| Document Title | AUTOSAR BSW & RTE Conformance Test Specification, Part 3: Creation & Validation |
|---|---|
| **Document Owner** | AUTOSAR |
| **Document Responsibility** | AUTOSAR |
| **Document Identification No** | 283 |
| **Document Classification** | Auxiliary |

| | |
|---|---|
| **Document Version** | 1.0.2 |
| **Document Status** | Final |
| **Part of Release** | 3.2 |
| **Revision** | 1 |

**Document Change History**

| Date | Version | Changed by | Change Description |
|---|---|---|---|
| 27.04.2011 | 1.0.2 | AUTOSAR Administration | 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 Specification Documents 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 Specification Documents for illustration purposes only, and they themselves are not part of the AUTOSAR Standard. Neither their presence in such Specification Documents, 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 Document Overview

This document describes the methodology to create and execute AUTOSAR BSW Conformance Test Specifications (CTSpecs). It describes how to realize the CTSpec creation process and details potential tool support.
This document mainly specifies how to apply the TTCN-3 test methodology to testing the conformance of AUTOSAR BSW module implementations. This includes defining the architecture of the CTSpecs and specifying the relevant execution environment components.
The document is therefore structured into the following parts:

1. Introduction (this chapter)
2. Realization of the CTSpec creation process (Chapters 2 to 8)
3. Further details on the methodology (Chapters 9 to12)

## 1.1 Focus and Scope

This document provides a detailed foundation for creating CTSpecs. It focuses more on the details of the work activities and less on the overall creation process. An overview of the CTSpec creation process and the description of the roles involved and their interactions is given in [3].

## 1.2 How to use the Document

This document presupposes a basic understanding of the CTSpec creation process, the roles involved and the associated terminology. These are all described in [3].
[4] is a supplement to this document that discusses issues related to executing CTSpecs after they have been created.
[5] defines a complete template for a BSW module conformance test.

## 1.3 Abbreviations

| Abbreviation | Description |
|---|---|
| API | Application Program Interface |
| BSW | Basic Software |
| 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 |
| ECU | Electronic Control Unit |
| ICC | Implementation Cluster Conformance Class |

| Abbreviation | Description |
|---|---|
| ICS | Implementation Conformance Statement |
| IP | Intellectual Property |
| PA | Platform Adapter (TTCN-3 – see Part 4) |
| PS | Product Supplier |
| RM | Requirements Management |
| RTE | Run Time Environment |
| SA | System Adapter (TTCN-3 – see Part 4) |
| 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 |

## 1.4 Bibliography

[1] TTCN-3 specifications
http://www.ttcn-3.org/Specifications.htm

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

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

[4] AUTOSAR BSW & RTE Conformance Test Specification Part 4: Execution Constraints
AUTOSAR_CTSpec_Execution_Constraint.pdf

[5] Template for AUTOSAR Conformance Test Specification Documents
AUTOSAR_CTSpec_Template.pdf

[6] Specification of Platform Types
AUTOSAR_SWS_PlatformTypes.pdf

[7] Specification of Development Error Tracer
AUTOSAR_SWS_DET.pdf

[8] AUTOSAR ECU Configuration Parameters

AUTOSAR_EcucParamDef.xml

[9] Specification of ECU Configuration
AUTOSAR_ECU_Configuration.pdf

[10] AUTOSAR Specification of C Implementation Rules
AUTOSAR_SWS_C_ImplementationRules.pdf

[11] Colin Willcock, Thomas Deiß, Stephan Tobies, Stephan Schulz, Stefan Keil and Frederico Endler, “An Introduction to TTCN-3”, John Wiley and Sons Ltd., May 2005, ISBN: 0470012232

# 2 Realization of the CTSpec Creation Process

Roles perform the activities in the CTSpec creation process. Different divisions of work can thus be realized when different persons and/or organizations adopt these roles. Furthermore tools are used to automate the generation of some work artifacts. [3] gives an overview of the CTSpec creation process. It focuses on defining the activities and the know-how requirements for the corresponding roles:

- Test Designer
- Test Implementer
- Test Validation Implementer
- Test Assessor

This document specifies how the process should be put into practice. It describes the process activities in more detail to enable their realization. From an activities point of view, four work phases can be identified in the CTSpec creation process:

| | |
|---|---|
| **Analysis Phase** | Analysis and refinement of the SWS document |
| **Design Phase** | Test case identification and design of the CTSpec |
| **Implementation Phase** | Implementation of the test cases and the module simulation |
| **Validation Phase** | Validation of test cases |

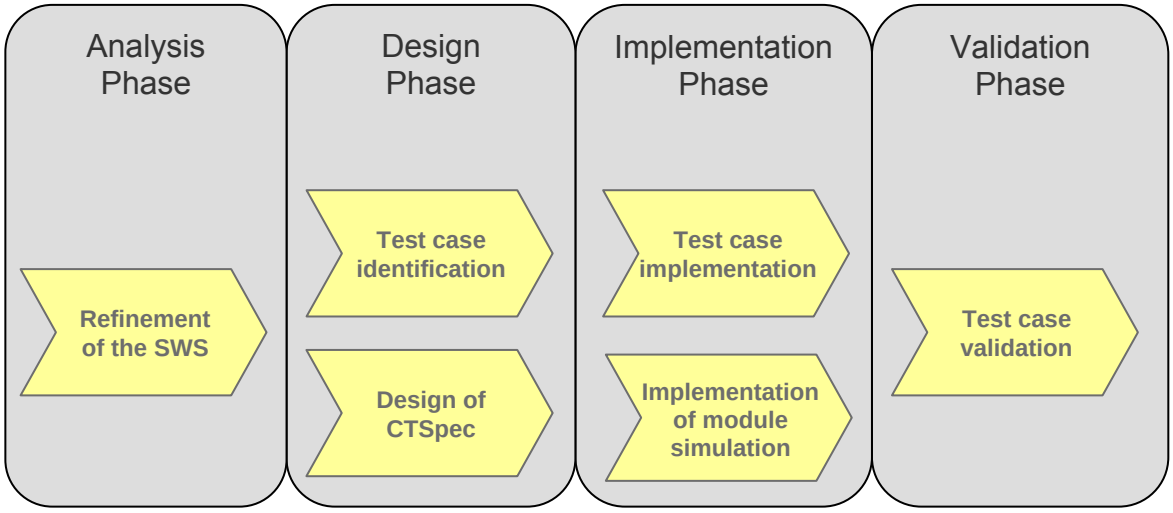


**Figure 1 - Phases and activities of the CTSpec creation process**

These phases are discussed in the following chapters.

# 3 SWS Analysis Phase: Refinement & Categorization

In the Analysis Phase, the Test Designer refines the SWS to be conformance tested. This refinement mainly consists of requirements engineering activities on the SWS document. A requirements management (RM) tool shall be used in order to automate change and version management, traceability among and between requirements and test cases and for working concurrently on and exchanging work results.
The objective of the analysis activities, as described in the following sections, is to produce a refined SWS document that is ready for test case identification.

## 3.1 Collect Test Input Baseline

First, all relevant documents and their right versions have to be identified. This includes at least the following documents:

- the SWS of the module itself,
- the SWSs of all “neighboring” modules that contain requirements that must be satisfied by the module,
- the deviation sheet for the module,
- the “ECU Configuration Parameters” [7] document

The baseline must be documented with the names of the documents, their versions and dates.
For example, the baseline for the NvM in the CTSpec Creation Pilot was:

| | | |
|---|---|---|
| Specification of NVRAMManager | 2.0.0 | 28.04.2006 |
| AUTOSAR BSW Deviations Validator 2 | 107 | 02.08.2006 |
| Specification of ECU Configuration Parameters | 0.08 | 11.06.2006 |
| Specification of Module Memory Abstraction Interface | 1.0.0 | 23.03.2006 |
| Specification of Module EEPROM Abstraction | 1.0.0 | 23.03.2006 |
| Specification of Flash EEPROM Emulation | 1.0.0 | 23.03.2006 |
| Specification of CRC Routines | 2.0.0 | 28.04.2006 |

## 3.2 Preparation for Analysis

In this step, all specification items in the baseline documents must be linked or copied into the RM tool[1]. At least the following attributes must be assigned to the module requirements:

[1] The exact procedure depends on the RM tool being used. For example, DOORS requires copying the requirements into the DOORS repository.

| | |
|---|---|
| **ID given by the RM tool** | For referring to the refined SWS requirement. |
| **Original requirements ID from the SWS** | For referring to the associated original SWS requirement. |
| **Related configuration parameter** | The configuration parameter related to the refined SWS requirement (if applicable). This attribute is later used in configuration analysis and for creating configuration sets. |
| **Category** | The category assigned to the refined SWS requirement. Test cases are later identified based on these categories (as defined in Chapter 3.4). |
| **References to test cases that cover the requirement** | The ID(s) of the test case(s) that test the requirement. This attribute realizes the traceability between refined SWS requirements and test cases. |
| **Analysis comment** | Additional information or comments that may arise. |

While the values of the first and second attributes can be defined during preparation, the values of the remaining attributes are defined in subsequent analysis activities.
The following general guidelines must be followed during this preparation activity:

- Keep the overall order of the SWS requirements in the RM tool the same as the order of the items in the SWS specification.
- All labeled SWS items must appear.
- Ensure that there is an entry for each configuration parameter in the module (typically from chapter 10.2). Document the name of the configuration parameter and add its description. Enter the SWS-Req-ID when it is available.
- Ensure that there is an entry for the signature of each function provided or required by the module.
- Ensure that the general properties of each API-call as defined in the SWS (e.g. reentrancy, synchronous/asynchronous) are entered in the table.

## 3.3 Analyze Specification Items

The actual analysis must be performed by personnel familiar with the BSW module's design and functionality.
The following guidelines must be followed during the analysis work:

- If a specification item is not atomic[2] (i.e. it contains several atomic requirements), divide it into its component atomic items and use extensions (e.g. "a", "b", …) in the reference to the original SWS ID[3].
- Ensure that a SWS item's contents can be interpreted independently of its context. If necessary, add context information to the item.

[2] In general, an "atomic requirement" refers to a single functionality of the BSW module.
[3] For example:

| | |
|---|---|
| NVM091a | NvM_Init shall be invoked by the ECU state manager exclusively. |
| NVM091b | The NvM_Init request shall not contain the initialization of the configured NVRAM blocks. |

- Prepend the API's name to all items (original SWS and newly created) pertaining to a specific API.
- Ensure that there are sufficient entries to describe the pre- and post-conditions of each function provided or required by the module. If necessary, add entries even if they do not have an ID in the SWS.
- Add any other information from the SWS document that is relevant to conformance testing, even if it does not have an explicit ID.

All information from the "neighboring" SWSs that is also relevant to conformance testing the module must also be included following similar guidelines. Typically, the entries from the neighboring SWSs can be limited to:

- Pre-conditions on functions used by the module under consideration.
- Post-conditions on functions provided by the module under consideration.

The "ECU Configuration Parameters" document contains a formal model of all the module's configuration parameters. As the configuration files, which are an important input for the conformance tests, are based on this model, the model has priority over the information in the "Configuration and Configuration Parameters" chapter of the SWS document.

During the activities described so far, it must be continuously ensured that the requirements are complete, understandable, logically correct and free from contradictions. If possible, the misunderstandings, errors, inconsistencies and missing information that emerge in the original SWS document must be raised (through "Bugzilla") as bug reports for the owner of the original SWS document.

The specification items must be modified based on the resolution of the bug reports,:

- When an item is changed, make a note under "analysis comment".
- When an item becomes obsolete, mark the requirements text appropriately (e.g. by crossing it out) and make a note under "analysis comment". Do not simply delete the item as it will then not be known whether this item has been overlooked or deleted.
- When additional configuration items are added, make a note under "analysis comment".

Finally, add relevant deviation sheet entries according to the following guidelines:

- When the deviation item changes an SWS item, integrate the deviation item in the existing SWS item (i.e. add the deviation number and change the description so that it reflects the contents of the deviation sheet).
- When the deviation is an addition or a more complex modification, add an additional line to the SWS analysis sheet at a logical place (next to the affected SWS-item or configuration parameter).

The activities up to now have produced a well structured and ordered SWS document that contains all specification information related to the BSW module. Incorporating an RM tool has prepared the refined SWS document well for use in the remaining phases of the CTSpec creation process.

## 3.4 Categorize Specification Items

In general, the refined SWS document contains many specification items that are in fact not relevant for conformance testing. Each specification item must therefore be categorized to establish *whether* and *how* it is relevant to conformance testing. The analysts must understand the schema well and apply it accurately.

The following information can be derived directly from a specification item's category:

- Relevant for conformance testing?
  - Yes / No / Pending on bug
- How to test
  - TTCN-3 test case / Manual inspection / …

The following table describes the specification item categories that AUTOSAR has identified so far, their relevance to conformance testing and the appropriate test approach:

| Category | Description (SI = specification item) | Relevance to CT | How to test |
|---|---|---|---|
| No Requirement | The SI is no requirement at all. It has been included for informational purposes | no | (irrelevant) |
| Redundant | The SI is already covered by one or several other SIs. | no | (irrelevant) |
| Informal Requirement | The SI contains an informal description on the module. This description usually contains non-functional requirements. | yes | Manual inspection |
| Definition of Configuration Parameter | The SI is a definition of a configuration parameter. | yes | Consider in configuration sets |
| Implementation of Configuration Parameter | The SI defines how the configuration parameter is to be realized (e.g. by #define). | yes | Manual inspection |
| Requirement on Configuration | The SI contains either general definitions on the configuration values or direct constraints on permitted configuration values. | yes | Consider in configuration sets |
| Detection of Wrong Configurations | The SI contains demands on how to detect wrong configurations or which configurations are wrong | no | (irrelevant) |
| Development Error | The SI defines behavior directly related to the Development Error Tracer (DET) | no[4] | TTCN-3 test case |
| Header Files for Internal Use | The SI contains definitions of header files that are used by the module only | no | (irrelevant) |
| Inside Source Code | The SI contains structural definitions of the source code that are not related to functionality. | no | (irrelevant) |
| Inside Header File | The SI contains structural definitions of the header file not related to functionality. | no | (irrelevant) |

[4] Development Error tracing is not relevant to conformance testing, but the SIs must be tested in the test suites. See section 3.4.8 for details.

| Category | Description (SI = specification item) | Relevance to CT | How to test |
|---|---|---|---|
| Provided Header Files for External Use | The SI contains definitions of header files that are to be included by other modules. | yes | Check header file |
| Provided Signature | The SI is a definition of a function that is provided to other modules as API | yes | Compile-build process |
| Required Signature | The SI contains demands on functions provided externally. | yes | Compile-build process |
| Requirement on Module Behavior | The SI defines module behavior and functionality. | yes | TTCN-3 test case |
| Requirement on Re-entrance of Module | The SI defines behavior and functionality of the module related to reentrancy. | no | (irrelevant) |
| Requirement on Execution in Interrupt Context | The SI defines behavior and functionality of the module related to execution in interrupt context. | no | (irrelevant) |
| Requirement on Other Module | The SI is not a requirement on the “module under specification” but a requirement on another module. | no | (irrelevant) |
| Direct Hardware Access | The SI defines behavior and functionality that involves direct access to hardware devices. | no | (irrelevant) |
| Vendor-Specific Extensions | The SI is a definition on possible extensions to be done by the vendor. | no | (irrelevant) |
| Pending on Bug | The SI has been identified as unclear and clarification has been requested. After clarification, the proper category will be defined for the SI. | undefined | undefined |

**Table 1 - Overview of the categories for specification items**

The following sections explain selected specification item categories in more detail.

