Testing with CANoe

Version 1.1
2009-10-08
Application Note AN-IND-1-002

Author(s) Stefan Krauß
Restrictions Public Document
Abstract This application note describes the basic concepts of CANoe's Test Feature Set in detail.

Table of Contents

1.0 Overview 2
2.0 Testing with CANoe 2
2.1 CANoe Test Concept 2
2.1.1 Architecture 2
2.1.2 Test Module, Test Group, Test Case, Test Step 3
2.1.3 Test Results – Verdicts 5
2.2 Reporting of Test Results 6
2.2.1 Overview 6
2.2.2 Manually and automatically generated report information 7
2.2.3 Use of identifiers 7
2.2.4 Test steps 7
2.2.5 Documentation of CAPL programs by test steps 8
2.3 Formulating test cases in CAPL 9
2.3.1 Principle 9
2.3.2 Setting up a CAPL test module 9
2.3.3 Wait commands 10
2.3.4 Wait commands with complex conditions 12
2.3.5 Event procedures in test modules 12
2.3.6 User-defined events 13
2.3.7 Differences between CAPL for simulation/analysis and for testing 13
2.4 Defining test cases in XML test modules 14
2.4.1 Principle 14
2.4.2 Setting up XML test modules 15
2.4.3 Working with test functions 16
2.5 Programming test cases in .NET test modules 16
2.5.1 Principle 17
2.5.2 Setting up purely .NET test modules 17
2.5.3 Wait points 18
2.5.4 Type Library 18
2.5.5 Event procedures in test modules 18
2.5.6 Observation of system conditions 18
2.6 XML test modules versus CAPL/.NET test modules 19
2.7 Constraints and conditions 20
2.7.1 Principle 20
2.7.2 Use of predefined checks in CAPL and .NET 22
2.7.3 Constraints and conditions in XML 23
2.7.4 User-defined test conditions 23
2.7.5 Influencing the test flow 24

Copyright © 2009 - Vector Informatik GmbH
Contact Information: www.vector.com or ++49-711-80 670-0

2.8 Test Service Library 24
2.8.1 Checks 24
2.8.2 Stimulus Generators 25
2.9 Test setup 25
3.0 Test strategies 26
3.1 Protocol tests 26
3.1.1 Test concept 26
3.1.2 Implementation in CANoe 27
3.2 Application Tests 27
3.2.1 Test concept 27
3.2.2 Implementation in CANoe 28
3.3 Invariants test 29
3.3.1 Test concept 29
3.3.2 Implementation in CANoe 30
4.0 Interface to a Test Management System 31
4.1 Fundamentals of test management 31
4.2 Principle of the interface 31
5.0 Contacts 32

1.0 Overview

Although CANoe was designed as a tool for analyzing and simulating bus systems, from its inception it was also used to test ECUs and networked systems. Effective with Version 5.0, CANoe[1] was expanded to include test support capabilities, and the so-called *Test Feature Set* was integrated. The Test Feature Set is not a self-contained component of CANoe; it encompasses a whole series of extensions that simplify the process of setting up tests and expand CANoe to include important capabilities such as test reporting.

Discussed in this White Paper are the testing concepts and potential applications of CANoe and the Test Feature Set. In particular, its objective is to show how tests are set up with CANoe and how test sequences are formulated.

2.0 Testing with CANoe

In this chapter the important concepts and components of the Test Feature Set are presented, and the use of CANoe as a platform for executing tests is explained.

2.1 CANoe Test Concept

2.1.1 Architecture

Besides its well-known analysis and simulation components, another component was introduced in CANoe for testing. *Test modules* can be started by the system or the user during measurement, and they each execute a test sequence. The test sequence may be formulated as a CAPL/.NET program or an XML file (these are also called

[1] The program package CANoe/DENoe supports different bus systems such as CAN, LIN, MOST and FlexRay. Therefore, the program is available under slightly different names, depending on the specific bus systems supported (e.g. CANoe.MOST supports CAN and MOST). In this White Paper “CANoe” represents all such bus system variants, i.e. the configurations apply to all variants.

CAPL test modules, .NET test modules and *XML test modules*). Since such a file describes exactly one test module, the description files themselves are often referred to as test modules.

Test modules access the remaining bus simulation, the buses (e.g. CAN, LIN, FlexRay, MOST), and access the digital and analog input and output lines of the Device Under Test using general purpose I/O cards or VT System via system variables . An ECU may be tested by itself, or as part of a network consisting of various ECUs where the object of testing is called a *System Under Test* or *SUT*. CANoe's options are available to the test modules, e.g. panels for user interaction or writing of outputs to the Write Window.

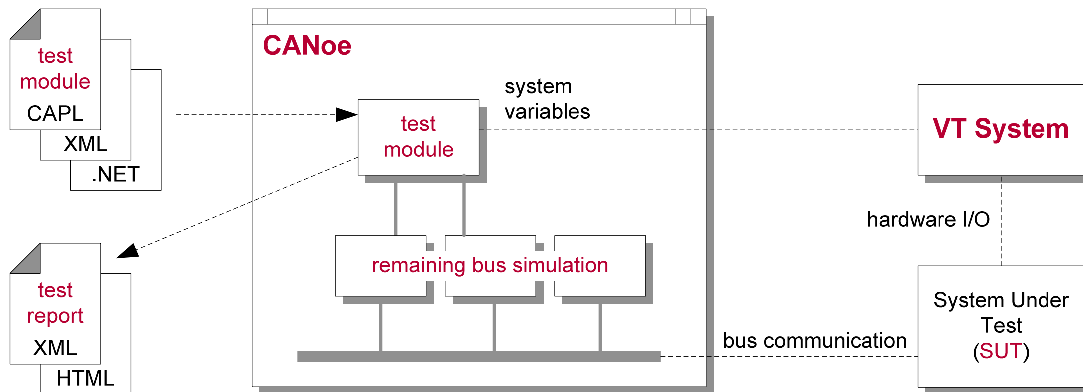

Figure 1: Setup of a test system with CANoe.

Besides using CANoe to execute the tests, other CANoe functions can also be used. For example, part of the total system might be simulated in CANoe using simulation nodes (*remaining bus simulation*) or bus communication and the behavior of the SUT might be analyzed (observation in the Trace Window, analysis by CAPL program, logging of statistics, etc.).

A *test report* is created as a result of the execution of a test module. The test report in XML format is written during test execution. After the test module has been executed the XML test report is converted to HTML format (see section 2.2).

2.1.2 Test Module, Test Group, Test Case, Test Step

In CANoe the *test module* is the execution unit for tests. A test module is always started for test execution, and the results of this execution are represented by exactly one test report. The test module contains the *test cases*. The test case is the central concept, in which the actual testing actions are collected. These in turn are organized by *test steps*. A test step is simply a piece of information indicating that the test sequence has reached a specific point.

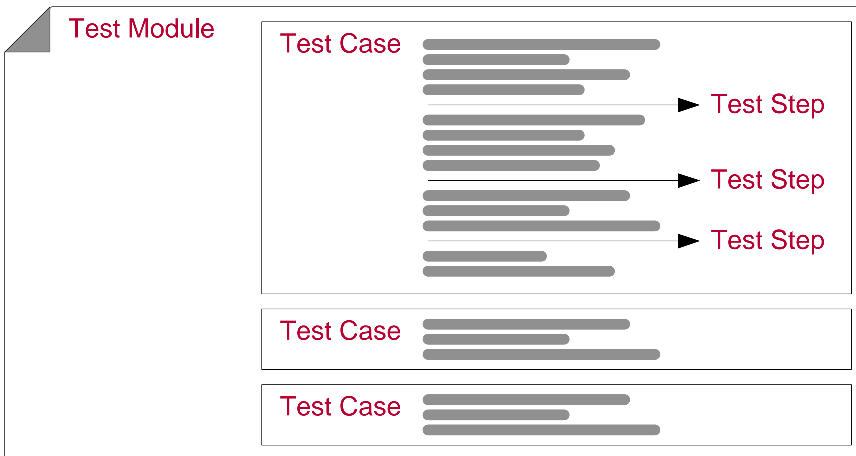

Figure 2: Test modules contain test cases that are organized by test steps.

The test case contains the actual instructions for test execution. If at all possible, test cases should be set up so that they work independently of all other test cases. Test cases are executed as part of a test module that includes other test cases as well, but they are developed and maintained independently of one another (see chapter 3.0 on Test Management Systems). The testing actions that serve to test a property or capability of the ECU under test are formulated in a test case.

For example, the entire network management of an ECU consists of a large number of properties and functions. Each of these properties and functions should be checked in a separate test case. E.g. the tester would formulate a separate test case for regular startup of the system, for handling a specific fault situation and so on.

one would not test the entire Network Management of an ECU in a test case, since it consists of a large number of properties and functions. Instead one would handle each of these properties in a separate test case, e.g. by formulating a separate test case for regular startup of the system or for handling a specific fault situation.

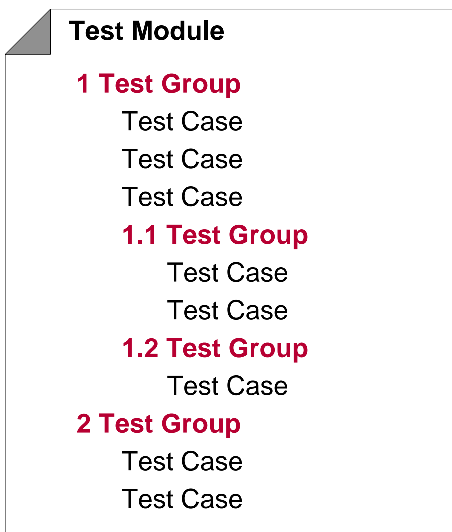

Figure 3: Test cases can be organized hierarchically by test groups.

In the test report the executed test cases are listed with their results. These *executed* test cases may be structured in the test report according to *test groups*. Test groups are mainly an organizational category for the executed test

cases. Like the titles of chapters in a book, the test groups may be created hierarchically, whereby the executed test cases would be the leaves on the tree of test groups.

Test groups are dynamically defined in CAPL and .NET test modules. A test case is assigned to a test group by the CAPL/.NET program at run time. In CAPL/.NET test modules, test cases may be executed a number of times during a test run. A test case might therefore be documented multiple times in a test report and in various test groups.

In XML test modules the structure of the test cases, i.e. their sequence and test group associations, is defined statically. A test case will only appear once here in the report, and it will always have the same test group assignment.

2.1.3 Test Results – Verdicts

Test results are automatically acquired and processed in CANoe. The Test Feature Set follows a very simple concept: Tests may be passed (*pass*) or failed (*fail*). This simple, two-value test result is called a *verdict*. The verdict is passed if the test could be executed and prove that the functionality that has to be checked by the test case or test module works. In all other cases the verdict is failed. Please note that a test case for example also fails if it could not be executed correctly because of external events (e.g. the presumptions are not valid). Therefore the statement of the verdict is “SUT is proven to be ok” or “it was not possible to check that the SUT is ok”. Especially failing does not mean “SUT is proven to be not ok”. Nevertheless, additional hints in the report may give the tester an idea why a test case has failed.

