VECTOR

vTESTstudio

Concept Manual

Version 10
English

Imprint

Vector Informatik GmbH
Ingersheimer Straße 24
D-70499 Stuttgart

Table of Contents

1 Introduction

In this chapter you find the following information:

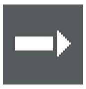
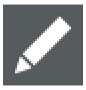
1.1 About this User Manual

1.1.1 Conventions

Conventions

In the two following charts you will find the conventions used in the user manual regarding utilized spellings and symbols.

Style	Utilization
bold	Blocks, surface elements, window- and dialog names of the software. Accentuation of warnings and advice. **[OK]** Push buttons in brackets **File|Save** Notation for menus and menu entries
vTESTstudio	Legally protected proper names and side notes.
`Source code`	File name and source code.
Hyperlink	Hyperlinks and references.
<CTRL>+<S>	Notation for shortcuts.

Symbol	Utilization
	Here you can obtain supplemental information.
	This symbol calls your attention to warnings.
	Here you can find additional information.
	Here is an example that has been prepared for you.
	Step-by-step instructions provide assistance at these points.
	Instructions on editing files are found at these points.
	This symbol warns you not to edit the specified file.

1.1.2 Certification

Certified Quality Management System

Vector Informatik GmbH has ISO 9001 certification.
The ISO standard is a globally recognized quality standard.

1.1.3 Warranty

Restriction of warranty

We reserve the right to modify the contents of the documentation or the software without notice. Vector disclaims all liabilities for the completeness or correctness of the contents and for damages which may result from the use of this documentation.

1.1.4 Support

You need support?

You can get through to our hotline at the phone number

+49 (711) 80670-200

or you send a problem report to the CANoe Support.

1.1.5 Registered Trademarks

Protected trademarks

All brand names in this documentation are either registered or non-registered trademarks of their respective owners.

2 Overview

This chapter contains the following information:

2.1 General

Overview

vTESTstudio is a multifaceted and integrated work environment for developing tests for embedded systems.

You can use different test design languages to write tests in vTESTstudio. The following editors are supported:

Test design editors

> CAPL Editor:
  CAPL is an event-oriented programming language of CANoe that can also be used for programming test sequences, test cases, and functions.

> C# Editor:
  C# is a .NET programming language that CANoe expands with libraries especially designed for testing and for accessing control units.

> Python Editor:
  Python is a high-level, general-purpose programming language. CANoe provides dedicated python classes for convenient test design.

> Test Table Editor:
  This editor enables writing of tests in tabular form and can be used without the need for programming expertise.

> Test Diagram Editor:
  Editor for modeling of test sequences in a graphic notation. This editor is contained in the **Graphical Test Design** product option.

> State Diagram Editor:
  Editor for modeling the expected behavior of the SUT as state model for automatic test case generation. This editor is contained in the **Graphical Test Design** product option.

User interface

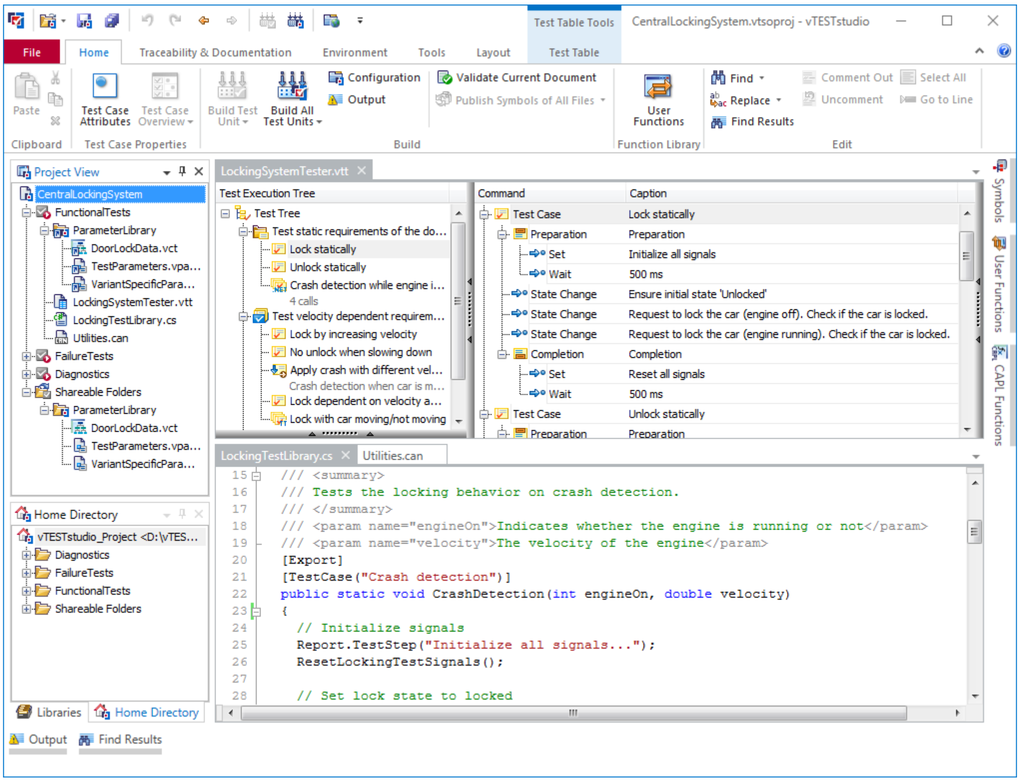


Figure 1: vTESTstudio user interface

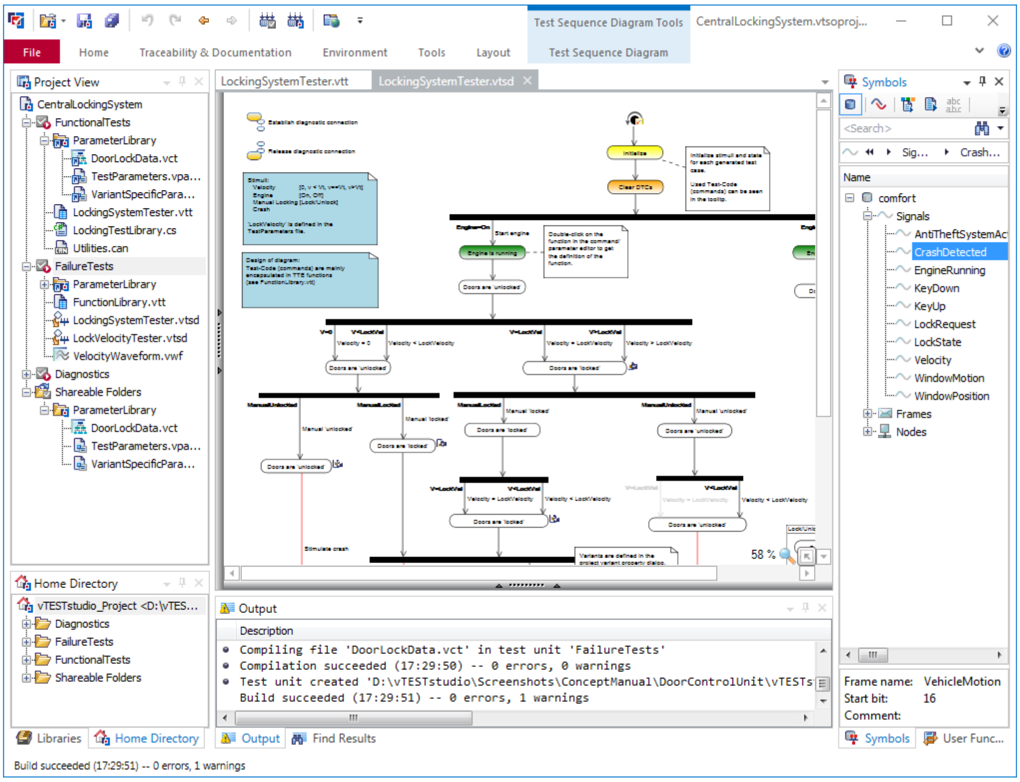


Figure 2: vTESTstudio user interface – Test Diagram Editor

2.2 Function Overview

Important features

Among the things that vTESTstudio provides for a successful test environment are:

> Integration of graphical test design editors, programming editors, parameter editors and curve editors in one tool
> Support of tests that contain several files for a structured test creation
> Comfortable capabilities for reviews by clear modeling of the test logic in the Test Diagram Editor or the expected behavior of the SUT in the State Diagram Editor
> Definition of stimulation curves with the Waveform Editor to stimulate the system under test (SUT) during the test.
> Comfortable administration of ECU configuration parameters and test vectors in separated files due to an integrated parameter concept
> Easy re-use of files as well as individual test cases in various tests through the use of libraries and a clear and easy to understand re-use concept, with separation of test logic, implementation, parameter values, and test vectors
> Possible pre-processing of events in CAPL and C# for access and evaluation in test sequences
> Continuous variant support for test logic, test implementation and parameter values allows an implementation of variant-dependent tests
> High test coverage through the use of test case lists.
> Universal traceability of externally defined requirements and test descriptions in the test implementation and in the test report
> Automatically generated test design documentation for documentation purposes as well as internal and external reviews

2.3 vTESTstudio and CANoe

Hand in hand

The interaction with CANoe enables easy access to the SUT in all languages. Database symbols, system variables, and diagnostic descriptions can be easily accessed in all languages. The bus systems and protocols CAN, LIN, FlexRay, Ethernet, WLAN and J1939 are supported.

Independent of their notation the tests are configured and loaded in CANoe, executed in real time, and documented in a detailed test report.

Schema

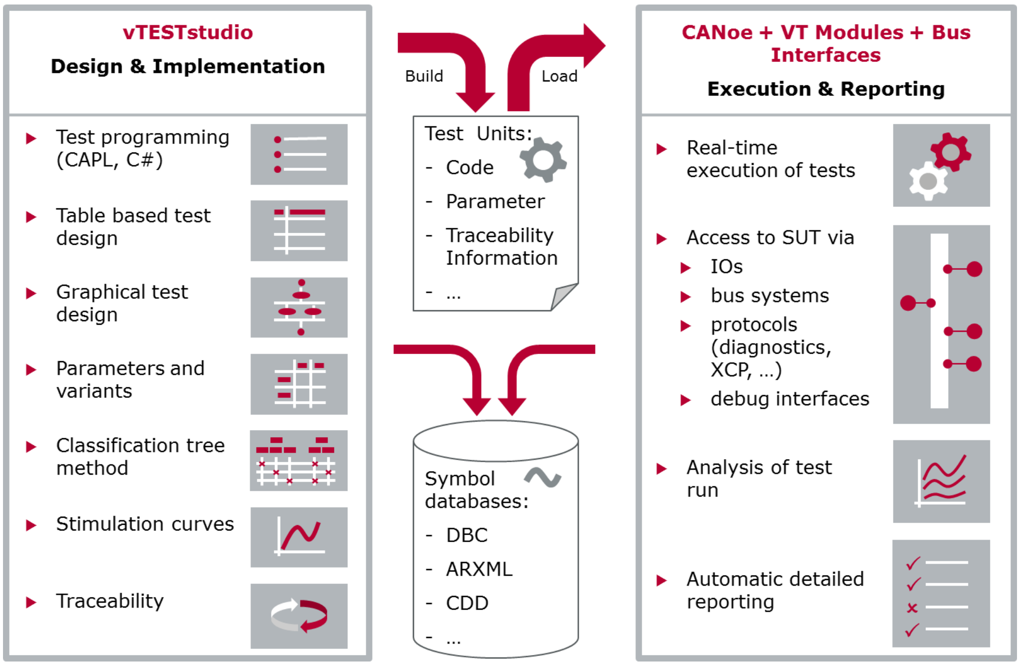


Figure 3: Schematic Overview – vTESTstudio and CANoe

2.4 Structure of a Project

2.4.1 Project Tree

Terms **project** and **test unit**

A vTESTstudio project consists of any number of **test units**. A test unit represents the configurable and executable unit in CANoe. It consists of a set of files that contain the test implementation (CAPL files, C# files, test tables, test diagrams and parameter files).