### 3.4.1 No Requirement

Some items are not requirements because they are too general and/or too weak.
Example: The following is formulated too weakly (“reasonably short”):

| | |
|---|---|
| NVM156 | Depending on implementation, callback routines provided and/or invoked by this module may be called on interrupt level. The module providing those routines therefore has to make sure that their runtime is reasonably short. |

### 3.4.2 Redundant

Some items are relevant to conformance testing, but are already covered by other items. The “internal analysis comment” must contain precise information about which items cover the redundant item.
Example: The following is redundant with respect to other SWS items:

| | |
|---|---|
| NVM007a | For each asynchronous request a notification of the caller after completion of the job shall be a configurable option. |

### 3.4.3 Informal Requirement

The specification item is either a general definition of terms or conditions (e.g. operating conditions) or it describes basic functionality on a very high level. In both cases, the item cannot be directly related to functional behavior that is testable with test cases because it is not specific enough and too “informal”.
The specification item is thus not directly relevant to conformance but must usually be considered when interpreting other items.
Example: The following must be considered when defining conformance test cases:

| | |
|---|---|
| NVM051 | The Memory Abstraction Interface and the underlying Flash EEPROM Emulation and EEPROM Abstraction Layer provide the NVRAM manager with a virtual linear 32bit address space. These logical 32bit addresses are composed of a 16bit logical block number and a 16bit block address offset. […] |

### 3.4.4 Configuration Parameter Definition

The specification item defines the meaning and/or use of a configuration parameter.
A correct understanding of configuration parameters and a correct selection of their values is important to the configuration generation process (see Chapter 5.2). Therefore, these specification items are relevant to conformance testing. The correct use of configuration parameters cannot, in general, be verified by automated test cases since expert knowledge of their context is required.
Example: The following must be considered when specifying configurations for conformance tests:

| | |
|---|---|
| NVM029 | NVM_DATASET_SELECTION_BITS:<br>Defines the number of least significant bits which shall be used to address a certain dataset of a NVRAM block within the interface to the memory hardware abstraction. |

### 3.4.5 Configuration Parameter Implementation

The specification item is related to implementation aspects of the configuration parameters. For example, Chapter 10 of an SWS specifies a configuration parameter as “#define”.
While it is important that a specific value of a parameter (as specified in the configuration XML-files) leads to correct module behavior (as observed during dynamic testing), the implementation of a configuration parameter or, in other words, the time at which a module can be configured (compile time, link time, post-build/run-time), can be specified in the SWS and influences the build and deploy process.
When the SWS specifies that a module should be configurable without recompiling (post-build configuration) but the parameter is implemented as a compile-time parameter, the implementation deviates from the specification. This violates the specification even when the behavior of the test object is not violated.
Example: The following is relevant to conformance testing:

| | |
|---|---|
| NVM124 | The block identifier shall be configured with the configuration tool and shall not be modified by the NVRAM manager. |

### 3.4.6 Requirement on Configuration

The specification item is a requirement for a correct configuration but not on the module itself. All conformance tests must respect the requirements on the configuration (i.e. only correct configurations are considered during conformance testing).
For example: The following is not relevant to conformance testing:

| | |
|---|---|
| NVM159d | To ensure that the DEM is fully operational at an early point of time, i.e. its NV data is restored to RAM, DEM related NVRAM blocks should be configured to have a low ID to be processed first within NvM_ReadAll. |

Consequently, the conformance testing does not include establishing whether configuration tools are intelligent enough to catch erroneous configurations.
For example: The following is not relevant to conformance testing:

| | |
|---|---|
| MemIf005a | All pre-compile time configuration parameters shall be checked statically (at least during compile time) for correctness. |

### 3.4.7 Detection of Wrong Configurations

Specification items of this category define the validity of configuration parameter values. They contain constraints on allowed configuration parameter values resulting from, for example, dependencies on other configuration parameters.

This type of specification item is not relevant to conformance testing since it is already the responsibility of the specific BSW module's configuration tool to verify the correctness of configuration parameter values.
Example: The following is not relevant to conformance testing:

| | |
|---|---|
| NVM089 | NvM.c shall check if the correct version of NvM.h is included. This shall be done by a preprocessor check of the version number NVM_SW_MAJOR_VERSION. |

### 3.4.8 Development Error Detection

These specification items define functionality related to Development Error Detection. Since errors detected and reported to the Development Error Tracer (DET) module are only in development code, this functionality is, strictly speaking, not relevant to conformance testing production BSW modules. The conformance test suites will test for anticipated DET errors, however (see below).
Example: The following is not relevant to conformance testing:

| | |
|---|---|
| NVM188 | If the NVM_DEV_ERROR_DETECT switch is enabled, API parameter checking is enabled. The detailed description of the detected errors can be found in chapter… |

***Note***: Error tracing is not needed to test production BSW modules, but it is needed when debugging conformance tests, however. Unanticipated DET error messages are useful indicators of problems in the tests.
During the validation phase, the conformance tests will be run with [ModuleName]_DEV_ERROR_DETECT==ON. During actual conformance testing, they will be run with the parameter set to OFF. The test cases for specification items related to Development Error Detection will therefore test for all anticipated (documented in the SWS) development error messages but make those tests conditional on [ModuleName]_DEV_ERROR_DETECT being set to ON.

### 3.4.9 Header Files for Internal Use

These specification items define header files to be provided and their inclusion structure. "Internal header files" are only used within the BSW module and not by other modules. The specification items are therefore not relevant to conformance testing as the header files do not affect the BSW module's interaction with its environment.
Example: The following is not relevant to conformance testing:

| | |
|---|---|
| NVM077 | NvM_Cfg.h shall include NvM_Types.h.<br>NvM_Types.h shall include Std_Types.h.<br>NvM.h shall include NvM_Cfg.h. |

### 3.4.10 Internal Source Code / Internal Header File

These items specify requirements on the contents of source code (*.c) or header (*.h) files. The requirements can vary from the inclusion of certain header files to high-level specifications of internal functionality or data structures.
Adherence to such requirements can only be verified through inspection of the respective source code or header file but not by examining the executable BSW module. Therefore, such specification items are not in scope of conformance testing.
Example: The following is not relevant to conformance testing:

| | |
|---|---|
| NVM150 | The NVRAM manager shall only contain that code that is needed to handle the configured block types. |

### 3.4.11 Header Files Provided for External Use

These specification items define the contents of header files required other BSW modules or SW components and specify how they will be provided.
Since the header files are used to exchange information and definitions between module generation processes, they influence the interaction of these BSW modules or SW components and are thus relevant to conformance testing. However, the correctness of the header files can only be verified by inspecting them and not by examining the executable BSW module.
Example: The following is relevant to conformance testing:

| | |
|---|---|
| NVM076b | The NVRAM manager shall provide: An API interface NvM.h providing the function prototypes to access the underlying NVRAM functions. |

### 3.4.12 Provided Signature / Required Signature

These specification items define function signatures (i.e. function name, return data type, parameter names and parameter data types) that are provided or required by the BSW module.
The correct implementation of these function signatures is of course crucial for conformance and is already checked while compiling and linking the BSW module for test into the dynamic test environment. These specification items are therefore relevant to conformance testing but are not tested by conformance test cases.
Example: The following is relevant to conformance testing:

| | |
|---|---|
| NVM044 | void NvM_SetDataIndex(NvM_BlockIdType BlockId, uint8 DataIndex) |

### 3.4.13 Module Behavior Requirement

This is the most important group of specification items. They define the functionality that interacts with the environment through the API and / or other kinds of interfaces (such as RAM blocks used for data exchange, hardware interfaces).
Specification items of this type are naturally relevant to conformance testing and are well suited to functional (black-box) testing with TTCN-3 conformance test cases.

Note that Section 3.8.2 of the Conformance Test Specification Background document [2] defines the types of bugs that are not relevant to AUTOSAR conformance. If a specification item defines behavior that falls into an excluded bug category, it will not be tested. All other forms of module behavior specifically defined in an SWS shall be tested by a test case.
Example: The following must be tested by a test case:

| | |
|---|---|
| NVM185 | NvM_ReadBlock: On successful enqueuing a request, the request result of the corresponding NVRAM block shall be set to NVM_REQ_PENDING. |

### 3.4.14 Module Reentrancy Requirement

These specification items define whether an API function is “re-entrant” or not. The current consensus is that they are not relevant to conformance testing.
Example: The following is not relevant to conformance testing:

| | |
|---|---|
| NVM045 | NvM_SetDataIndex must be reentrant. |

### 3.4.15 Execution in Interrupt Context Requirement

This kind of specification item defines whether an API function must have properties that allow its execution when called by an interrupt service routine. The current consensus is that they are not relevant to conformance testing.
Example: The following is not relevant to conformance testing:

| | |
|---|---|
| NVM060 | NvM_JobEndNotification: This routine might be called on interrupt level, depending on the calling function. |

### 3.4.16 Requirement on Other Module

Some specification items actually pose requirements on BSW modules that interface with the module under test. These specification items contain, for example, requirements on how certain API functions shall be used by other modules.
This group of specification items is not relevant for the creation of the CTSpec.
Example: The following is not relevant to conformance testing:

| | |
|---|---|
| NVM077g | Only NvM.h shall be included by the upper layer. |

### 3.4.17 Direct Hardware Access

SWS items that relate to interaction with hardware components are especially important. Such SWS items can often only be tested with individually designed hardware interfaces and thus require more effort to test than SWS items that relate to pure software functionality.
Whether these specifications items are relevant to conformance testing must be decided individually.

Example: The following is not relevant to conformance testing:

| | |
|---|---|
| EEP052 | In normal EEPROM mode, the EEPROM driver shall access the external EEPROM by usage of SPI channels that are configured for normal access to the SPI EEPROM. |

### 3.4.18 Vendor-Specific Extensions

The SWS documents leave some room for vendor-specific extensions and definitions such as error codes. This category consists of SWS items that define how this room is to be used.
Since the CTSpec can only test standardized functionality, vendor-specific extensions are naturally out of scope of conformance tests.
Example: The following is not relevant to conformance testing:

| | |
|---|---|
| NVM186 | Values for production code Event Ids are assigned externally by the configuration of the Dem. They are published in the file `Dem_IntErrId.h` and included via Dem.h. |

### 3.4.19 Pending on Bug

The category of these SWS items cannot be determined until a bug entry in AUTOSAR's Bugzilla system has been clarified. They are therefore categorized temporarily as "Pending on Bug". After the open issue has been clarified, the category must be changed according to the result.

## 3.5 Associate Test Method with Test Category

The "How to test" column in Table 2 indicates how to test specification items for conformance. The second column details the different ways of ensuring conformance:

| How to test | Explanation |
|---|---|
| **Manual inspection** | The SUT must be verified manually since no automation is possible. This means:<br>1. manually deducing the object to be considered (as part of the SUT) from the specification item<br>2. manually deducing the criteria to be verified from the specification item<br>3. verifying whether the deduced criteria are fulfilled by the object under consideration |
| **Consider in configuration sets** | The specification item has to be considered when the configuration sets for CTSpec execution are being defined and/or generated. In particular, the meaning of and interdependencies between configuration parameters have to be taken into account. |

| How to test | Explanation |
|---|---|
| **Check header file** | Adherence to the specification item is verified by checking for the presence of the specified contents in the affected header file (as part of the SUT). This can be done manually or can be automated. |
| **Compile-build process** | Adherence to the specification item is verified by mechanisms in the (already automated) compile-build process. Typically, violations result in errors which are reported by the tools used (e.g. compile error if API function is defined incorrectly). |
| **TTCN-3 test case** | A test case implemented in TTCN-3 can be defined to verify adherence to the specification item. |

**Table 2: Conformance Test Methods**

## 3.6 Review SWS Analysis Phase Results

A person other than the person who did the analysis must review the analysis result. The review criteria can be derived directly from the analysis guidelines given in the previous sections.
The refined SWS document produced in the analysis phase must state the names of both the analyst and the reviewer.

## 3.7 Delivery of the Refined SWS Document

The refined SWS document package must contain the modified specification text, the analysis results/comments and the categorization of all specification items.

# 4 The Main Phases of Conformance Test Case Creation

The main phases take the analysis phase output, the corrected SWS document and the categorized specification items and create the conformance test cases.
Depending on the category of the refined SWS items, the corresponding test cases are specified either in TTCN-3 (for automated execution) or by other means.

## 4.1 “Non-TTCN-3” Test Cases

The “non-TTCN-3” test cases are not applied to the executable module, but to other parts of the BSW module configuration / build process (e.g. source and header files).
There are many checks to be performed in the “Non-TTCN3” test cases and a generic automation methodology cannot be defined. However, some generic test methods have been identified in Table 3. These tests must be performed manually.

| Conformance Object | Test Method |
|---|---|
| Existence of file (.c/.h files) | Check whether file is provided |
| Include statements in .c/.h files | Check whether the file contains the statement |
| Definitions in .c/.h files | Check whether definitions are made in file |
| Implementation in .c files | Check implementation properties according to test criteria |
| Configuration data properties | Check properties according to test criteria |
| Configuration process | Check configuration tool output for success / error reports |
| Compile / build process | Check of compile / build tool output for success / error reports |

**Table 3 - Generic test methods for non-TTCN-3 test cases**

Due to their informal character, the “non-TTCN-3” test cases can be created in one step directly from the SWS item as depicted in Figure 2. The “non-TTCN-3” test cases shall produce a table in a format based on the template defined in the CTSpec template [3]. Additionally, the validation of these test cases is limited to manual reviews.

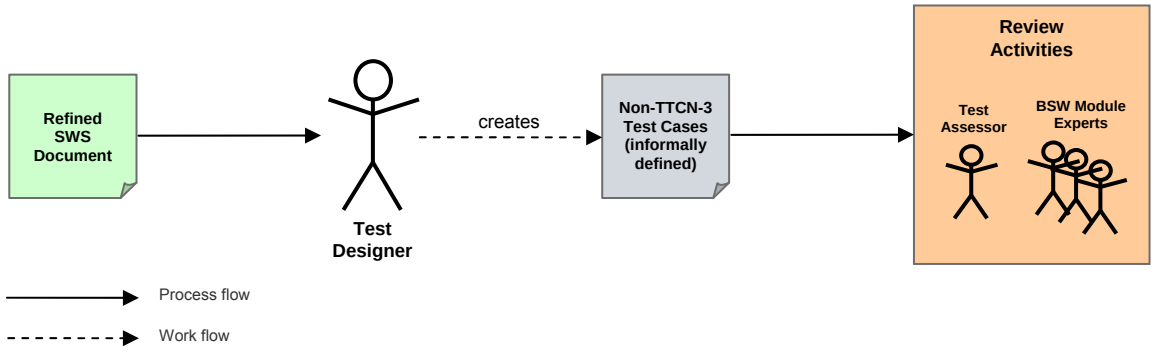


**Figure 2 - Overview of the creation process for "non-TTCN-3" test cases**

## 4.2 TTCN-3 Test Cases

TTCN-3 conformance test cases test the dynamic behavior of the BSW modules. The majority of functional requirements relevant to conformance relate to this behavior.
These test cases execute against a fully configured BSW module executable and use the API specified in the BSW module's SWS.
The TTCN-3 test cases are created in three phases: design, implementation and validation. Figure 3 shows an overview of the different roles involved, their activities, the tools and the work artifacts. These phases produce the TTCN-3 test cases together with the sets of configuration and test parameters in both AUTOSAR XML and TTCN-3 formats.

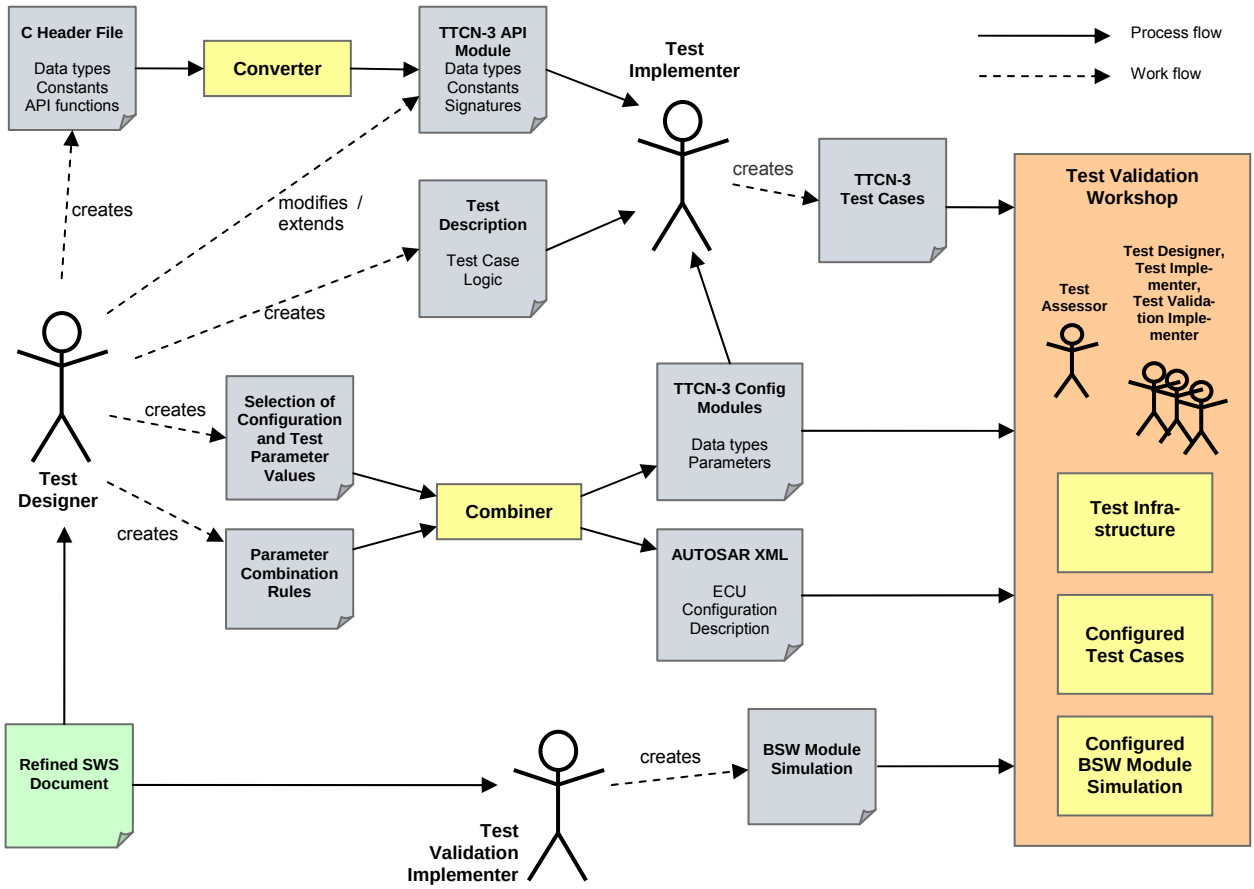


**Figure 3 – Roles, deliverables and tools in the design, implementation and validation phases**

The following chapters describe in detail the steps to be carried out within the three phases.