Test execution is evaluated on two levels, the level of test cases and the level of the entire test module. In CANoe, test results are not formed on the level of test groups since assignments of test cases to test groups are not static, at least in CAPL test modules, and consequently they are not uniformly determinant over multiple test runs. This means that it is not possible to draw any conclusions of the type “Test group XY passed/failed”.

In principle, a test case is considered passed if there were *no* errors during its execution. Therefore, error-free execution does not need to be reported explicitly in the test flow. An error might be reported by different sources, especially these:

- An error is indicated in the CAPL program by `TestStepFail` or `TestCaseFail`,
- A failure of test functions (see section 2.4.3) is automatically reported,
- A failure of a condition or constraint (see section 2.5) is automatically reported,
- A problem in the execution of test commands is evaluated as an error (e.g. input of illegal parameter values).

Generally a CAPL/.NET test case is continued after an error occurs. Subsequent tests might run just fine with no reported errors and an explicit “pass” in the test report. However this does not mean that the verdict of the test case is reset to “pass”. Clear rule: Once an error has occurred the test case has always failed. In XML no control structures (like conditions) are available. Thus, it is impossible for the user to react on failures. Instead of this the XML test cases are canceled after an error occurs and the test module is continued with the next test case.

The verdict of the test module is reached by the same rules as those used to arrive at verdicts for the executed test cases. As soon as a test case has failed, the entire test module is also considered failed. The verdict is automatically determined by CANoe during execution of the test module.

Essentially a verdict is a conclusion regarding an executed test module or an executed test case. So there is no verdict of the type “not executed” or similar. Nevertheless, unexecuted test cases may be documented in the test report. They have no verdict, and the information that is passed simply indicates that these test cases were specified in the test description but were not executed.

The current verdict may be queried, e.g. to automatically terminate execution of a very long test sequence at specific points if an error has already occurred up to that point. On the module level the verdict may also be set explicitly. This option makes it possible to implement your own verdict logic in exceptional situations.

Test steps, test functions, test patterns and others may also have a verdict like result. These results are not verdicts in the closer sense. They typically contain more information than just pass/fail. For example, test steps

may also have a result “not available” if the test case is just information about the ongoing test execution and reports no checking result.

2.2 Reporting of Test Results

A test report documents one execution of the test module and the results of this test run. The test report is written during execution.

2.2.1 Overview

CANoe starts writing the test report to the hard drive in XML format during test execution, and it supplements it continuously. This assures that even very long test runs can be documented without having internal buffers overrun or requiring excessive amounts of memory (RAM). Immediately after the end of the test run this XML file is converted into HTML with the help of a XSLT stylesheet.

The XSLT stylesheet determines the form and contents of the HTML test report. Using the stylesheet, it is easy to display a report on a simple HTML page, but complex HTML documents consisting of many files may also be generated. Furthermore, the stylesheet can be used to filter out information, since in the conversion not all information needs to be copied from the XML report to the HTML report. It is easy to modify the supplied stylesheets for one’s own needs or to use a custom stylesheet.

The conversion of the XML report to HTML is performed by a XSLT processor that is controlled directly by CANoe. However, the user may also call the XSLT processor from a DOS prompt without CANoe (see online help of CANoe).

It is therefore possible to convert an existing XML report to a HTML report at a later time. Various XSLT stylesheets may be used to generate different representations of the same report. For example, the user could create a simple test summary for filing purposes as well as a very detailed report for error analysis. Similarly, different representations that depend on the specific output medium are feasible, e.g. an easy-to-print report and a version equipped with lots of navigation controls for interactive viewing with a Web browser. Using XSLT stylesheets it is even possible to generate reports in other output formats such as RTF or text files.

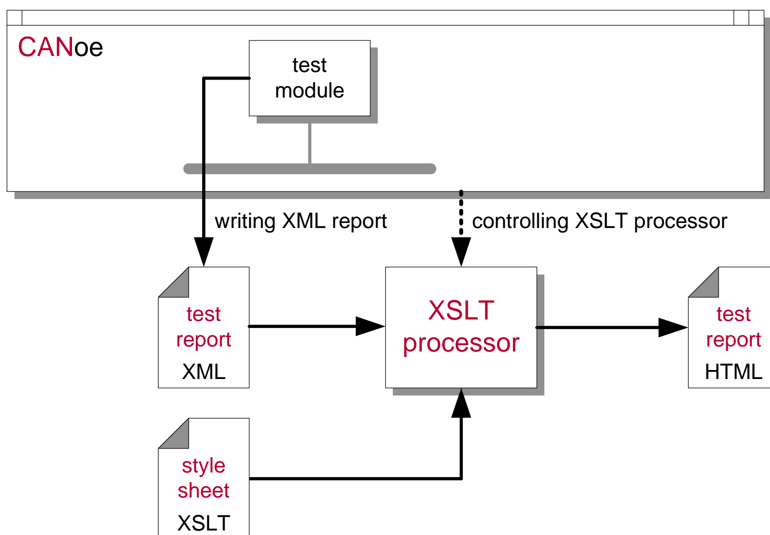

Figure 4: Creating a report using CANoe.

The XML test report is the primary report output of CANoe that contains all of the recorded results. The amount of reported results can be influenced by the report filters in the configuration dialog of a test module. Filtered information will not be generated at all. The reason for using this filter is to get smaller XML files and to disburden

CANoe from handling unneeded reporting events. HTML reports are simply views that can be generated from the XML report at any time.

Besides reporting, CANoe also supports other modes of recording information, especially the so-called *Logging* Feature. Test reporting does not replace logging; the two output types complement one another. While logging records bus communications and interaction with the environment (e.g. I/O via system variables) the test report gives information on which test cases were executed when and what their results were.

2.2.2 Manually and automatically generated report information

Essentially the test report is automatically generated by CANoe[2]. This means that it is not necessary to add supplemental commands or information to the description of test sequences. CANoe might use test case names as they appear in the CAPL, .NET or XML definition, for example.

The existing information is sufficient to generate a simple test report that lists the executed test cases with their results and the specific start and end times. On the other hand, CANoe can supplement the report to include expanded descriptions of the test setup, device under test, test engineer and individual test cases. This information is added to the test description by suitable CAPL/.NET commands (e.g. `TestCaseComment`) or XML elements (e.g. `<description>` in `<testmodule>` or `<testcase>`).

2.2.3 Use of identifiers

In many test specifications the test groups, test cases and individual test steps are assigned a unique identifier in addition to their name. Often the test cases are just numbered sequentially, but sometimes a more complex system of identification is used (e.g. “NWM01” through “NWM15” might identify test cases for network management functionality).

Test groups, test cases and test steps may be assigned identifiers in CAPL, .NET or XML test modules. This is the purpose of the parameter respective field “ident”. The identifier will be copied to the report. They might be used to uniquely identify the test groups, test cases or test steps in the test module and thereby establish a reference between the test module and the test report.

If identifiers are not needed, these entries may be omitted (enter an empty string). Identifiers should not be used to store longer names, since the test report formatting is not laid out for this purpose. Separate fields are available for names and descriptions.

External references are another possibility to link test cases to test specifications or other external sources. Test cases written in CAPL, .NET or XML can contain reporting commands/elements with references that are copied to the report. The DOORS integration in the Test Automation Editor TAE for example works in this way. The reference to the representation of the test case in DOORS is stored in the test case of the XML test module. During execution this reference is copied to the report. At the end, the user can directly navigate from the test case in the report to the test case in DOORS.

2.2.4 Test steps

The test steps are the most important means of reporting on the test sequence in detail. Test steps are statements about the status of test execution at a specific point in time[3]. They contain:

- The level of detail,
- an identifier,
- a description, and

---

[2] Just like the conversion to HTML in CANoe, the creation of the XML test report may also be configured separately for each test module.

[3] Test cases are not subdivided into test steps, rather the execution of test cases is reported by test steps at specific points in the sequence. In spite of this, test steps may be utilizes so that they describe the test cases section by section.

- a result.

Entry of the *level of detail* is optional and serves to differentiate between fundamental test steps and those that just provide details in the test report. This makes it possible to vary the level of detail in generating the HTML report. The *level of detail* is input as a number; if no number is input, 0 is assumed (0 represents important test steps; larger values describe increasing detail in test steps).

Identification of test steps as described above (see 2.2.3) is used to reference individual descriptions of test steps in the user's existing test documentation, e.g. reference to a test step number in a test specification.

Test steps are assigned a *result* resulting from the specific test step command used, and this is output in the report. The actions documented by the test step are what is being evaluated:

- *pass*: The reported actions were executed as expected. For example, a specific message was received as expected in the test sequence. The test sequence is thereby reported accurately, since error analysis often requires precise knowledge of the previous passed test steps.
- *fail*: The reported action indicates an error in the test flow. For example, an expected message may not have been received within a specified time period (timeout).
- *warning*: The reported action was executed as expected, but the result indicates a potential problem.
- *none*: The test step only indicates that a specific action was executed (e.g. that a message was sent out), but a verdict in the strict sense of the term is not possible (documentation of a fact).

Essentially test steps only serve to transfer information on the test sequence to the report. They do not influence the test execution itself except of the failed test steps. Output of a test step with the "fail" verdict only makes sense if the test case also fails at the same time. Therefore the CAPL command `TestStepFail` for example automatically causes the test case to fail as well. If the user works with test steps in this way, the CAPL command `TestCaseFail` is not really necessary.

In CAPL/.NET test modules the user can only define test steps. In XML test modules, tests are defined by the more abstract test functions which output the test steps without user intervention.

2.2.5 Documentation of CAPL programs by test steps

In CAPL test steps may be used to organize the program and substitute for program comments. Test step commands can be used in a "two-part" form for this purpose: An initial test step command initiates the test step and essentially serves as a title for the program lines that follow it. Then the concluding test step command is called at a prescribed time within this first test step. The second command sets the time stamp for the test step and causes it to be copied to the test report.

Consider the following program section that was formulated with a comment and only one test step:

```
// Test step 27: waiting for answer
if (1 != TestWaitForMessage (Answer, 500))
  TestStepFail ("27", "Waiting for answer - answer not received");
else
  TestStepPass ("27", "Waiting for answer – answer received");
```

This program section could be replaced by the following program section:

```
TestStepBegin ("27", "Waiting for answer");
if (1 != TestWaitForMessage (Answer, 500))
  TestStepFail (" - answer not received");
else
  TestStepPass (" – answer received");
```

Depending on the outcome of the wait operation, the output for the test step of the example might be a "fail" verdict with the comment "Waiting for answer – answer not received", or it might be a "pass" verdict with the comment "Waiting for answer – answer received".

2.3 Formulating test cases in CAPL

Test modules can be defined in various ways in CANoe. A very flexible option for programming test modules is the use of CAPL.

2.3.1 Principle

The CAPL program supplied in CANoe is an easy-to-learn programming language that is well-suited to the problem area of "control module communications". This programming language was expanded in the framework of the Test Feature Set so that it not only could master simulation and analysis tasks efficiently, but could also formulate tests in a clear and concise manner. As already mentioned above, CAPL programs that implement tests are executed as test modules in CANoe. Therefore these CAPL programs are also referred to as *CAPL test modules*.

