SAE INTERNATIONAL

Arttest – a New Test Environment for Model-Based Software Development

2017-01-0004
Published 03/28/2017

Norbert Wiechowski
RWTH Aachen University

Thomas Rambow and Rainer Busch
Ford Motor Company

Alexander Kugler, Norman Hansen, and Stefan Kowalewski
RWTH Aachen University

CITATION: Wiechowski, N., Rambow, T., Busch, R., Kugler, A. et al., "Arttest – a New Test Environment for Model-Based Software Development," SAE Technical Paper 2017-01-0004, 2017, doi:10.4271/2017-01-0004.

Abstract

Modern vehicles become increasingly software intensive. Software development therefore is critical to the success of the manufacturer to develop state of the art technology. Standards like ISO 26262 recommend requirement-based verification and test cases that are derived from requirements analysis. Agile development uses continuous integration tests which rely on test automation and evaluation. All these drove the development of a new model-based software verification environment. Various aspects had to be taken into account: the test case specification needs to be easily comprehensible and flexible in order to allow testing of different functional variants. The test environment should support different use cases like open-loop or closed-loop testing and has to provide corresponding evaluation methods for continuously changing as well as for discrete signals. In a joint project of RWTH Aachen University and Ford, a new tool, Arttest, has been developed for testing model-based software. The tool uses a domain specific language to specify the tests. It offers different test evaluation methods for automated open- and closed-loop testing and reactive testing. It automatically executes the tests, evaluates the outputs and generates summary reports indicating passed tests and errors found. The paper presents the tool and its various unique propositions such as domain specific test language, the evaluation properties and other features like open-loop and closed-loop capabilities.

Introduction

Model based software development methods can have significant savings in time and cost over traditional design methods [1]. One advantage is to have a model that can be used throughout different phases of software development. The development typically starts with the capturing and the analysis of requirements. Afterwards the functionality is implemented in an executable graphical model. This model can then be used to automatically generate production code. In order to avoid late detection of errors, the models should be tested systematically for their compliance with the functional requirements. For this purpose functional tests have to be specified with predefined input stimuli. According to [2] functional testing is the task to evaluate the generated outputs of software that are generated in response to selected input and execution conditions. A test case contains a set of test inputs, execution conditions and the predicted results developed for a certain test objective [2]. If tests are done open-loop then the interface, including all the inputs and outputs of the model, can be used to specify the test case. The test case defines the initial conditions, the input changes in the test steps and the acceptance criteria that are used to evaluate whether the simulation results comply with the expected results (see Figure 1).

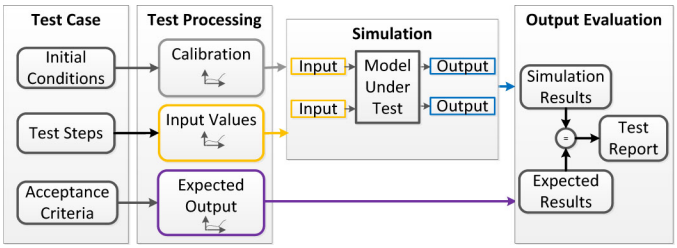

Figure 1. Functional Open-Loop Test Environment

If the model under test needs to be tested and simulated in a closed-loop together with a plant model, then parts of the input data for the model under test are generated by the plant (see Figure 2). These signals cannot be changed directly by the test environment, but the model under test reacts to value changes from the plant. For this use case it is important that the test case specification allows the change of input or expected output values based on conditions or events that occur during simulation.

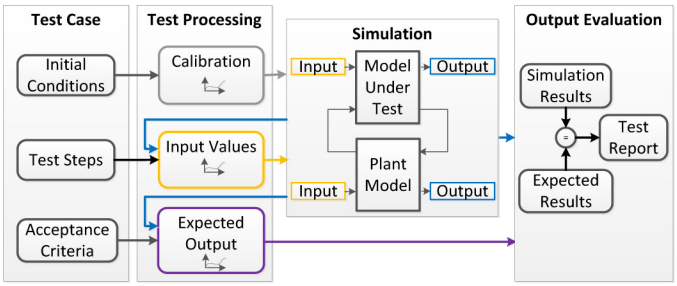

Figure 2. Functional Closed-Loop Test Environment

To be able to write tests efficiently the test case specification needs to be easily comprehensible and flexible. Engineers should be able to understand how signals are changed and what parts of the function are tested. Further requirements to such a tool are:

1. Test preparation
   a. The tool should be able to access all input and output signals as well as internal signals and configuration parameters of the model under test.
   b. The test harness for the model under test should be generated automatically.
2. Test specification
   a. The tool should be able to specify signal changes at certain points in time.
   b. Signal changes for the inputs and expected output based on conditions or events during test execution should be possible.
   c. It should be possible to define input signals that are changing continuously with a certain pattern over time (e.g., sin, cos, sawtooth, and ramp) or based on arbitrary functions including references to other signals.
   d. It should be possible to change the calibration parameters of the model under test.
   e. Evaluation methods and tolerances are needed for discrete as well as continuous signals.
   f. Changing values of calibration parameters and the definition of different acceptance criteria for different calibrations should be possible.
   g. It should be possible to define tolerance-based evaluation regions with specific start and stop intervals and change those tolerances during test execution dynamically if certain predefined events occur.
   h. To support modularity, it should be possible to reuse predefined test sequences.
   i. The import of measured data, e.g. from test drives with a vehicle, should be possible.
3. Test execution
   a. It should be possible to execute one or a sequence of tests automatically.
   b. The definition of a sequence of tasks before and after test execution should be supported to allow data pre- and post-processing.
4. Report generation
   a. It should be possible to generate a test report with information about the verified model, an overview of the executed tests and the verification results with a statement whether tests are passed or failed.
   b. The report should also include graphical representations of the input data and the evaluation results.
5. Requirement traceability
   a. Requirements import and linking of requirements to test should be supported in order to allow requirement traceability and coverage reporting.

Ford is in need of a tool which fulfills these requirements for software tests. The solution developed in cooperation with the RWTH Aachen University has the potential to address all of these needs. In the following sections we will discuss the proposed solution including several tool features, the developed domain specific test language as well as the evaluation methods and requirements traceability. A case study within a Ford project will demonstrate the usability of the tool and the last section includes a discussion and conclusion.

Arttest Concepts

The success of testing activities is significantly impacted by the methods used to specify test cases. The specification of test cases should be intuitive, simple and effortless. Resulting test cases have to be maintainable and reviewable.

In order to fulfill these requirements and the recommendation of the ISO 26262 [3], we decided to create a new language from scratch. The core of Arttest, a stand-alone test-tool, is a newly developed domain specific language which has been specifically designed for signal-based test cases. With closed-loop testing in mind, the language has been tailored to not only enable black-box but also white-box testing. In particular it is possible to override or evaluate arbitrary internal signals of a system. Existing solutions usually focus on the interface of a SUT, i.e. the direct inputs and outputs of the system. The ability to override internal signals is achieved by automatically enriching the test harness with specific blocks. Depending on timing constraints, which are specified in a respective test case, either the original signal values from the model or the signal values specified by the tester are used during the simulation. With the domain specific language from Arttest it is also possible to react to events that might occur during the simulation of a model. For instance, if a certain event occurs, the tester might want to change the input stimuli, start overriding a specific signal or change the evaluation tolerances. This entails several challenges, e.g., the duration of a test case and thus simulation may vary depending on certain event occurrences.

One general aim during the development of Arttest was a high degree of automation throughout the complete testing process. For that reason, the model is analyzed to detect the inputs and outputs, the internal signals and the data types of all signals. This enables the automation of the harness generation and is also the basis for handling internal signals correctly and automatically. Figure 3

illustrates a common structure for testing in closed-loop. Due to the fact that signals between the plant model and the model under test are all internal signals which are generated during simulation (see Figure 2), it is natively not possible to stimulate the input from the model under test with a test case specific signal. Another common way of testing together with a plant model, e.g. in an open-loop, is to integrate one or multiple plant models directly into the model under test (see Figure 4).

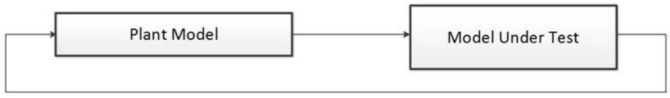

Figure 3. Closed-loop testing together with a plant model

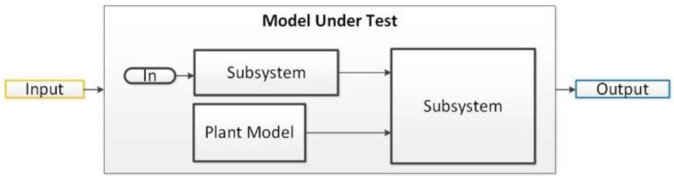

Figure 4. Open-loop testing together with a plant model

Both variants require a modification of the model to make the signals available for testing. In principle, internal signals that need to be overridden are enriched with a block which is able to switch between two signals based on its activation signal (see Figure 5).