# 5 Design Phase

The Design Phase consists of the following two activities:

1. Identification of test cases
2. Design of the CTSpec

The “test case identification” activity has three main objectives:

- Identify the test cases based on the testable specification items
- Define the test steps based on the identified test cases
- Specify the rules for generating configuration sets

The objectives for the “CTSpec Design” activity are:

- Specify the TTCN-3 architecture for the test cases (test components, port etc.)
- Specify the behavior of test components that simulate neighboring modules
- Provide definitions (test functions, function signatures, data types etc.) for implementing the test cases

Note that these two activities – although described separately – cannot usually be performed sequentially. As the sub-activities are interdependent, (e.g., the test cases influence the test case architecture, test case architecture influences the test steps) these two activities are mostly performed in parallel.

## 5.1 Test Case Identification

The test cases and test steps must be defined based on the refined and atomized specification items. This involves four steps:

1. Definition of a unique test case identifier
2. Definition of the test purposes (i.e. test case objectives)
3. Definition of the logical test steps (i.e. test cases’ behavior)
4. Definition of additional conditions for the test case (i.e. test cases’ constraints)

### 5.1.1 Definition of a unique Test Case Identifier

Conformance test cases must be unambiguously identifiable. A unique test case number must therefore be defined whenever a new test case is created. This number shall be preceded by `[ModuleName]_TC` to associate the test case with a specific BSW module. The test case numbers must not be in sequence as there is always the possibility that certain test cases will be removed from the CTSpec.

Example:

```
NvM_TC020:        Test case number 20 for the NvM module.
```

### 5.1.2 Definition of the Test Purpose

The test purpose contains the well-defined objective of the test cases. It is a short description of what the test case verifies or ensures. Note that the test purpose is *not* identical to the specification item as the specification item describes what the BSW module does and the test purpose describes what the test case does.
Test purposes often use phrases as for example

```
Verify / Ensure / Check that ...
... on reception / in the event / under the condition of ...
... does / does not ...
```

Example:

```
"Ensure that NvM_MainFunction() when called before NvM_Init() returns
without changing the NvM module's state and without interacting with
other modules."
```

The test purposes must be defined "atomically" enough to permit easy separation and handling of the test cases.

### 5.1.3 Definition of the Test Steps

Refer to Chapter 3 in the Conformance Test Specification Background document [2] for guidance on the nature and extent of AUTOSAR testing. Section 3.8.1 defines which types of bugs should be tested. Note additionally that insofar as a specification item was not rejected completely in the SWS analysis phase, Section 3.8.2 defines which types of module behavior should not be tested. Section 3.6 defines which test methods are allowed and which are excluded in AUTOSAR conformance testing.
Although it is possible to define the test steps directly in the test execution language (TTCN-3), the work has been divided into test step definition and test step implementation. This allows the Test Designer to quickly define the test steps without having to account for structural completeness, formal correctness and error conditions.
However, the logic of the test steps must be defined precisely enough to be implementable in TTCN-3. This is accomplished with a pseudo code notation incorporating keywords and logical expressions based on the C programming language. Refer to Chapter 8 for a description of this notation.
The CTSpec design, which specifies the TTCN-3 test components, their interfaces and the functional decomposition of the test functions, must be followed when defining the test steps. Refer to Chapter 5.2 for detailed information on CTSpec design.

### 5.1.4 Definition of Additional Conditions

Conditions that are relevant to test case execution must be added to the test case bodies to complete the definitions:

**Pre-conditions** Conditions that must be fulfilled before the test case is executed (e.g. "module is not initialized when testing an init function like `NvM_Init`").

**Post-conditions** States in which the test case can leave the BSW module both in production and in during validation (e.g. development error conditions, see Section 3.4.8)

## 5.2 Design of the CTSpec

The overall CTSpec design must be specified before the test steps for each test purpose can be defined,.
The CTSpec design consists of the following:

| | |
|---|---|
| **Test case architecture** | Contains the structural information about the TTCN-3 test components being used and their interfaces (i.e. function signatures) to each other and to the BSW module under test. |
| **Decomposition principle for test functions** | For some BSW modules which require complex test cases, the principle for the functional decomposition of the test functions needs to be described. |
| **Specification of rules for generating configuration sets** | The rules to generate valid configuration sets which are suitable for conformance testing must be defined. |

### 5.2.1 Test Case Architecture

The test case architecture defines the TTCN-3 test components, their ports and the function signatures for communicating internally and with the module under test. UML component and class diagrams are a straightforward means to define this architecture.
Example:

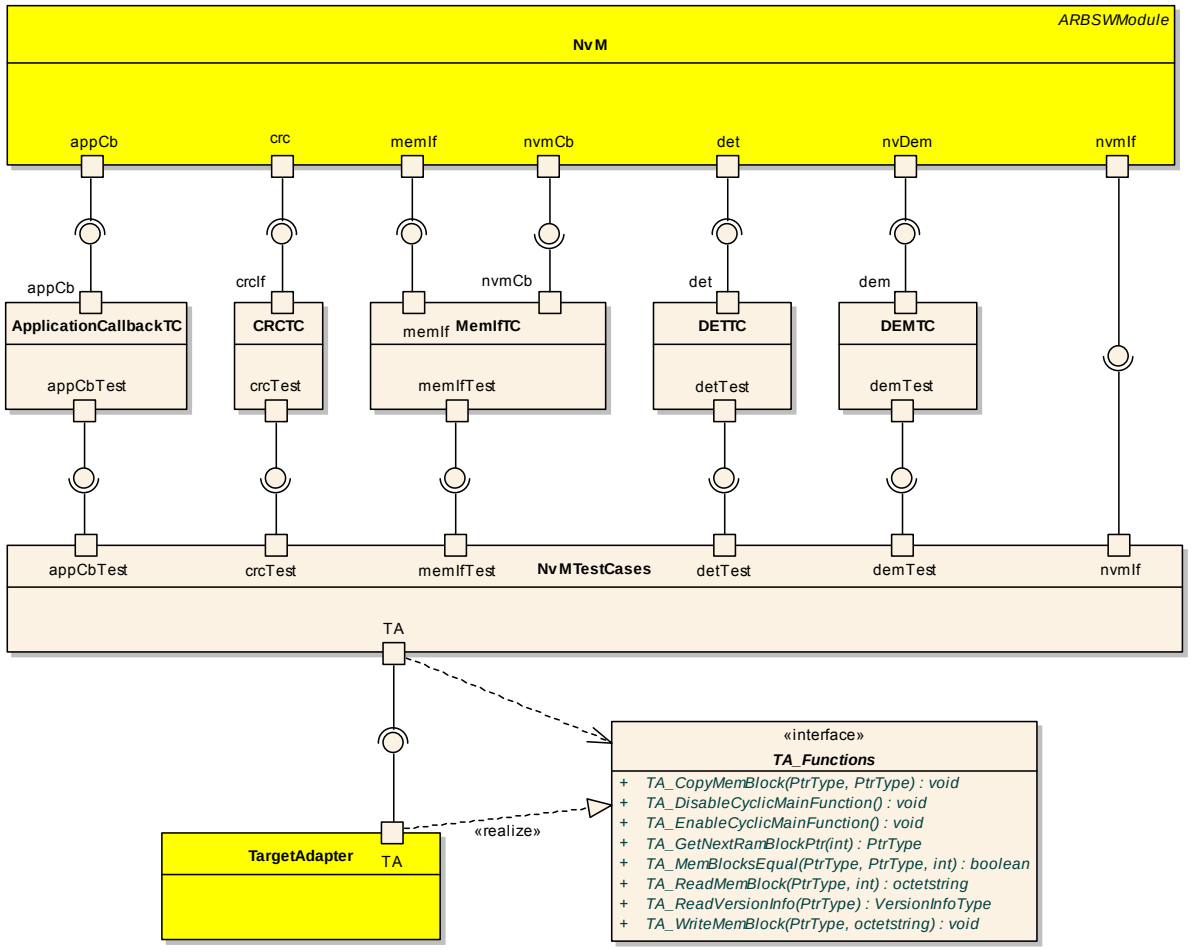


**Figure 4: UML Test Case Architecture for the NvM**

Refer to Chapter 9 for further details on the design of the CTSpec architecture.

## 5.3 Decomposition Principle for Test Functions

BSW modules with complex functionality usually require conformance test cases with complex, recurring test steps. To reduce redundancy, recurring test steps shall be defined as test functions (also called “base functions” on TTCN-3 level, see Chapter 6.3) that can be called by the test cases.
In order to make the concept of defining these test functions comprehensible, the CTSpec design for complex BSW modules must include either

- a description of the approach to defining the test functions or
- definitions of typical test functions from which the approach can be deduced.

This information enables the Test Implementer to correctly implement the TTCN-3 base functions based on the test functions, to correctly use the base functions in the test cases and to maintain consistency among the base functions.

## 5.4 Specification of Configuration Sets

AUTOSAR BSW modules can be configured by parameters to a very high degree. Configuration parameters not only define quantifiable attributes of the module (e.g. queue size, address, bit size, id number), more importantly, they sometimes define the qualitative behavior of the module.
Defining suitable configuration sets is therefore as fundamental to conformance testing as defining the right test cases and test steps. They are used to generate BSW modules which are configured correctly for test execution. The module can then be tested by a CTSpec that has also been configured with the same configuration set.
The CTSpec cannot define all AUTOSAR configuration parameters generically in advance, however. Some configuration parameters are constrained by the ICS[5] specified by the vendor of the BSW module. Therefore, the configuration sets used for CTSpec execution must be generated individually for each ICS.
In addition to the configuration parameters defined by AUTOSAR, the CTSpecs use test-specific parameters to define different test execution behavior variants. This concept is an efficient way to test different behavioral conditions without increasing the number of test cases.
AUTOSAR configuration parameters and test-specific parameters are defined in two steps:

1. During CTSpec creation, the rules for the configuration generation are defined with as many concrete values as possible.
2. After provision of the ICS, the rules are refined so that the final configuration sets can be generated.

The following sections describe the strategy and process for generating (in two steps) a number of configuration sets which result in good “configuration coverage” while remaining reasonable.

[5] The details of this concept have not been finalized at the time of writing. The ICS concept is outlined in the Background Document [2]

Good configuration coverage ensures that the most important and critical “working conditions” of the BSW module are tested by the CTSpec. This ultimately leads to good test coverage of the module’s functionality.

### 5.4.1 Input to the Generation Process for Configuration Sets

Both the SWS document and the “AUTOSAR ECU Configuration Parameters” document [7] define a BSW module’s configuration parameters. In case they are inconsistent, the ECU Configuration Parameters document shall take precedence.
Most AUTOSAR configuration parameters are relevant to conformance testing and one or more values must be defined to obtain “the right test data”. AUTOSAR configuration parameters can be further categorized as *module-specific configuration parameters* (one value per BSW module) or *“entity”-specific configuration parameters* when the module handles several individually configured “entities” (e.g. memory blocks with different types and properties for the NVRAM Manager).
As already mentioned above, the CTSpecs may define their own test-specific parameters to specify test variants within the test cases. These test-specific parameters are defined in the design phase of the CTSpec creation process [3] and are only used to configure the CTSpecs, i.e. they do not affect the BSW modules under test.

### 5.4.2 Strategy

For most BSW modules, the number of relevant configuration parameters is high enough to make manually defining appropriate configuration sets highly tedious, error-prone and thus practically hopeless.
Hence, automation and tool support is required to efficiently identify and combine the relevant configuration parameter values and to write the resulting configuration sets in the XML format defined by AUTOSAR.
Defining combinations of test input data is an inherent problem in the domain of functional testing which has been widely addressed by researchers. There are several appropriate methods (e.g. classification tree method) and quite a number of tools (e.g. CTE). A best-practice method which AUTOSAR deems adequate for conformance testing BSW modules is defining “equivalence classes” and combining the resulting configuration values in “pair-wise combination”.
This is a general approach to comprehensibly defining “equivalence classes” (i.e. the set of “representative” configuration parameter values) and a set of pair-wise combinations with an effective trade-off between configuration coverage and the total number of configuration sets. The application of these methods to AUTOSAR conformance testing is described in Chapters 5.4.3.3 and 5.4.3.4.

### 5.4.3 Generation Process

Figure 5 presents an overview of the configuration set generation process consisting of four steps with several activities. The steps performed during CTSpec creation are:

- “Analysis”
- “Clustering and Formalization”

- parts of “Value Selection and Refinement”

The remaining “Value Selection and Refinement” activities are performed when the ICS is provisioned and finally the “Combination and Generation” can be done.

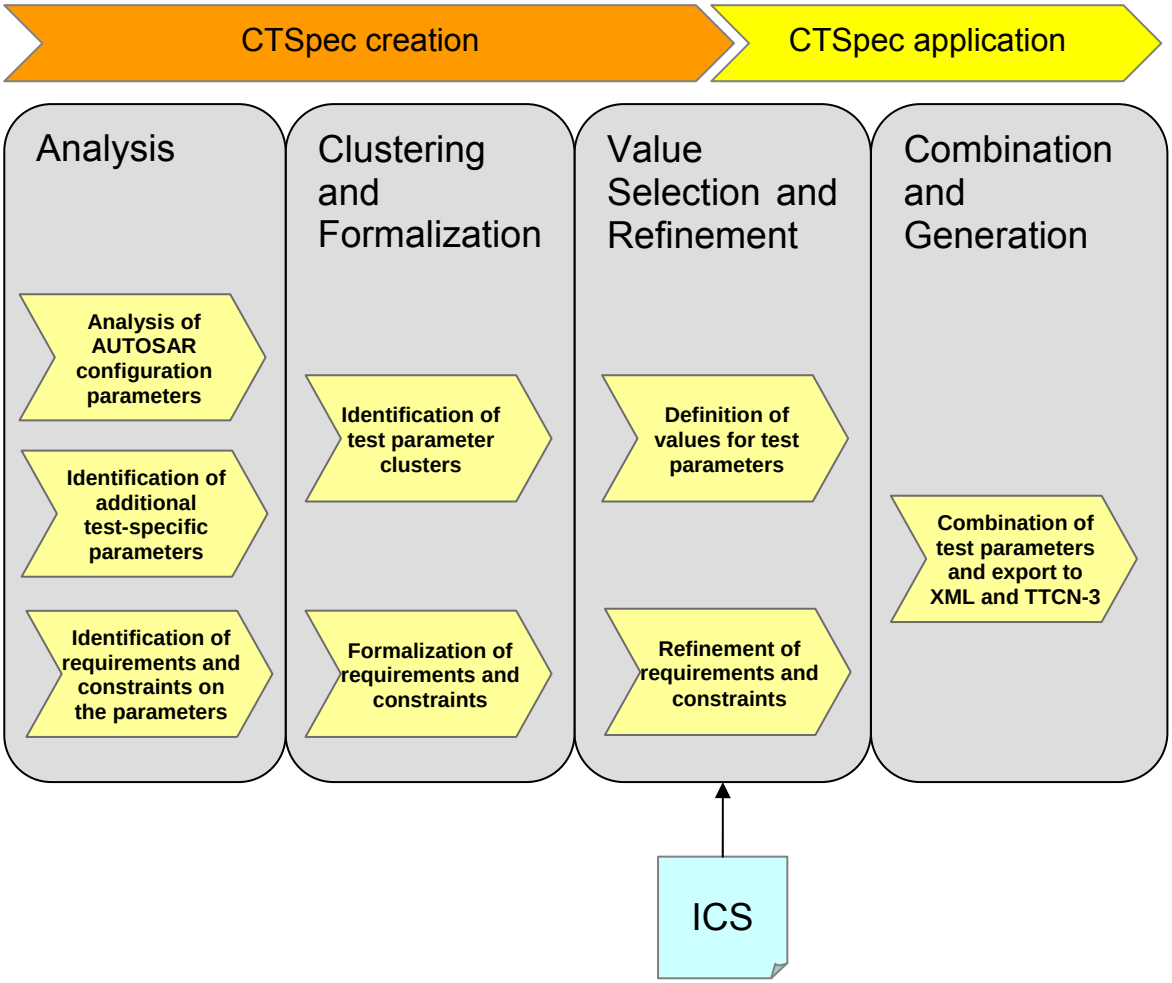


**Figure 5 - Overview on the generation process for configuration sets**

The following sections describe the steps and activities required to create configuration sets using the approach outlined in the previous sections.

#### 5.4.3.1 Step 1: Analysis

The analysis step is part of the first two phases (“Analysis Phase” and “Design Phase”) of the CTSpec creation process described in [3]. In this step, the AUTOSAR configuration parameters required by the BSW module are analyzed from the conformance testing point of view.

This step determines whether a configuration parameter value can and should be defined during the CTSpec creation process or whether the value depends on execution, hardware or other external conditions. In the latter case, the value can only be determined after the external conditions (e.g. hardware devices for test execution) have been defined and is therefore out of the scope of the CTSpec creation process

Test specific test parameters that are identified during test case design are defined and combined with the AUTOSAR configuration parameters[6]. This results in a collection of *test parameters*.
Requirements and constraints on the test parameters can then be identified either directly or from insights gained from the SWS document. These requirements and constraints must also be documented.
The remaining work steps treat both the AUTOSAR configuration parameters and test specific parameters in the same way and the same automatable combination algorithms can be applied to both types of parameters.