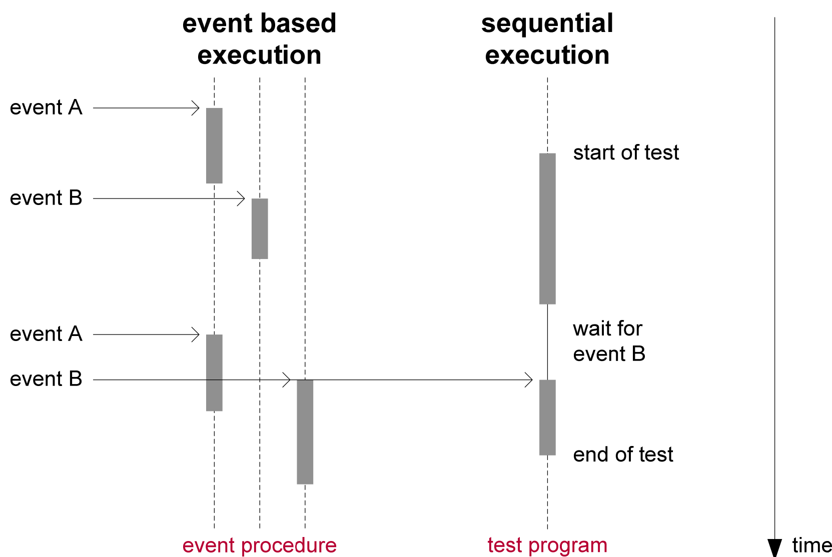

Figure 5: Event-oriented and sequential execution.

For the most part, tests consist of a sequentially executed flow of test actions that has defined start and end points. Therefore CAPL test modules do not follow a strict event-oriented execution model, rather they follow a sequential execution path that may also contain wait points.

A test module is started explicitly during the measurement. When the test module is started the CAPL function `MainTest` is called. This function and the test cases it calls are executed sequentially. As soon as the `MainTest` function ends the test module is also automatically terminated. That is, the run time of the `MainTest` function determines the run time of the test module.

A test module can be restarted after it has been stopped, even during the same measurement. This means that a test module may be used multiple times during a measurement, but only one instance of the same test module may be executed at any given time. Different test modules may be operated simultaneously in CANoe, since test modules, like simulation nodes, represent independent execution blocks. However, this is only practical if the test sequences are fully independent of one another.

2.3.2 Setting up a CAPL test module

A CAPL test module consists primarily of the `MainTest` function and the implementation of test cases. Test cases are defined in special functions:

- That are identified by the key word `testcase`,
- That have no return value,
- That can have parameters like other functions,
- Whose call is automatically recorded in the test report (see 2.2.2), and
- For which CANoe automatically computes a test result, i.e. a verdict (see 2.1.3).

The `MainTest` function is really the “main program” of a test module. The task of this function is to call the necessary test cases in the desired sequence. `MainTest` provides control of the test sequence. In its simplest form `MainTest` could just consist of calls of test cases.

```
void MainTest ()
{
  Testcase_1 ();
  Testcase_2 ();
  Testcase_3 ();
}
```

Even if it is technically feasible to implement any desired test sequences using output and wait commands in `MainTest`, `MainTest` should only contain test flow control and not enter into interaction with the system itself. This should be reserved for the individual test cases. The flexibility of `MainTest` can be exploited to implement complex flow logic, however. For example, when errors occur certain test cases could be ruled out as sources if they build upon a functionality that has already been detected as faulty. A test case might be executed multiple times by calling it in a loop, e.g. with a different set of parameters each time.

The wide range of programming options in `MainTest`, and the capability of using global variables in test case implementations too, both carry the risk that test cases might be interdependent from a technology perspective on implementation. Such dependencies make troubleshooting difficult in the test cases and prevent the reuse of test cases. Therefore, in general efforts should be made not to have test cases build upon one another and to undo changes made to the ECU under test at the end of the test case.

Event procedures can also be defined and used in test modules (only during run time of the test module, see also 2.3.5), but the system event procedures `Start` and `MeasurementStop` are not available in the test modules, since test modules are not active from the start to the end of the measurement. Nevertheless, `Prestart` and `Prestop` can be used to initialize and finalize resources.

A CAPL file with test cases in CAPL may also be assigned to XML test modules as a test case library. In this case the XML test module “calls” the test cases in the library. The control flow of the test module is defined in XML (see 0) but several test cases are implemented in CAPL. The XML test module only contains the calls with the concrete parameters.

2.3.3 Wait commands

Sequential execution of a CAPL test program can be interrupted by *wait commands*. The special aspect of these commands is that they return flow control back to CANoe and do not resume processing until a specific event occurs. From a technical perspective, when a waiting CAPL function is called flow control does not return to the caller until a specified event has occurred.

If CANoe events are registered – e.g. reaching a dedicated signal value range, change to an environment variable or expiration of a prescribed time span – then these events get a time stamp according to the CANoe clock and are queued in a wait buffer for handling. First, all those events that occurred at a specific time point are handled then the internal clock advances.

The execution of an event procedure in a CAPL program is not interruptible. An event procedure is always executed completely, and only afterwards is the next event taken from the wait queue and handled. This means that all operations within a CAPL program are executed at the same time, as measured by the CANoe clock.

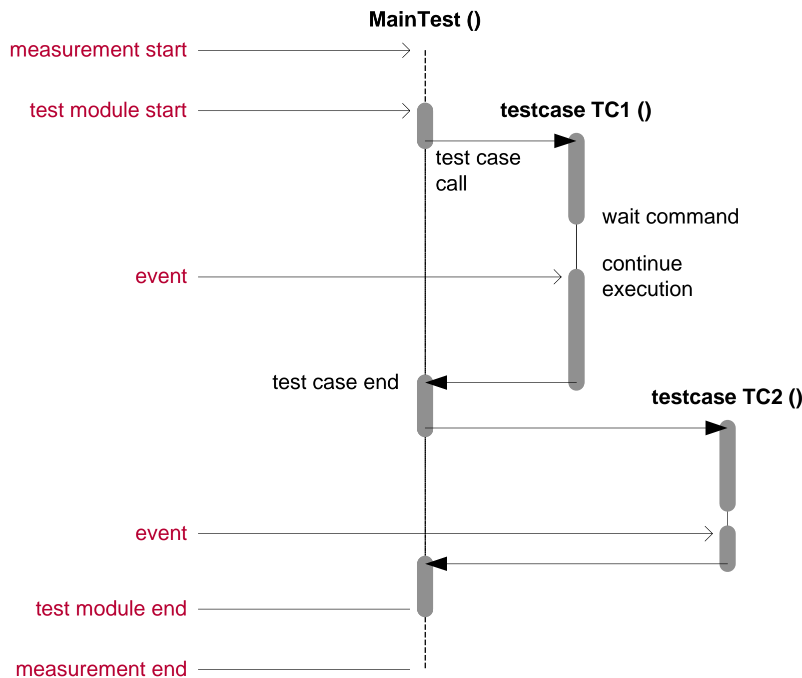

Figure 6: Execution principle for a test module with wait commands.

This execution model for CAPL programs is breached by the wait commands. Initially the start of a test module is a CANoe event like any other. It starts processing of the test module. At first this processing cannot be interrupted; it runs at its own timing until it encounters a wait command. The wait command now interrupts the CAPL program and makes sure that all other events in the queue are processed first, and the clock is advanced if applicable. The CAPL test module does not resume execution until an event occurs that was the object of the waiting. The other CAPL program is then executed again without interruption up to the next wait command.

A wait condition is triggered when the event specified in the wait command occurs, e.g. a specific message is received. A maximum wait time (*Timeout*) must always be given in the wait commands, so that the wait condition is removed when the maximum wait time has elapsed.

The reason the wait condition was cleared is indicated in the return value of the wait condition. Generally it is not necessary to consider all return values individually. It is sufficient to differentiate between the expected return value and all others. The latter is evaluated as an error. The expected return value does not necessarily need to be the return value indicating that the event given in the wait instruction has occurred. It may actually be the case that expiration of the maximum wait time is expected in the test case, for example (statement: “The specified event did not occur within this time period”).

If at all possible, the various wait commands return the same values for similar events. Negative return values identify events that are generally evaluated as errors in the test sequence. However, in an actual specific case such an event might also be expected in the test sequence, and it does not necessarily lead to a test error. This distinction only reflects the assessments made in most cases; otherwise it has no influence on test execution.

```
if (1 == TestWaitForSomething (…))
{
  … // expected event received, go ahead with normal test course
}
else
{
  … // expected event not received, handle this error
}
```

The reason for clearing a wait condition is automatically recorded in the test report. However, the test case only fails automatically if real errors occur such as entry of invalid parameters. In all other cases the CAPL program itself must evaluate and decide whether to clear the wait command.

The CAPL program would be structured even more clearly if there were only an explicit reaction to the unexpected termination of the wait command. In this case it is possible to avoid nested if-conditions in the CAPL program for the most part.

```
if (1 != TestWaitForSomething (…))
{
  // Some thing went wrong – handle this error
}
… // all correct, go ahead with the normal test course
```

The main program flow now reflects the expected course of testing; the if-branches after the wait commands handle errors. In the simplest scenario, the test case is terminated as soon as an error occurs (but not the test module). Error handling would be to report the failure with `TestStepFail` and exit the test case with `return`.

Example:

```
if (1 != TestWaitForMessage (msg, 1000))
{
  TestStepFail (“”, “Message msg not received within 1000ms.”);
  return;
}
TestStepPass (“”, “Message msg received.”);
… // go ahead with normal test course
```

In this example, commenting is also output to the test report if the test is successful (see also 2.2.4). This is not absolutely necessary, but it makes it easier to follow the test flow in the test report.

2.3.4 Wait commands with complex conditions

Besides simple wait commands that wait for exactly one event, it is possible to set up wait commands with combined wait conditions. This involves defining the individual conditions (`TestJoin...` CAPL commands) before the actual wait command. These definitions do not start the wait process, nor do they monitor the conditions. That is done by `TestWaitForAnyJoinedEvent` (waiting for one of the defined events with OR logic) or `TestWaitForAllJoinedEvents` (waiting for all events with AND logic).

```
TestJoinMessageEvent (msg1);
TestJoinMessageEvent (msg2);
TestJoinEnvVarEvent (envvar);
if (0 < TestWaitForAnyJoinedEvent (2000))
…
```

The `TestJoin` commands return a value greater than zero as the unique identifier for the wait condition. With `TestWaitForAnyJoinedEvent` the return value indicates which event occurred.

After the wait operation all wait conditions are cleared internally. This means that the user would need to input all of the wait conditions again if a combined wait condition is needed more than once.

2.3.5 Event procedures in test modules

A test module may also contain *event procedures* outside of the test sequence that consists of the `MainTest` function and the test cases it calls. Event procedures are executed quasi in parallel to the actual test sequence. However, since as described above execution is not interruptible between two wait commands, and on a logic level it occurs within a single time step, an event procedure can only be executed if the “main program” of the test module is waiting. An event procedure is not interruptible, and it cannot contain any wait commands itself. Therefore synchronization tools such as semaphore may be omitted.