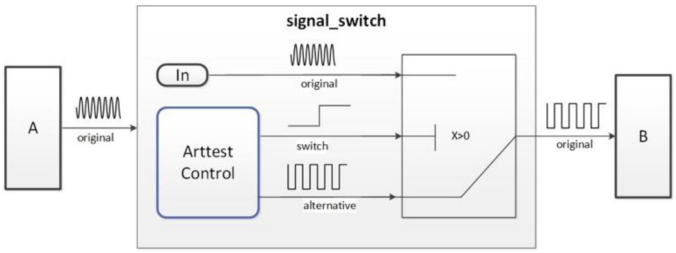

Figure 5. Principle of switching signals

The activation signal is controlled by a dedicated block which evaluates the previously specified conditions for overriding a signal during simulation and controls the switching. As soon as a condition is satisfied, the simulation is paused and this block communicates with Arttest to trigger a complete recalculation of all stimuli and reference signals based on the current simulation time and state of the model. After refreshing the stimuli, the simulation is continued. Since the conditions may also include other signals, each signal which is included in a condition is enriched with a block, that logs the current value and makes it available for condition evaluation as well as for an evaluation of the internal signal (see Figure 6).

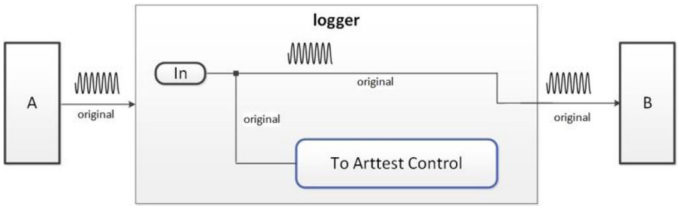

Figure 6. Access to internal signals

Apart from the textual specification via the domain specific language, Arttest enables the visual test case specification by specifying the resulting signal curves within an editor. It is possible to compose input stimuli and reference signals from predefined elements, e.g. a ramp, and manipulate respective control points on a signal curve via drag&drop.

When compared to a textual representation, a visual representation of a test case has distinct advantages and disadvantages. One advantage is that the user does not need any knowledge about a syntax of a language. In addition the direct visualization of the specified signals helps understand the timing relations between two or more signals. The disadvantage of a purely visual representation is the lack of context information to support the readability and maintainability of a test case, e.g. by writing comments within a textual test case. Furthermore within a textual representation it is possible to efficiently reuse parts of a test case, i.e., by copying and pasting the relevant lines.

In order to leverage the advantages of both techniques the visual specification synchronizes with the textual representation without losing comments or custom text formatting. In a similar way changes in the textual representation also update the visual representation of the test case.

Other supporting features of Arttest include error checking, warnings, hovering for meta-information and content completion in the textual editor. A screenshot enriched by the key components of Arttest, which will be further elaborated in the following sections, is shown in Figure 7.

Test Language

To increase the readability of test cases, the syntax and semantic of the language constructs are inspired by the vocabulary regularly used during requirements engineering in the automotive domain and also by conventions known from Matlab®/Simulink®/Stateflow®.

A minimal test case is shown in Listing 1. It consists of four main sections. The *Initial Conditions* section is used to specify the initial values for each stimulus, as well as the expected value and the corresponding tolerances of each output signal. As long as the value remains unchanged, the initial value will be hold For many test cases a continuously repeating stimulation pattern on a subset of inputs is required, e.g. a rolling counter from 0 to 255 (see line 14, Listing 1). For specifying such signals, the section *Continuous Signal Change* can be used. The desired sequences of signal changes, which make up the test case, are specified in the *Test* section within a *Step*. Depending on the test case it is necessary to specify simultaneously changing signals or a sequence of signal changes. The default behavior is a sequential execution of signal changes. If a simultaneous execution of signal changes is desired, the actions can be encapsulated by the keyword *simultaneously*. The concurrent use of sequential and simultaneous signal changes within a *Step* eases the specification of a correct timing. To further ease the specification of a correct timing, each *Step* is coupled to exactly one *Criterion* within the *Acceptance* section. Changes to reference signals specified in a *Criterion* are executed as soon as its corresponding *Step* is executed and are not allowed to have a longer duration than its corresponding step. A typical use case is to drive the model under test to a desired state in a dedicated step before simulating a concrete behavior in a different step. By allowing to import already specified steps and corresponding criteria from other test cases, the separation into steps facilitates readability, reusability and modularity.