#### 5.4.3.2 Step 2: Clustering and Formalization

Some test parameters are inherently interdependent and cannot be treated separately when determining test combinations. This happens, for example, when a set of parameters all relate to a specific functionality (e.g. job queuing and prioritization).
In this step, the strongly interdependent parameters are identified and are put into clusters. The clustered test parameters are regarded thereafter as “a whole”.
In addition to the clustering activity, the requirements and constraints on the test parameters are formalized into expressions that can be processed automatically.

#### 5.4.3.3 Step 3: Value Selection and Refinement

The value selection and refinement step determines the values for both non-clustered and clustered test parameters and refines the requirements and constraints on the test parameters. As depicted in Figure 5, this step is subdivided into activities performed during CTSpec creation and those that must be done after provision of the ICS. i.e. when the CTSpec is to be executed[7].
Creating equivalence classes can be helpful in finding concrete, representative values for the test parameters. In general, the following guidelines should be applied.

| | |
|---|---|
| **Boolean or Enumeration test parameters** | All possible values are selected except those which are irrelevant a-priori (e.g. not in focus of conformance testing) or otherwise irrelevant to conformance testing. These exclusions must be documented clearly. |
| **Integer test parameters** | If possible, equivalence classes for the test parameter should be defined and reasonable values picked from these classes.<br>If equivalence classes cannot be identified, then usually the lowest value, the highest value (if reasonable) and at least one intermediate value are selected. |

The value combinations of clustered test parameters are defined manually according to the inherent constraints. Each combination is then regarded as a “cluster value”.

[6] In this document, the terminology “AUTOSAR configuration parameters” refers to those parameters for which values can and should be defined during the CTSpec creation process.

[7] The details of this division of activities before and after provision of the ICS must be further specified after the ICS concept has been finalized.

After the ICS has been provisioned and checked for validity, previously generated configuration parameter values for must be checked or regenerated according to the constraints stated in the ICS. Further ICS requirements and constraints on the test parameters must be added as formal expressions.

#### 5.4.3.4 Step 4: Combination and Generation

After the ICS has been taken into account, the configuration sets are generated from the relevant refined test parameters definitions, clusters, selected values and formalized constraints. All pair-wise combinations of test parameters and clusters are generated using the selected values.
A TTCN-3 file and an ECU Configuration file in XML are then generated from the configuration sets.
Generally, generating the combinations cannot be done manually and must be automated with a custom tool.

### 5.4.4 Example of Configuration Set Creation

This section presents a possible realization approach to generating configuration sets based on the steps described in the previous sections. It uses a spreadsheet (e.g. *Microsoft Excel*) and a custom tool, *Combiner,* that has been developed by Carmeq.
The spreadsheet (also referred to as “combination table”) should be filled out during the “Analysis”, “Clustering and Formalization” and “Value Selection and Refinement” steps. In the last step, “Combination and Generation”, the custom tool takes the combination table, evaluates the constraints, applies the pair-wise combination algorithm and writes out the generated configuration sets.

#### 5.4.4.1 “Analysis”, “Clustering” and “Value Selection and Refinement”

Figure 6 shows a combination table with module-specific configuration parameters and their selected values for an NVRAM Manager module.

| [MODULE_PARAMETERS] | | Selected Values | | | | | |
|---|---|---|---|---|---|---|---|
| {NVM_JOB_PRIORITIZATION, NVM_SIZE_IMMEDIATE_JOB_QUEUE, NVM_SIZE_STANDARD_JOB_QUEUE} | | {FALSE,0,1} | {FALSE,0,10} | {FALSE,0,255} | {TRUE,1,10} | {TRUE,5,10} | {TRUE,255,10} |
| NVM_MULTI_BLOCK_CALLBACK | | TestStubMultiblockCallback | NULL | | | | |
| NVM_DYNAMIC_CONFIGURATION | | TRUE | FALSE | | | | |
| NVM_MAX_NUM_OF_WRITE_RETRIES | | 0 | 2 | 7 | | | |
| NVM_CRC_NUM_OF_BYTES | | 1 | 5 | 65535 | | | |
| NVM_DRV_MODE_SWITCH | | TRUE | FALSE | | | | |
| NVM_DATASET_SELECTION_BITS | | 1 | 3 | 8 | | | |
| NVM_SET_RAM_BLOCK_STATUS_API | | TRUE | FALSE | | | | |
| NVM_POLLING_MODE | | TRUE | FALSE | | | | |
| NVM_DEV_ERROR_DETECT | | TRUE | | | | | |
| NVM_API_CONFIG_CLASS | | NvmApiConfigClass3 | | | | | |
| NVM_COMPILED_CONFIG_ID | | 0x1234 | | | | | |
| NVM_VERSION_INFO_API | | TRUE | | | | | |
| TCConfigIdBehavior | | MATCH | FAIL | MISMATCH | | | |

**Figure 6 - Example: Combination table for module-specific configuration parameters of NvM**

The first column contains the test parameters (or clusters). The identifiers of AUTOSAR configuration parameters consist of capital letters and underscores (e.g. `NVM_JOB_PRIORITIZATION`). The identifiers of test-specific configuration parameters are denoted by a "camel-case" notation (e.g. `TCConfigIdBehavior`).
The "Selected Values" columns contain the test parameter (or cluster) values. Clustered test parameters and clustered values are enclosed by curly brackets.
The second column in the combination table of Figure 6 provides for the formalized constraints. No interdependencies or constraints have been identified for the module-specific parameters in this example. This topic is discussed in the following sections.

### 5.4.5 Interdependence between Test Parameters

One way of dealing with interdependent test parameters is clustering. The cluster parameter values must be selected manually while respecting the given constraints and interdependencies.
Some BSW modules have an additional level of interdependence between test parameters that cannot be resolved by clustering, however.

#### 5.4.5.1 Module-specific Parameters and "Entity"-specific Parameters

In general, two groups of BSW module test parameters can be distinguished:

| | |
|---|---|
| **Module-specific test parameters** | These are defined once for a BSW module instance. |
| **"Entity"-specific test parameters** | Some BSW modules have a number of "entities" (e.g. memory blocks) which must be configured individually and a specific set of configuration parameters must be provided for each of these "entities". |

Generating the pair-wise combinations for "entity"-specific test parameters results in a set of differently configured "entities". These "entities" are collected into the same configuration set. In this way, a test case can then iterate through the "entities" and apply the tests on those "entities" that are appropriately configured.

#### 5.4.5.2 Example of Interdependence

The NVRAM Manager (NvM) has both "module-specific" and "block-specific" test parameters. The module-specific parameters are unique per NvM module. There are block-specific parameters for each NvM block controlled by the NvM module.
Example for module-specific parameter:

| | |
|---|---|
| **`NVM_DATASET_SELECTION_BITS`** | Defines the number of least significant bits used to address a certain dataset of a NVRAM block within the memory hardware abstraction interface. |

Example for block specific parameter:

**NVM_NV_BLOCK_NUM** Defines the number of NVRAM blocks in a contiguous area according to the block management type.

The number of bits used to address the dataset NVRAM blocks is defined by NVM_DATASET_SELECTION_BITS. However, NVM_NV_BLOCK_NUM is applied to dataset NVRAM blocks and specifies how many NVRAM blocks are available in total. Obviously, there is an interdependence between NVM_DATASET_SELECTION_BITS and NVM_NV_BLOCK_NUM:

- When, for example, NVM_DATASET_SELECTION_BITS is set to 4, only $2^4$ = 16 dataset NVRAM blocks can be addressed.
- Thus, in this case, it would not be correct to define NVM_NV_BLOCK_NUM to be greater than 16.

#### 5.4.5.3 Solution Approach to Interdependence

The logic behind the interdependence between module specific and “entity specific” parameters (e.g. for memory blocks) is extensive. A generic solution to handle this interdependence in configuration set generation is thus required.
It is assumed that certain combinations of module-specific parameters imply certain groups of “entities”. When defining and combining these “entity” groups’ configuration parameters, dependencies on module-specific parameters must be respected.
A generic solution based on algebra and evaluation of the interdependence between configuration parameters has been developed for creating the configuration sets:

- The constraints that describe the interdependence between module specific and entity specific test parameters are formulated as algebraic expressions.
- When creating the pair-wise combinations for the “entity specific” test parameters, only those entity specific parameter values are used that satisfy the given constraints.

Example
Values selected for pair-wise combination:

```
NVM_DATASET_SELECTION_BITS = { 2, 4, 8 }
NVM_NV_BLOCK_NUM = { 3, 6, 10, 32, 50 }
```

The constraint between these two configuration parameters is formulated as:

```
NVM_NV_BLOCK_NUM <= 2^NVM_DATASET_SELECTION_BITS
```

Applying this constraint results in the following:

- When 2 is selected for NVM_DATASET_SELECTION_BITS , only 3 is used for NVM_NV_BLOCK_NUM in pair-wise combination.
- When 4 is selected for NVM_DATASET_SELECTION_BITS , then the values { 3, 6, 10 } are used for NVM_NV_BLOCK_NUM in pair-wise combination.
- When 8 is selected for NVM_DATASET_SELECTION_BITS , then all values { 3, 6, 10, 32, 50 } are used for NVM_NV_BLOCK_NUM in pair-wise combination.

#### 5.4.5.4 Exclusions

In addition to the interdependence between module-specific parameters and “entity-specific” parameters, there is interdependence with test parameters.
Example:

NVM_BLOCK_USE_CRC == FALSE and NVM_CALC_RAM_BLOCK_CRC == TRUE are mutually exclusive[8].
The exclusion relations between certain combinations of test parameters values must also be defined and taken into account when generating the pair-wise combinations.
The exclusion relations must be expressed as pairs using arbitrary combinations of equalities and inequalities.
For example:
NVM_BLOCK_CRC_TYPE == NVM_CRC16 and NVM_NV_BLOCK_LENGTH <= 3
NVM_BLOCK_CRC_TYPE == NVM_CRC32 and NVM_NV_BLOCK_LENGTH <= 5

#### 5.4.5.5 Example of a Combination Table after Formalization of Constraints

Figure 7 shows the block-specific configuration parameters part of the combination table from Figure 6. The second column now contains the formalized constraints between module-specific and block-specific parameters. The exclusion constraints have been added in rows.

### 5.4.6 Output of the Configuration Generation Process

The configuration generation process produces a number of configuration sets in both TTCN-3 and AUTOSAR XML format.
The TTCN-3 format for configuration data is described in Chapter 10.1.2.
The AUTOSAR XML format for configuration data is defined in [7] and [9]

## 5.5 Prepare Specification of Test Cases

As preparation for the test case specification (i.e. implementation) phase, the TTCN-3 API module (containing the BSW module's data type definitions and API functions) and the TTCN-3 Config module (containing the test parameters) must be created.
These two TTCN-3 modules are created automatically by two tools (see Figure 3):

**Converter** This tool converts a C header file containing the BSW module's API functions and data types into the appropriate TTCN-3 syntax.

**Combiner** This tool combines the definitions and rules in the "configuration combination table" (see Chapter 5.4.3) and produces configuration sets in both AUTOSAR XML and TTCN-3 formats.

[8] When CRC is disabled generally, it cannot be enabled for a specific block.

| [BLOCK_PARAMETERS] | Constraint | Selected Values | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| NVM_BLOCK_USE_CRC | | TRUE | FALSE | | | | | | | |
| NVM_CALC_RAM_BLOCK_CRC | | TRUE | FALSE | | | | | | | |
| NVM_BLOCK_CRC_TYPE | | NVM_CRC16 | NVM_CRC32 | | | | | | | |
| NVM_BLOCK_WRITE_PROT | | TRUE | FALSE | | | | | | | |
| {NVM_BLOCK_MANAGEMENT_TYPE, NVM_NV_BLOCK_NUM, NVM_ROM_BLOCK_NUM} | NVM_NV_BLOCK_NUM <= 2**NVM_DATASET_SELECTION_BITS | {NATIVE,1,0} | {NATIVE,1,1} | {REDUNDANT,2,0} | {REDUNDANT,2,1} | {DATASET,1,0} | {DATASET,10,0} | {DATASET,1,10} | {DATASET,10,0} | {DATASET,10,10} |
| NVM_WRITE_BLOCK_ONCE | | TRUE | FALSE | | | | | | | |
| NVM_RESISTANT_TO_CHANGED_SW | (NVM_RESISTANT_TO_CHANGED_SW eq TRUE) \|\| (NVM_DYNAMIC_CONFIGURATION eq TRUE) | TRUE | FALSE | | | | | | | |
| NVM_BLOCK_JOB_PRIORITY | | 0 | 1 | 2 | 255 | | | | | |
| NVM_NV_BLOCK_LENGTH | | 1 | 5 | 8 | 100 | | | | | |
| NVM_RAM_BLOCK_DATA_ADDRESS | | RamAddress<N> | NULL | | | | | | | |
| NVM_ROM_BLOCK_DATA_ADDRESS | | RomAddress<N> | NULL | | | | | | | |
| NVM_INIT_BLOCK_CALLBACK | | TestStubInitBlockCallback<N> | NULL | | | | | | | |
| NVM_SINGLE_BLOCK_CALLBACK | | TestStubSingleBlockCallback<N> | NULL | | | | | | | |
| NVM_SELECT_BLOCK_FOR_READALL | | TRUE | FALSE | | | | | | | |
| TCInitialRamBlockState | | VALID_INVALIDCRC | VALID_VALIDCRC | INVALID | | | | | | |
| TCMemIfBehavior | | DO_NOT_ACCEPT_JOB | FAIL_JOB | CONTENTS_INVALID | WRONG_CRC | SUCCESS | | | | |
| TCMemIfBehaviorRedundantBlock | | DO_NOT_ACCEPT_JOB | FAIL_JOB | CONTENTS_INVALID | WRONG_CRC | SUCCESS | | | | |
| | | | | | | | | | | |
| [EXCLUSION] | | | | | | | | | | |
| NVM_BLOCK_USE_CRC | | FALSE | | | | | | | | |
| NVM_CALC_RAM_BLOCK_CRC | | TRUE | | | | | | | | |
| | | | | | | | | | | |
| [EXCLUSION] | | | | | | | | | | |
| NVM_SELECT_BLOCK_FOR_READALL | | TRUE | | | | | | | | |
| NVM_BLOCK_MANAGEMENT_TYPE | | DATASET | | | | | | | | |
| | | | | | | | | | | |
| [EXCLUSION] | | | | | | | | | | |
| NVM_SELECT_BLOCK_FOR_READALL | | TRUE | | | | | | | | |
| NVM_RAM_BLOCK_DATA_ADDRESS | | NULL | | | | | | | | |
| | | | | | | | | | | |
| [EXCLUSION] | | | | | | | | | | |
| NVM_SELECT_BLOCK_FOR_READALL | | TRUE | | | | | | | | |
| NVM_RESISTANT_TO_CHANGED_SW | | FALSE | | | | | | | | |
| | | | | | | | | | | |
| [EXCLUSION] | | | | | | | | | | |
| NVM_ROM_BLOCK_DATA_ADDRESS | | NULL | | | | | | | | |
| NVM_ROM_BLOCK_NUM | | 1 | | | | | | | | |
| | | | | | | | | | | |
| [EXCLUSION] | | | | | | | | | | |
| NVM_ROM_BLOCK_DATA_ADDRESS | | NULL | | | | | | | | |
| NVM_ROM_BLOCK_NUM | | 10 | | | | | | | | |

**Figure 7 - Example: Combination table for block-specific NvM configuration parameters**

# 6 Implementation Phase

Two sets of products are developed in the implementation phase:

- The TTCN-3 test cases
- The BSW module simulation

While the BSW module simulation is based solely on the refined SWS document, the implementation of the TTCN-3 test cases has the following inputs:

- TTCN-3 API module containing function signatures and data type definitions for the BSW module's API
- TTCN-3 Config module containing parameters for the test object (AUTOSAR configuration parameters) and the test cases (test-specific parameters
- UML model specifying the CTSpec architecture
- Test steps for each test case in pseudo code notation

## 6.1 Implementation of the BSW Module Simulation

The BSW module simulation simulates the module's correct visible behavior. It is not necessarily efficient and makes certain assumptions to reduce implementation effort. "Visible behavior" is the module's behavior at its interfaces. SWS requirements related to "internal mechanisms" that, for example, were stated for performance reasons, can be ignored as long as the module's behavior at its interfaces is unaffected. Since the BSW module simulation does not run on an embedded system, the following simplifications can be applied:

| | |
|---|---|
| **Dynamic memory management** | Embedded systems usually treat memory as static and the (maximum) memory size for data structures must be defined during coding. Dynamic memory management offers the programmer additional possibilities as memory can be allocated and released as needed at runtime. |
| **Simplified execution model** | Ensuring reentrancy (when required) is a major difficulty in implementing embedded systems. The simulation uses a simplified model (i.e. sequential function execution). Reentrancy and race conditions for shared resources thus become "non-issues". |
| **Complex data structures** | Embedded systems must handle their resources (i.e. CPU and memory) economically and complex data structures, possibly based on dynamic memory management, are often prohibited. The simulation can use them, however. |
| **Advanced function libraries** | Function libraries that implement commonly used functionality (e.g. queues) can be used. This can also involve "language extensions" implemented in libraries such as System-C. |
| **Advanced design methodologies** | Advanced design methodologies, such as object orientation, and their supporting tools (CASE) can be used. |

To summarize: the simulations can be implemented efficiently by applying state-of-the-art methods. In general, embedded system constraints need not be considered.