When using event procedures in test modules it should be noted that the test module is inactive before the start of `MainTest` and after its end, and therefore no event procedures may be executed during those times either. Furthermore, all CAPL variables are reset at each start of a test module. In this way each test run has the same initial conditions.

2.3.6 User-defined events

Not all conceivable wait conditions can be implemented with the existing wait commands. Therefore provisions are also made for triggering *user-defined events* and waiting for them. User-defined events are referred too as *"text events"*, because they are identified by a unique character string selected by the user.

The event procedures existing in test modules are also used to implement user-defined wait conditions. In these procedures a check is made to determine whether the incoming events satisfy the wait condition. If this is the case, a user-defined event is initiated that can be waited for in a test case. CAPL programming in the event procedures permits implementation of any desired wait conditions.

A simple example:

```
on sysvar sysvar::VTS::Speed::PWMFreq
{
  if (@sysvar::VTS::Speed::PWMFreq > 100)
    TestSupplyTextEvent ("Speed100");
}

testcase TC ()
{
  …
  TestWaitForTextEvent ("Speed100", 2000);
  …
}
```

User-defined events only act within a test module. Therefore, it is not possible to initiate a user-defined event in one test module or simulation node and react to it in another test module.

Besides user-defined events ("text events") there are also the very similar *"auxiliary events"*. These events are generated by the *Checks* of the *Test Service Library* when errors occur (compare 2.8) or by other extension libraries (DLLs). It is possible to wait for these events.

2.3.7 Differences between CAPL for simulation/analysis and for testing

The following table summarizes the differences between CAPL in simulation/analysis and for testing:

	Simulation and Analysis	**Testing**
Execution period	Active over entire measurement	Only active from start of the test module until end of the function `MainTest`
Execution model	Event-driven	Sequential execution Additionally: Execution of event procedures is possible
	Event procedures are conceptually designed to be atomic; waiting is not possible	Conceptually atomic execution between two wait commands and of event procedures
Available CAPL commands	CAPL test commands cannot be used	Supplemental CAPL test commands are also available
Reporting	–	Automatic

Essentially CAPL for testing is an extension of the more familiar CAPL implementation, so nearly all existing CAPL capabilities may also be used in CAPL test programs. Nonetheless, because of the changed execution model there are a few principle differences between CAPL programs for simulation or analysis and CAPL for test modules.

2.4 Defining test cases in XML test modules

Besides being defined in CAPL or .NET programming, test modules can also be defined in a more abstract way in XML. Furthermore, CAPL test cases, .NET test cases and test cases defined in XML can be mixed within one test module. Thus, test cases should be written in the language that fits best to the requirements of each test case.

2.4.1 Principle

Test modules in XML take a different approach to describing test cases than CAPL test modules, but both approaches are based on the same underlying concepts, structures and properties of the Test Feature Set. The most important difference, besides the file format, is the way in which test cases are described.

In XML test modules test cases are described by a sequence of *test functions* and *control functions*. Test functions consist of *test primitives* and the *test patterns*. A test pattern implements a typical test sequence including a check of whether the test was passed. The test pattern is generic in the sense that specific key information must be provided. For example, one of the available test patterns implements a check of a state change. How this query is executed, however, is fixed in the pattern. The signals and signal values that describe the start and end states must be communicated to the test pattern by means of parameters.

A test pattern contains:

- The actual test sequence,
- The check of the condition by which the verdict is set,
- Generation of report outputs, and
- Parameters for configuration of the test patterns.

All that needs to be done in the XML file is to name the test patterns to be executed and configure them with parameters. Therefore, test cases are *defined* in XML, while in CAPL and .NET test cases are *programmed*.

An example of a simple test module with a test case that is implemented by exactly one test pattern:

```
<testmodule title="Mirror Control Test" version="1.2">
  <testcase ident="TC 2.4.1" title="Mirror Close Right" >
    <statechange title="Close Mirror, check motor" wait="1000">
      <in>
        <cansignal name="Mirror_2">1</cansignal>
      </in>
      <expected>
        <cansignal name="MirrorMotor2"><gt>5</gt></cansignal>
      </expected>
    </statechange>
  </testcase>
</testmodule>
```

The available test patterns are components of a library in CANoe. This is where the `statechange` test pattern is used, which executes the following steps:

1. The signals or environment variables given in the section `<in>` are set to specific values. In the example the “Mirror_2” signal is set to the value 1.
2. Waiting occurs for the prescribed time. This wait time is necessary so that the system is able to transfer the signals. The state changes can be made in the ECU, and the changed signals can be sent back to the tester. In the example waiting is set to 1000ms.
3. The conditions for signal values and environment variables are checked in the `<expected>` section. In the example the value of the “MirrorMotor2” signal must be greater than 5.

The set of input signals is often referred to as the *input vector*, and the set of prescribed output signals is known as the *output* or *target vector*. The input vector establishes the values that are set to stimulate the system. The output vector, on the other hand, defines a *condition* against which the actually occurring values are checked. If one of these conditions is violated the verdict of the test case is set to “fail”.

Test primitives and control functions can also be used like the test patterns in the sequence of a test case. Test primitives do also fulfill test functionalities like the test patterns. But they are simpler and do a more basic job than the more elaborated test patterns. Initializing a couple of signals with the `initialize` primitive is an example.

Control functions are used to influence the test run (e.g. wait for a defined time span) or to document the test sequence in the report.

It is possible to use multiple test and control functions (test patterns, test primitives) in a test case. The test and control functions are executed sequentially in the specified order. In the above example, an additional test primitive initialize may be used so that the actual test takes place in an assured initial state.

In XML test modules too, test cases should not be built upon one another, and the ECU under test should be reset to its initial state after each test. However, it is not so easy to avoid all dependencies between test cases in XML test modules.

2.4.2 Setting up XML test modules

XML test modules contain a collection of test cases. Just as in CAPL test modules they are organized in test groups, and their execution is documented in test steps. But the test steps do not need to be defined themselves, rather they are automatically output when the test and control functions are executed.

In XML test modules there is no flow control comparable to `MainTest`. It is the XML file that defines the hierarchy of test groups and test cases and their sequential order. Once the test module is started, CANoe automatically executes all test cases, one after another, in this order.

The CANoe user interface offers the option of selecting a subset of test cases for execution. Here too, the sequence of execution is determined by the XML file, but the unselected test cases are skipped over. In the test report these unselected test cases are noted as not executed ("skipped").

Example of a complete test module (including XML header):

```
<?xml version="1.0" encoding="UTF-8"?>
<testmodule title=" Window Lift System Test" version="2.1"
    xmlns=http://www.vector-informatik.de/CANoe/TestModule/1.15
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xsi:schemaLocation="http://www.vector-informatik.de/CANoe/TestModule/1.15
      testmodule_1.15.xsd">
  <description>Test the states of the window lift system</description>
  <sut>
    <info>
      <name>Name</name>
      <description>Central Locking System</description>
    </info>
    <info>
      <name>Version</name>
      <description>0.9 a3</description>
    </info>
  </sut>
  <engineer>
    <info>
      <name>Test Engineer Name</name>
      <description>Stefan Krauss</description>
    </info>
  </engineer>
  <testgroup title="Function Tests">
    <testcase ident="TC201" title="Open The Window">
      <description>Initiate window open procedure over CAN bus.</description>
      <initialize wait="2000" title="Initialize signals, ensure window closed">
```

```
        <cansignal name="LockRequest">0</cansignal>
        <cansignal name="EngineRunning">0</cansignal>
        <cansignal name="Ignition15">2</cansignal>
        <cansignal name="KeyUp">1</cansignal>
      </initialize>

      <initialize wait="100" title="Reset window up key">
        <cansignal name="KeyUp">0</cansignal>
      </initialize>
      <statechange wait="3000" title="Open window">
        <in>
          <cansignal name="KeyDown">1</cansignal>
        </in>
        <expected>
          <envvar name="WindowMotion"><ge>2</ge></envvar>
        </expected>
      </statechange>
    </testcase>
  </testgroup>
</testmodule>
```

Besides the `statechange` test pattern described above, the `initialize` test primitive is used here, too. It serves to initialize signals and environment variables, and it works similar to the `statechange` pattern, but without the condition section.

If a test case fails, the test cases that follow the failed test case are still executed, even though this always lead to failure of the entire test module. Within a test case CANoe behaves differently: As soon as a test function fails, the test case (not the test module!) is terminated. That is, the test or control functions that follow a failed test function are no longer executed. This behavior serves to accelerate the test flow.

2.4.3 Working with test functions

Many test patterns and primitives operate with signals. For stimulation of the system, messages are not generated directly on the bus, rather the input signals of an embedded Interaction Layer are changed in the remainingbus simulation. Exactly when this layer passes the modified signals to the communication bus depends on the selected Interaction Layer and is not directly influenced by the test function. In principle, the mechanism used by the test functions does not differ from work on the signal level in CAPL programs.

In defining signal values it should be noted that test functions always work with physical values. This means that the transferred signal values are converted to physical values with the help of the database (CANdb). Raw values of signals cannot be accessed directly from the test functions and checks (see 2.7.2).

The values of the results represent physical variables, therefore ranges should always be used instead of simple comparisons. This method will cover any inaccuracies that might result from the conversion of raw values. Strictly speaking, this is not a special issue with test functions, rather it should be observed whenever working with physical parameters, even in CAPL programs. When working with signals containing floating point numbers, ranges should always be used in comparisons.

2.5 Programming test cases in .NET test modules

To program a .NET test module, you need to specify a .NET file as the source file for the test module. The recommended programming language is C#.

Although there can be several test modules running in parallel, these modules are not executed in separate threads. Because of the CANoe internal concurrency architecture, you are not permitted to use synchronization primitives (such as locks) or start additional threads in a .NET program. Use of the System.Thread class leads to undefined behavior and may cause invalid results or program instability.

The .NET API is an additional way to specify and implement test cases in CANoe. It extends CANoe for the use with standard object oriented programming languages: e.g. C#, J# and VB.NET (C# recommended).

Features like extended capabilities to structure test programs, objected oriented concepts for reuse of test code, and debugging brings advantages over CAPL.

Standard development environments like MS Visual Studio can be used for test development.

The CANoe installation comes with a .NET demo containing detailed code examples.

The following chapters describe the application of .NET test modules only on a high level. For detailed information please refer to the separate document for the usage of .NET within CANoe.

2.5.1 Principle

The .NET Test API in CANoe can be used either for creating a pure .NET test module, or a so-called test library with independent test cases that can be called from an XML test module. In the latter case the test sequence is controlled via the XML module and there is no Main() method in the .NET code.

Not all features known from CAPL Test Feature Set are available in the .NET API but CAPL can be called from .NET using a CAPL test library.

The difference in programming .NET test modules and a test library in .NET, is that a pure test module is instantiated at the start of the test module and has a life-time until the Main() loop ends, A test library on the other hand contains independent test cases that are instantiated at the test case start. Out of this reason variables are not valid between different test cases in a test library.