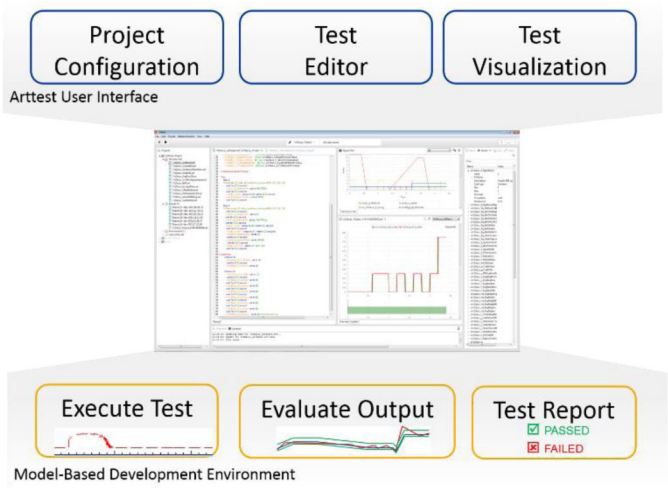

Figure 7. Arttest

```
Initial Conditions:
    Inputs
        <in_signal>  [0]
    Outputs
        <out_signal> [0] (ON;-0.1;0.1;1;.5;0.1;0.001)
    Internal Signals
        <int_signal> [0]
    Calibratables 1
        <period_1>   [2.5]
    Calibratables 2
        <period_1>   [4]

Continuous Signal Change:
    <in_signal_2> varies with sawtooth (0,255,period)

Test:
  Step 1:
    Import from "file.art"
  Step 2:
    % specify changes to input stimuli

Acceptance:
  Criterion 1:
    Import from "file.art"
  Criterion 2:
    % specify changes to reference signals
```

Listing 1. Test case specification

Specification of Signal Changes

Signals in the test case are described as known from Simulink using the angle brackets <*SIGNAL_NAME*>. Square brackets are used for value settings [VALUE] and curly brackets for time parameters {TIME}. Setting a constant value of 1 as input stimuli for the input *in_signal* is specified as:

<in_signal> set to [1]

To ease the specification of correct timings, a wait command can be used. For example, the following command delays all further actions for 3 seconds.

wait for {3} seconds

Another possibility to specify a delay is with an after construct which delays a single action by a specified amount of time. This is especially useful if signal changes shall be executed simultaneously since a wait command would have no effect in the following example.

```
simultaneously (
    after {1} second <in_signal> set to [0]
    after {1} second <int_signal> set to [2]
)
```

Ramping a signal to a target value within a specified amount of time e.g. to 20 within 3 seconds, is defined as:

<in_signal> ramps to [20] within {3} seconds

Multiple ramps can be written in a short form by enumerating the target values and the ramping time. Example:

<in_signal> ramps to [20,10,15,-5] within {3,1,2,1} seconds

Examples

The signal pattern specified in the *Continuous Signal Change* section lasts as long as the *Test* section. On the assumption that the imported step has a duration of 18 seconds, *in_signal_2* is the signal curve as shown in Figure 8 for the first calibration. The first parameter defines the lower value, the second parameter defines the amplitude and the third parameter defines the period.

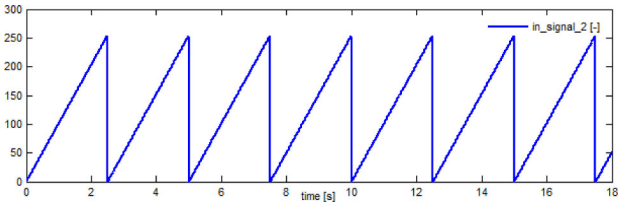

Figure 8. Sawtooth signal

Listing 2 is an example for the specification of signal changes. The example has three input stimuli and translates to the signals illustrated in Figure 9 to Figure 11. All signals have an initial value of 0.

```
Step 1:
    wait for {1} seconds
    <input1> ramps to [-1;3;-2;0] within {2;2;1;1} seconds

Step 2:
    simultaneously (
        after {0} seconds <input1> varies with function
          [sin(time)] for {10} seconds
        after {1} second <input2> set to [3]
        after {4.5} second <input2> set to [-1]
        <input3> varies with function [tan(input1)]
    )
    <input2> set to [3]
    wait for {1} second
```