## 6.2 Implementation of the TTCN-3 Test Cases

The test cases are TTCN-3 files that, along with the appropriate TTCN-3 configuration file, can be executed against a SUT. If the test designer and the test implementer are the same person, the TTCN-3 test cases are implemented directly from the test purpose. If not, the test cases are derived from the pseudo-code test step definitions. The TTCN-3 source code structure should be the same for all CTSpecs to simplify handling and maintenance. The basic structure is specified in the following section. Refer to Chapter 10 for further TTCN-3 related CTSpec implementation details.

## 6.3 CTSpec File Structure

The test cases are structured into various TTCN-3 files. Common definitions & functions (e.g. standard types, DET and DEM interfaces) are put in files with fixed names:

| | |
|---|---|
| **`std_types.ttcn3`** | Specifies AUTOSAR standard types and other common types. |
| **`det_dem.ttcn3`** | Specifies the Development Error Tracer (DET) and Diagnostic Event Manager (DEM) interfaces that the test cases provide the SUT for error reporting. |

The BSW module specific files have the module's official abbreviation (e.g. NvM for NVRAM Manager) as prefix and a suffix indicating the file type. The following files make up the conformance test cases (* indicates the module's prefix):

| | |
|---|---|
| **`*_test_suite.ttcn3`** | Contains generic test parameters (e.g. test case timeouts) and a control part to select/deselect test cases for execution. |
| **`*_test_cases.ttcn3`** | Specifies all conformance test cases belonging to the BSW module. Each test case can potentially call the base functions |
| **`*_base_functions.ttcn3`** | Contains the CTSpec's base functions that encapsulate shared test steps. |
| **`*_test_architecture.ttcn3`** | Specifies the test components, the SUT's interfaces and their interactions. Furthermore, it provides generic functions for preparing and finalizing test cases and handling failure behavior. |
| **`*_api.ttcn3`** | Specifies the "abstract test system interface", i.e. the interface between the test case and the SUT. Test cases use this interface to stimulate and observe the SUT. |
| **`*_test_parameters.ttcn3`** | Contains AUTOSAR and test-specific configuration parameter values. |

`*_test_suite.ttcn3` is the main CTSpec file. It imports the required TTCN-3 modules from other files. They, in turn, may also import further modules.

# 7 Validation Phase

The following human errors could be expected when implementing the CTSpecs:

- Typos and syntactical errors
- "Qualitative" errors:
  - Logic errors in the sequence of test stimulus and observation
  - Logic errors in algebraic operations (e.g. comparisons to deduce test verdict)
  - Logic errors in the use of control structures
  - Wrong API calls and callbacks
  - Wrong constants or variables (e.g. for configuration values)
  - Structural errors in implementing the CTSpec architecture
- "Quantitative" errors:
  - Wrong parameters values in API calls
  - Wrong configuration parameter values
  - Wrong values in algebraic operations

The validation phase therefore has the following three main objectives:

1. Verify that the CTSpec complies to the TTCN-3 coding style defined in Chapter 12 and to the file structure described in Chapter 6.3.
2. Verify that the implemented test cases comply to the test steps as described in Chapter 5.1.3.
3. Verify the overall correctness of all CTSpec creation process activities by executing the CTSpec against a simulation of the BSW module.

The following verify and validate the CTSpecs with respect to these objectives:

1. "Shallow" code review to verify that the implemented TTCN-3 test cases adhere to the TTCN-3 coding guidelines and to the specified file structure.
2. Compiling and linking with the TTCN-3 API and Config modules to ensure formal correctness and executability.
3. "Deep" code review to verify that the TTCN-3 test cases correctly implement the test steps.
4. Execution of the conformance test cases against the BSW module simulation to validate whether they correctly report "pass" and "fail".

The first three activities are straightforward but the fourth needs further explanation.

## 7.1 Test Case Validation using a Simulation of the BSW Module

This section describes the concept of validating conformance test cases by executing them against a simulation of the BSW module under test in more detail.

### 7.1.1 Motivation

Validating test cases with simulation has the following advantages:

- The simulation focuses on the BSW module's functionality as a whole and not on individual requirements. This avoids code duplication and thus reduces implementation effort compared to preparing test cases for the CTSpec.

- The simulation is implemented in a high-level programming language on a standard PC. This reduces the overall implementation effort compared to implementing on an embedded system.
- The personnel familiar with the BSW module's specification can use their preferred high-level programming language (e.g. Java, C++). This accelerates implementation of the simulation and results in fewer errors
- Code coverage in the simulation can be analyzed during execution of the conformance test cases. The functionalities within the BSW module that have not been covered by the conformance test cases can thus be identified.

### 7.1.2 Validation Setup

Figure 8 shows the setup for validating the conformance test cases against a BSW module simulation. The Test System is realized according to [4]. However the SUT is replaced by a "SUT Substitute" composed of

- the BSW module simulation
- the validation target adapter

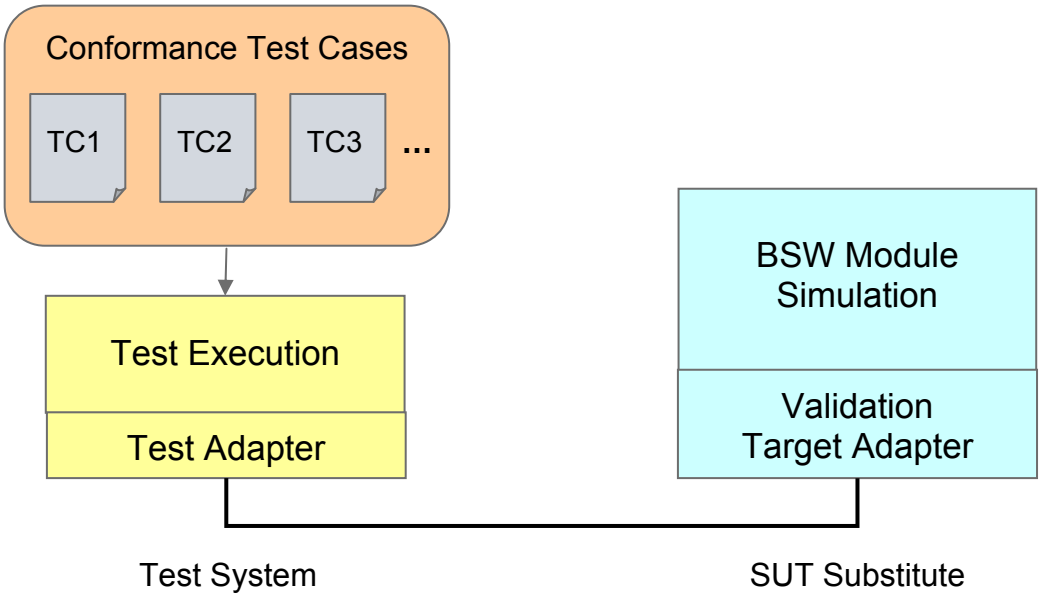


**Figure 8 - Validation Setup**

### 7.1.3 Error Tracing during Validation

The module simulations used to validate the test cases shall all implement DET development error detection mechanisms as specified in [7]. This ensures consistent interfaces among the simulations and should ease their integration, if necessary.

## 7.2 The Validation Workshop

A “validation workshop” shall be held to validate the test cases by executing the CTSpec against the BSW module simulation. The following should attend:

- the Test Assessor
- the Test Designer
- the Test Implementer
- the Test Validation (Simulation) Implementer

Involving knowledgeable representatives from test case, simulation and test infrastructure development ensures that problems can be quickly identified and solved.

### 7.2.1 Preparation for the Validation Workshop

Before the workshop is held, the Test Assessor must develop the test infrastructure (i.e. test adapters, test tools on Test-PC, communication network etc.) and set it up.
The Test Assessor must also make end-to-end communication tests using the API of the BSW module under test. These tests verify that communication events are correctly transferred between the test cases and the target adapter.

### 7.2.2 Conducting the Validation Workshop

The target test adapter (implemented by the Test Assessor) is first integrated with the BSW module simulation (implemented by the Test Validation Implementer). After that, the test cases can be compiled and validation can start with simple test cases.
The subsequent activities depend on the problems and errors found during test case execution. The workshop participants must work together to identify the causes of problems and to define the right approaches to solving them.
The issues identified and the solutions must be documented for traceability.

## 7.3 Result of the Validation Workshop

The validation workshop finally results in:

- an improved version of the BSW module simulation,
- an improved version of the TTCN-3 test cases,
- an issue list composed of generic issues and test case-specific issues.

All issue list items must be resolved (by involving the SWS authors, if necessary) before the CTSpec can be regarded as “validated”.

## 7.4 “Validation against Misbehavior”

The validation activities, especially the validation workshop, focus on eliminating errors in both the module simulation and the test cases. The simulation’s objective is to emulate correct module behavior and verify that the test cases correctly report “pass”
This approach does not validate that the test cases correctly report “fail” when executed against a non-conformant implementation, however.

In theory, “validation against misbehavior” is far more complex than “validation against correct behavior” because the number of conceivable faults is much higher than the number of correct results for any given module implementation.
Therefore, the confidence in the CTSpec gained through “validation against misbehavior” is inherently limited by the number of faults injected into the simulation.
“Validation against misbehavior” must be carried out after the CTSpec has been validated against correct behavior since confidence in the correctness of the simulation must be gained before errors are purposely injected into the simulation.
The steps to “validate against misbehavior” are as follows:

1. For each test case, identify possible BSW module implementation errors that should result in a “fail” verdict. These errors must be linked to the test cases to maintain traceability.
2. Implement these errors in the BSW module simulation. It must be possible to switch each error on or off individually.
3. Execute the CTSpec several times against the BSW module simulation with implemented errors. For each run, switch a different error on.
4. Verify that for each CTSpec execution, only those test cases report “fail” that are associated with the error that were switched on during execution.

Some tasks must be carried out manually (e.g. error identification and implementation) and some tasks that can be automated (e.g. execution with different errors activated, verification of failed test cases).

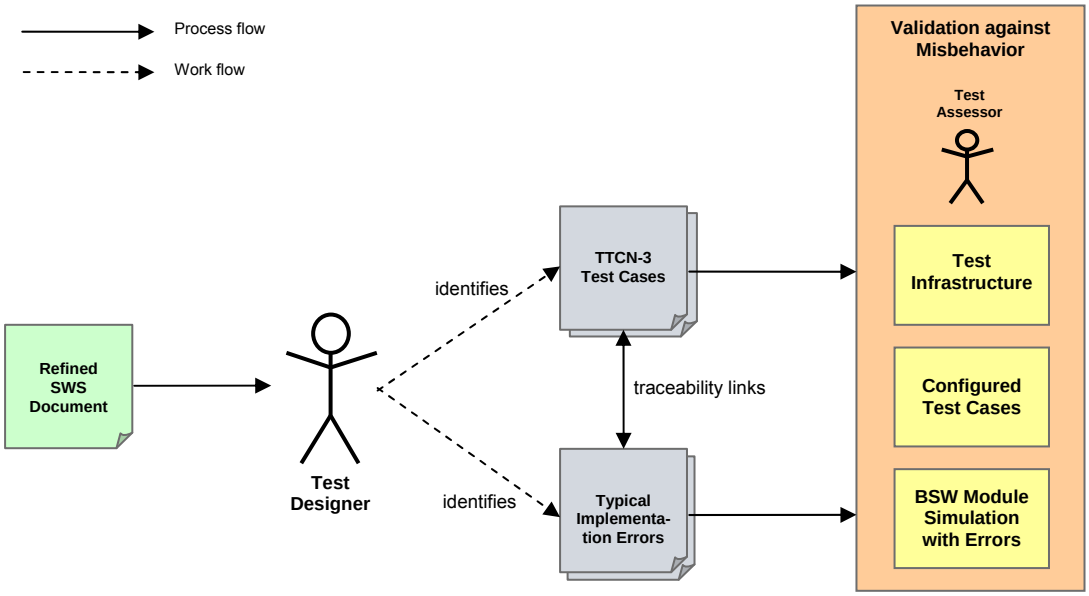


**Figure 9 - Work and process flow for "validation against misbehavior"**

The Test Designer must define the extent of “validating against misbehavior” (i.e. the number of errors injected into the simulation) based on the impacts of false positives on the quality of the test case implementation. There is no general “rule of thumb”.

# 8 Pseudo-Code for Test Step Descriptions

Test steps can be described in a semi-formal notation, a pseudo code, as an intermediate step before coding in TTCN-3. This chapter describes the semantics of this pseudo-code.

### 8.1.1 Objective

The Test Description document contains the results of the SWS analysis, test case descriptions and configuration parameter definitions and combinations. The test case descriptions are the basis for implementing the conformance test cases in TTCN-3 and must therefore comprise all information necessary for test case implementation.
Test step descriptions are used as a means of communication between the test case designer and the test case implementer. This is necessary when two different persons or organizations design and implement the test cases. Otherwise the test cases would be written in TTCN-3 directly. The “logical flow” of the test case must be specified in sufficient detail to support this “arms-length” communication.
Message sequence charts are commonly used for this purpose. However, the circumstances of BSW module APIs are generally already clear from the definition, i.e., the function name’s prefix indicates the caller and the callee and the BSW module’s location in the overall AUTOSAR architecture (e.g. `NvM_ReadBlock()` indicates an API provided by the NVRAM Manager to be called by an Application SW-C).
The pseudo-code notation is used in the Test Description documents to efficiently describe logical test steps involving stimulating the BSW module under test and observing its reactions. This notation is described in the following sections.
Note that the pseudo-code is a means to document test steps and their logic quickly and efficiently. It is not intended to be complete in every aspect or to be executable.

### 8.1.2 General Conventions

The notation combines elements from structured programming languages with the ease of informal descriptions. However, being informal, it is often not precise enough and a basic technical background is needed to understand it correctly. Additional explanatory documents (e.g. a UML model) or workshops must be provided to ensure this basic understanding on the receiver side (i.e. the TTCN-3 implementers).
The following general conventions are defined for this notation:

- Test steps are defined on individual lines terminated by semi-colons.
  Example: EXPECT NvM_JobEndNotification();
- Configuration parameters are used directly with their names in capital letters.
  Example: IF (NVM_POLLING_MODE == TRUE) { ... }
- Variables must not be declared if their meaning and data type is clear from the context (e.g. loop counter variables, variables containing return values, etc.).
  Example: FOR n = 0 to NVM_NV_BLOCK_NUM { ... }
- Sub-functions should be defined for test steps that recur in Test Descriptions.
- Functions other than BSW API functions (e.g. sub-functions defined in Test Descriptions, target adapter functions) can be invoked by just writing down the

function name and possible parameters in parentheses (or "()" for no parameters)[9]. This is possible since these function invocations return immediately.

### 8.1.3 Keyword Descriptions

In general, each test step line begins with a keyword written in capital letters. The following sections give the meaning of the keywords that have been defined so far.

#### 8.1.3.1 CALL

As opposed to invoking sub-functions or target adapter functions, BSW module API function always invoked with the `CALL` keyword. They must be `CALL`ed since they are potentially asynchronous (i.e. do not return immediately). The API function name to be invoked together with its parameters must be included after the `CALL` keyword.

Example: CALL NvM_ReadBlock(BlockId, DataBufPtr);

Since all API functions are defined in TTCN-3 as "blocking" functions, they always result in a return event. This return event has to be `EXPECT`ed in subsequent test steps to account for the asynchronous execution behavior.

#### 8.1.3.2 EXPECT

The `EXPECT` keyword indicates that the test case is expecting an event coming from the SUT. Basically, there are two kind of possible events:

- Incoming function invocation.
  Example: EXPECT NvM_JobEndNotification();
- Incoming return from a previous function call. In this case, the extension "`return of`" must be used.
  Example: EXPECT return of NvM_ReadBlock();

#### 8.1.3.3 REPLY

For incoming function calls from the SUT, the test case must indicate the (possibly simulated) function completion together with an optional return value to the SUT. This is done using the `REPLY` keyword extended by "on", the replied function's name and optionally the return value preceded by a "`with return value of`".

Example:

```
REPLY on Ea_GetJobResult() with return value of MEMIF_JOB_OK;
```

#### 8.1.3.4 CHECK

The CHECK keyword indicates checks that are relevant to the test verdict . The description of what is to be checked can be informal and/or an algebraic expression. If the check passes, the test verdict is set to "passed". Otherwise, it is set to "failed".

Example: CHECK that rv is 0;

[9] This is identical to the C syntax for function calls.

or CHECK(rv == 0);

#### 8.1.3.5 Structural Elements

Common structural programming elements such as

IF ...
FOR ... TO
WHILE ... DO
REPEAT ... UNTIL

can be used with conditions expressed informally and/or using algebraic expressions. Using structural programming elements similar to C is recommended.

### 8.1.4 Handling Pointers and Addresses

Pointers and addresses that are provided to API functions under test are handled transparently[10] within the test cases. In other words, the test case must provide real existing addresses to the API function that it calls. The target adapter offers functions to the test case that provide real existing addresses (e.g. for a free memory block to be used temporarily).
Test steps involving pointer handling and interactions with the target memory (see [4]) are part of the test case description and must either be stated explicitly (i.e. writing down these steps) or implicitly (i.e. providing a general description from which the implementer can derive the required steps).
Refer to Chapter 10.5 for more information on the memory handling concept.