2.5.2 Setting up purely .NET test modules

A .NET test module inherits from the class Vector.CANoe.TFS.Testmodule and consists primarily of the Main() method and the implementation of test cases. Test cases are defined by the custom attribute [TestCase]. The Main() method is the main program of the test module where the test cases are called in a specified sequence. It is required to override this method.

Structuring of a .NET test module can be done by using TestGroups, TestCase and TestStep analog to the structuring in a CAPL test module.

```
public class CentralLockingSystem : TestModule
{
  public override void Main()
  {
    TestGroupBegin("Test the windows", "Checks the window functions");
      SimpleWindowTest(true);  // test case - open the window
      SimpleWindowTest(false); // test case - close the window
       TestGroupEnd();
  }

  [TestCase("Open/close the window")]
  public void SimpleWindowTest(bool open)
  {
    // some test code
  }
}
```

2.5.3 Wait points

As in CAPL, it is possible to wait for various events in .NET test modules. Events could be e.g. timeout, bus signal value, I/O signal value, or a user-defined event. In .NET, wait points for signal types are resolved as soon as they are called if the condition is already met at that point in time.

Use the Execution class and it's Wait method to wait for a certain event:

```
Execution.Wait(50);              // Wait for 50 ms
Execution.Wait<LockState>(1); // Wait for the signal LockState to reach value 1
```

2.5.4 Type Library

The generation of the type libraries for the access to the database objects (signals, environment and system variables, messages) is done completely automatically by CANoe. This ensures a type-safe access to bus and I/O signals. CAN, LIN, and FlexRay signals are currently supported.

In case of name ambiguities of signals, the namespaces of the generated libraries (qualification of signals --> DB::Node::Message::Signal) can be adjusted in the global options dialog. After changing this setting the type libraries will be regenerated.

2.5.5 Event procedures in test modules

So-called event procedures are used to process events in .NET test modules. No event procedures may be executed when the test module is inactive (see also 2.3.5). Custom attributes are used to define an event procedure. The following types of custom attributes are available:

Handler for signal, environment variables and system variables:

[OnChange(typeof(LockState))] // Signal LockState

Handler to process a key event:

[OnKey("k")]

Handler to process a timer event cyclically:

[OnTimer(500)]

Handlers to process CAN message events

[OnCANFrame], [OnAnyCANFrame()]

For more information and code examples regarding the available attributes and their usage, please refer to the online help.

The .NET API only supports on change handlers for signal types (in contrast to CAPL that supports both on change and on update handlers). One reason is the performance aspect in .NET programs. And another reason is that the signal based tests work state-based.

The message based tests on the other hand work event-based.

2.5.6 Observation of system conditions

The observation of system conditions n parallel to the test sequence is realized with checks. There are three types of checks in .NET: Constraints, Conditions, and Observation. Constraints/conditions (as known form CAPL and XML) influence the verdict of the test module but Observation does not.

- Signal based checks can be easily setup by using the ValueCheck class:
  ICheck observeLockState = new ValueCheck<LockState>(1)
- It is also possible to let a user-defined handler decide about the check result:

```
    [Criterion] // Define custom criteria
    [OnChange(typeof(AntiTheftSystemActive))]
    [OnChange(typeof(LockState))]
    bool AntiTheftSystemCriterion()
    {
       …
    }
ICheck observeAntiTheftSystem = new Check(AntiTheftSystemCriterion)
```

- As in CAPL there are also predefined chechs for e.g. cycle time observation.
- Additionally the user gets the possibility to define its own check in .NET.

The checks should only be defined, started and stopped in the “main test program”, not in event procedures.

2.6 XML test modules versus CAPL/.NET test modules

XML test modules do not replace CAPL or .NET test modules, rather they supplement them as another description type that in many cases permits simpler definition of the tests. Stated quite simply, the important difference is that in XML test sequences the user only assigns parameter values, while in CAPL or .NET the test sequences are programmed.

CAPL/.NET and XML test modules have many similarities, but technically they differ in the aspects listed in the table below.

	XML test module	**CAPL/.NET test module**
Test case execution	Each test case maximum once	Each test case as often as desired
Execution order	Statically fixed by XML file	Set dynamically by call in the `MainTest` function
Execution control	Linear; test cases can be selected by the user at the GUI	Programmed in the `MainTest` function
Test groups	Statically defined by XML file → Fixed, one-to-one assignment of test cases to groups	Dynamically defined by code in `MainTest` → *Executed* test cases are assigned to test groups at run time
Test case definition	Defined by test patterns	Programmed in CAPL or .NET
Test report	Contains information on all test cases *defined* in the XML file	Contains information on the *executed* test cases

The type of test module used to describe the test cases depends on the specific testing task and the existing test environment (e.g. programming knowledge of colleagues, use of a test management tool). Strengths of the different description types are listed below and are intended to make it easier to choose between them. Nevertheless, it is possible to mix test cases written in different languages within one test module (especially to call test modules from a CAPL or .NET library in XML test modules).

Strengths of XML test modules:

- *Easy application*: Execution control, verdict handling and implementation of test sequences (test and control functions, especially test patterns) are an integral component and the user does not need to be concerned with them.
- *Generation*: Test modules in XML format are easy to generate using other tools (e.g. test generators, test management system).
- *Parameterization* of existing test sequences, the test sequences themselves are implemented in the test patterns and primitives.
- *No programming* necessary: Test modules contain just a static description of the test cases and the necessary parameters.

Strengths of CAPL test modules:

- *Maximum flexibility*: The CAPL programming language offers maximum flexibility in the setup of test modules, in programming flow control and in implementing test cases. In particular, it is possible to use this method to program complex test cases that would otherwise be impossible in XML test modules.
- *Complex flow control*: The flow control programmed in `MainTest` can dynamically control the calls of test cases, and it can make the calls dependent upon conditions that are checked during the test sequence.

Strengths of .NET test modules:

- *Full-grown programming language*: .NET test modules can be written in one of the programming languages that are supported by the .NET platform. Especially programming of complex test cases benefits from the elaborated constructs in such programming languages.
- *Powerful programming environment:* .NET test modules can be written using a powerful programming environment like Microsoft's Visual Studio. Many test programmers are used to these tools that let them write test code very efficiently.

Somewhat superficially one might say that XML test modules are well-suited for tests in which many test cases are performed with similar sequences but different parameters (especially if these test parameters are stored in a database, Excel file or test management system). CAPL and .NET test modules, on the other hand, are especially well-suited for programming complex test sequences that cannot be generated automatically.

In case of doubt, an initial attempt should be made to implement XML test modules with CAPL or .NET test case libraries, since XML permit easier and quicker definition of test cases with their test functions. The additional CAPL/.NET test cases provides the means to program the test cases with a more complex test execution flow.

2.7 Constraints and conditions

2.7.1 Principle

Constraints and conditions are checks that are performed in parallel to the specified test sequence. They are used to detect deviations of the test environment or of the ECU under test from certain target values during the testing. This may be a part of the actual test (the ECU is stimulated in the test sequence; the purpose of the test is to check whether the ECU violates an invariant condition at some time during the test). Or they may be used to ensure that the test environment operates properly during the test.

In principle the user could insert such checks in the test sequence, but then it would be necessary to program the same checks at a large number of points in the sequence. Constraints and conditions, on the other hand, can be implemented with very little effort. Furthermore, they are based on a concept that ensures that the conditions are really checked for conformance at all times during the test flow.

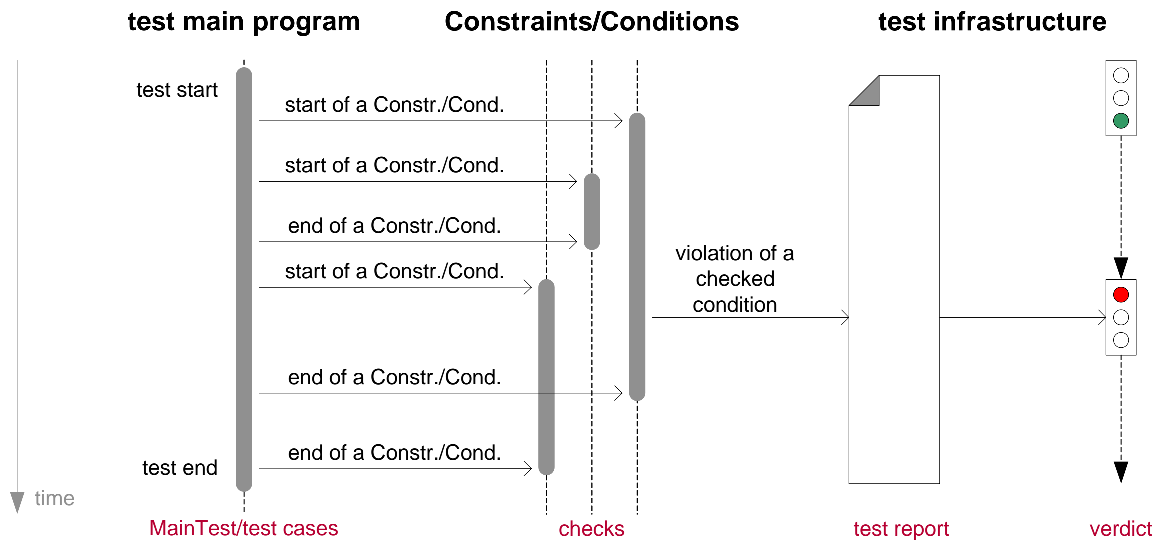

Figure 7: Constraints and conditions are executed in parallel to the actual test program.

The conditions to be checked are formulated and started in the test sequence. The test conditions are checked continuously in parallel to the test flow until the constraint/condition is stopped in the test sequence. The constraint/condition is automatically stopped when exiting the context (test case, test module, or test group in XML test modules) in which the constraint/condition was started. In CAPL, constraints and conditions should only be defined, started and stopped in the “main test program”, not in event procedures.

In a test module any desired number of constraints and conditions may be used, and they may also be simultaneously active. However, there is always just one defined test sequence, the so-called “main test program”.

Conceptually, constraints are distinguished from conditions as follows:

- *Constraints* ensure that the test environment is in a state in which a valid test may be performed. Constraints serve to check test assumptions but not to check the ECU under test. A constraint violation indicates a problem in the test environment, not a defect in the ECU.

  Example: During the test the ECU must maintain a stable supply voltage. If this voltage drops below a specified Minimum at any point in time, then the test is invalid.

- *Conditions* check assumptions related to the ECU under test. Violation of a condition therefore indicates a fault in the ECU.

  Example: During test execution the ECU must maintain the specified cycle time for messages that are sent periodically. The ECU does not satisfy its specification is this cycle time exceeds an upper or lower limit (e.g. due to overloading of the processor).

In application, the primary difference between the two types of checking is that the result is represented differently and must be interpreted accordingly. If the test fails due to a condition violation then the test objective has been achieved, i.e. the ECU was tested, albeit with a failed result. A constraint violation, on the other hand, states that the test could not be performed due to deficiencies in the test environment. In this case the test objective was not achieved, and it is not possible to reach a verdict about the quality of the ECU under test.