Listing 2. Specification of signal changes

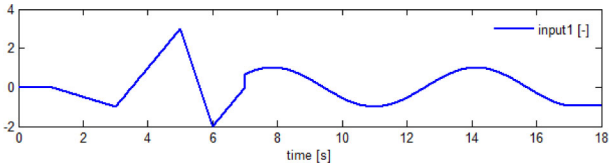

Figure 9. Signal curve of input1

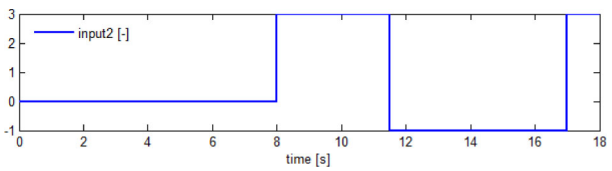

Figure 10. Signal curve of input2

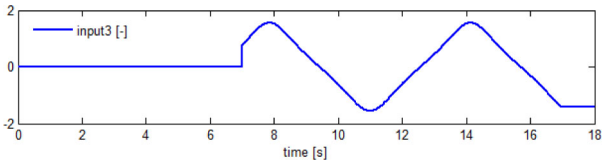

Figure 11. Signal curve of input3

Reference signals are specified using the same commands on the output signals. In addition to the commands for changing signal values there are also commands to change the tolerances.

Internal Signals

Manipulating internal signals during a simulation is useful for interacting with plant models but also for many other use cases e.g. for robustness testing. Depending on the test case and the reusability of test steps, it is important to not only replace the generated internal signal by an alternative signal but rather to switch between the signals at specific points in time. To comply with this requirement, the test language introduces the statements:

override <SIGNAL> on
override <SIGNAL> off

A typical use case is to override internal signals when a predefined condition is fulfilled, e.g. the system reached a desired state.

Event Handling

Reacting to events that occur during simulation is another important use case for testing. It may ease the specification of a concrete test case since the tester does not need to calculate exact timings of signal changes and the expected behavior of the model under test. If it is possible to specify a condition which triggers the signal changes, the tester can wait for the condition to be fulfilled. The test environment supports this use case with the following statement:

```
when [CONDITION] during {TIME} seconds then (
    override <int_signal> on
    wait for {1} second
    <int_signal> ramps to [0] within {1} second
    override <int_signal> off
)
```

Listing 3. Reacting to events

Listing 3 demonstrates the overriding of an internal signal *int_signal* as soon as a condition is satisfied. Further specified signal changes will only be executed when the condition is satisfied. To execute the signal changes which are contained within the *when* construct, it is necessary that the condition is satisfied within the given *TIME* parameter. Further signal changes will only be executed if the when construct has been processed or if the given condition is not satisfied within the given time. This guarantees the termination of a simulation. In general, conditions are expected to be satisfied during simulation. Therefore, for creating a preview of the signal curves, Arttest assumes the condition to be satisfied at the end of the waiting time. To enhance the preview, the tester can choose an expected point in time for each condition to be satisfied to create a different preview of the signal curves.

Special about the event handling technique is the condition that can be used. Due to the internal signal handling of Arttest, the condition can be an arbitrary expression evaluable by Matlab®. It can for example include calls to the Matlab® API, references to the currently simulated value of arbitrary signals (inputs, outputs and internal), constants and Matlab® scripts. The satisfiability of the condition depends always on the simulation time and the generated signal value used within the condition. For that reason, the concrete input stimuli cannot be calculated in advance. Thus, as soon as the condition is satisfied, the simulation will be paused until all stimuli have been recalculated by Arttest. This can not only influence the input stimuli and the reference signals, but also the length of a test case. To ensure a correct termination of the test case, Arttest analyzes the worst case duration based on the conditions and stops the simulation beforehand in case the test case simulates in less time.

Evaluation

Arttest provides two reference signal based evaluation methods, the sequence evaluation and the tubular evaluation. The sequence evaluation targets the evaluation of signals with discrete value changes, whereas the tubular evaluation targets the evaluation of continuous-valued signals. The evaluation methods are based on a specified reference signal and corresponding tolerances, which may vary during test evaluation according to the specification. Since it is possible to use the different evaluation methods on the same reference signal, tolerances are specified using a tuple of values as shown in the *Initial Conditions* section in Listing 1. Both tolerances and reference signals are specified within the test case based on the syntax known from stimuli specification. Each reference signal relates to a simulated signal, either an output of the model under test or an internal signal, being generated by Matlab®/Simulink® during test execution. If the simulated signal corresponds to the reference signal with regard to the according tolerances, the evaluation passes. Otherwise, the evaluation fails. Although the reference signal is as long as the test case, the values of a simulated signal might only be of interest for a certain time period. Consequently, Arttest allows to disable and enable evaluations at any time in the test specification, causing the simulated signal not to be compared to the reference signal and the tolerances if the evaluation is disabled.