### 8.1.5 Limitations

There is currently one type of test step that cannot be described directly with pseudo-code:

**Irrelevance in the order and/or number of observations**

Some test cases must specify that events observable at the SUT (e.g. incoming callbacks from the SUT) are expected to occur at arbitrary times and in arbitrary order (or, arbitrary to a certain degree). Although these kinds of test cases can be implemented in TTCN-3 without problem, the pseudo-code described here cannot express them directly. Additional informal descriptions (e.g. comments in the pseudo-code) must be added to make the correct way of implementing clear to the test case implementer.

[10] This means, the pointer variables used within TTCN-3 in association with the BSW module's API functions contain the actual address values used on the target system running the BSW module.

# 9 Further Details: Test Case Design

This chapter contains further information and recommendations on designing and implementing the conformance test cases.

## 9.1 Design Elements for TTCN-3 Test Cases

Conformance testing complex BSW modules efficiently and with good coverage requires an appropriate test case design. The following sections describe the different design elements which must be considered for TTCN-3 test cases.

### 9.1.1 Test Components

Within TTCN-3 test cases, *test components* execute the test steps. Since several test components can be created within a test case, test steps can be executed in parallel. BSW conformance testing uses this parallelism, e.g. for error reception through the DET and DEM interfaces.
Defining roles for test components makes the test cases more understandable and easier to re-use. There are three roles for test components:

- Test Case Controller (TCCO)
- Test Case Client (TCCL)
- Test Case Stub (TCST)

The following sections define these test component roles.

#### 9.1.1.1 Test Case Controller (TCCO)

The purpose of the TCCO is to control the execution of the test case. The TCCO creates further test components (i.e. the TCCL and any optional TCSTs) and stops them to end the test case gracefully. After the test case completes, the TCCO collects the (local) test verdicts from the other test components and summarizes them in the final test verdict[11].
The TCCO is the entry point to TTCN-3 test cases. Thus, it is implemented on the MTC (Main Test Component, see [1]). Note that the TCCO does not communicate with the SUT directly.

#### 9.1.1.2 Test Case Client (TCCL)

TCCLs actually implement a test case's purpose. That is, they send stimuli to the SUT and wait for responses from the SUT. They then validate the responses and finally calculate the (local) test verdict (pass, fail or inconclusive). Thus only TCCLs execute test steps involving interaction with the SUT.

[11] The rule for summarizing the test verdicts is defined in the TTCN-3 standard. Basically, the final test verdict can only be "pass" when all (local) test verdicts are "pass" as well.

TCCLs communicate with the SUT through defined SUT Adapter ports using procedure-based communication methods. They are implemented on a PTC (Parallel Test Component, see [1]) and are created and controlled by the TCCO.

#### 9.1.1.3 Test Case Stub (TCST)

A TCST is needed if the SUT requires some external functionality. The TCST provides this functionality to the degree necessary to run the test case successfully. Thus, the TCST is in general a limited simulation of an external BSW module.
Typically, the TCST provides services to the SUT by waiting for incoming SUT requests and providing appropriate responses. The TCST makes its own (local) test verdict based on the interactions with the SUT and the test purpose.
Test functions implemented by a TCST are, for example:

- Error handling according to the Development Error Tracer (DET)
- Event handling according to the Diagnostic Event Manager (DEM)
- CRC calculation

The TCCL is implemented on a PTC (Parallel Test Component, see [1]) and created and controlled by the TCCO, as required by the test purpose.

### 9.1.2 Ports

Ports must be defined for each test component so that the test component can interact with other test components or with the SUT.
In TTCN-3, ports can be defined as `message`, `procedure` or mixed ports (i.e. `message` and `procedure` at the same time). In general, AUTOSAR BSW module conformance testing only uses `procedure` ports.
Details and examples of defining ports can be found in [1].

### 9.1.3 Interfaces

`Procedure` ports convey test components' function calls and their parameters. The group of function calls that can be transmitted over a port is referred to as an *interface* or *Application Programming Interface* (API) in the case of a standardized interface.
The signatures of function calls that can be transmitted over ports must be defined resulting in interface definitions for conformance test cases. These signatures are either translated directly from the API definitions in the AUTOSAR SWS documents, or, in case of function calls among test components or towards the target adapter (see also [4] for target adapter functions), are specified in the CTSpec itself

## 9.2 Modeling with UML

The TTCN-3 test case design elements described in the previous sections can be represented very well with UML component diagrams. Therefore, the CTSpec design specification must use UML component and class diagrams. The CTSpec design should provide at least the following to improve understandability:

| | |
|---|---|
| **Generic Test Architecture View** | This view (Figure 10) shall contain all possible test component interactions with the BSW module under test (SUT) used in the test cases. It shall depict all test component and SUT ports and their interconnections. The interface definitions associated with these ports can be omitted. The structural information contained in this view is helpful when implementing the framework of the CTSpec design in TTCN-3 |
| **Test Case Client View** | This view (Figure 11) specifies all ports and their interface definitions from the Test Case Client point of view (i.e. the test component that executes the main test steps, see Chapter 9.1.1.2). , This view gives an overview on all possible interaction methods available for the main test steps of a test case. This is helpful when implementing the test steps. |
| **Test Parameters View** | The test parameters view contains all relevant test parameter information required during TTCN-3 implementation. This view (Figure 12) presents all AUTOSAR configuration and test specific parameters relevant to the CTSpec. Module-specific and entity-specific parameters shall be defined separately as attributes of “parameter classes” which have appropriate TTCN-3 data type names. Enumeration types shall be defined as an “enumeration class”. |

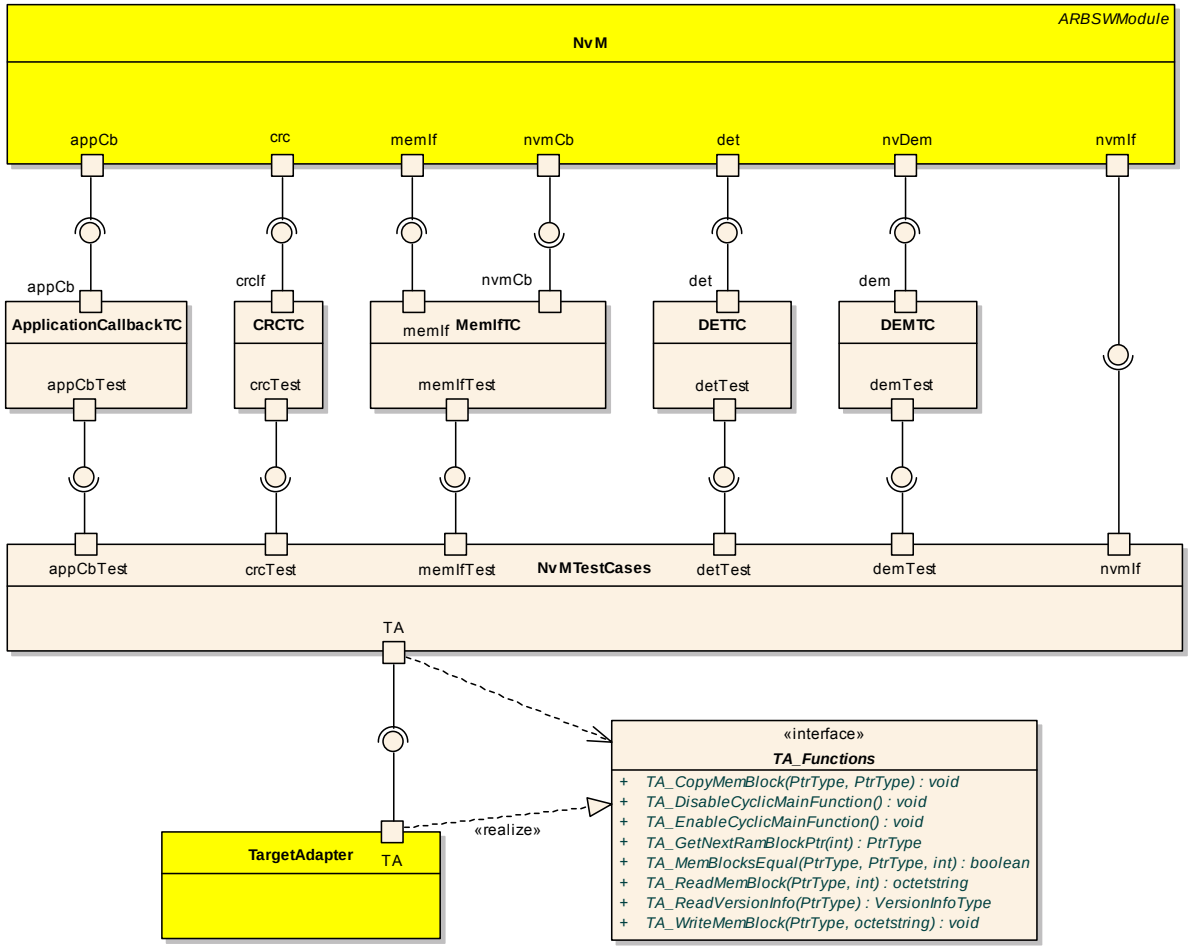


**Figure 10 - Example: Generic Test Architecture View on the CTSpec for the NVRAM Manager**

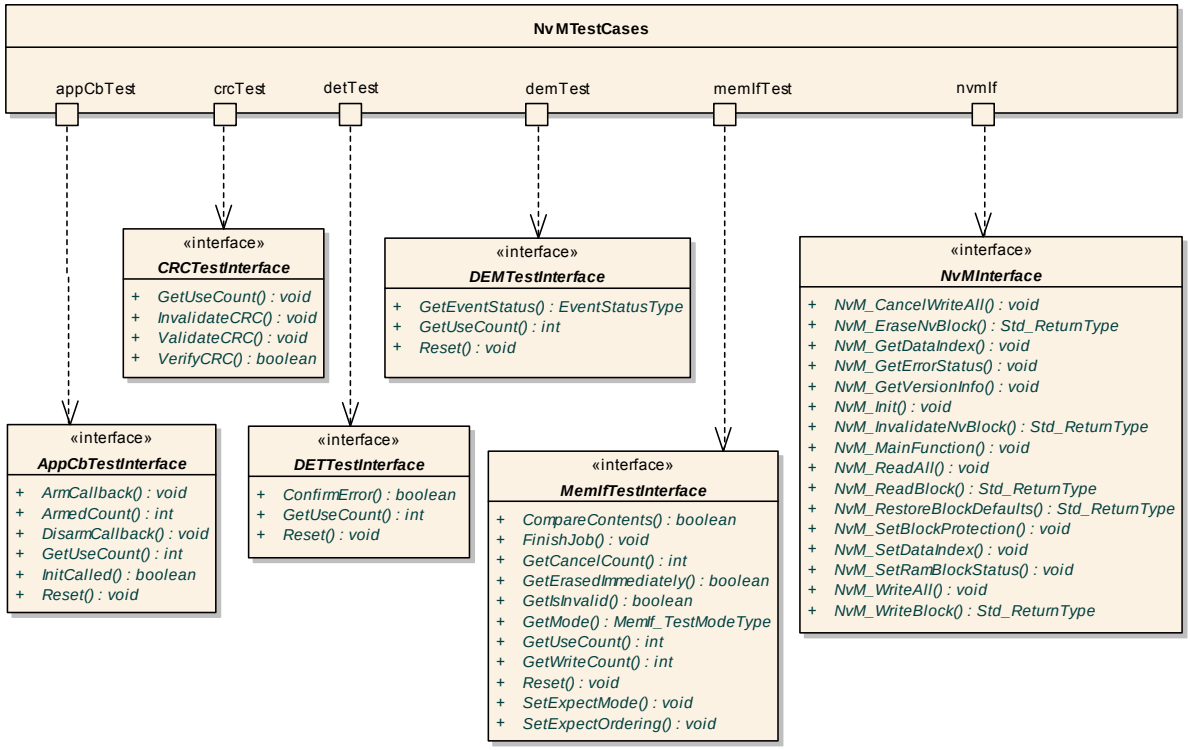


**Figure 11 - Example: Test Case Client View on the NVRAM Manager CTSpec architecture**

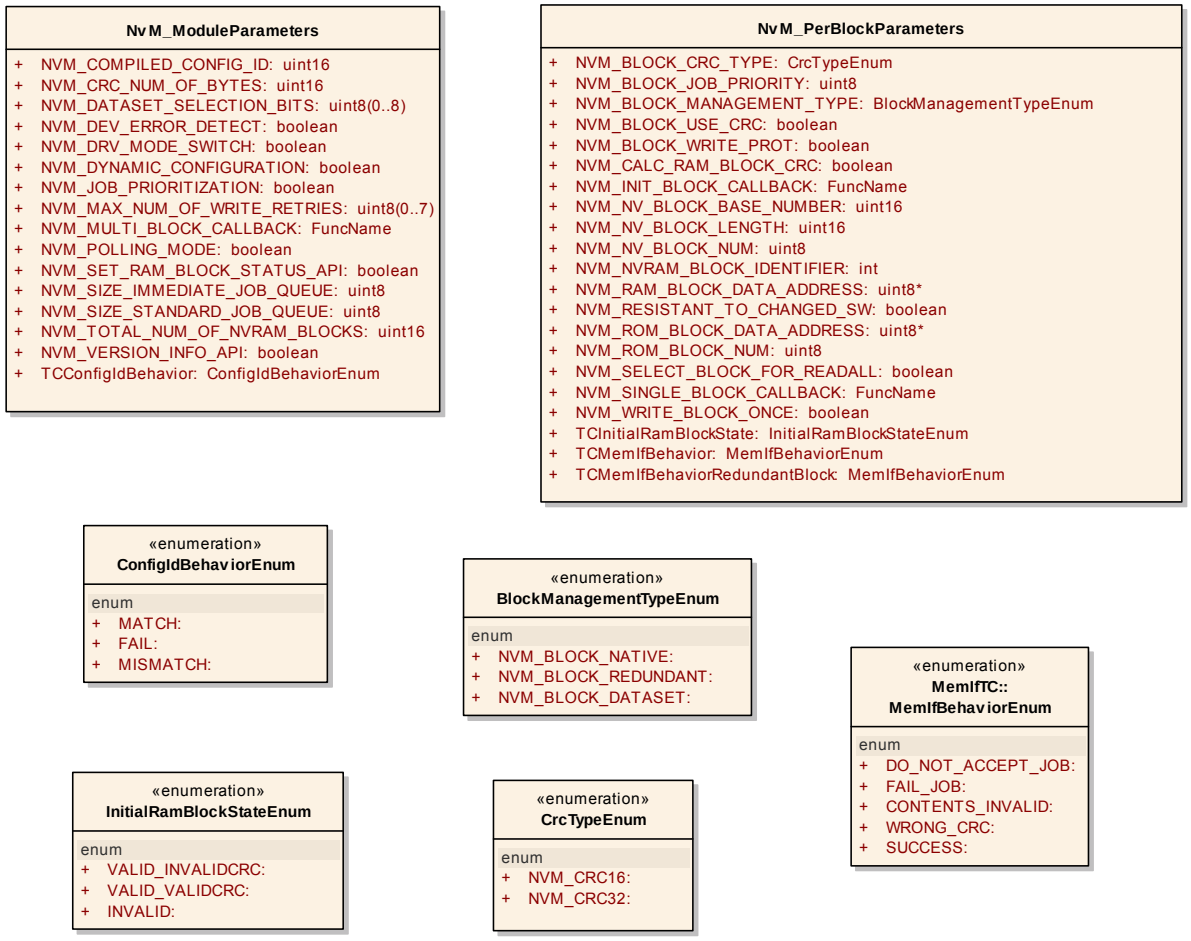


**Figure 12 - Example: Test Parameter View on the CTSpec architecture for the NVRAM Manager**

# 10 Further Details: Test Case Specification / Implementation

## 10.1 Configuration Mechanism

It is essential to BSW conformance testing that parameters (both AUTOSAR configuration parameters and test specific parameters) can be evaluated within test cases.
Since providing values for TTCN-3 module parameters is not standardized and has therefore been implemented differently by various TTCN-3 tools, the configuration mechanism used for BSW conformance testing defines the configuration parameter values directly as TTCN-3 code inside a parameter module. This parameter module is imported by the test case definitions.
The `*_test_parameter.ttcn3` TTCN-3 file contains the values of all configuration and test specific parameters contained in the conformance test cases. In general, this file is created by a tool.

### 10.1.1 Data types of Test Parameters

The rules for data type mapping described in Chapter 9 apply when converting AUTOSAR configuration parameter values into appropriate TTCN-3 definitions.