Whenever a violation of a constraint or condition is detected, it is noted in the test report, and the verdict of the currently active test cases (and thereby also of the entire test module) is set to “fail”. In a CAPL or .NET test module this does not affect the test sequence that is executed in parallel. Therefore, it is possible for a test to be conducted right to the end without any problems being detected in the test sequence yet still have a “fail” verdict because a constraint or condition failed. In a XML test module, on the other hand, the test case being executed is terminated, but not the test module.

The time point of the constraint/condition violation is indicated in the test report as the time at which CANoe recognized the violation. Precisely when this occurs will depend on the test condition that is used. To ensure that the report does not become unwieldy only the first occurrence of a constraint/condition violation in a test case is reported together with a supplemental statistical summary. This summary states how often the violation was detected and how often violation checking was performed[4]. However, the exact definition of these two numbers will depend on the test condition used and its implementation.

2.7.2 Use of predefined checks in CAPL and .NET

Checking conditions for constraints and conditions can be implemented in CAPL test modules by means of predefined checks in the Test Service Library (see also 2.8.1) or user-defined events implemented in CAPL or .NET. The checks provided in the Test Service Library are very easy to use and should be adequate in most cases. User-defined checking of conditions in CAPL/.NET should only be necessary in exceptional cases such as those described in section 2.7.4 below.

Four steps must be performed to use checks as constraints or conditions:

1. Create, parameterize and start the necessary check from the Test Service Library. The associated functions begin with the prefix `ChkStart_`. The functions each return a unique ID for the newly created check.
2. A constraint or a condition is generated by `TestAddConstraint` or `TestAddCondition`. The check created in Step 1 represents the constraint/condition statement, and its ID is now passed as a parameter. From this time point forward the constraint/condition monitoring that runs in parallel takes effect, i.e. “the check becomes semantic”.
3. Constraint/condition monitoring is removed again by `TestRemoveConstraint` or `TestRemoveCondition`.

The check is deleted by ChkControl_Destroy.

The two last steps are optional, since constraints/conditions are automatically removed at the end of the program structure in which they were created. Checks are automatically destroyed at the end of the measurement. Example in CAPL:

```
testcase TC ()
{
  dword check_id;
  check_id = ChkStart_MsgAbsCycleTimeViolation (StatusMsg, 90, 110);
  TestAddConstraint (check_id);
  …
  // The actual test sequence is located here
  …
  TestRemoveConstraint (check_id);
  ChkControl_Destroy (check_id);
}
```

The check is identified in the system via the ID returned when the check is created. In the Test Feature Set this ID is also referred to as the Auxiliary ID, since it is also used to identify the events sent by checks and the components they utilize. Therefore, from a technical perspective `TestAddConstraint (check_id)` signifies that from this point forward a constraint violation will be reported if an (Auxiliary) event of the ID `check_id` is received. As soon as the check detects a violation of the check assumption this event is sent by the check running in parallel to the actual test sequence. The check does not become a constraint until the event is linked via the ID.

Monitoring is in force between `TestAddConstraint` and `TestRemoveConstraint`. These two commands must be executed in the same context, i.e. in the same test case or in `MainTest`. The same check may be used more

---

[4] Conceptually the check runs in background. In actuality the check only occurs if events arrive that are relevant to the condition to be checked.

than once as a constraint/condition; there might be several monitored sections in a test case, for example. In principle, a check does not need to be created by `TestAddConstraint` just before its first usage, but this is recommended in the spirit of localizing the code and to make it more readable.

2.7.3 Constraints and conditions in XML

Constraints and conditions can be used in XML test modules, too. Checks from the Test Service Library are available as checks. Constraints and conditions are defined in a separate XML element at the beginning of a test case, test group or test module, and they then apply to that section of the XML test module. Example:

```
<testcase ident=”TC2” title=”Wiping Level 2”>
  <conditions>
    <cycletime_abs min="0" max="1100">
      <canmsg id="WipingStatus"/>
    </cycletime_abs>
  </conditions>
  …
</testcase>
```

Similarly, it is not necessary to create the check or constraint/condition separately, nor is it necessary to explicitly delete checks, constraints or conditions. Nevertheless, the basic functionality is the same as that of constraints and conditions in the CAPL test module.

Monitoring covers the entire execution of the test sequence defined by test patterns within this test case. If constraints and conditions are defined in test groups or in test modules, then monitoring covers the execution of all test cases and test groups they contain, including all of the sections contained in these test cases. Failure of a constraint or condition leads to the “fail” verdict of the test case just executed.

2.7.4 User-defined test conditions

User-defined events can be used to implement any desired constraints and conditions in CAPL. Essentially the procedure is the same as in implementation of user-defined wait conditions (compare 2.3.6). The actual check is implemented in the event procedures of the test module, which initiates a user-defined event (“text event”) if a violation of a constraint or condition is detected. Example:

```
on message Status
{
  if (this.Temperature > 100)
    TestSupplyTextEvent (“Overheated”);
}

testcase TC
{
  TestAddConstraint (“Overheated”);
  …
  // The actual test sequence is located here
  …
  TestRemoveConstraint (“Overheated”);
}
```

In this case the user-defined event represented by the character string is input and serves as a constraint or condition. Entry of a user-defined event as a constraint or condition means “make sure that this event does *not* occur”.

2.7.5 Influencing the test flow

Constraints and conditions do not affect the test flow in CAPL/.NET test modules. In XML test modules they lead to termination of the test case being processed. Generally the results of constraint/condition monitoring are not evaluated by the test program. If a constraint or condition fails, this impacts the verdict directly and is noted in the test report. No further steps are required in the program.

If the further course of a test sequence actually depends on the results of constraint/condition monitoring, then the status of a constraint or condition running in CAPL/.NET may be queried explicitly (`TestCheckConstraint`, `TestCheckCondition`). However, because of the dependency of the verdict on the current state it is possible to just query the verdict by `TestGetVerdictLastTestCase` and `TestGetVerdictModule`. One potential application would be to only start a very lengthy test if the verdict indicates a “pass”. However, such queries are only recommended in exceptional cases, and generally they should not be necessary.

2.8 Test Service Library

The Test Service Library is an integral part of the CANoe Test Feature Set. It contains prepared testing and value generating functions intended to simplify test setup:

- *Checks*: Checking functions that operate in parallel to the defined test sequence. Checks are used primarily in constraints and conditions (compare 2.5).
- *Stimulus generators*: These commands are used to generate a specific sequence of values for a signal or environment variable in order to stimulate an ECU.

Checks and stimulus generators must be generated and parameterized before their usage. With a generator, for example, the sink for a series of generated values and the cycle time of the generated values may be specified. Suitable control functions may be used to activate or deactivate, reset or clear a check or a generator. These control functions immediately roll back to the calling function, i.e. they do not wait. The check or generation is executed in background, in parallel to the continued test processing.

Once a check or generator has been set up, it may be used multiple times during test execution, i.e. it may be started and stopped multiple times. However, once a check or stimulus generator has been started, it may not be started a second time without having been stopped first. In general it is advisable to insert the call of `ChkControl_Reset` just before its reuse as a constraint or condition. It is possible to create multiple similar checks and use them simultaneously. A check or generator is identified by a unique number (ID) that is returned when a check or generator is created.

2.8.1 Checks

Checks can be implemented in CAPL simulation nodes and in test modules. They are used differently in these two cases. In CAPL simulation nodes, a violation of a test condition detected by the check is reported via a callback function. In CAPL and XML test modules the checks are implemented as constraints or conditions (see 2.7.2). Feedback via a callback function can also be used in CAPL test modules, but this is only advisable in very few isolated cases.

The Test Service Library has some functions for controlling checks (deleting, reinitializing, starting, stopping, finding the status) and for querying results information (e.g. statistics on detected fault events). These functions may be used in test modules, but they are not absolutely necessary. Generally, only the function for creating, parameterizing and simultaneously starting the check is used in test modules. The check is then used by entering the returned ID as a constraint or condition.

```
dword id;
id = ChkStart_MsgAbsCycleTimeViolation (MsgSpeed, 80, 120);
TestAddConstraint (id);
…
TestRemoveConstraint (id);
ChkControl_Destroy (id);
```

Checks report on their execution and detected faults in the test report. No additional programming work is needed here.

2.8.2 Stimulus Generators

Stimulus generators generate series of values that are applied to signals or environment variables. The cycle time is the time until the next value of the generated value series is copied to the environment variable or signal. That is, the cycle time sets the sampling rate for the generated value series and thereby the rate of change of the sink.

The generators repeat the specified value series periodically, so that a value series of any desired length can be created. The time required to execute one value series until its repetition is called the period. In most cases it is obtained from the specified value series and therefore is not usually specified explicitly when creating the stimulus generator (Example: A square-wave signal with 10ms High and 30ms Low level has a period of 40ms).

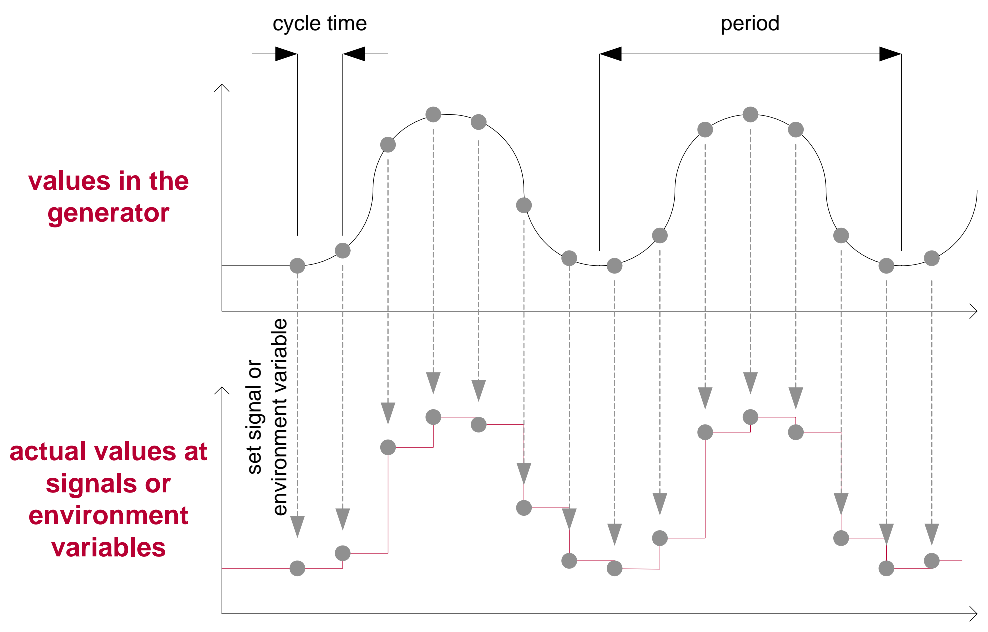

Figure 8: Stimulus generators for signals and environment variables.