Sequence Evaluation

The sequence evaluation is meant to evaluate whether a signal corresponds to a specified sequence of discrete values. For instance, to express that an algorithm shifts from gear four to gear two without shifting to gear three in between. Figure 12 shows a visualization of the sequence evaluation.

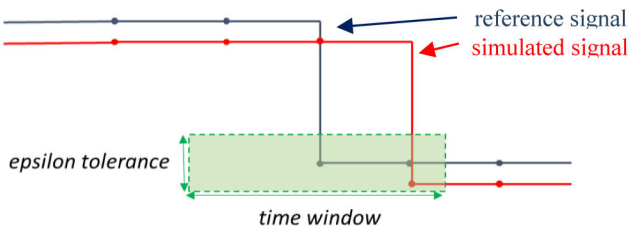

Figure 12. Sequence evaluation visualization

The tolerances describe a time window around every value change of the reference signal during which the value of the simulated signal has to follow the reference signals change. To cope with rounding of IEEE 754 [5] floating point values generated by the model under test, an epsilon tolerance specifies a value tolerance in the upper and lower

direction of the reference signal values. Thus, for the sequence evaluation to pass, it is required that the sequence of pairwise value changes of both the reference and the simulated signal are such that the value the simulated signal changes to lies within the current tolerances around the reference signal's value.

Tubular Evaluation

Although tubular evaluations are common to evaluate signals [6] and used in existing tools [4], the presented approach differs with regard to its flexibility and expressiveness.

There are currently two variants of the tubular evaluation supported by Arttest, which differ with regard to the extent of the tolerances being specified, see Figure 13.

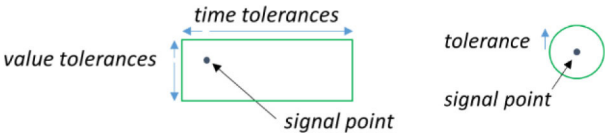

Figure 13. Tolerance variants for tubular evaluations

The tubular evaluation using rectangles allows to specify four tolerances around a signal point, time tolerances to the left, right and value tolerances in upper and lower direction. The second tubular evaluation variant uses a single absolute value as tolerance which specifies a circular region around the signals points.

By shifting the geometric objects defined by the tolerances along the signal points of the specified reference signal, an acceptance region for the tubular evaluation method is generated. By connecting the outermost points of neighboring geometric objects, it is ensured that a single coherent acceptance region is generated. Figure 14 shows the construction of an acceptance region for both tubular evaluation variants.

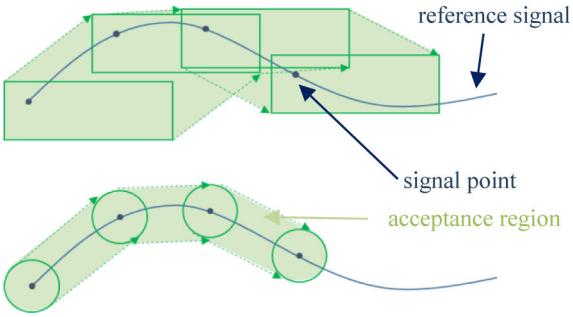

Figure 14. Tolerance based acceptance region construction

Since tolerances are in general larger than the used sampling time, connecting neighboring objects seems not necessary to create coherent regions. However, in case of value changes bigger than the value tolerances within a single time step, as shown in Figure 15, connecting neighboring objects is necessary. Otherwise, simulated values in between the two objects would cause the evaluation to fail since they would not be inside an acceptance region.

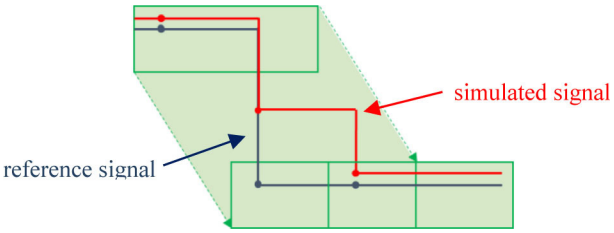

Figure 15. Interpolation between geometric objects

The tubular evaluation method supports varying tolerances. These cause the acceptance regions to be of different size. Figure 16 shows how varying tolerances may influence the constructed acceptance region. Although arbitrary tolerances may change during the test, the shifting of the geometric objects and the interpolation procedure applies.