### 10.1.2 Format of Test Parameters

In general, several configuration sets with different combinations of test parameter values are needed for conformance tests. It is therefore crucial that the test parameters be in a form that is easily integrated with the rest of the CTSpec. All test parameters are defined within one TTCN-3 module. The module containing the test case specifications can then simply import the test parameter module to access the parameter values.
As described in Chapter 5.4.5.1, two kinds of configuration parameters exist for BSW modules: module-specific and “entity”-specific configuration parameters.
Since module-specific parameters exist only once per SUT, they can be defined as simple TTCN-3 constants.
Example:

```
const boolean NVM_POLLING_MODE := false;
```

There can be several entities in a SUT, however, and entity-specific parameters are defined per entity. Furthermore, it must be possible to iterate through these entities in order to find an entity with specific test parameter values.
Entity-specific parameters can be defined along with module-specific parameters within one module with an “accessor” function that returns the test parameters for an entity at a certain index.
Example:

```
function f_NvM_ReturnBlock(in integer BlockSelect)
runs on NvMPTC return NvMBlockParType
{
  var NvMBlockParType CfgElement;

  select(BlockSelect)
  {
    case(0)
      {
        CfgElement.NVM_WRITE_BLOCK_ONCE := true;
        CfgElement.NVM_BLOCK_MANAGEMENT_TYPE := e_REDUNDANT;
            :
      }
    case(1)
      {
        CfgElement.NVM_WRITE_BLOCK_ONCE := false;
        CfgElement.NVM_BLOCK_MANAGEMENT_TYPE := e_DATASET;
            :
      }
    case(2)
      :
  }
  return CfgElement;
}
```

## 10.2 Control Part

The control part executes all test cases belonging to a CTSpec and thus specifies the test case sequence. The test case itself evaluates whether a test case is actually valid for the currently active configuration set. Examples of control part code can be found in Part 1 “TTCN-3 Core Language” of [1].

## 10.3 Data Type Mapping

In [6], AUTOSAR has defined basic data types to be used within BSW module source code. These are then further mapped to C data types in order to account for particular target system properties (e.g. CPU type) and to optimize execution time.
Since conformance tests are independent of the target platform, only the basic AUTOSAR data types must be converted to TTCN-3 data types. The System and Platform Adapters must convert the TTCN-3 data types to the specific target platform data types (see [4]).
In addition to the mapping of basic AUTOSAR data types, the mapping of enumeration data types and data type compositions has to be defined.
C pointers are handled differently, however. All C pointer data types are mapped to the generic TTCN-3 `PtrType` data type. Refer to Section 10.5 for details.

### 10.3.1 Guideline

Generally the AUTOSAR BSW module data types should be mapped as directly as possible to the TTCN-3 data types used in the CTSpecs. The TTCN-3 data type definitions should also use the equivalent AUTOSAR data type names.

Applying this guideline to basic AUTOSAR data types (e.g. `uint8`) is generally straightforward as one mapping rule is usually obviously the most direct one.
Complex and composed C data types (e.g. enumerations, `struct` data types) can sometimes be mapped to TTCN-3 data types in different ways. Here, a specific mapping must be defined.

### 10.3.2 Mapping Rules

#### 10.3.2.1 Basic AUTOSAR Data Types

Table 4 maps the basic AUTOSAR data types (taken from [6]) to TTCN-3 data types:

| AUTOSAR data type | Corresponding TTCN-3 data type definition |
|---|---|
| `boolean` | `boolean` (basic TTCN-3 data type) |
| `uint8` | `type integer uint8 (0..255)` |
| `uint16` | `type integer uint16 (0..65535)` |
| `uint32` | `type integer uint32 (0.. 4294967295)` |
| `sint8` | `type integer sint8 (-128..127)` |
| `sint16` | `type integer sint16 (-32768..32767)` |
| `sint32` | `type integer sint32 (-2147483648..2147483647)` |
| `uint8_least` | not relevant |
| `uint16_least` | not relevant |
| `uint32_least` | not relevant |
| `sint8_least` | not relevant |
| `sint16_least` | not relevant |
| `sint32_least` | not relevant |
| `float32` | `type float float32 (-3.4E38 .. 3.4E38)` |
| `float64` | `type float float64 (-1.7976931348623157E308..1.7976931348623157E308)` |

**Table 4: Mapping between AUTOSAR data types and TTCN-3 data types**

Note that the optimized AUTOSAR integer data types (`*_least`) must not be used in BSW module APIs (see SWS item `PLATFORM032` in [6]). Therefore, they are not relevant to conformance test cases.
Mapping the basic AUTOSAR data types to TTCN-3 data types in this way makes it possible to use the same names for these data types in both the C source code and the TTCN-3 test cases.

#### 10.3.2.2 Indefinite AUTOSAR Data Types

Some AUTOSAR data types are indefinite due to unresolved dependencies, e.g. hardware dependencies: `Eep_AddressType` of type "`uint8 …uint32`". Three approaches are possible:

1. **Map the indefinite AUTOSAR data type to the most comprehensive data type possible (in the example above, map to `uint32`).**
   Adopting this approach shifts the issue of converting the comprehensive data type (e.g. uint32) to the currently valid data type (e.g. uint16) to the TTCN-3 Coder/Decoder (see [4]). The Coder/Decoder must be altered to the valid data type and this approach shall therefore be avoided.
2. **If possible, map the indefinite AUTOSAR data type to a compatible TTCN-3 data type that leaves room for the indefiniteness (in the example above, map to `octetstring` of `length (1, 2, 4)`.**
   This approach is quite elegant since the different AUTOSAR data types can be mapped to one "flexible" TTCN-3 data type. The different AUTOSAR data types must be "compatible" with the "flexible" TTCN-3 data type (i.e. there is an unambiguous mapping) so that the Coder/Decoder can convert between the AUTOSAR data types and the TTCN-3 data type. However, the "flexible" TTCN-3 data type may not be "compatible" in all aspects. For example, the TTCN-3 "octetstring data type of length (1, 2, 4)" used to map the AUTOSAR "uint8, uint16, uint32" data types cannot be used for arithmetic calculations directly. In this case, type conversion routines must be implemented before doing arithmetic operations.
3. **Leave a placeholder for the mapping (e.g. "`type integer indefinite_integer`"). Later when the concrete data type for the indefinite AUTOSAR data type is known, replace the placeholder by the matching TTCN-3 data type according to Chapter 10.3.2.1.**
   This approach requires changing the CTSpec and shall thus be avoided if possible. However, when the second approach is not possible, only this approach remains.

The approach should be chosen based on the recommendations given above.

#### 10.3.2.3 AUTOSAR Enumeration Data Types

AUTOSAR enumeration values are either implemented directly as C enumerations ("`enum`") or defined as constants ("`#define`"). In both cases, an integer value (unique within the enumeration type) is associated with the enumeration value. The integer value associated with the enumeration type in a SWS document is either stated explicitly or given by the enumeration sequence (counting starts at zero).
Due to this strong association between AUTOSAR enumeration value and integer value, AUTOSAR enumeration values are mapped to TTCN-3 integer constants which are grouped according to the AUTOSAR enumeration type[12].

[12] TTCN-3 offers an „enumerated" type of its own. However, it is not clear from the current TTCN-3 specifications (V3.1.1.) how the TTCN-3 Coder/Decoder handles the enumerated type and how the associated pre-defined integer values are to be taken into account by the Coder/Decoder. Therefore, the solution described here has been used.

Example:

AUTOSAR enumeration definition:

```
MemIf_StatusType:
    Type:   Enum
    Range:  MEMIF_UNINIT
            MEMIF_IDLE
            MEMIF_BUSY
            MEMIF_BUSY_INTERNAL
```

Equivalent TTCN-3 definition:

```
group g_MemIf_StatusType
{
    const EMemIf_StatusType e_MEMIF_UNINIT := 0;
    const EMemIf_StatusType e_MEMIF_IDLE := 1;
    const EMemIf_StatusType e_MEMIF_BUSY := 2;
    const EMemIf_StatusType e_MEMIF_BUSY_INTERNAL := 3;
}
```

## 10.4 Handling Open Implementations

Extensive analysis of four BSW Software Specification documents[13] during the CTSpec pilot project has shown that SWS requirements often leave the implementation of certain functionality open within certain limits.
From a conformance testing viewpoint, granting this degree of implementation freedom often dictates that the functionality be tested indirectly. , For example, the strategy (in terms of step size etc.) for realizing the multi-step CRC calculation in the NVRAM Manager is left to the implementer. This means that the classic black-box test approach – stimulation, observation, evaluation – with single events is not applicable. Multiple events (e.g. multiple partial CRC calculations) must be observed instead and their conformance with certain conditions (e.g. partial CRC calculations cover the whole memory block) must be evaluated. This shall be realized by Test Case Stubs in the test cases, i.e. independent test components that provide services to the Test Case Clients which execute the main test steps.

## 10.5 Pointer Handling

AUTOSAR BSW module APIs use pointers heavily to realize “`out`” or “`inout`” parameters i.e. the parameter passes data back (e.g. `GetVersionInfo` API calls). This is a “pass by reference” mechanism, as the parameters actually contain the addresses of the data. Basically there are two ways to handle this functionality in TTCN.

1. Abstract from pointers
   TTCN-3 supports the notion of "out" and "inout" parameters natively. Therefore, it does not require pointer mechanisms as in C to pass parameter data from the called function to the calling function. The "abstract from pointers" approach therefore defines the type of "out" or "inout" data directly within the TTCN-3 function signatures. In other words, a "pass-by-value" semantic is used in TTCN-3 functions that represent BSW module APIs.
2. Transparent pointer handling

[13] NVRAM Manager, Memory Abstraction Interface, EEPROM Abstraction Interface and EEPROM Driver

The alternative approach is to handle pointers transparently in TTCN-3; i.e. defining a TTCN-3 user type representing C pointers (a "pointer type"). This type is then used in TTCN-3 function signature representing BSW module APIs with pointer parameters. Calling these TTCN-3 functions then requires providing address values for these "pointer type" parameters. In this case, a "pass-by-reference" semantic is applied to the TTCN-3 functions.

The second approach, “Transparent Pointer Handling”, shall be used for the following reasons:

- Abstracting pointer parameters would require performing memory transactions and data type conversions on the SUT Adapter and Platform Adapter level with associated functionality in the Target Adapter. These operations would need to be implemented in these adapters making them more dependent on the BSW module under test, i.e. a greater part of these adapters would need to be implemented specifically for the BSW module under test.
- Handling pointers transparently in the CTSpecs reduces the System and Platform Adapters’ (including the Target Adapter) dependency on the BSW module’s APIs and they can be reused for other BSW modules with less modification. This approach puts additional effort in test case implementation, as the memory transaction and data type conversion operations involved with pointer parameters must be handled in the test cases. However, these operations can be defined generically (see [4]) and encapsulated in base functions (see Chapter 6.3). Therefore, the advantage of the System and Platform Adapters being more generic outweighs the additional effort in test case implementation.

Figure 13 depicts the pointer handling concept used in the TTCN-3 conformance testing. Each pointer parameter’s data type is mapped to the TTCN-3 `PtrType` data type. In this way, the pointer type information (e.g. whether it points to an `uint8` or an `uint16`) is lost in the TTCN-3 domain. However, the advantage of this generic `PtrType` is its simplicity – no pointer type conversions are required. When a test case evaluates data addressed by a pointer (e.g. version information), its structure must be known anyways. This also eliminates the need for typed pointers.

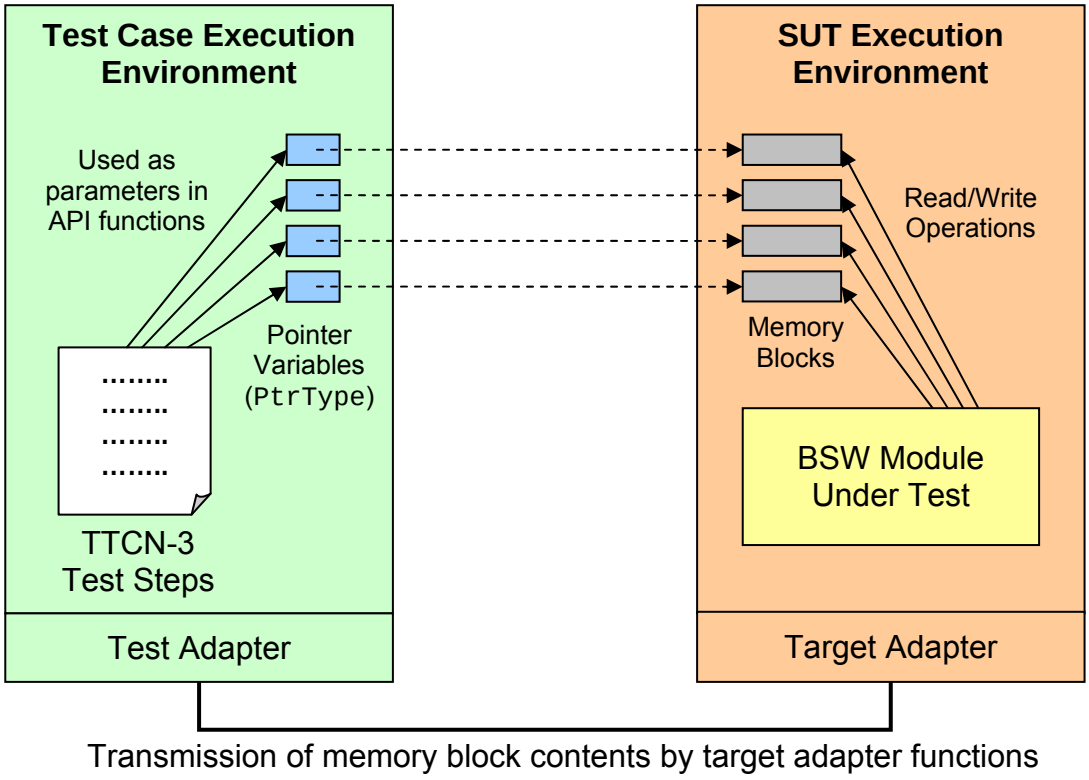


**Figure 13- Overview of the pointer concept in TTCN-3 conformance test cases**

[4] specifies the memory block operations needed to handle BSW module APIs with pointer parameters in detail. The realization of the pointer handling concept depends heavily on the memory concept used on the SUT side:

- For validation, a BSW module simulation is used as SUT. The pointer handling concept must be mapped to the simulation's memory handling concept.
  - For a JAVA simulation, a mapping between the memory addresses (i.e. content of TTCN-3 pointer variables) and the JAVA objects that represent memory blocks has to be implemented in the target adapter.
  - A simulation implemented in C can make use of direct references in the TTCN-3 pointer variables, i.e. the TTCN-3 pointer variables contain the addresses of the associated memory blocks in the simulation.
- A C implementation with real memory blocks is usually used for Class A and Class B test setups. Here, the TTCN-3 pointer variables refer directly to the associated memory blocks on the target system.
  - Configuration set constraints (e.g. "permanent RAM block" addresses in the NVRAM Manager configuration sets) may require additional mapping schemes between TTCN-3 pointer variables and the real target memory block addresses. These additional mapping schemes must then be implemented as module-specific target adapter functionality.

The Test Assessor must pay attention to the mapping between TTCN-3 pointer variables and the (real or simulated) target memory blocks when integrating the target adapter with the BSW module under test. If necessary, the target adapter has to be modified to fulfill the requirements of the SUT.

## 10.6 Error Condition Handling

During test case execution error conditions due to misbehavior or unexpected events occur primarily on the SUT side. While the BSW module under test generally reports error conditions to the test executable through DEM[14] and DET[15] API functions, error conditions in the target adapter must be reported explicitly to the test executable.
Since the target adapter mostly contains passive functionality that is triggered by the Test PC, TTCN-3 exception mechanisms are sufficient for reporting error conditions. The CTSpec shall define possible exceptions for each API and target adapter function with a common type (e.g. uint8 values for error codes).

Example:

```
signature NvM_Init ()
            exception (TA_ErrorType);
```

When it detects an internal error (e.g. out of memory), a target adapter should raise an exception with appropriate error code instead of replying to its invoking test executable. The test executable catches the exception, makes a test log entry and sets the test case verdict to “inconclusive” or “fail” as appropriate.

[14] “Diagnostic Event Manager“, a standardized AUTOSAR BSW module

[15] „Development Error Tracer“, a standardized AUTOSAR BSW module

# 11 Further Details: Illustration of System Dynamics

This chapter describes aspects of the TTCN-3 test case architecture, implementation and execution based on example test logs.

## 11.1 Defining the Main Function's Calling Mode

In general, a test case first defines the calling mode for the module's main function (automatically by the target adapter or controlled by the test case, Figure 14).

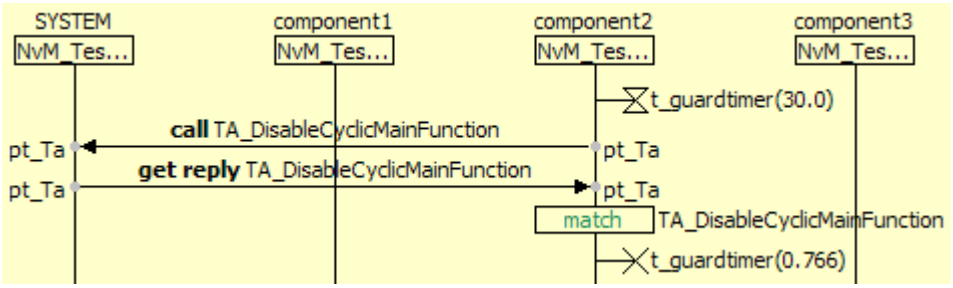


**Figure 14 - Specifying the calling mode for the BSW module's main function**

## 11.2 Initialization of the BSW Module under test

Most test cases require that BSW module under test be initialized (Figure 15). In these test cases, it must be ensured that the module's initialization function has been invoked (usually done in the basic "Init" test case).