Examples

Example 1

> Project for tests of the overall functionality of an ECU
> Test units for core features, error detection, diagnostics, …

Example 2

> Project for tests of the entire product line of an ECU
> Test units for tests of a variant of an ECU

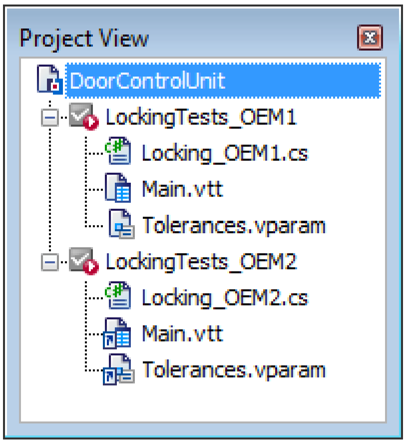


Figure 4: Project view

2.4.2 File-Based Storage

Storage of project components

All vTESTstudio project components are stored using **file-based** storing. A vTESTstudio project consists of a project file and any number of source code and parameter files that are referenced in the project file.

Files relevant for the test (CAPL files, C# files, test table files, parameter files, etc.) are created in the context of a test unit. Within a test unit, for example, test cases and test functions can be accessed across file- and language boundaries. Similarly, the parameters of a parameter file are known (only) in the context of the corresponding test unit.

2.4.3 Re-Use of Files

Linking of files

Files can easily be re-used by linking them to other test units. This is done by moving them from the source to the destination using a drag & drop operation. Thus, for example, the same test sequence can be re-used in multiple test units, e.g. with only using different parameter files for parameterizing the test sequence.

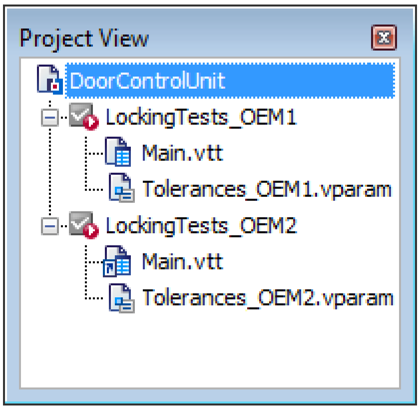


Figure 5: Project view with re-used file Main.vtt

2.4.4 Home Directory and Libraries

Home directory

The directory where the project file is located is called the **Home Directory**. Files from the root directory and files from specially defined **Libraries** can be used in the project.

Libraries

The Libraries view provides a filtered view of the file system. A library represents only a folder on the file system that is made known in the project. Files from libraries can also be linked to test units using a drag & drop operation.

If a library folder is moved on the file system, in vTESTstudio only the link to the library has to be updated. The paths of the linked files in the test units are updated automatically.

2.4.5 Storage Location of Files on the File System

Storage of files

If a file is created in the context of a test unit, it is automatically created on the file system under the path `<Home Directory>\<Test Unit Name>`.

If the files are to be organized differently on the file system, this can be done using the Home Directory view. This view shows the file system located underneath the project directory. If a file is moved in this view, possible links to this file in the test units will be automatically updated.

2.4.6 Structuring Using (Shareable) Folders

Re-use of folders

In order to structure the contents of test units more clearly, folders can be created within a test unit (as many folders, including nested folders, as required). This has no effect on the content of the test and is only used to organize files.

In addition to re-use individual files, the re-use of whole folders is also possible. For example, if a folder contains a test sequence description and an associated parameter file, these two files can be re-used together by linking the whole folder to another test unit.

If a folder with files is created underneath a test unit and then linked to another test unit, the folder with the original files is located on the file system underneath the first test unit. If such a "master test unit" is not desired, the folder can be created as a **shareable folder** instead. This is done by the project's shortcut menu command **New Shareable Folder**. Folders and files created by this are created on the file system under the path `<Home directory>\<Shareable Folders>`. From there they can also be linked to the test units using a drag & drop operation.

2.5 Concepts for High Test Coverage

Parameterized test case lists

For high test coverage without significant programming effort, so-called parameterized test case and test sequence lists are supported.

To define a test case list, a test case with input parameters must be defined, for example, as test case definition or in C#:

```
[Export][TestCase]
public static void CheckMotor(int temperature, double voltage)
{
  // ...
}
```

The test case list can then be defined in the Test Table Editor. The command selection contains a **CheckMotor [list]** entry for this purpose. If this entry is chosen, an editor appears that can be used to assign multiple values for each test case parameter:

Combinatorics:	Sequential	
Name:	**temperature**	**voltage**
Type of Values:	Single Values	Value Range
Value Source:	-	[min=1, max=11, inc=0.5]
Values:	-20	1
	-15	1.5
	0	2
	10	2.5
	40	3
	Add value...	3.5
		4
		4.5
		...
		11

Figure 6: Parameterized test case list in the Test Table Editor

In addition to the input of individual values or a range, the use of list parameters and struct list parameters from parameter files is supported as well (also see section Parameters):

Combinatorics:	Sequential	
Name:	**temperature**	**voltage**
Type of Values:	List from Parameter File	Value Range
Value Source:	TemperatureList	[min=1, max=11, inc=0.5]
Values:	-20	1
	-15	1.5
	0	2
	10	2.5
	40	3
		3.5
		4
		4.5
		...
		11

Figure 7: Using list parameters in test case lists

By a sequential combination, a number of test case calls are now generated automatically so that each parameter value is used at least once:

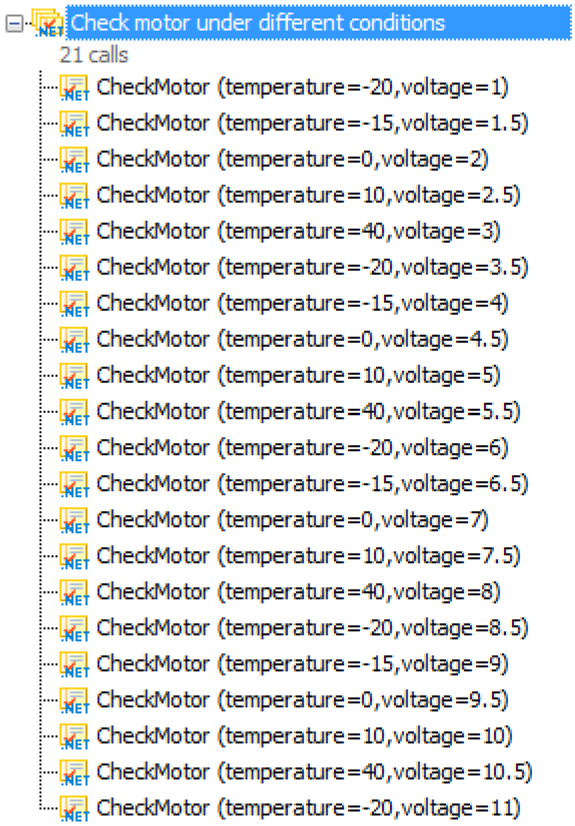


Figure 8: Generated test case list using **Sequential**

If a higher test coverage is to be achieved by testing pairwise or every combination of parameter values, the **Combinatorics** property must be changed from **Sequential** to **Pairwise** or **Combinatorial**:

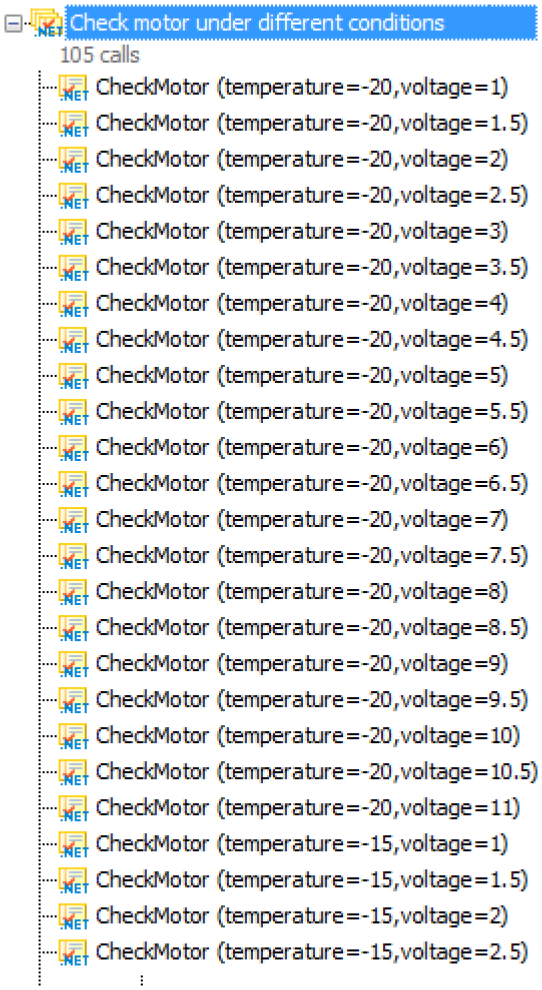


Figure 9: Generated test case list using **Combinatorial**

For the definition of a test case list in the Test Table Editor or in C#, all language combinations or crossovers listed in Language Interaction are supported.

2.6 Fuzz Testing

Concept

Fuzz test cases test the robustness of a software system (partly) automated with mainly invalid and/or unexpected input values. They provide a high level of test coverage and can be created in the test table.

The configuration parameters are set in the fuzz test generator, based on which test cases are automatically generated and executed at test execution time. The specific values of the signals are not yet known at the time of test design. The test values are rule and type-based, i.e. certain properties of the selected signals are used from the database (e.g. bit length or data type). This allows limit and special values to be determined for the signal.

Test values can also be generated randomly.

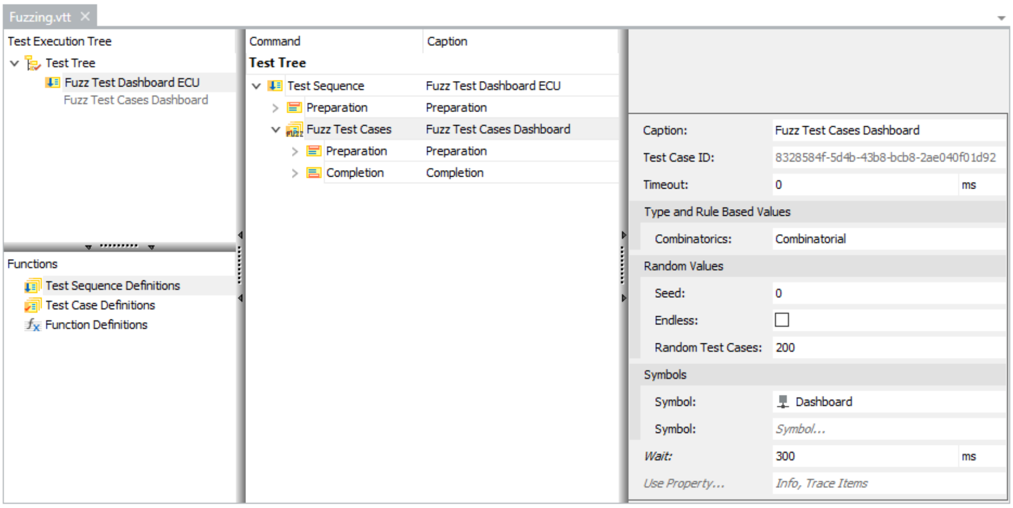


Figure 10: Fuzz Testing

2.7 Traceability by the Connection of REQM/TDM Tools

Concept

You can use vTESTstudio to trace externally defined requirements and test descriptions during test implementation and in the test report (traceability). This is done using exchange files in an open XML format. As a result, any REQM/TDM system can be used coupled with vTESTstudio.