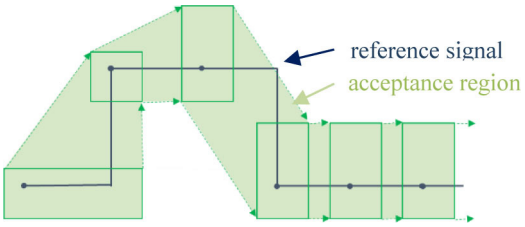

Figure 16. Varying tolerances

Assertions

Besides the introduced evaluation methods, Arttest enables the user to specify user-specific expressions, called *assertion,* such as *"signalA > 2*signalB"*. Assertions are evaluated by Matlab® if they become active during test execution and are thus plain Matlab®-expressions with the signal names used in the Arttest test specification as variables. Although it is possible to call arbitrary Matlab®-functions using assertions, the evaluation of the expression should return a value which can be interpreted as a logical value indicating that the assertion passed or failed. Similar to the evaluation methods, the assertions can be enabled or disabled based on conditions using the *when* language construct introduced in the section about the test language. Similar to evaluations, a failed assertion does not affect or stop a test execution. However, the results of assertions are not mixed with evaluation results. Assertion results are presented separately in Arttest and generated reports.

Reporting

The results of the evaluations and assertions are persisted in files which can be opened and inspected with Arttest. Besides overviews on the test data, visual representations and detailed views of the recorded signals and acceptance regions are available to the user. To provide the possibility to exchange the results with third parties, Arttest supports a HTML-export of test results. Configuring the HTML-export to contain, for instance, only evaluated signals helps the user to improve readability of the report depending on the reports purpose.

Requirements and Traceability

According to ISO 26262, functional test cases have to be performed based on requirements. Moreover, traceability between requirements, test cases and test results shall be provided. To address these topics,

Arttest features the import, export and management of corresponding data. Requirements and traceability data is managed using a central database. Since Arttest has to be integrated into the existing development process, standardized excel templates are chosen as data exchange format.

Besides the import of the requirements, Arttest supports the creation, modification and import of data describing test objects and the automatic creation of corresponding test case files. To provide traceability between the test objects and the corresponding requirements, Arttest features the creation of links between requirements and test objects. Subsequent test executions produce test result data. Arttest creates implicitly links between the test results and the corresponding test objects.

Since test object and test result data can be created by Arttest, it features the export of corresponding Excel files to the database. Furthermore, link documents can be exported, describing the links between requirements and test objects or between test objects and test results.

Link documents describe links to be imported and interpreted by the database. Thus, the linking has to be performed using identifiers known to the database and Arttest needs to be provided with these identifiers to create appropriate link documents. Consequently, objects created by Arttest, such as test results or test objects which were not imported, must be exported to the database in a first step. The database then creates identifiers for the objects and only afterwards can the objects be re-imported by Arttest.

The described workflow allows changes of data being imported or exported from the database and Arttest, which might produce inconsistencies. Thus, Arttest allows the user to reimport data in order to update, for instance, changes on the requirement specification or test objects. This is done by creating a diff view letting the user decide which changes to be applied on the Arttest side, e.g. deleting an existing requirement which is not part of the newly imported file.

Simulator

In previous sections we have considered the default testing procedure, i.e., a previously specified test case has to be executed and evaluated. However, developing a test case is an iterative process. For instance, the tester might specify the input stimuli or acceptance criteria incorrectly in an early version of a test and only after the simulation and evaluation have been finished he notices an error by analyzing the simulated output signals. In this use case a significant time overhead is introduced with the default procedure as in each iteration the initialization of the workspace, the compilation of the model and the simulation and evaluation of the revised test case is necessary. In order to eliminate this time overhead during an iterative development of a test case, Arttest features a widget-based simulator. Inputs and internal signals can be connected to typical control elements, e.g. sliders, while outputs and acceptance regions are displayed in signal plots. Arttest starts an infinite simulation of the model, the tester can alter the stimuli and acceptance regions using the widgets and is provided with a real-time visualization of the simulated output values. Additionally, the speed of the simulation time is controlled so that one second in real time corresponds to one second within the simulation. For debugging purposes, it is also possible to decelerate the speed of the simulation, to skip a specified amount of seconds or to set break points. Break points make use of the same technique as the event handling, which means that arbitrary conditions may be used to define break points. Furthermore the sequence entered into the simulator is recorded and may be used to automatically generate a first version of a test case.

The simulator may also be used to support the development of a model. For instance, a developer might be working on a subsystem of a model and may have extensive knowledge about this particular subsystem but only limited knowledge about the other parts of the system. In this case the simulator within Arttest may be used to rapidly test the behavior of the unknown parts of the system, eliminating the necessity to manually infer their behavior by looking at the implementation. In addition the simulator may be used to test the interaction of the developed subsystem with the rest of the model for early verification purposes.

