst, 10, pattern);
```

5.2.6.7 De-allocate (free) target memory

Function Name	TestMemoryAccess_Deallocate
Syntax	TestMemoryAccess_Deallocate ()
Parameters In	-
Parameters Out	-
Return Value	-

Description:

The function TestMemoryAccess_Deallocate frees all the memory that was allocated previously using either the function TestMemoryAccess_Allocate (AND / OR) Test-MemoryAccess_AllocateSpecfic

Example:

```
/* allocate destination chunk with 10 bytes */
PointerAddr_ hDst = TestMemoryAccess_Allocate(10);

/* write 10 bytes pattern into destination block */
TestMemoryAccess_Write(hDst, 10, pattern);

/* allocate chunk of 10 bytes with start address 1000 */
PointerAddr_ hMem1 = TestMemoryAccess_AllocateSpecfic = (1000,
10);

/* De-allocate all the memory that was allocated earlier */
TestMemoryAccess_Deallocate();
```

5.2.7 Also known as

TestMemoryAccess

5.2.8 Related patterns

The application of the network Stub pattern in most cases requires an application of this pattern also as the data of message to be transmitted over a network is usually supplied to the SUT using pointers.

5.3 Average Neighbor

How to provide interfaces to a module that expects neighboring modules to be present in its environment?

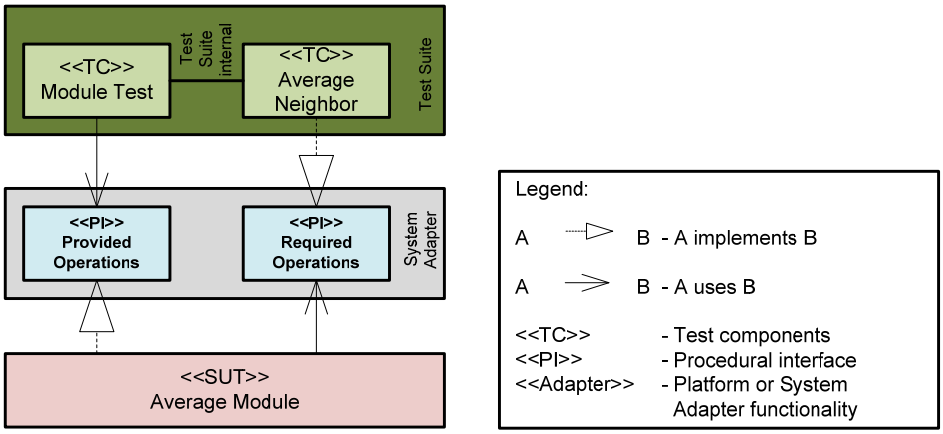


Figure 32: Overview Average Neighbor

5.3.1 How it works

A module under test assumes neighboring modules to be present. Within its functionality the module under test calls functions of these neighboring modules throughout the course of the tests. Instead of the actual modules being present on the target, the system adapter provides the functions of the neighboring modules, suspends the execution of the module under test and relays the calls to the TTCN-3 test system where test components handle the calls and respond. The responses are again relayed to the system adapter which resumes the execution of the module under test and provides the results of the call to the same.