Schema

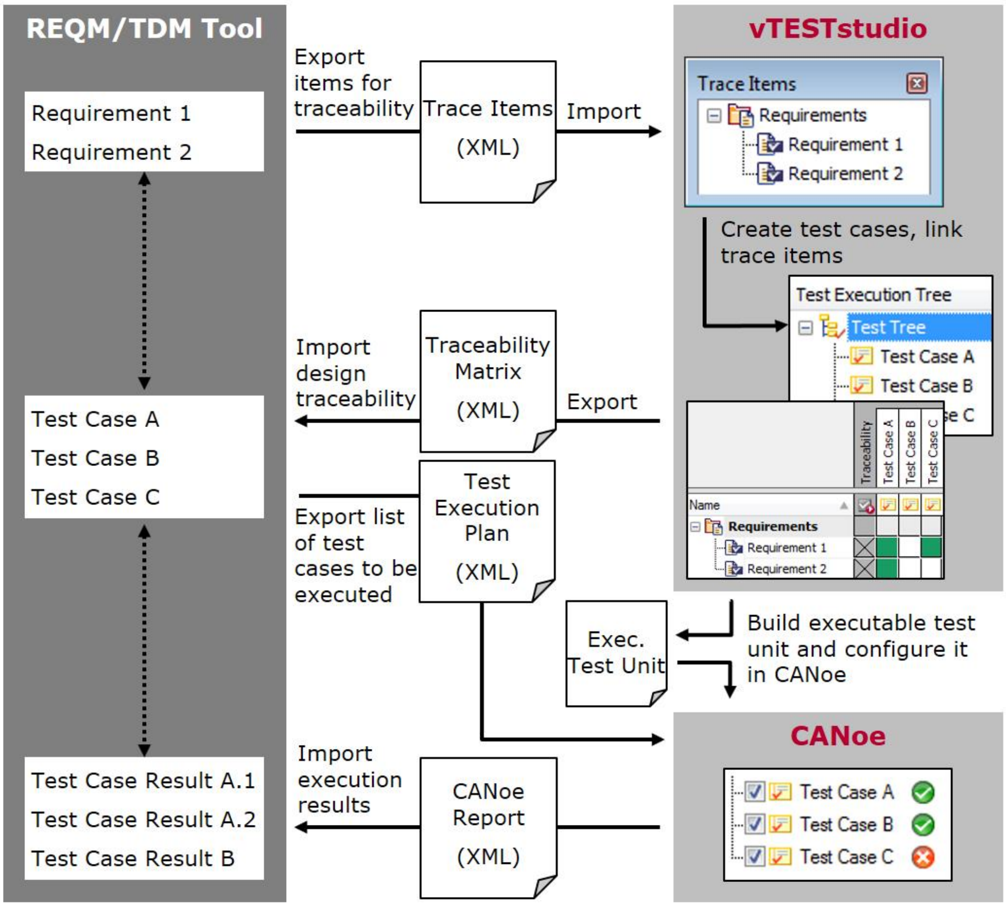


Figure 11: Traceability workflow

Import of requirements or test descriptions in vTESTstudio

The information exported from the REQM/TDM system is imported into vTESTstudio and can be linked there to test cases. The linked elements can be requirements or test descriptions. Because these elements are used to achieve traceability, they are designated as **trace items**.

Trace Item Explorer

All trace items imported into a vTESTstudio project are available via the Trace Item Explorer. From there, they can be easily linked to test cases using drag & drop.

In the REQM/TDM system, requirements and test descriptions might be structured in hierarchical folders or comparable elements. These folders are displayed in the Trace Item Explorer in the same hierarchy. However, folders cannot be directly linked to test cases.

Traceability Matrix

The **Traceability Matrix** in vTESTstudio gives an overview over all trace items of the project and their test case links. By this overview of test design coverage you can easily see, for example which trace items are not covered yet by any test case implementation. Furthermore, from within the traceability matrix comfortable navigation from a trace item to linked test cases is possible. For documentation purposes the traceability matrix can be exported to Excel.

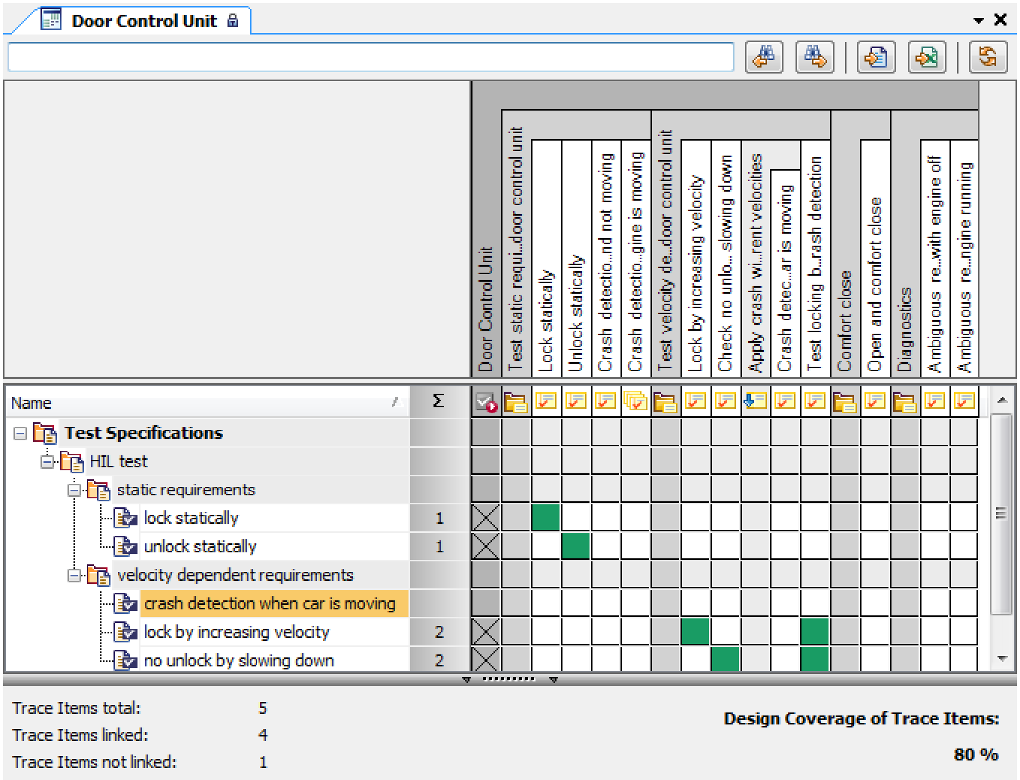


Figure 12: Traceability

Import of design traceability in REQM/TDM system

For synchronization with the REQM/TDM system an export of the traceability matrix to XML is supported. This enables importing the information about implemented test cases and linked requirements or test descriptions to the REQM/TDM system.

Test report

For each trace item linked to a test case, the test report contains a corresponding reference at the test case. This enables traceability from the test case result back to the requirement or test description.

Import of test results in REQM/TDM system

The test report can be played back to the REQM/TDM system to be able to analyze test results within this system.

Supported REQM/TDM systems

The connection of the following systems is supported out-of-the-box:

> Vector PREEvision TDM
> Squore / Software Analytics
> IBM Rational DOORS (classic)
> IBM Rational DOORS NG, IBM Rational Quality Manager / IBM Engineering Test Management
> Siemens Polarion ALM
> PTC Integrity Lifecycle Manager / Windchill
> Jama Connect
> Intland codeBeamer ALM

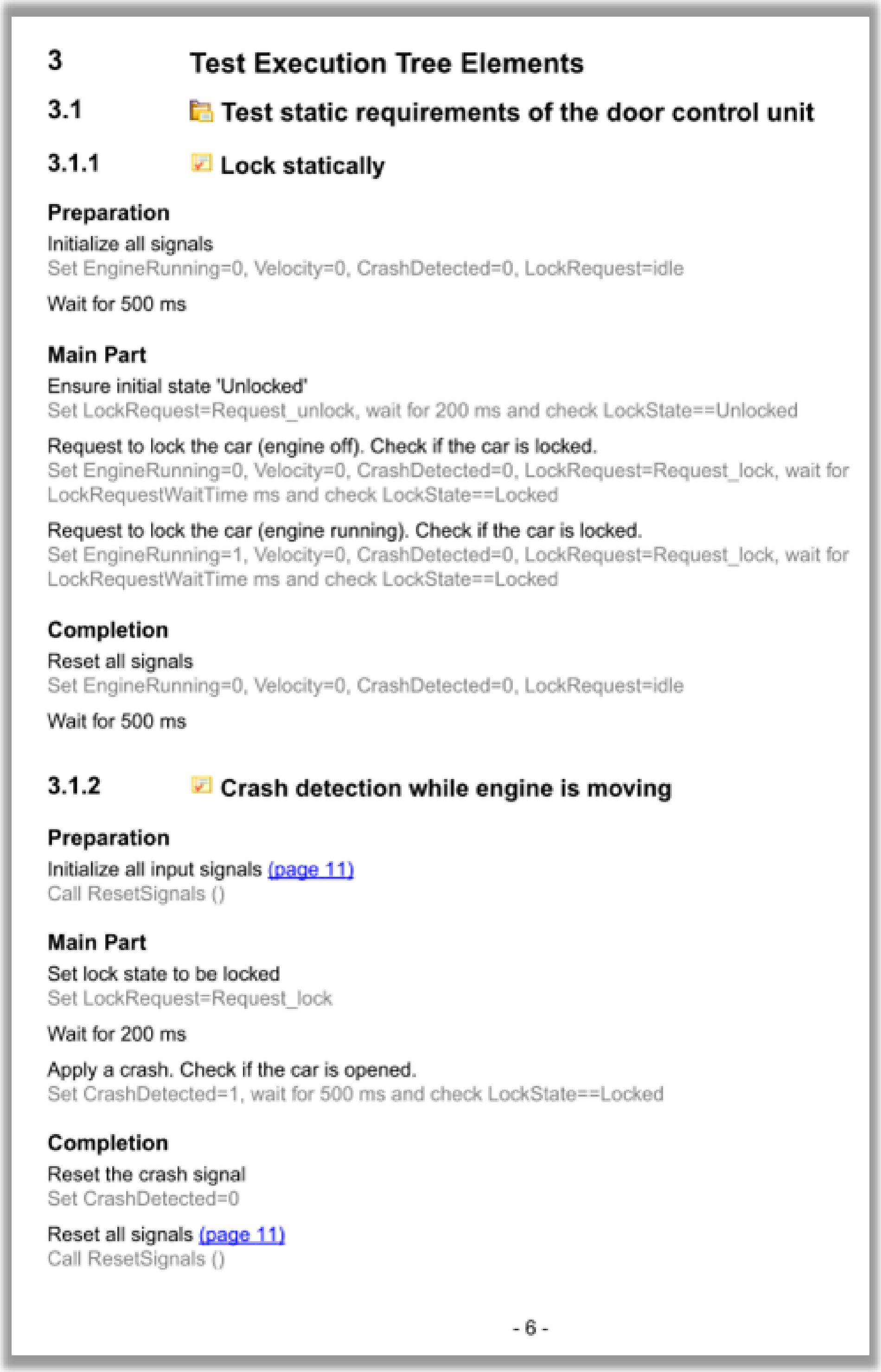
So-called “Connection Utilities” are available for free. By a command line interface, they can be used for continuous integration and continuous testing as well (e.g. with Jenkins).

Further REQM/TDM systems can be connected by the use of the open interfaces.

2.8 Test Design Documentation

Concept

For a test unit in vTESTstudio a test design documentation in PDF format can be generated automatically. This test design documentation provides an overview of all implemented test cases and test steps. Beyond the documentation it can be used for internal and external reviews of the test design.