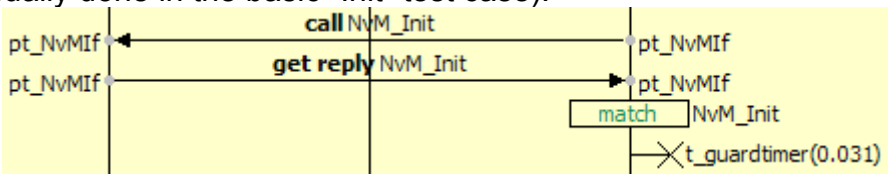


**Figure 15 - Initialization of the BSW module**

## 11.3 Allocation of Memory Blocks used by the Test Case

In case the test case uses memory blocks, it must allocate them using target adapter functions (Figure 16).

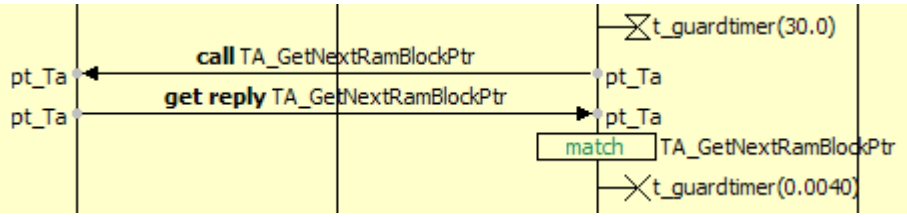


**Figure 16 - Allocation of a target memory block for use by the test case**

## 11.4 Interaction with Target Memory Blocks

After allocating memory blocks on the target, the test case can read from these blocks (Figure 17) and write to them (Figure 18).

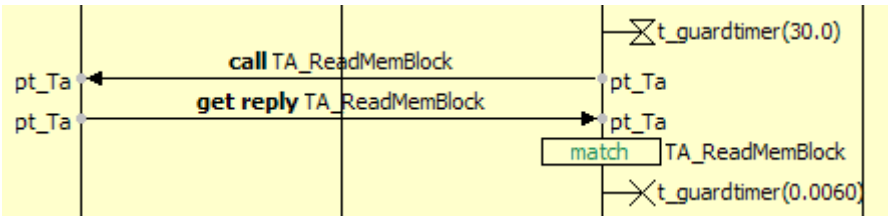


**Figure 17 - Read operation on a target memory block**

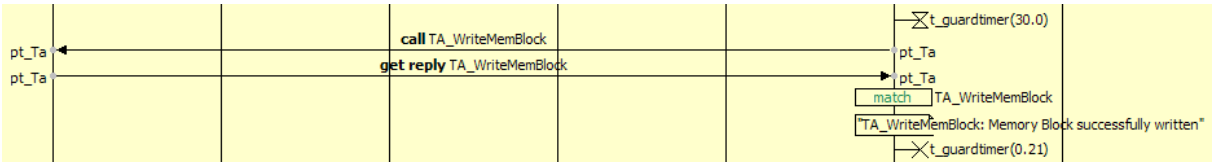


**Figure 18 - Write operation on a target memory block**

## 11.5 Invocation of API Functions

Test cases interact with the BSW module under test by invoking its API functions. In Figure 19, the API function "`NvM_GetErrorStatus()`" is called and returns an error status code in a memory block that has been allocated earlier. After the API function returns, the error status code is read out of the memory block.

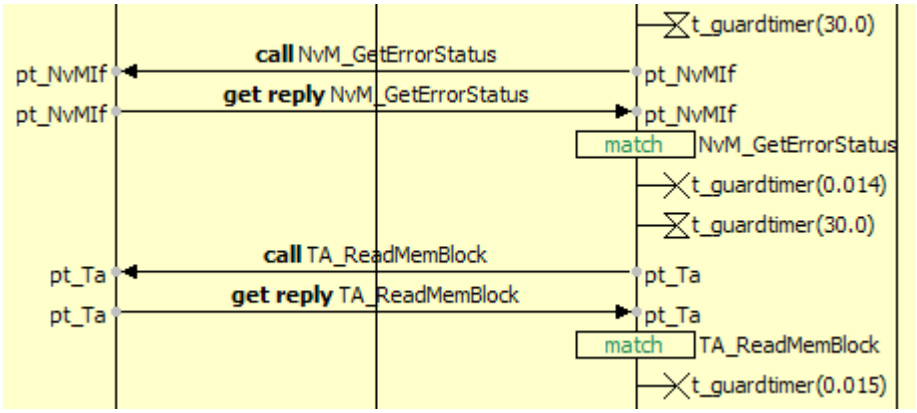


**Figure 19 - API function invocation and retrieval of the result from the target memory block**

API functions can, in general, be invoked in two directions:

- From test case to BSW module
- From BSW module to test case

The example illustrates both cases.
In Figure 20, the test case first invokes the main function which results in a number of calls to the lower layer API (here: `MemIf_GetStatus()`) and to the CRC module (here: `Crc_CalculateCRC32()`). These calls are received by the TTCN-3 test components in charge of these API functions. The main function returns after these activities have finished.

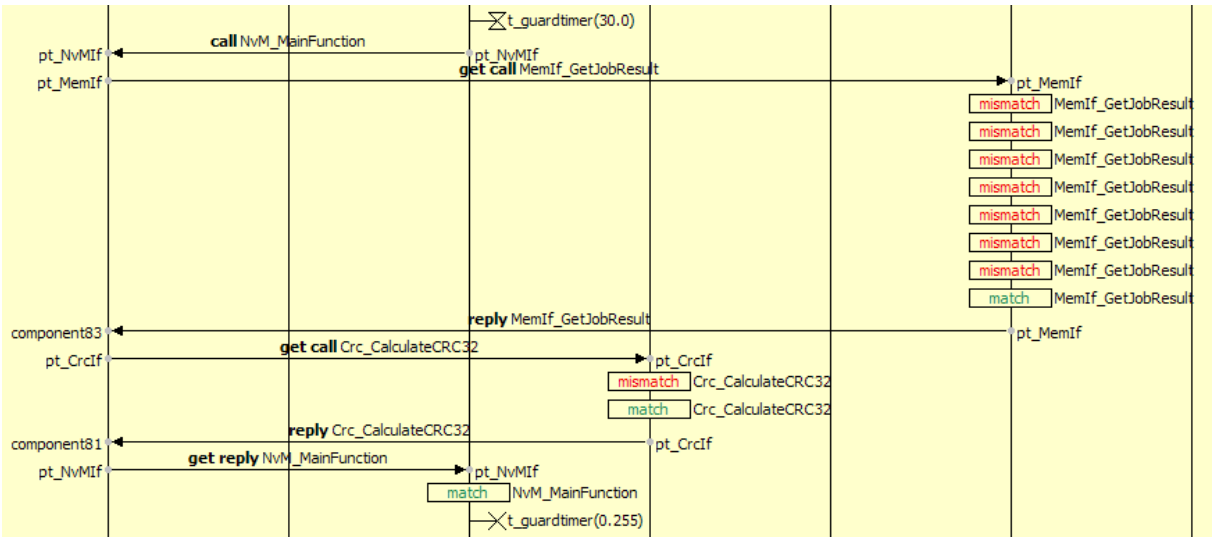


**Figure 20 - Invocation of the main function with subsequent BSW module activities**

## 11.6 Callbacks towards the BSW Module

From the test case execution viewpoint, callbacks are equivalent to API functions. They can be invoked in both directions.
In Figure 21, the test case invokes a callback at the BSW module.

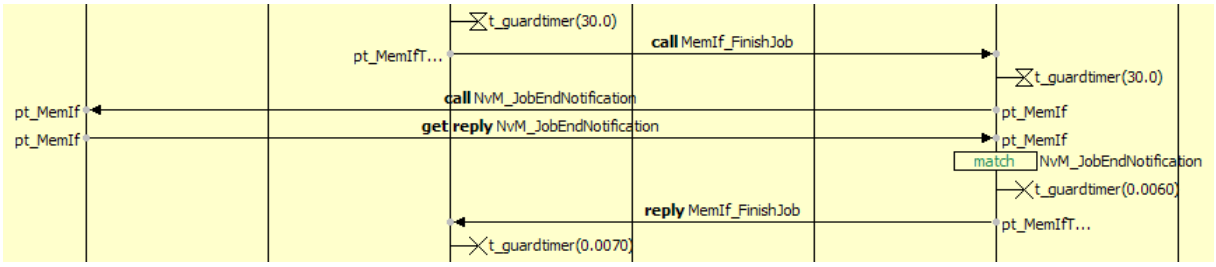


**Figure 21 - Callback invoked by the test case towards the BSW module**

## 11.7 Error reported to DET

During test case execution, the BSW module may detect development errors that must be reported to the DET module. As the DET is simulated by the test case, the DET error report is received by the TTCN-3 test component in charge of DET.
In Figure 22, the API function `NvM_SetDataIndex()` is called on a block for which this API function is actually not applicable. This results in a DET error.

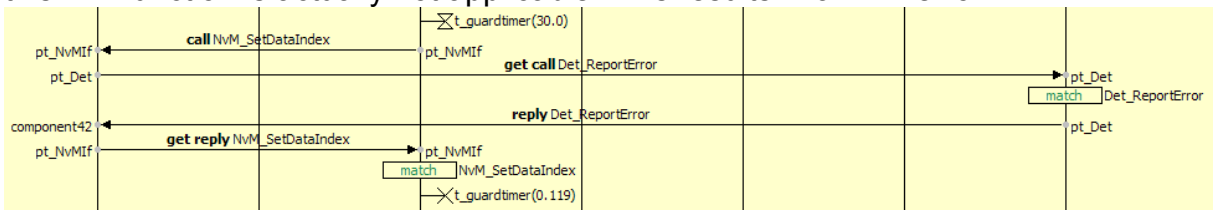


**Figure 22 - Report of an development error to the DET**

## 11.8 Error reported to DEM

The BSW module may also detect errors and other irregularities that must be reported to the DEM module. As the test case also simulates the DEM, the TTCN-3 test component in charge of DEM receives the DEM error report.
In Figure 23, a CRC calculation is initiated by the BSW module. The resulting CRC value is then compared with a CRC value inside a memory block. In this case, the CRC value of the memory block is wrong and a DEM error is reported.

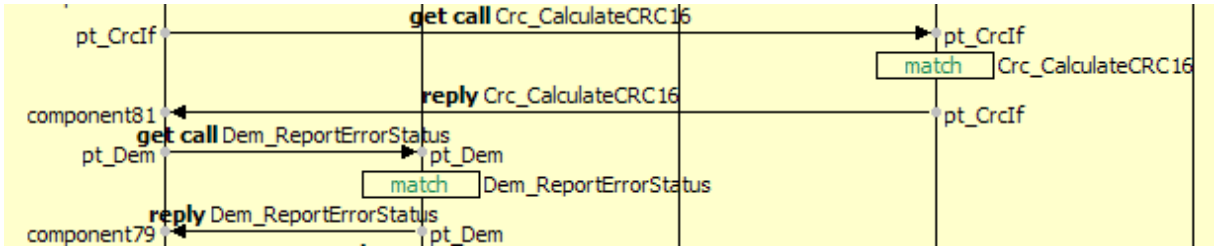


**Figure 23 - Report of an error condition to the DEM**

# 12 TTCN-3 Coding Style

A common, well defined style guide guarantees a common layout and a consistent and understandable structure for all parts of the test specification. This style guide is similar to other style guides used during software development.

## 12.1 Documentation

Every TTCN-3 test case shall be complemented by an explicit, natural language, test case description. This test case description shall contain a description of:

- The system under test (SUT)
- The SUT's interface:
  - Functions provided by the SUT
  - Valid ranges for the functions' input values
  - Functions required by the SUT
- The purpose of the test case
- Test case preconditions
- Test case post-conditions

Every TTCN-3 test case shall contain in-line documentation (comments within TTCN-3 code) comprising:

- A description of each implemented function, type, template etc, including:
  - Purpose
  - Input/output parameters and return values (if applicable)
- An in-line process description to explain the behavior.

## 12.2 Document Tagging

It shall be possible to automatically extract traceability information from the Test Case documentation defined above. Table 5 defines a system of tags that shall be used to mark the relevant information for later extraction in an appropriate form with a suitable tool.

| Tag | Delimiters | | R/O | Comment |
|---|---|---|---|---|
| | Start | End | | |
| Test Case ID | { | } | R | The unique ID for this test case. The actual form of the ID is left to the discretion of the test case specifier. The critical point is that it be unique across all test cases and all BSW modules.<br>This tag must always be first in the sequence |
| $SWS | : | $ | R | The SWS-item that the test case tests. There may be more than one. |
| Test Purpose | | | R | A short description of the purpose of the test. |
| Pre-condition | | | R | A description of the state in which the SUT must be put before the test can begin |

| Tag | Delimiters | | R/O | Comment |
|---|---|---|---|---|
| | Start | End | | |
| Post-condition | | | O | A description of the state in which the SUT will be after successful completion of the test. |
| Parameters | | | O | Any parameters (input or output) or return values that this test case requires. |
| Depends on | | | O | A comma-separated list of the test-case IDs of other tests upon which this test case depends. |
| Required by | | | | A comma-separated list of the test-case IDs of the other tests that depend upon this test case. |

**Table 5: Documentation Tags**

The following points apply to the information presented in Table 5:

- The tag text is not case sensitive.
- Spaces in a tag can be replaced with hyphens ('-') or underscores (' ').
- Not all tags are required, only those noted with an 'R' in the 'R/O' column in Table 5. Those marked with 'O' are optional.
- If no delimiter is specified, the tagged text begins with the first non-white-space character after the tag, and continues until the next blank line, the next end of (multi-line) comment, or the next tag.
- Where start and end delimiters are specified, the tagged text must reside on the same line as the tag. End-of-line ends the tagged text.
- Tags can appear in any order, with the exception of the Test-case ID, which must be first in any set of tags.
- A warning will be generated for a duplicated tag - the tagged text will be appended to whatever has been collected already.
- A warning will be generated for a required tag that is missing.

## 12.3 Naming Conventions

Identifiers shall be categorized using the prefixes specified in Table 6, which is a copy of the corresponding table (page 219) in "An Introduction to TTCN-3" [11]

| Prefix | TTCN-3 construct | Comment |
|---|---|---|
| (none) | Types | The type identifiers are written without prefix, instead the first letter is written in upper case. This is consistent with type definitions written in ASN.1, which can be imported directly to TTCN-3 |
| a | Templates | |
| alt | Altstep | The same prefix is used for `altsteps` independent of whether they are used as defaults or not. |
| c | constants | |

| Prefix | TTCN-3 construct | Comment |
|---|---|---|
| e | Enumeration elements | |
| f | Functions | |
| p | Parameters | |
| pt | Ports | |
| tc | Test cases | |
| t | Timers | |
| v | variables | |

**Table 6: Identifier Prefixes**

## 12.4 Restricted TTCN-3 Features

The “GOTO” expression cannot be used.

## 12.5 Value Ranges

Every test case description shall contain the valid parameter ranges for the tested functions.

## 12.6 Implementation Rules

The TTCN-3 implementation rules are based on the “AUTOSAR C Implementation Rules” [10] as the C-Language is very similar to TTCN-3. However there differences which mean that the “C Implementation Rules” must be adapted.
Each AUTOSAR C implementation rule has its own number. Table 7 details whether the rule has been adapted, rejected or accepted for TTCN coding.

| Reference | Rule Topic | Adaptation |
|---|---|---|
| [PROG 003] | File extensions | – not applicable – |
| [PROG 000] | English language | valid |
| [PROG 008] | Declarations per line | valid |
| [PROG 087] | Brackets in expressions | valid |
| [PROG 023] | No space before and after ‘.’ | valid |
| [PROG 024] | No space between operators and operand | – not applicable – |
| [PROG 025] | No space after unary operator | – not applicable – |
| [PROG 030] | Commenting functions | valid |
| [PROG 086] | Commenting violations of MISRA rules | valid |

| Reference | Rule Topic | Adaptation |
|---|---|---|
| [PROG 090] | Commenting violations of AUTOSAR C Implementation rules | – not applicable – |
| [PROG 034] | Double underscore | valid |
| [PROG 038] | Notation of macros | – not applicable – |
| [PROG 039] | Notation of function-like macros | – not applicable – |
| [PROG 042] | Form of #include statements | – not applicable – |
| [PROG 044] | Protection against multiple inclusion | – not applicable – |
| [PROG 048] | Inclusion of own header file | – not applicable – |
| [PROG 050] | Body inclusion | – not applicable – |
| [PROG 052] | Macros with several statements | – not applicable – |
| [PROG 055] | Declarations of global functions | – not applicable – |
| [PROG 063] | Declarations of global variables | – not applicable – |
| [PROG 061] | No function definition within header file | – not applicable – |
| [PROG 057] | No variable definition within header file | – not applicable – |
| [PROG 062] | Declaration of function parameters | valid |
| [PROG 056] | Declaration and definition of local functions | – not applicable – |
| [PROG 058] | Explicit definition of types | valid |
| [PROG 071] | Multiple Assignments | valid |
| [PROG 072] | Use of ’++’ and ’−−’ | – not applicable – |

**Table 7: Applicability of AUTOSAR C Implementation Rules**