5.3.2 When to use it

This pattern should be used whenever a neighboring module needs to be emulated towards the module under test unless an application of one of the above patterns is strictly required.

5.3.3 Implementation Notes

The signatures of the operations of all the AUTOSAR modules are specified in the SWS documents. Along with these the signatures are also formally modeled in a so called UML model. Due to the large number of AUTOSAR signatures the creation of the above mentioned system adapter that allows the relay of function calls to TTCN-3 can be created using model based techniques.

The mapping of signatures, the notation and types in the UML model to the TTCN-3 language is done as part of the creation of the CTSpecs. The TTCN-3 signatures are thus standardized along with the tests themselves.

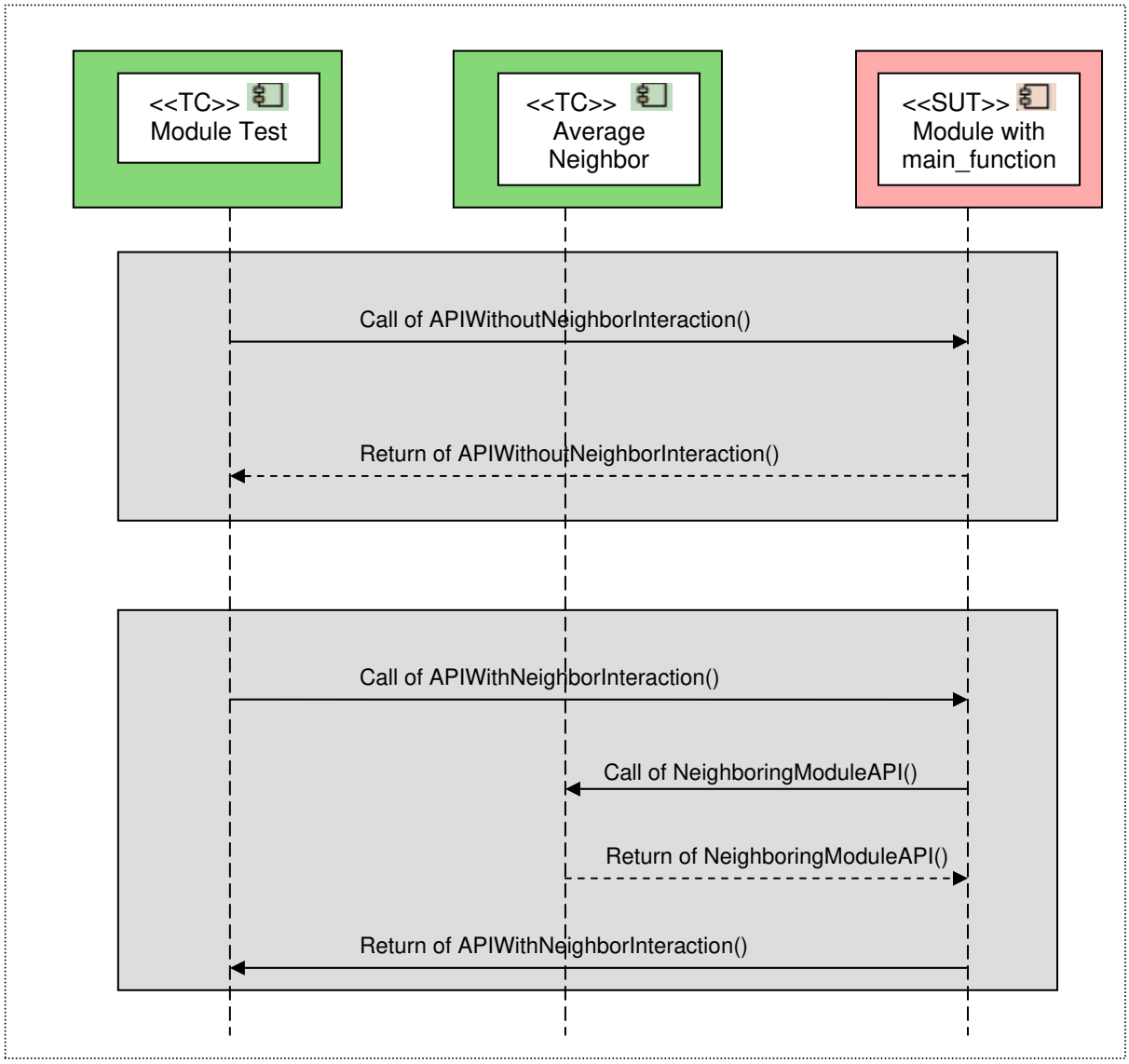


Figure 33: Comparison of API call with and without neighbor interaction

5.3.4 Consequences

The application of this pattern leads to test suites that do not only contain the tests themselves but also provide all necessary functionality that is required by the module under test. This leads to more convenient test suites for the CTAs as there is no need for implementing any additional stubs.

5.3.5 Examples

These patterns are used for every module that has neighboring modules for which none of the other patterns need to be applied. For the module CAN the following example can be given: The All_Modules_Api.ttcn contains the generated AUTOSAR

APIs in the TTCN-3 notation. This API would be called by any higher layer module that might require to call the module CAN, e.g. CanIf:

```
signature Can_CheckWakeup ( in integer Controller )
  return Std_ReturnType_ exception (Std_ExceptionType_);

signature Can_EnableControllerInterrupts ( in integer Control-
ler )
  exception (Std_ExceptionType_);

signature Can_Write ( in integer Hth, in Can_PduType_ PduInfo
)
  return Can_ReturnType_ exception (Std_ExceptionType_);
```

5.3.6 Also known as

There is a large number of applications of this pattern. Hence there is no individual name for each of its applications.

5.4 Important Notes

5.4.1 Specification for configuration parameter field “Type”

The AUTOSAR SWS documents use the representation “Reference to [...]” for the “Type” field in the configuration section (usually Section 10 of AUTOSAR SWS).

e.g. CANIF636_Conf
Type: Reference to [CanController]

Hence the conformance test specification uses the same representation for the configuration parameter types, what is used in AUTOSAR SWS documents.

In the conformance test specifications, if a parameter “Type” is specified as “Reference to [...]", then the exact type of the parameter could be EcucSymbolicNameReferenceDef (OR) EcucReferenceDef (OR) EcucChoiceReferenceDef (OR) any other reference type which is specified in specification of ECU configuration.

5.4.2 Order and values of enumeration literals

Background:

In general most of the cases AUTOSAR SWS documents do not specify the values for the liters of enumeration type definitions. Hence the implementer is free to choose values for enumeration literals.
e.g. Icu_ModeType has the definition {ICU_MODE_NORMAL, ICU_MODE_SLEEP}

Possible implementations for Icu_ModeType:

(a)

```
typedef enum
{
  ICU_MODE_NORMAL = 0,
  ICU_MODE_SLEEP
} Icu_ModeType;
```

(b)

```
typedef enum
{
  ICU_MODE_SLEEP = 0,
  ICU_MODE_NORMAL
} Icu_ModeType;
```

Both implementations ((a) and (b)) are valid implementations, since the SWS document do not specify the order and values of enum literals. In real scenario both implementations ((a) and (b)) will work without any problems (since the module who is defining Icu_ModeType (i.e. ICU Driver) and the module who is using the Icu_ModeType both will use the same header file for the type Icu_ModeType).

However in the conformance tests, the test scripts will not use the “C” header file defined by ICU Driver. Since the test scripts are written TTCN-3 language and resides external to the target where ICU Driver is residing.

Hence there is mapping required between the enumeration definitions (i.e. order of the enumeration literals and their values) in the BSW module implementation and the enumeration definitions in the conformance test scripts.

5.4.2.1 Point to be considered while implementing adapters

The conformance test scripts define the enumeration literals in the same order that appears in the AUTOSAR SWS document. The first literal in the enumeration type is assigned with value “0” and the subsequent literal values are incremented by “1”.
e.g.
The type definition in the AUTOSAR SWS

Name:	Lin_FrameResponseType	
Type:	Enumeration	
Range:	LIN_MASTER_RESPONSE	Response is generated from this (master) node
	LIN_SLAVE_RESPONSE	Response is generated from a remote slave node
	LIN_SLAVE_TO_SLAVE	Response is generated from one slave to another slave, for the master the response will be anonymous, it does not have to receive the response.

Description:	This type is used to specify whether the frame processor is required to transmit the response part of the LIN frame.

The definition in the conformance test scripts in TTCN-3:

```
type enumerated Lin_FrameResponseType_
{
   LIN_MASTER_RESPONSE (0),
   LIN_SLAVE_RESPONSE (1),
   LIN_SLAVE_TO_SLAVE (2)
}
```

Hint: Each module <Msn> TTCN-3 type definitions will be available in the TTCN-3 scripts file <Msn>_api_types.ttcn (e.g. Lin_api_types.ttcn)

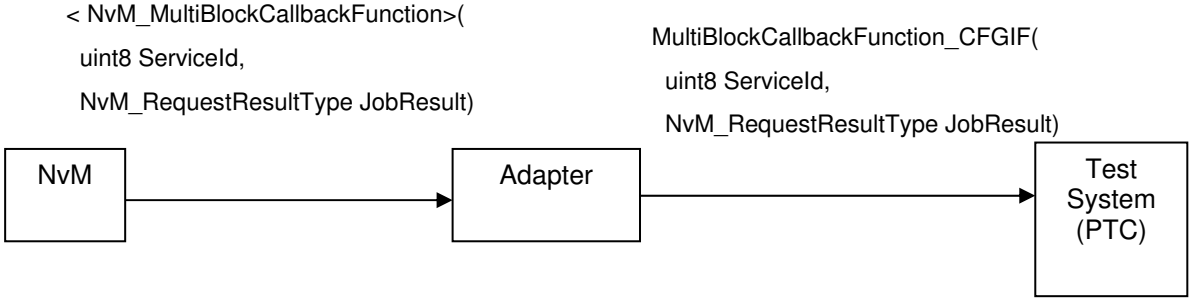
If a BSW implementer defined the enumeration literals of Lin_FrameResponseType with different values (compared to that of above defined TTCN-3 implementation values), then the adapter shall handle the mapping of enumeration literal values from BSW (SUT) to the TTCN-3 scripts and vice versa.

5.4.3 Handling configurable interfaces

AUTOSAR specifications uses configurable interface concept. That is the operation name (either a part of the operation name or the complete operation name) can be configurable using a configuration item. Implementation of configurable interface would be possible in real BSW module implementation in “C” language (e.g. using function pointers). However this is not possible to implement in the conformance test scripts which are developed in TTCN-3 language. Hence the following method is used to test the configurable interfaces.

SUT invokes the configurable interface with the name configured by the corresponding configuration parameter. Test case always uses a predefined fixed interface name for the configurable interfaces. The adapter translates the interface used in test case to the interface used in SUT.

e.g.



The details of the configurable interfaces handling of a BSW module is described in the Section 3 of the respective conformance test specification document.

5.4.4 Handling of DET stub

Each BSW module might report the specified errors to DET during development mode. The conformance tests are developed for the production mode BSW implementation. Hence the development errors are not expected by any test case in the conformance tests. That means if a SUT reports an error to DET then it is considered as a failure in the conformance tests. The main test component used for testing the SUT will not read the information from DET stub to set the test verdict. The DET stub used for the conformance tests will automatically set the test verdict as failure if the DET stub receives an error.

5.4.5 Pointer handling

Conformance test scripts are developed using the TTCN-3 language. In TTCN-3 language there is no concept of pointers. However the AUTOSAR specifications (SWS) use the concept of pointers. Hence a special mechanism is required to handle the pointers in conformance tests.
The following rules have been used while handling pointers in TTCN-3 files.

- If an AUTOSAR API parameter (of pointer type) is used to transfer the number of data bytes, then the parameter type shall be represented as type integer in TTCN-3 signatures. The value of the integer parameter shall be the address location where data is placed (by test case) or where data shall be placed (by SUT).

Example:

```
AUTOSAR Signature:
void IpduM_TriggerTransmit(PduIdType PdumTxPduId, uint8* SduPtr);

TTCN-3 Signature:
signature IpduM_TriggerTransmit(in PduIdType_ PdumTxPduId,
  in PointerAddr_ SduPtr) exception(Std_ExceptionType_);
```

In the above example, the SUT writes the data bytes to the location pointed by "SduPtr". In TTCN-3 signature "uint8*" of AUTOSAR type is replaced with "PointerAddr_" (which is of type integer (type integer PointerAddr_)). Hence the TTCN-3 test system shall create a memory buffer location and provide the absolute address of the memory location for the parameter "SduPtr".
E.g. A memory buffer with starting address 1000 is created in test case. Provide the value 1000 for the parameter "SduPtr". SUT is assumed to provide the data starting from the memory location 1000.

- If an AUTOSAR API parameter (of pointer type) is used as out parameter of a simple or complex type, then the parameter type shall be represented same as that of AUTOSAR parameter type (without pointer '*') in TTCN-3 signa-

tures. The value of the parameter shall contain the actual content of the simple or complex type parameter (instead of pointer).

Example:

```
AUTOSAR Signature:
void IpduM_GetVersionInfo(Std_VersionInfoType* versioninfo);

TTCN-3 Signature:
signature IpduM_GetVersionInfo(out Std_VersionInfoType_ versioninfo)
  exception (Std_ExceptionType_);
```

In the above example, SUT will provide the pointer reference to version information structure (record) through the parameter "versioninfo". However the TTCN-3 sees the version information as "out" parameter containing the actual version information (updated by SUT) instead of pointer address. The assumption here is system adapter will receive the pointer from SUT and provide the content of pointer to TTCN-3 test case.

- In module initialization API, AUTOSAR modules use a configuration pointer, where the pointer represents one out of multiple configurations which is loaded by the module under test. Location of such pointer will not be really known to TTCN-3 test case. Basically the configuration pointer points to one instance of multiple configuration container of the AUTOSAR module. In these cases TTCN-3 test case shall pass the index (of type integer) of the multiple configuration containers as parameter to module initialization operations. The assumption here is system adapter will convert the index (given TTCN-3 test case) to appropriate pointer (to SUT).

Example:

```
AUTOSAR Signature:
void IpduM_Init(IpduM_ConfigType* config);

TTCN-3 Signature:
signature IpduM_Init(in integer config) exception (Std_ExceptionType_);
```

The integer parameter in the TTCN-3 signature represents the index of multiple

5.4.6 Handling array types

AUTOSAR specification uses array type to represent an unsigned integer byte stream. TTCN-3 language which is used to develop the conformance test scripts does not have the equivalent feature. Hence a special mechanism is used to achieve this feature.
Hence in TTCN-3 language “record of integer” shall be used to represent the array type (which holds the integer values).

e.g. `type record of integer Com_IpduGroupVector_;`

5.4.7 External stubs used in conformance test specification

In the conformance tests some special cases it is not feasible to use TTCN-3 stubs (e.g. Hardware dependency, function pointer handling, etc.). In such cases the conformance tests uses external stubs, these external stubs have to implement by CTA in actual conformance testing (in Class B tests). This section lists the external stubs used by conformance tests.(Note that this list excludes the external stubs which are defined in the Section 2 and Section 3 of this document)
The following BSW module uses the external stubs:

- AUTOSAR COM
- DIO Driver
- AUTOSAR OS
- RTE

The details of the external stubs are documented in their respective conformance test specification document.

5.4.8 Order of test steps for synchronous APIs

If an operation call to SUT leads a synchronous call to the neighboring module operation within the SUT operation call then the following is apply:

Once the SUT operation is invoked from the test case, the control goes to SUT and SUT starts executing the requested operation. During execution of the SUT operation, SUT invokes the neighboring module operation which will be captured and call information is stored by the stub. Since the test case does not know how much time it would take to SUT to complete the requested operation, the test case simply waits for the reply (return) from the SUT Once the SUT is returned from the requested operation then the test case reads the information (related to the SUT call of neighboring module operation) from the stub and sets the test verdict accordingly. Due to this process, the test steps would look little different from the actual module execution

e.g. SUT = FrIf, neighboring module = FrTrcv
If test system invokes FrIf_EnableTransceiverBranch then FrIf will invoke the FrTrcv operation FrTrcv_EnableTransceiverBranch and then FrIf returns to test system. This will be tested with the below sequence of test steps:

01:	Invoke FrIf_EnableTransceiverBranch(FrIfCtrlIdx, FrIfClusterChannel, CTFrIf-BranchIdx)
02:	Verification Point : Condition: FrIf_EnableTransceiverBranch returns E_OK if true: proceed else: stop
03:	Verification Point: Condition: FrTrcv_EnableTransceiverBranch(FrTrcv_TrcvIdx == CTFrIfFrTrcvChannelRefIdx, FrTrcv_BranchIdx == CTFrIfFrTrcvBranchIdx) is invoked

if true: ...
else: stop

That is SUT operation is invoked then SUT operation return is verified and then it is verified if whether the neighboring module operation was invoked by SUT.