3 Test Execution Tree Elements

3.1 Test static requirements of the door control unit

3.1.1 Lock statically

Preparation

Initialize all signals
Set EngineRunning=0, Velocity=0, CrashDetected=0, LockRequest=idle

Wait for 500 ms

Main Part

Ensure initial state 'Unlocked'
Set LockRequest=Request_unlock, wait for 200 ms and check LockState==Unlocked

Request to lock the car (engine off). Check if the car is locked.
Set EngineRunning=0, Velocity=0, CrashDetected=0, LockRequest=Request_lock, wait for LockRequestWaitTime ms and check LockState==Locked

Request to lock the car (engine running). Check if the car is locked.
Set EngineRunning=1, Velocity=0, CrashDetected=0, LockRequest=Request_lock, wait for LockRequestWaitTime ms and check LockState==Locked

Completion

Reset all signals
Set EngineRunning=0, Velocity=0, CrashDetected=0, LockRequest=idle

Wait for 500 ms

3.1.2 Crash detection while engine is moving

Preparation

Initialize all input signals (page 11)
Call ResetSignals ()

Main Part

Set lock state to be locked
Set LockRequest=Request_lock

Wait for 200 ms

Apply a crash. Check if the car is opened.
Set CrashDetected=1, wait for 500 ms and check LockState==Locked

Completion

Reset the crash signal
Set CrashDetected=0

Reset all signals (page 11)
Call ResetSignals ()

- 6 -

Figure 13: Automatically generated test design documentation as PDF

3 Test Design Editors

In this chapter you find the following information:

3.1 CAPL Editor

Features

The CAPL Editor integrated in vTESTstudio provides the functions of a modern development environment such as

> Code completion and syntax checking while writing
> Configurable syntax highlighting
> Syntax-sensitive indentation
> Expandable function blocks and function reference in a tree structure for faster navigation
> Hierarchical function list with search function for direct transfer to the source text
> Direct transfer of network symbols, environment data, diagnostic symbols, and parameters from the Symbol Explorer

User interface

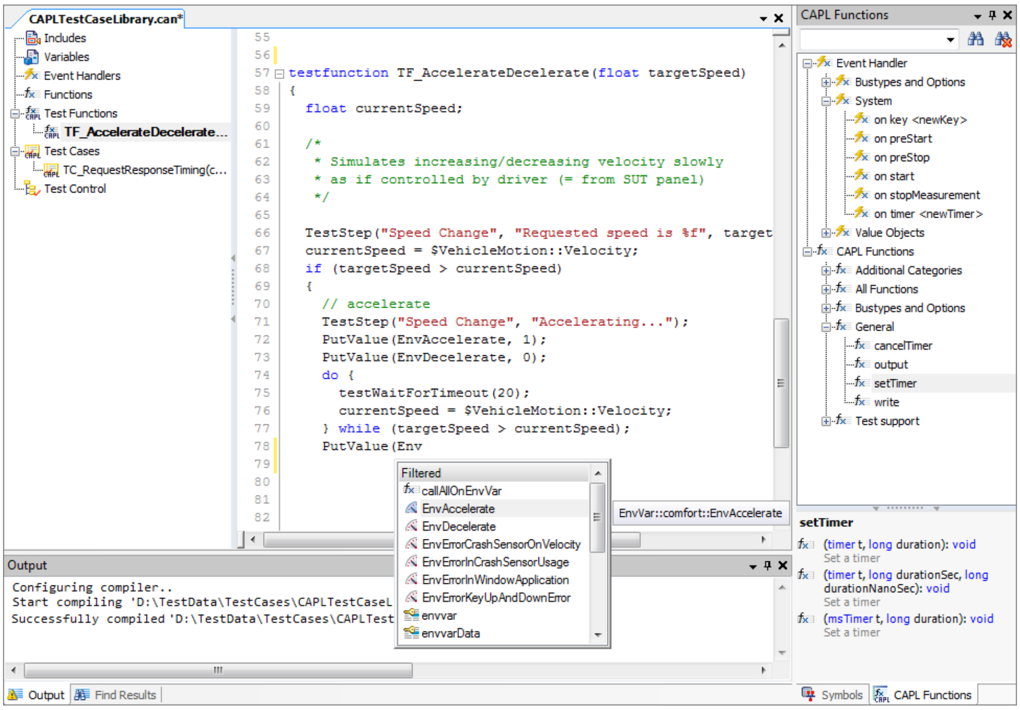


Figure 14: CAPL Editor

Access to symbols

Direct access to signals, frames and environment data is possible by the CAPL API. This enables e.g. read and write access to signal values in the CANoe runtime environment. Example:

```
$Velocity = 100;
```

Event procedures

In addition to the programming of sequential test sequences, test cases, and functions, the programming of event procedures is possible. In order to react to events (e.g., a signal change) in CANoe, event handlers can be defined. The methods are then called as soon as the event occurs during the test.

3.2 C# Editor

Features

Similar to the CAPL Editor, the C# Editor integrated in vTESTstudio provides the functions of a modern development environment such as

> Code completion and syntax checking while writing

> Configurable syntax highlighting

> Syntax-sensitive indentation

> Expandable function blocks

> Direct transfer of network symbols, environment data, diagnostic symbols, and parameters from the Symbol Explorer

User interface

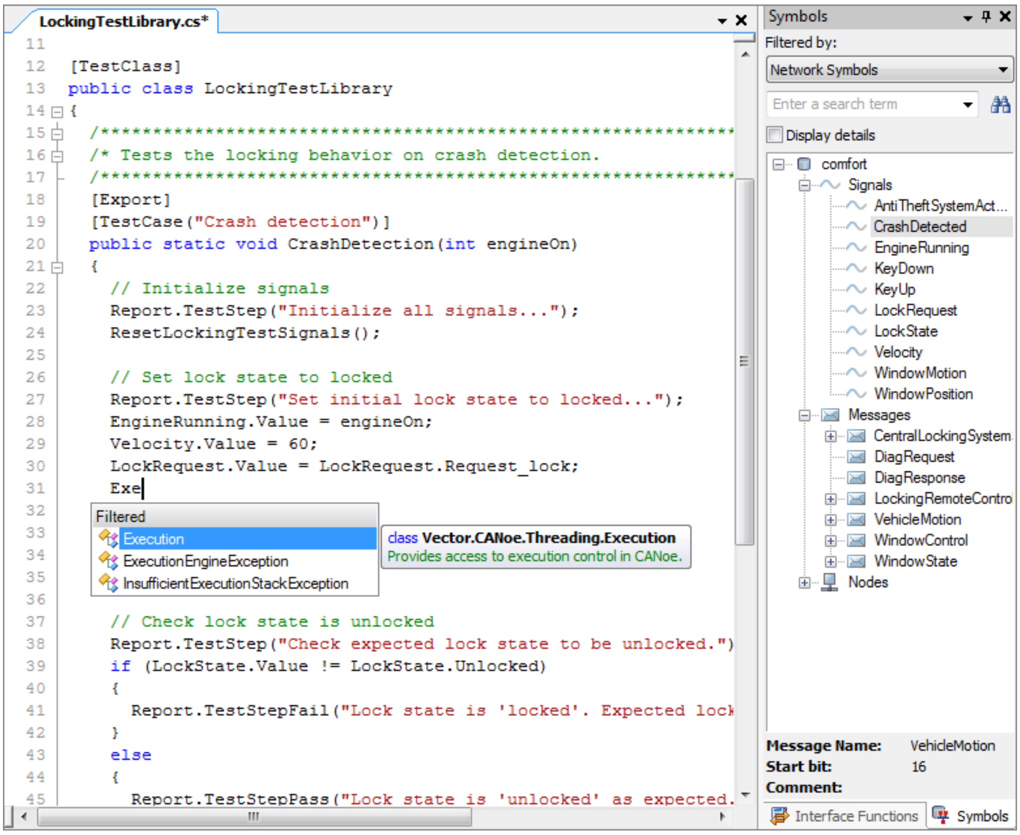


Figure 15: C# Editor

Access to symbols

Automatically generated .NET classes are available for accessing signals and environment data as well as CAN frames. These enable access to the values in the CANoe runtime environment. Example:

```
Velocity.Value = 100;
```

Event procedures

In addition to the programming of sequential test sequences, test cases, and functions, the programming of event procedures is also possible with the C# Editor. In order to react to events (e.g., a signal change) in CANoe, event handlers can be defined by the use of a special C# attribute. The methods are then called as soon as the event occurs during the test (exactly the same as in CAPL).

Connection to MS Visual Studio

By an integrated connection it's also possible to use Microsoft Visual Studio for the development of tests in C# as an alternative to the C# editor in vTESTstudio.

3.3 Python Editor

Features

Similar to the CAPL and C# Editor, the Python Editor integrated in vTESTstudio provides the functions of a modern development environment such as

- Code completion and syntax checking while writing
- Configurable syntax highlighting
- Syntax-sensitive indentation
- Expandable function blocks
- Direct transfer of distributed objects and parameters from the Symbol Explorer

User interface

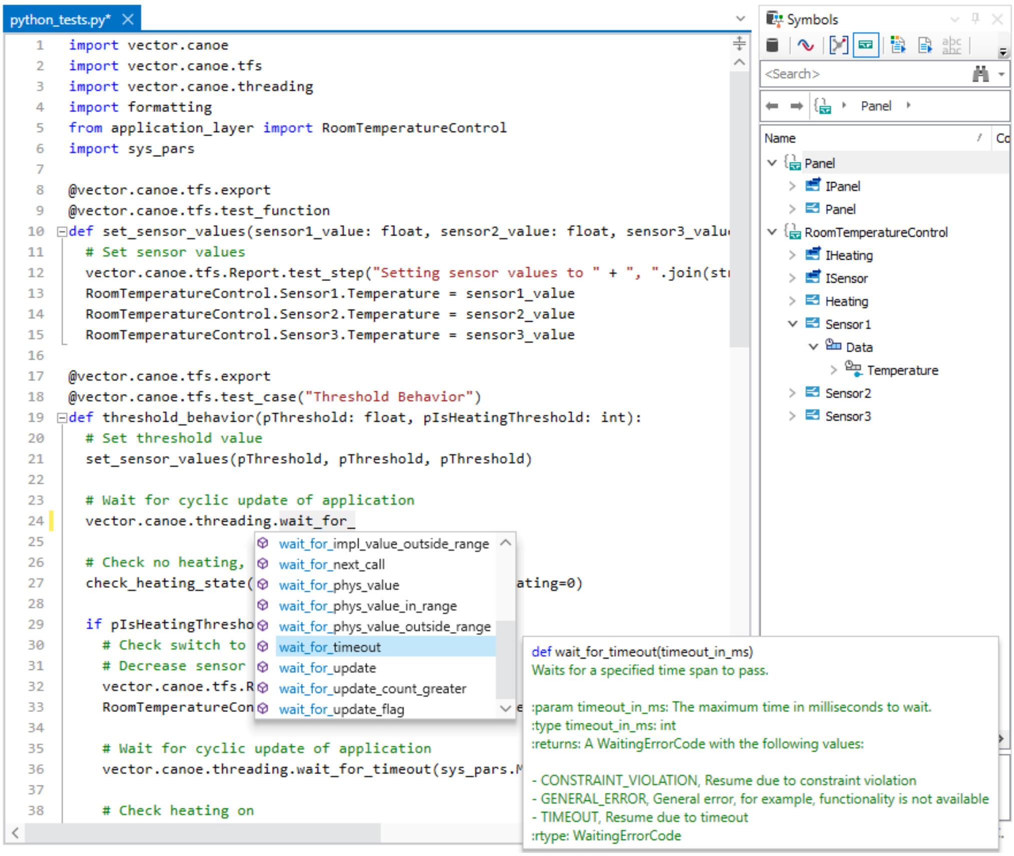


Figure 16: Python Editor

Access to symbols

Automatically generated Python classes are available for accessing distributed objects. These enable access to the values in the CANoe runtime environment. Example:

```
RoomTemperatureControl.Sensor1.Temperature = 30
```

Event procedures

In addition to the programming of sequential test sequences, test cases, and functions, the programming of event procedures is also possible with the Python Editor. In order to react to in CANoe, event handlers can be defined by the use of a special decorator. The methods are then called as soon as the event occurs during the test (exactly the same as in CAPL / C#).