The change of a signal does not necessarily cause an associated message to be sent to inform the entire system of the new signal value. An *Interaction Layer* (e.g. Vector IL that is supplied with CANoe) or a suitable CAPL program is responsible for sending out messages. The stimulus generator only sets the appropriate signal value. With messages that are sent periodically, the cycle time of the stimulus generator also does not need to match the cycle time of the message. The first cycle time determines when the signal is set according to the value series, and the latter cycle time determines when the message is sent out.

2.9 Test setup

In principle, test modules can be used both in the simulation setup and in the test setup. However, the test setup, as the name implies, is the preferred location for tests. In the test setup the test modules are not assigned to specific buses, because it may be assumed that tests relate to the total system. To make it easy to reuse prepared tests even after the simulation model is changed (e.g. a new version of the simulation model is introduced), the test setup may be saved and reloaded separately.

Similarly, the test setup offers a few advanced options such as consecutive execution of multiple test modules and merging of test reports from different test modules. These functions do indeed simplify working with multiple test

modules, but they do not represent a modularization concept for testing (distribution of tests among individual test modules). Test cases that belong together and are always executed together should reside within a single test module.

3.0 Test strategies

Tests can be developed and implemented according to different strategies. This chapter describes a few such test strategies and their implementation in CANoe. In practice, however, mixed forms are encountered frequently, and some test strategies may be implemented in parallel.

3.1 Protocol tests

Tests that primarily involve direct communication between tester and ECU to stimulate and evaluate the ECU under test are called *protocol tests*. This type of testing does not utilize any abstraction layers in the tester, which for example map signals to communication by messages, rather the tester communicates directly with the ECU under test. It thereby replaces even the communication protocol; hence its name.

3.1.1 Test concept

Protocol tests are described by a test sequence that is basically structured as follows: The tester stimulates the ECU under test, e.g. by sending messages or activating I/O lines, then it waits for the reaction of the ECU and finally checks whether the observed reaction matches the expected one. Depending on the results of the last test the tester might resume the stimulation, wait for the reaction, check it, and so on. These sequences may of course be more complex too. The “ping pong game” of action and reaction with subsequent checking of the reaction by the tester is a typical characteristic of this test concept.

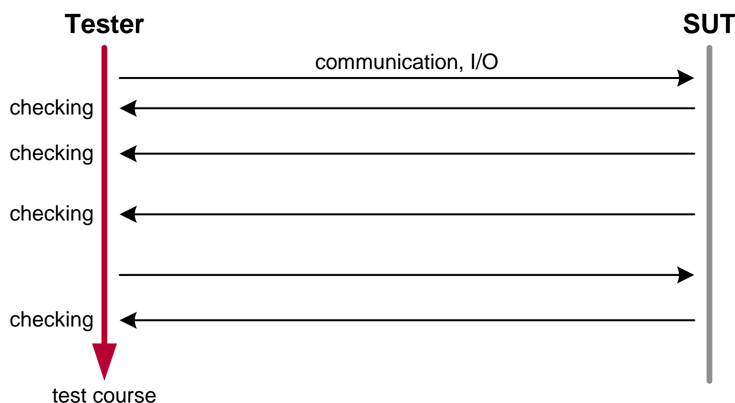

Figure 9: Schematic diagram of the Protocol Test.

The test sequence and the test conditions are very tightly interwoven. The test sequences simulate the specified behavior of the ECU precisely and check whether the ECU under test actually behaves according to specification. Even if the test conditions can only be formulated somewhat vaguely, e.g. if a range needs to be specified for an expected value, correct behavior of the ECU under test must be known somewhat precisely.

Example: An ECU for the windshield wiper system is to be tested. This involves connecting the ECU’s CAN bus to the tester as well as the various I/O lines, e.g. to drive the windshield wiper motor or the end switches of the windshield wipers. A test case might be formulated as follows: The tester sends a CAN message to the controller telling it that the windshield wiper has been turned on. Then the tester waits until the ECU activates the voltage for the windshield wiper motor. This must occur within a specific time span, or else the test case fails.

3.1.2 Implementation in CANoe

Protocol tests can be implemented very linearly in CAPL test modules using existing CAPL functions and the supplemental wait functions of the Test Feature Set. Typically the test sequence follows a simple scheme that repeats as often as needed:

- Output of a message to the ECU under test (Stimulus).
- Waiting for the ECU's response.
- Evaluation of the response if necessary.

In the same way as for messages, environment variables can also be set and read, in order to influence or measure the ECU's I/O lines, for example.

Excerpt from a test case as example of a protocol test implementation in CAPL:

```
message SeatSensorLeftRequest  msgLeftRequest;
message SeatSensorRightRequest msgRightRequest;
…
// TESTER  ---->>  SUT
output(msgLeftRequest);

// TESTER  <<----  SUT
if (1 != TestWaitForMessage(SeatSensorLeftResponse, 1000))
{
  TestStepFail ("", "Response message left not received within 1000ms.");
  return; // cancel test case
}

// TESTER  ---->>  SUT
output(msgRightRequest);

// TESTER  <<----  SUT
if (1 != TestWaitForMessage(SeatSensorRightResponse, 1000))
{
  TestStepFail ("", "Response message right not received within 1000ms.");
  return; // cancel test case
}
…
```

In this example the test case is terminated when an error occurs. However, this need not be the case. It is also conceivable that the test case might make further communication dependent on the output of a wait operation.

In principle, protocol tests can also be implemented by XML test modules. This involves the use of test patterns that work on the message level (e.g. `requestresponse`). Such possibilities are limited, however. In particular, the test sequence cannot be made dependent upon previous results, since classic programming capabilities such as conditions and loops are not available in the XML test modules.

3.2 Application Tests

Application tests is the name for tests in which the test sequence is primarily formulated on the level of an ECU application. In the tester an abstraction layer adapted to the system of the ECU under test is used for communication; this layer is often called the Interaction Layer. The tester uses it to communicate like an ECU application with the ECU under test via a system-conformant communication layer.

3.2.1 Test concept

Just as with the protocol test, in the application test the test sequence is used to stimulate the ECU, receive reactions and evaluate them. However, the actual communication is abstracted via an Interaction Layer. In the

CAN area this largely consists of reading incoming signals or setting outgoing signals. How and when these should be sent out and received via the CAN bus is determined and implemented by the Interaction Layer.

The use of an Interaction Layer simplifies the test sequences significantly, since otherwise all properties of the communication protocols would need to be emulated precisely in the test sequence. However it must be assumed that the Interaction Layer will operate error-free in the tester. Unfortunately, in many cases the abstraction layer also prevents errors in the ECU's Interaction Layer from being detected. Therefore, in this test strategy it is the application of the ECU that is being tested, not the components of the communication layers.

At many points the test sequences differ from those of protocol tests in one important detail: The tester does not wait for the arrival of the reaction, rather it waits a defined time and then checks the states in the Interaction Layer. The tester might set certain signals in the Interaction Layer, wait and then read signal values from the Interaction Layer for checking purposes. The wait time must be of such a length that it always allows sufficient time for the Interaction Layer to transmit the set signals to the ECU, the ECU can react to them, and the changed signals can be received in the tester. The time span used is therefore a maximum value that is not needed in every case. Nevertheless, the test sequence generally waits for the defined time, because the actual processes are hidden by the Interaction Layer.

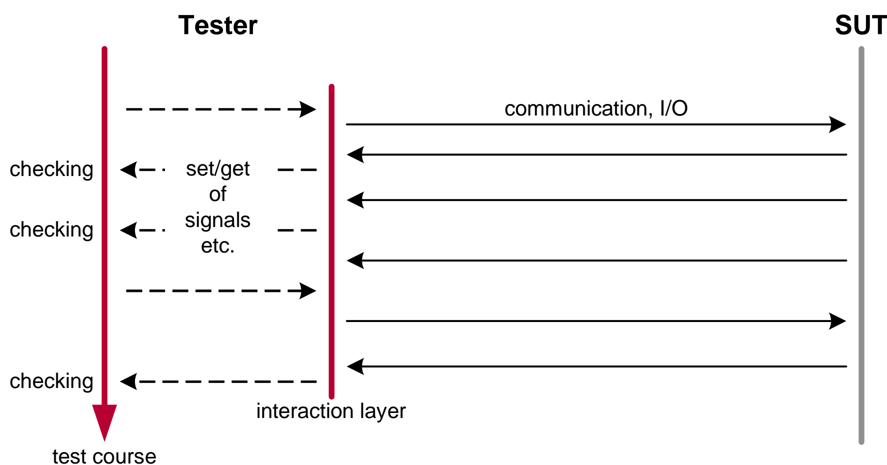

Figure 10: An abstraction of the actual communication is made in the Application Test.

Example: An ECU of the windshield wiper system is to be tested. A test case might consist of having the tester set the “Windshield wiper adjustment stalk” signal to “automatic” and the “rain sensor” signal to “strong rain”. The tester waits for a specific length of time and then reads the signal “Windshield wiper activity”. If this signal has the value “Windshield wiper active”, then the test case has been completed successfully. If the signal has the value “Windshield wiper inactive”, then the test case failed.

3.2.2 Implementation in CANoe

The Interaction Layer, the abstraction layer needed for the application tests, is also used for simulation of ECUs in CANoe. As soon as an Interaction Layer is integrated in CANoe's rest-of-bus simulation[5], the relevant signals may be changed in test modules.

XML test modules provide a number of test patterns for signal-oriented tests, and they are especially useful when larger quantities of signals and signal values must be covered in the test. In this case, the XML file could be generated from an Excel table or a database in which the relevant signal names and signal values are saved.

[5] Usually the Interaction Layer exists as a DLL and is configured by entries in the CANdb database. However, it may also be implemented in CAPL.

Example of a test case of an application test in XML:

```
<testcase ident="1" title="Test Application Door Lock">
  <initialize title="Ensure system is running" wait="200">
    <cansignal name=""></cansignal>
  </initialize>
  <statechange title="Lock door, check door locked response" wait="1200">
    <in>
      <cansignal name="Lock">1</cansignal>
    </in>
    <expected>
      <cansignal name="DoorLocked"><eq>1</eq></cansignal>
    </expected>
  </statechange>
</testcase>
```

The test only handles signals (writing and reading); it does not access bus communication directly. The timing of the sequence is not determined by bus communication either; rather it is based on fixed wait times. Wait times are set as the maximum allowable time within which the system must react to signal changes.

Access to signals is also provided in CAPL. Therefore application tests can also be formulated in CAPL, which presents itself as a solution especially for tests with complex sequences.

3.3 Invariants test

Tests may consist of operating the ECU under various limit conditions and continuously checking it for conformance to certain specified behaviors. These specified behaviors are conditions that must be maintained under all circumstances, and they are frequently referred to as “invariants”. Therefore we also call this type of testing the *invariants test*.

3.3.1 Test concept

For the devices or systems under test one can formulate conditions that they must fulfill in certain states. These conditions describe the desired properties of the ECU under test. A simple condition for an ECU on the CAN bus might be: Periodically sent CAN messages must be placed on the CAN bus at the specified cycle time in every operating state. However, such conditions could be much more complex, and they might exhibit dependencies to specific operating states.