3.4 Test Table Editor

Overview

The Test Table Editor allows users to easily define test sequences in tabular form without the need for programming expertise. Special commands are available for stimulating and testing the SUT.

Features

To make work easy, the following features are available:

> Command and symbol completion while writing
> Direct transfer of network symbols, environment data, diagnostic symbols, and parameters from the Symbol Explorer
> Symbol and value type checks while editing
> Access to user-defined functions and test cases in other languages
> Ease of operation using both the keyboard and the mouse

User interface

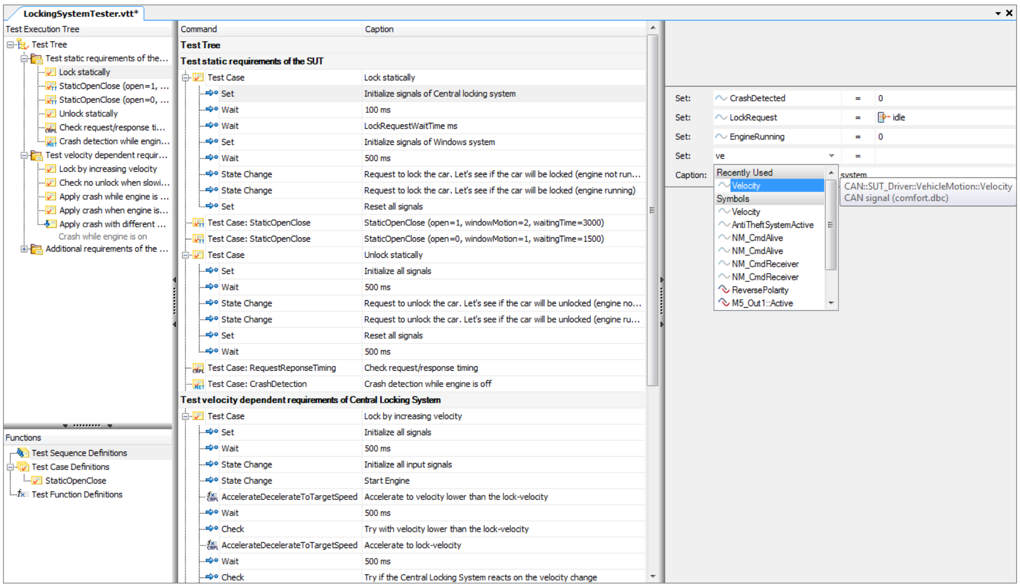


Figure 17: Test Table Editor

3.5 Test Diagram Editor

Overview

The Test Diagram Editor can be used to define tests in a graphical way. The graphical notation is particularly suitable for reviews. Test code in tabular notation is located behind each graphical element. By default, one test case is generated for each path through the diagram; a more fine-grained test case definition is possible as well.

Features

To make work easy, the following features are available:

> Easy configuration through the insertion of graphic elements into the graphic interface using a drag & drop operation
> Command and symbol completion while writing
> Direct transfer of network symbols, environment data, diagnostic symbols, and parameters from the Symbol Explorer
> Symbol and value type checks while editing
> Access to user-defined functions and test cases in other languages
> Usage and re-use of sub-diagrams possible
> Display and preview of generated test cases and their content in table form

User interface

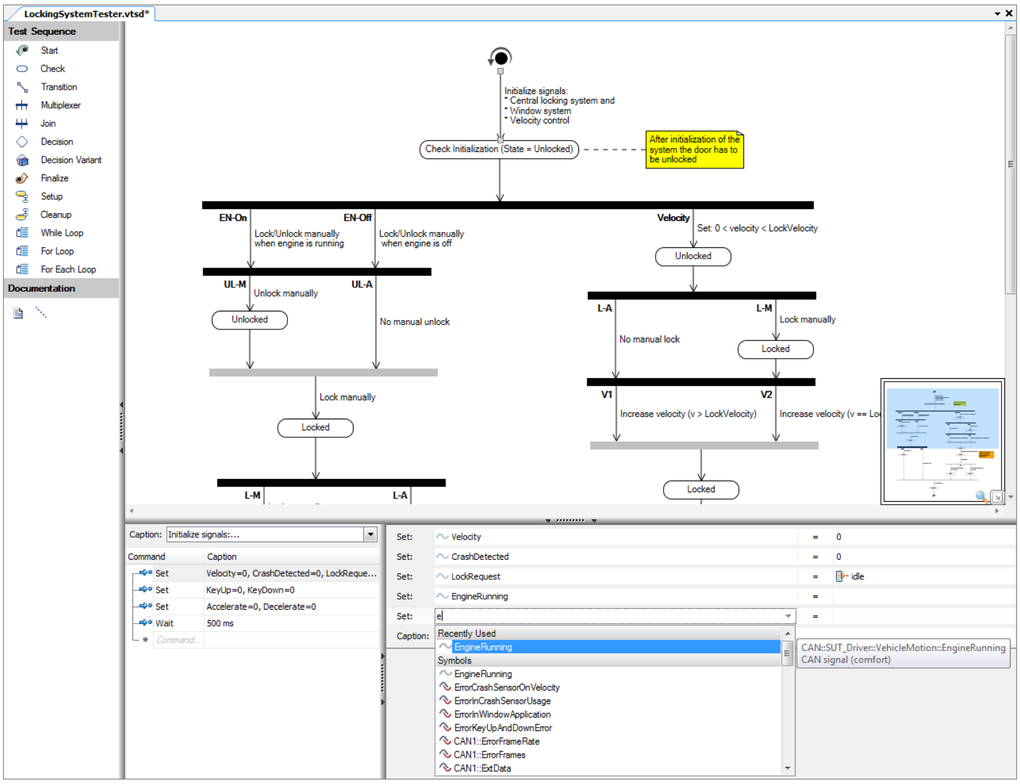


Figure 18: Test Diagram Editor

Note: The Test Diagram Editor is only available with the option **Graphical Test Design**.

3.6 State Diagram Editor

Overview

The State Diagram Editor can be used to model the expected behavior of the system under test as state model. Test code in tabular notation is located behind each graphical element. Based on transition coverage test cases are automatically generated out of the model. Different generation algorithms are supported, e.g. Chinese Postman algorithm and a breadth search based algorithm.

Features

To make work easy, the following features are available:

> Easy configuration through the insertion of graphic elements into the graphic interface using a drag & drop operation

> Command and symbol completion while writing

> Direct transfer of network symbols, environment data, diagnostic symbols, and parameters from the Symbol Explorer

> Symbol and value type checks while editing

> Access to user-defined functions and test cases in other languages

> Display and preview of generated test cases and their content in table form

User interface

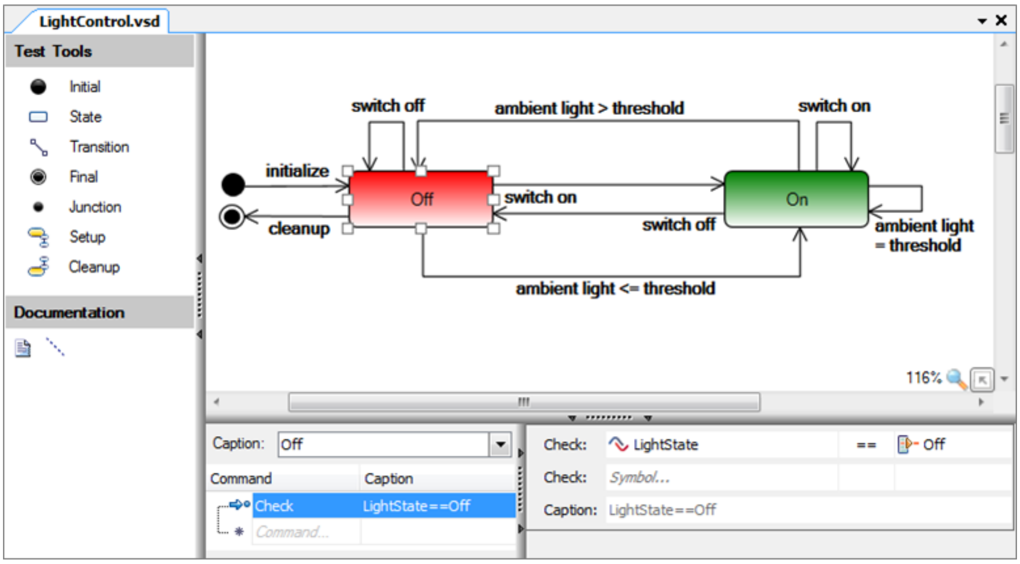


Figure 19: State Diagram Editor

Note: The State Diagram Editor is only available with the option **Graphical Test Design**.

4 Interaction with CANoe

In this chapter you find the following information:

4.1 CANoe System Environment

Access

The integration of description files for the system environment (databases, diagnostic descriptions…) enables easy direct access to symbols of the CANoe system environment. The database symbols, system variables, etc., contained in the system environment are available for use in the test coding. The symbols can be inserted into the test sequences in two different ways: via text completion in the individual editors or using a drag & drop operation from the Symbol Explorer.

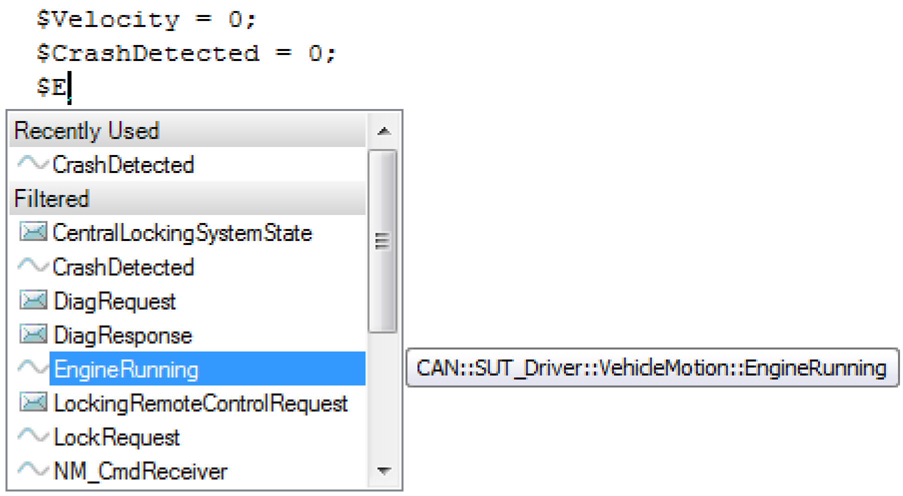


Figure 20: Symbol completion in the programming editors

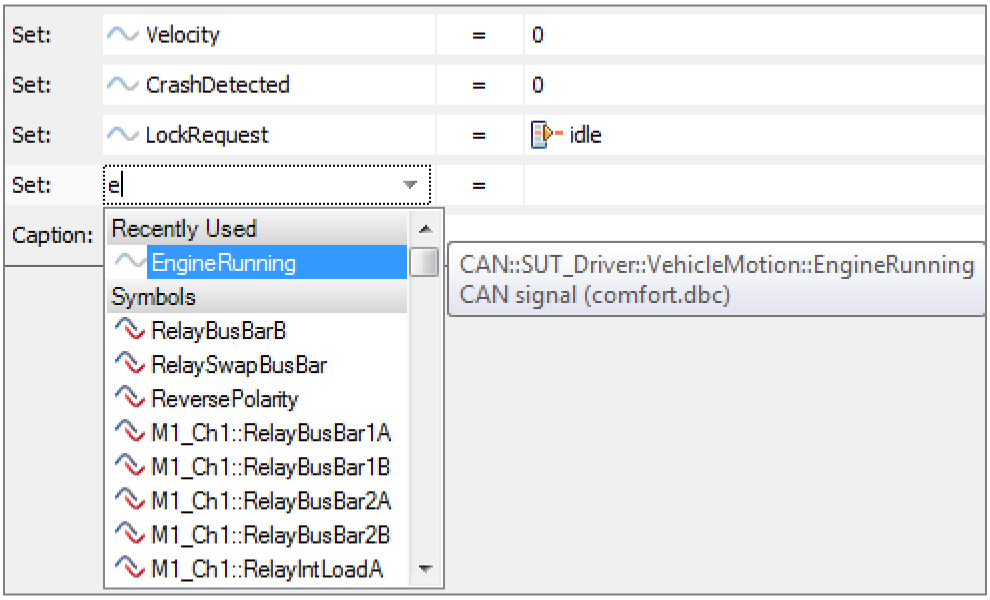


Figure 21: Symbol completion in the Test Table Editor

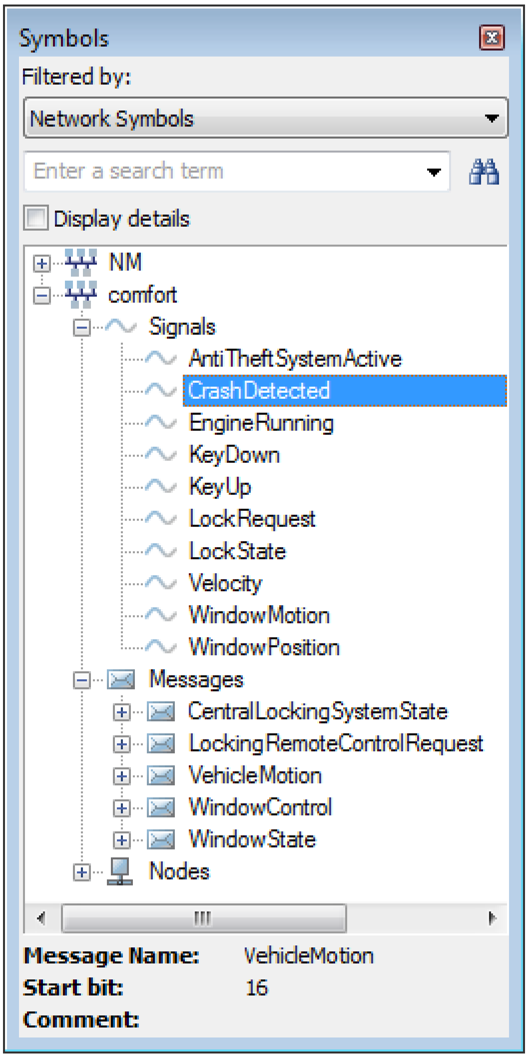


Figure 22: Symbol Explorer for inserting symbols using drag & drop

4.2 Execution in CANoe

Creating a test unit

In vTESTstudio an executable test unit can be created. An executable test unit has the extension *. VTUEXE and contains all necessary data for the test (source files, parameter files…).

Configuring and executing a test unit in CANoe

The executable test unit can be configured and executed in CANoe.

For the execution in CANoe in the CANoe configuration any number of test configurations can be created. A test configuration can again contain any number of executable test units that will be executed in sequential order.

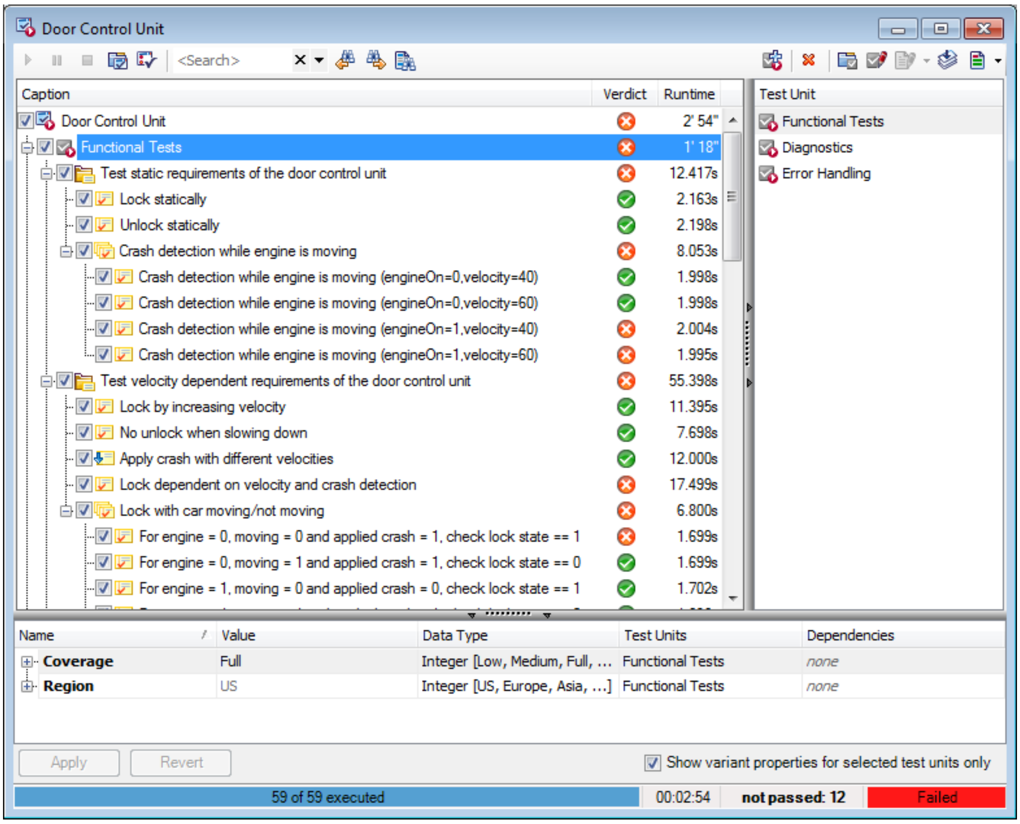


Figure 23: Test execution in CANoe

Test Trace Window

Details of the test run are visualized in the Test Trace Window already during the test execution.

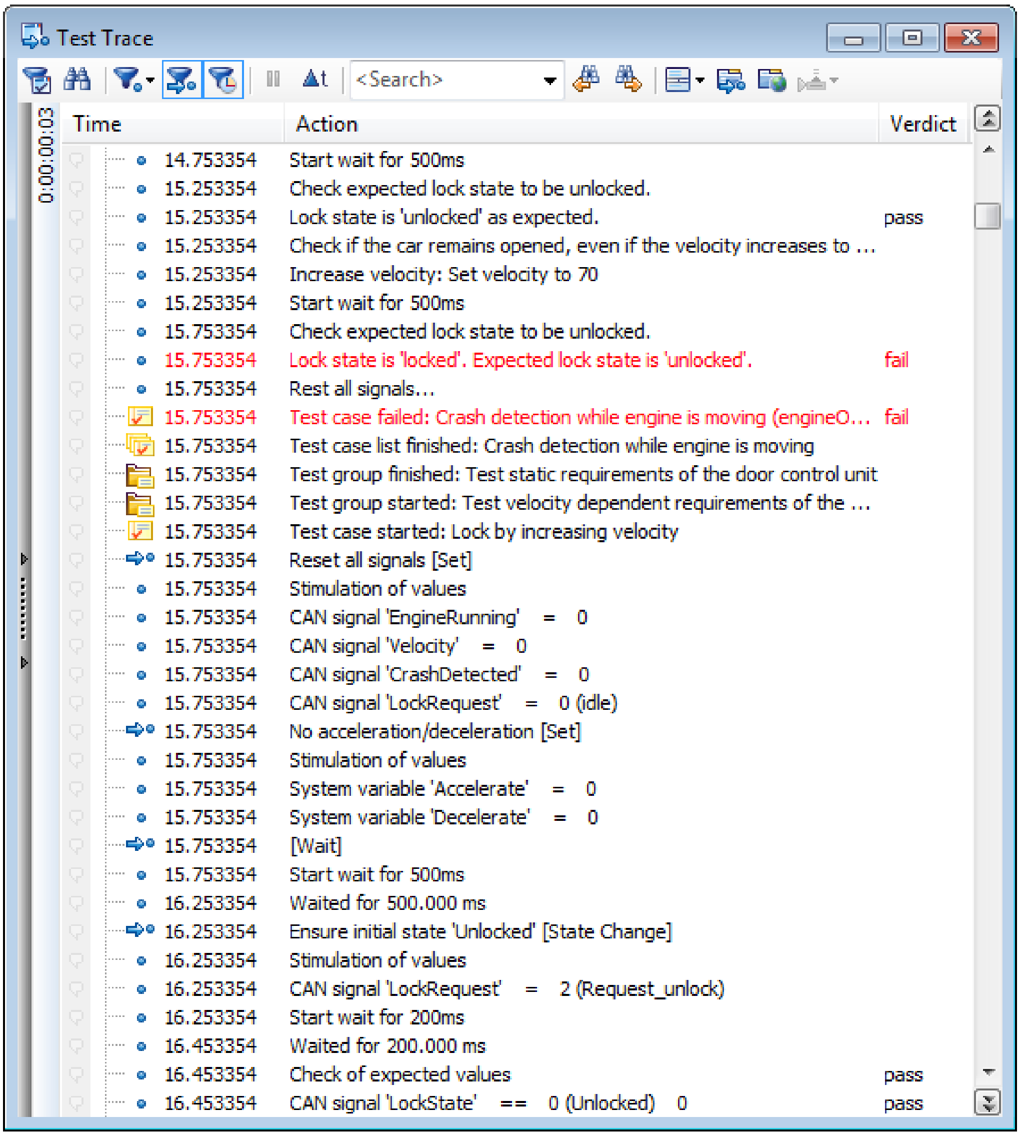


Figure 24: Test Trace Window in CANoe

Debugging

CANoe also supports debugging of CAPL and C# code that is part of a test unit.

CANoe Execution Adapter

The CANoe Execution Adapter is available as free add-on on the Vector homepage. It is used for automated test case selection and execution based on a XML-based planning list. Three planning formats are supported:

- Trace Item Format (*.vti-tso): The requirements and test case specifications can be exported directly from the REQM/TDM system.
- vTESTstudio Traceability Matrix (*.vtc-tso): The traceability matrix can be exported to vTESTstudio with trace items selected for test execution.
- Execution Plan (*.vexecplan): The execution plan can be exported from the vTESTstudio Traceability Matrix, the vTESTstudio Test Case Overview, the REQM/TDM system or the CANoe Test Report Viewer.

4.3 Reporting

CANoe Test Report Viewer

During the execution of a test unit in CANoe a detailed test report is created automatically. The CANoe Test Report Viewer supports a lot of filter and grouping functionality as well as user defined queries for a comfortable and comprehensive analysis of the test report.

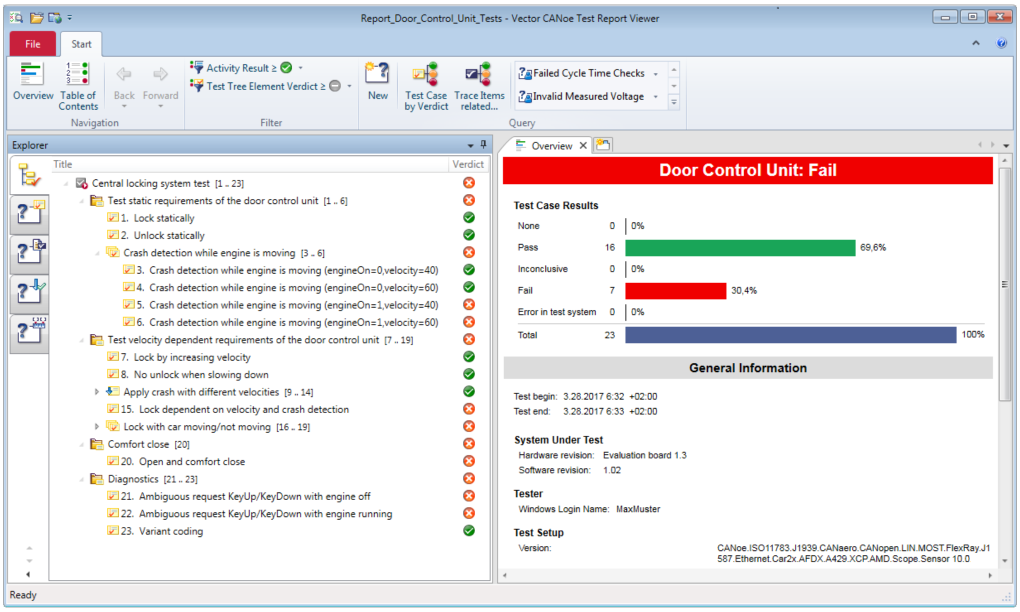


Figure 25: CANoe Test Report Viewer

5 Language Interaction

In this chapter you find the following information:

5.1 Interface Functions

Overview

Interface functions allow for functions of one language to be used in another one. An interface function can be a test case, a test sequence, a test function, or a simple function. Other than the test function the simple function is not represented as a block in the report.