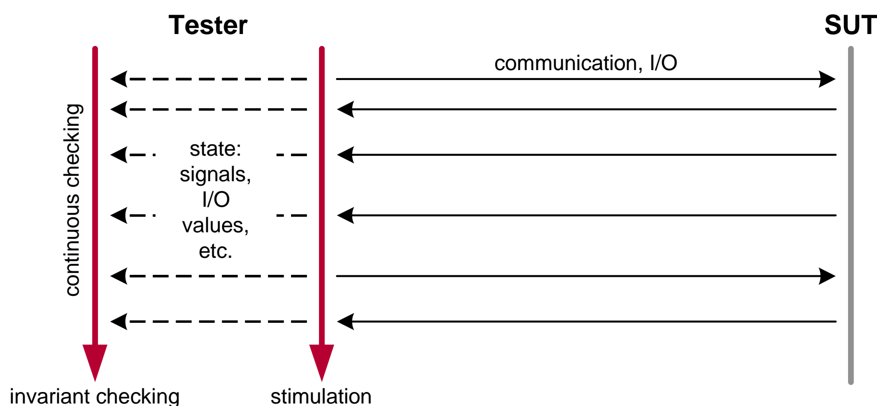

Figure 11: Principle flow in the Invariants test.

The actual test now consists of continuously monitoring for these conditions while the device under test is subjected to various situations (the latter might involve a “vehicle driving cycle”). What is important is that the ECU under test satisfies the required conditions in all operating situations. The test is passed if the conditions were not violated at any time.

Example: An ECU of the windshield wiper system is to be tested. Behaviors that might be monitored as invariant conditions include: Cycle times of periodic CAN messages, the value ranges of various ECU signals, and the reliability of the windshield wiper position (determined by test instrumentation at the ECU outputs). The test program then subjects the ECU to various situations. The test program executes a type of “vehicle driving cycle” with the ECU. For example, various user inputs at the control stalk and various “rain sequences” (information from rain sensor) are applied to the ECU. The invariants are tested continuously in parallel. The ECU passes the test if no condition is violated during the entire test sequence.

Stimulation and test conditions are independent of one another in this test strategy, and they can therefore be developed separately without any problems. In developing the stimulation what is important is to run through all classes of operating situations and all special cases if at all possible (code coverage, feature coverage, etc.). It is relatively easy to develop the stimulation, and random elements may be included (Basic approach: “Playing with the user controls), since the reaction of the ECU under test does not need to be evaluated.

The invariant test conditions can usually be derived from the ECU specification, since they essentially describe the ECU’s properties. The conditions usually check whether certain properties and values lie within their defined allowable limits. These conditions are genuine invariants, because they apply equally to all operating states. However, it is also possible to check specific properties in specific operating states, whereby identification of the desired operating state must be made part of the condition.

3.3.2 Implementation in CANoe

Conditions are available for implementing invariants tests in CANoe. These may be used in both CAPL and XML test modules (compare 2.5). In both cases the invariant is formulated as a condition that is active while the test case is being run. The actual test sequence consists primarily of stimulation of the ECU under test. Evaluation of the ECU’s responses finds very limited use here. The ECU is operated, not tested.

The ECU can also be stimulated via recorded communication, as shown in the following example with the `replay` test pattern:

```
<testcase ident=”1” title=”Check cycle time of status message”>
  <conditions>
    <cycletime_abs min=”80” max=”120”>
      <canmsg id=”Status”>
    </cycletime>
  </conditions>
  <replay title=”Replay signal curve” file=”curve.asc” />
</testcase>
```

The sequence saved in a CANoe log file can be stored by logging or recorded as a macro beforehand with the help of CANoe. Typically the ECU is first operated in a rest-of-bus simulation, and some of the functional sequences are played back there. The resulting communication, which may include changes at I/O lines, is recorded and used as a stimulus in the test case.

As an alternative, a log file may be fully generated by a simple tool or be created manually. The stimulus functions of the Test Service Library may also be used to drive the system (compare 2.8.2).

The desired control input sequence can also be programmed in CAPL for a CAPL test module or formulated as a sequence of test patterns for a XML test module. This is how a test can be created that contains application and invariants tests.

4.0 Interface to a Test Management System

Besides using CANoe as a standalone test tool, it is often necessary to interface a higher-order test system. CANoe assumes the role of the test execution tool in such a network. Embedding options are the main topic of this chapter. In individual cases details will depend on the specific problem and the tools and test approaches used.

4.1 Fundamentals of test management

The *Test Management System* concept is not rigidly defined. It is understood to cover systems that save and manage information for testing.

For example, individual test cases could be described in a test management system, and information could be saved regarding which test cases should be executed for which ECUs. Other organizational units are also feasible, e.g. configuration of different test types (short test, regression test, etc.). Moreover, the test management system could also manage the different versions of the test case descriptions.

After tests have been executed, a test management system could save the results. The system must then be able to answer queries of the type: “Has test case X already been executed for Version Z of ECU Y, and if so what were the results?” Statistical evaluations of the results data would make sense too, e.g. to reach a conclusion about the test status of a specific version of an ECU.

These are just a few examples of the possible functional scope of a test management system; others are conceivable. In particular, a test management system could also be integrated in a more advanced process tool, which might, for example, also produce and manage the interrelationships between a specification and the test cases.

4.2 Principle of the interface

In interplay with a test management system, however it is set up, CANoe plays the role of the test execution machine. Information may flow in two directions:

- The test management system provides the test cases/test modules for test execution in CANoe.
- The test results are read into the test management system.

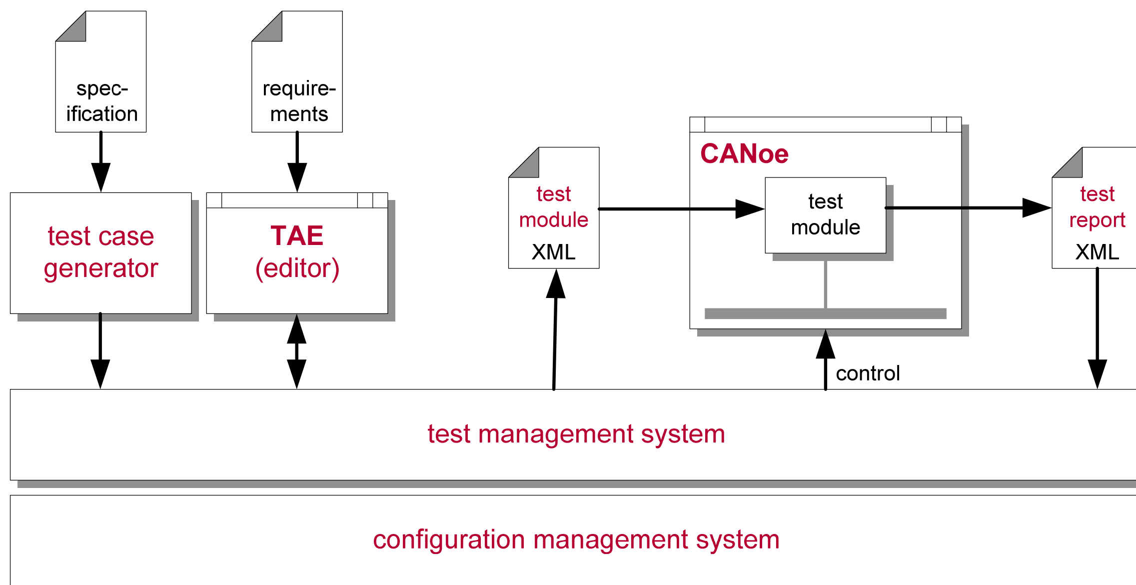

Figure 12: Interfacing a test management system to CANoe.

Furthermore, the test management system could control test execution itself by utilizing CANoe's remote control capabilities via the COM server.

Best suited as an interface to the test management system are the XML-based test modules and test reports, because the XML format is easy to generate in other applications and can be read into them and interpreted.

For this reason the XML test module has intentionally been kept simple. It is expected that the test management system will manage the volume of test cases and their assignment to specific ECU versions and variants. Once a test has been performed, the test management system creates a precisely-adapted test data record in the form of a XML test module.

In this scenario it is the task of the XML test modules to define the test cases to be executed and prepare the data needed for them. The actual test sequences are programmed in the test patterns. Therefore the XML file should primarily be understood as a parameter description and not as a test program. Accordingly, the distribution of tasks between the test management system and CANoe should be this: The test management system determines which test cases should be executed and with which data, while CANoe handles their actual execution.

In many cases, evaluation of the XML test report and thereby the reading back of test results is performed with the on-board tools of the test management systems. Programming capability in one of the script languages in common use such as JavaScript, Visual Basic or Perl is generally adequate, since these languages usually offer support for XML (XML Parser). XML test reports follow a rather simple structure, so it is possible to read and interpret them using simple XML tools.

5.0 Contacts

Germany
and all countries not named below:
Vector Informatik GmbH
Ingersheimer Str. 24
70499 Stuttgart
GERMANY
Phone: +49 711-80670-0
Fax: +49 711-80670-111
E-mail: info@de.vector.com

France, Belgium, Luxemburg:

Vector France SAS
168 Boulevard Camélinat
92240 Malakoff
FRANCE
Phone: +33 1 42 31 40 00
Fax: +33 1 42 31 40 09
E-mail: information@fr.vector.com

Sweden, Denmark, Norway,
Finland, Iceland:
VecScan AB
Theres Svenssons Gata 9
41755 Göteborg
SWEDEN
Phone: +46 31 764 76 00
Fax: +46 31 764 76 19
E-mail: info@se.vector.com

United Kingdom, Ireland:
Vector GB Ltd.
Rhodium
Central Boulevard
Blythe Valley Park
Solihull, Birmingham
West Midlands B90 8AS
UNITED KINGDOM
Phone: +44 121 50681-50
E-mail: info@uk.vector.com

China:
Vector Informatik GmbH
Shanghai Representative Office
Suite 605, Tower C,
Everbright Convention Center
No. 70 Caobao Road
Xuhui District
Shanghai 200235
P.R. CHINA
Phone: +86 21 - 6432 5353 ext. 0
Fax: +86 21 - 6432 5308
E-mail: info@vector-china.com

India:
Vector Informatik India Private Ltd.
4/1/1/1 Sutar Icon
Sus Road
Pashan
Pune 411021
INDIA
Phone: +91 9673 336575
E-mail: info@vector-india.com

USA, Canada, Mexico:
Vector CANtech, Inc.
39500 Orchard Hill Pl., Ste 550
Novi, MI 48375
USA
Phone: +1 248 449 9290
Fax: +1 248 449 9704
E-mail: info@us.vector.com

Japan:
Vector Japan Co. Ltd.
Seafort Square Center Bld. 18F
2-3-12, Higashi-shinagawa,
Shinagawa-ku
Tokyo 140-0002
JAPAN
Phone: +81 3 5769 7800
Fax: +81 3 5769 6975
E-mail: info@jp.vector.com

Korea:
Vector Korea IT Inc.
#1406, Mario Tower,
222-12 Guro-dong, Guro-gu
Seoul, 152-848
REPUBLIC OF KOREA
Phone: +82 2 807 0600
Fax: +82 2 807 0601
E-mail: info@kr.vector.com