Definitions

An interface function can be defined as follows in the respective languages:

> Python:
Decorator **@vector.canoe.tfs.export** on test cases, test sequences, test functions or simple functions.

> C#:
Attribute **[Export]** on test cases, test sequences, test functions or simple functions.

> CAPL:
Keyword **export** ahead of test cases, test sequences, test functions or functions.

> Test Table Editor:
Export setting on test cases, test sequences or functions which are defined in the functions view.

All interface functions which are available within a test unit are displayed in an explorer.

Explorer

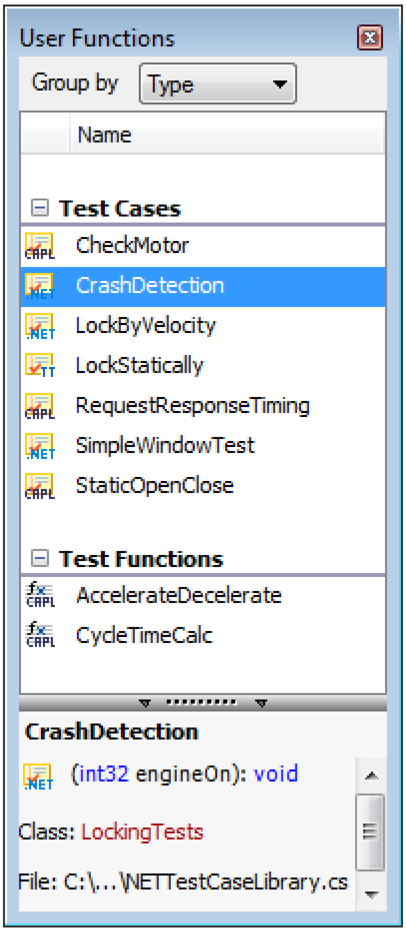


Figure 26: Explorer for interface functions

Out of the explorer, the functions can be added to the respective editors using drag & drop. Alternatively the functions can be added to the editors by the use of the text input completion feature.

Language interaction The following language interaction is supported:

Calling Language	Defining Language				
	CAPL	C#	Python	Test Table	Diagram
CAPL	●	—	—	—	—
C#	●	●	—	—	—
Python	●	—	●	—	—
Test Table	●	●	●	●	—
Test Sequence Diagram / State Diagram	●	●	●	●	—

6 Parameters, Curves and Variants

In this chapter you find the following information:

6.1 Parameters

6.1.1 Concept

Definition

The term "parameter" refers to any constant value that can be accessed within the test sequence from all implementation languages.
Examples of parameters: Configuration parameters for a control unit, test vectors, tolerances, etc.

Parameters are defined and maintained in separate files. Available parameters are displayed in the Symbol Explorer within the tab **Parameters**.

Kinds

There are several kinds of parameters:

> **(Scalar) Parameter**:
A scalar parameter represents exactly one constant value that can be accessed in the test sequence.

Example:

Name	Value
CycleTimeTolerance	50

> **(Scalar) List Parameter**:
A list parameter has 1…n values for the same variable. In the test sequence, iteration over all values of the list can be performed in order, for example, to perform a test under different temperature values.

Example:

Name	Value
OutsideTemperature	-40, -10, 0, 15, 30, 50

> **Struct Parameter**:
A struct parameter represents a set of associated (scalar) values. In the test sequence, the individual values of a struct can be accessed in order, for example, to apply a test vector (stimulating and expected values) to the test system.

Example:

Struct	Member	Value
LockingTestVector	Velocity CrashDetected Wait LockState	60 0 500 1

> **Struct List Parameter**:
A struct list parameter corresponds to a list of value tuples for a struct. In the test sequence, iteration over all list elements can be performed in order, for example, to apply various definitions of a test vector onto the system.

Example:

Struct	Member / Value			
LockingTestVectorList	Velocity	CrashDetected	Wait	LockState
	60	0	500	1

	40	0	500	0
	60	1	250	0

Hierarchical structure Parameter definitions can be structured hierarchically using namespaces.

6.1.2 Find Test Case Data by the Classification Tree Method

Classification tree method By an integrated editor for the classification tree method test case data - in terms of test vectors - can be defined. The graphical user interface supports finding the relevant input data for a test. Automatic or manual combination of all crucial input values allows to efficiently define the minimum number of required test vectors.

Boundary value analysis A dedicated support of boundary values enables the targeted testing in critical value ranges of the input data.

User interface

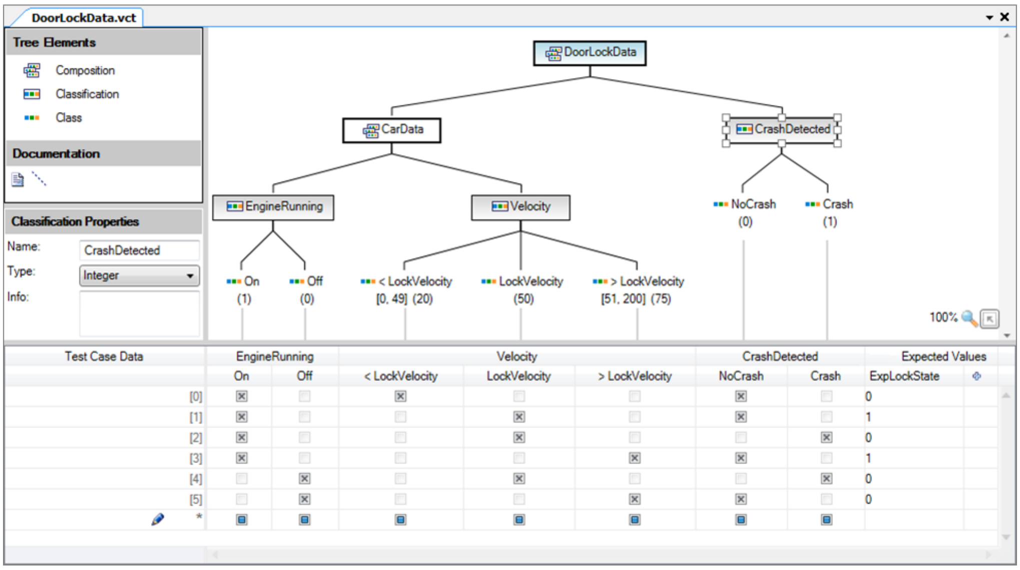


Figure 27: Definition of test vectors by the editor for the classification tree method

Parameterization of test case lists with test vectors

The test vectors can be used in implemented test cases, e.g. for the parameterization of test case lists in the Test Table Editor (see section Concepts for High Test Coverage).

Signature:	CheckAutomaticLock(int64 engineRunning, double velocity, int64 crash, int64 lockState) : void			
Parameter Values				
Struct List:	DoorLockData			
Combinatorics:	sequential			
Name:	**engineRunning**	**velocity**	**crash**	**lockState**
Type of Values:	Member of Struct List	Member of Struct List	Member of Struct List	Member of Struct List
Value Source:	DoorLockData.EngineRunning	DoorLockData.Velocity	DoorLockData.CrashDetected	DoorLockData.LockState
Values:	1	20	0	0
	1	50	0	1
	1	50	1	0
	1	75	0	1
	0	50	1	0
	0	75	0	0
Test Case Caption:	CheckAutomaticLock (engineRunning = {engineRunning}, velocity = {velocity}, crash = {crash}, lockState = {lockState})			
Use Property...	*Trace Items, Variant Dependencies, External References*			
Caption:	Check locking behavior			

Figure 28: Usage of test vectors for the parameterization of a test case list in the Test Table Editor

6.2 Curves

Stimulation curves

By the use of the so-called Waveform Editor, curves for the stimulation of the SUT can be defined. Predefined segment types (sinus, pulse …) enable easy definition of e.g. voltage curves defined by ECU test standards like LV124.

Multiple curves can be synchronized easily within the same editor.

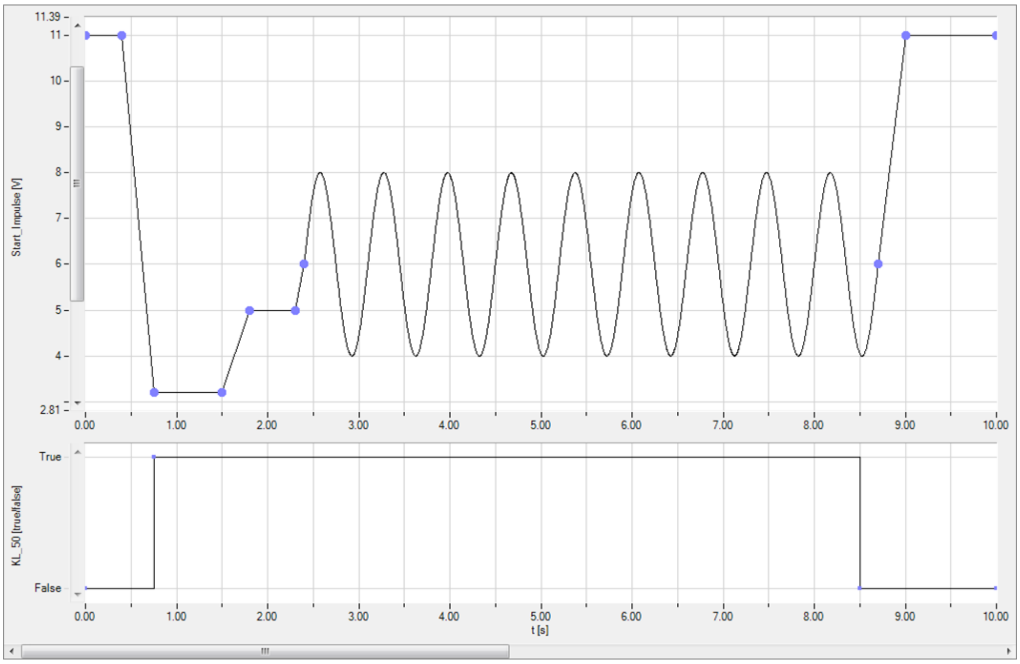


Figure 29: Definition of stimulation curves with the Waveform Editor

Synchronized check points

By the definition of check points for a stimulation curve the reaction of the SUT triggered by the stimulation can be verified.

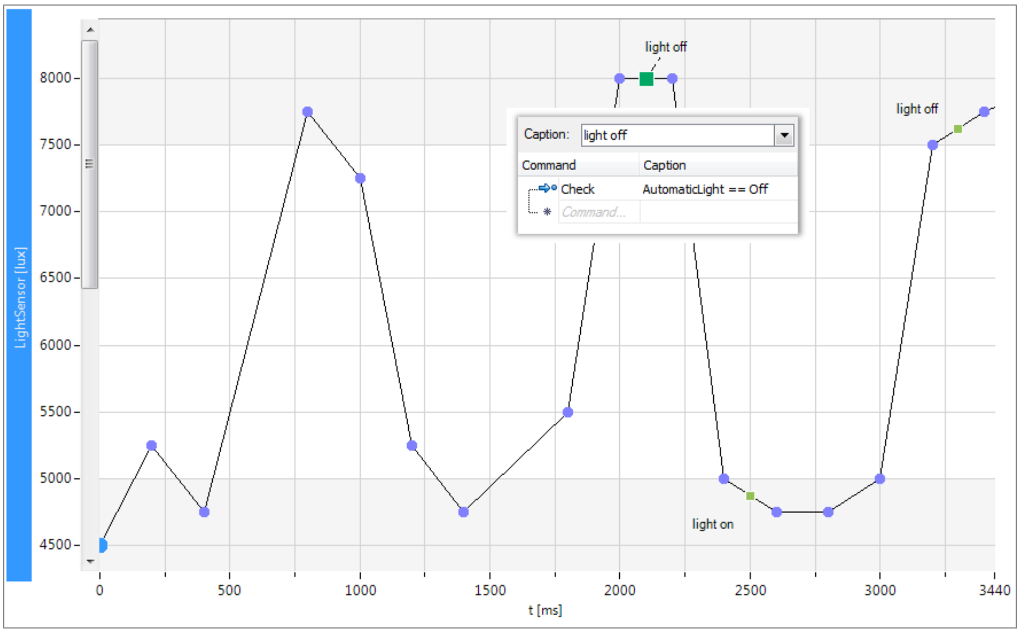


Figure 30: Definition of check points

6.3 Variants

Variant properties

ECU variants and test variants can be realized using so-called variant properties.

Examples:

Variant Property	Possible Values
Region	US, Europe, Asia
Coverage	full, low, medium
Model	OEM1, OEM2

Access

Variant properties can be used for conditional test coding as well as for access to variant-dependent parameter values and for defining variant-dependent test cases or test groups.

Access to variant property in CAPL

```
if (varprop::Region == varprop::Region::US)
{
  // ...
}
```

Parameter value dependent on a variant property

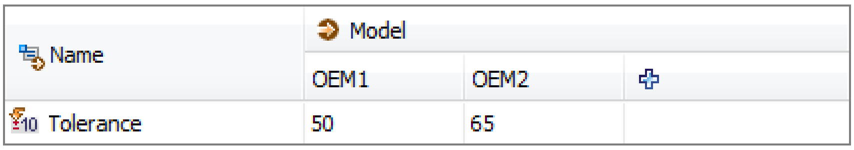


Figure 31: Parameter value dependent on a variant property

Variant-dependent test cases

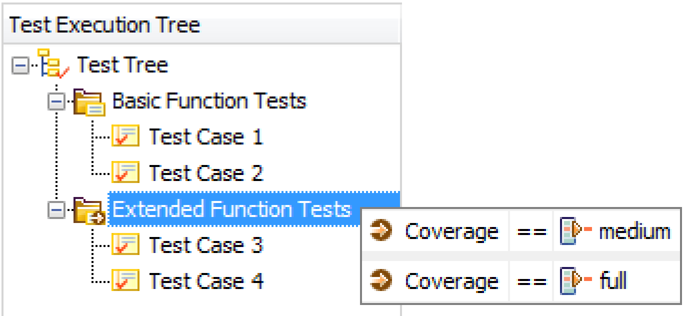


Figure 32: Variant-dependent test cases

Transfer

Variant properties can be easily transferred to the test code from the Symbol Explorer or using text completion.

Variant properties in the Symbol Explorer

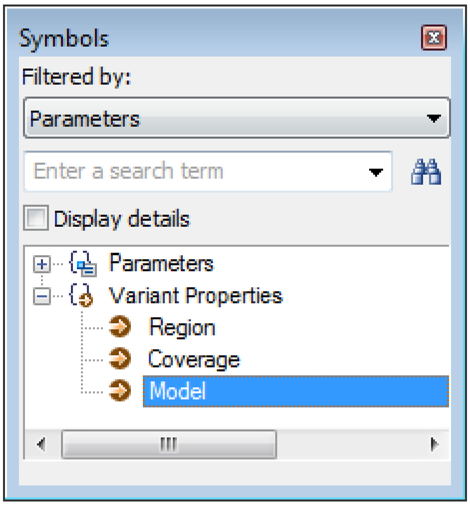


Figure 33: Variant properties in the Symbol Explorer

The value of a variant property is either already defined at design time in vTESTstudio or can be set in CANoe up until just before the start of the test.

Variant properties can be dependent on one other, i.e., the value of one variant property (e.g., `Region`) can determine the value of another variant property (e.g., `Model`).

7 Use Cases

In this chapter you find the following information:

7.1 Generating Two Similar Test Units for Different OEMs

Setup

There is a vTESTstudio project for the test of the door control unit. It consists of two test units – one for OEM1 and one for OEM2. The actual sequence logic is the same for both OEMs. It is implemented in a test table **Main.vtt** and is used in both test units by a file link. For information on linking files to test units, also see Re-Use of Files.

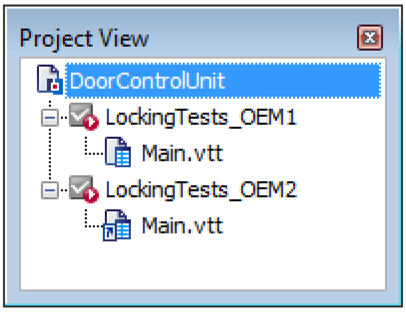


Figure 34: Using test logic in both test units

Various tolerance values are accessed during the test. These are dependent on the OEM. An **OEM** variant property is declared for this purpose with two possible values: **OEM1** and **OEM2**.

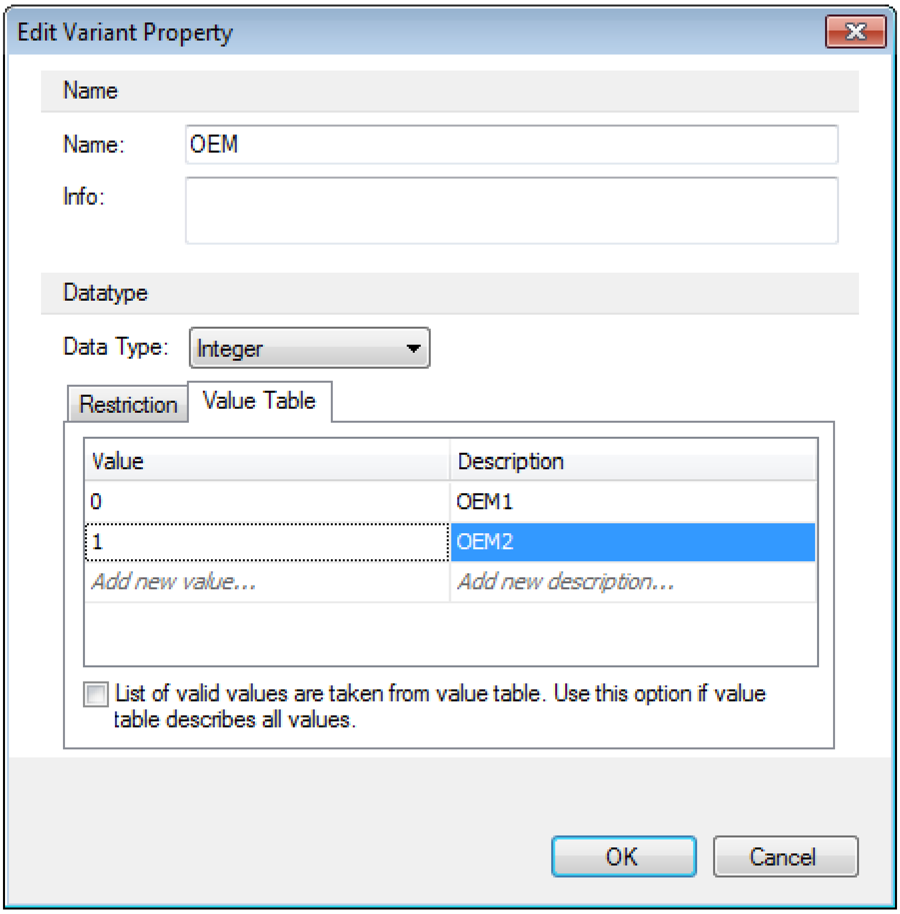


Figure 35: Definition of the variant property **OEM**

The tolerances themselves are defined in a parameter file. The specific tolerance values are defined according to the **OEM** variant property.

Name	Variant	
	OEM1	OEM2
MaxReactionTime	500	600
Tolerance1	10.2	15.1
Tolerance2	79.9	80.0

Figure 36: Parameters with variant-dependent values

The parameter file is also used in both test units.

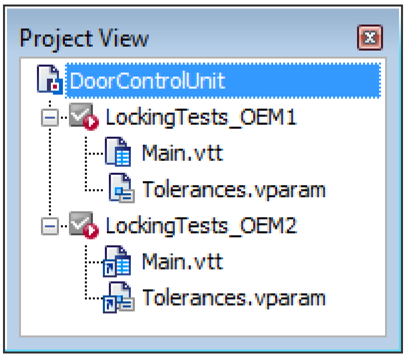


Figure 37: Using same parameter files in both test units

To ensure that the correct parameter value is used during the test execution, the user only has to set the correct value of the **OEM** variable property in each case in the configuration dialog of the test units.

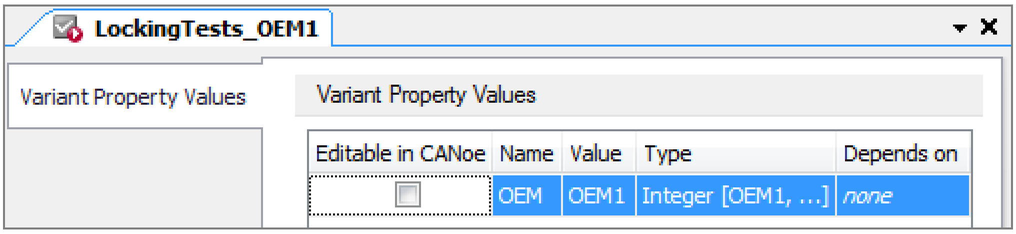


Figure 38: Defining the variant property value for the **LockingTests_OEM1** test unit

In addition to the OEM-dependent parameter values, an OEM-dependent test function is also to be used. For this, a C# function is called in the test table:

Test Case	Lock by increasing velocity
Set	Stimulate input signals
Wait	MaxReactionTime ms
Check	Check expected output signals
TestLockByVelocity	Do OEM specific locking test
Wait	500 ms
Set	Reset signals

Figure 39: Calling a C# function in the test table

The implementation of the function takes place in two C# files, one of which contains the implementation for **OEM1** and the other the implementation for **OEM2**:

Locking_OEM1.cs*

```
[TestFunction]
public static void TestLockByVelocity(double initialSpeed)
{
  // OEM1 specific test coding...
}
```

Figure 40: C# file with OEM1-specific implementation for **TestLockByVelocity**

Locking_OEM2.cs*

```
[TestFunction]
public static void TestLockByVelocity(double initialSpeed)
{
  // OEM2 specific test coding...
}
```

Figure 41: C# file with OEM2-specific implementation for **TestLockByVelocity**

The file for OEM1 is added to the first test unit and the file for OEM 2 is added to the second test unit:

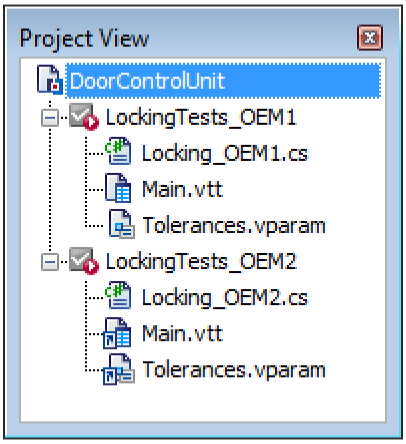


Figure 42: Using different C# files for variant-dependent implementations

VECTOR >

More Information

- > News
- > Products
- > Demo Software
- > Support
- > Training Classes
- > Addresses

www.vector.com