Raw Data

Another important topic is the integration of raw data recorded e.g. from a HIL environment or during a test drive in a real car. Typical use cases are to evaluate this data and generate reports or to run a simulation using the recorded data as stimuli. For that reason, Arttest has two options to deal with raw data. The first is to analyze and convert the data into an editable test case. The second option is to use the raw data directly as stimuli for the model under test. In both cases Arttest allows the visualization of the signal curves prior to the simulation.

Case Study

In this section we want to demonstrate the usage of the tool and how it may be used to specify and execute functional tests. As an example we will use the start-stop control feature that reduces fuel consumption by shutting down the engine in standstill phases (e.g. while waiting at a traffic light). The following two requirement examples describe conditions that need to be fulfilled to allow an engine shutdown.

An inhibit stop request shall be generated after a manual key start until the vehicle speed is greater than a calibratable scalar (cal: STOP_INHIBIT_SPEED).

The engine stop request shall be generated by setting the <engine_command> to STOP after a calibratable period of time (cal: STOP_DELAY) if all of the following conditions are true:

1. The <clutch_pedal> state is RELEASED;
2. The <vehicle_speed> is below a calibratable threshold (cal: MAX_STOP_SPEED);
3. The <gear_lever> is in neutral.

The request shall be cleared if one of the above conditions is violated.

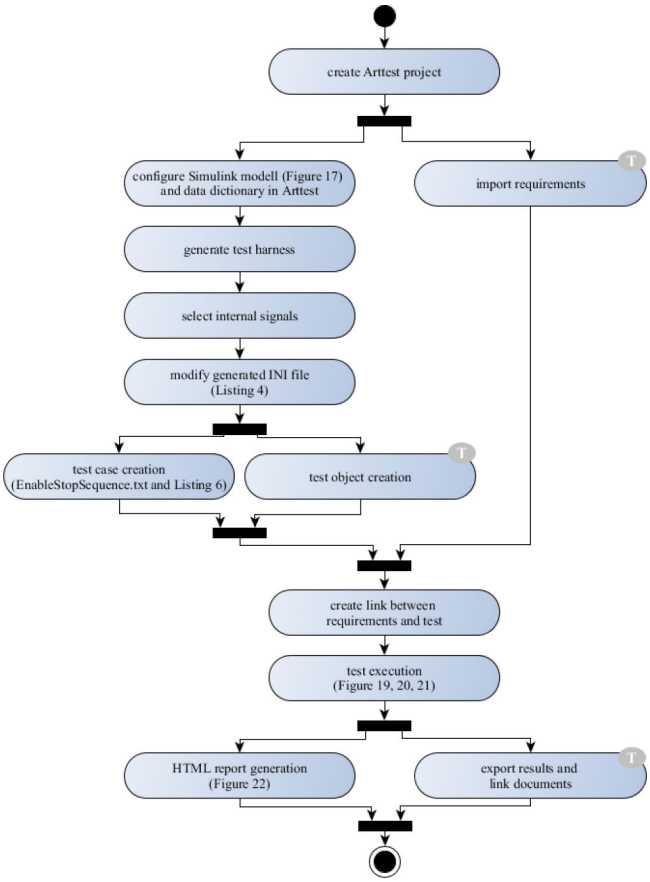

Figure 17. Activity diagram describing the test setup and workflow of the case study

Figure 17 shows the basic workflow, describing the test setup and process with Arttest to create tests for the mentioned requirements. The annotation "T" indicates that these activities are required to provide traceability as described in section "Requirements and Traceability".

To be able to test these requirements, the vehicle has to be started and accelerated by using the clutch pedal, the accelerator pedal and the gear lever. After the vehicle speed is above a threshold, automatic stops are no longer inhibited. The next step would be setting the conditions for an engine stop request.

The stop start control software has been developed using Matlab®/ Simulink®. All interface signal variables, constant values and calibration parameters of the control model are defined in a data dictionary and loaded to the Matlab workspace. When these variables are used in the test, the data values have to be read from the workspace, or have to be overridden in order to induce specific calibration changes.

For verification purposes it has been integrated into a closed-loop model together with a vehicle plant model (see Figure 18).

The test environment needs to be able to override signals in the driver interface, as it has to take over control for the clutch pedal, the accelerator pedal, the brake pedal and the gear lever. Based on the position of the clutch pedal, the clutch status is calculated in the Control subsystem. The evaluated output of the control model is the engine command. As the conditions for allowing stop commands depend on the vehicle speed, we want to be able to use the vehicle speed values generated during simulation as criteria for changes in the inputs as well as expected changes at the output.

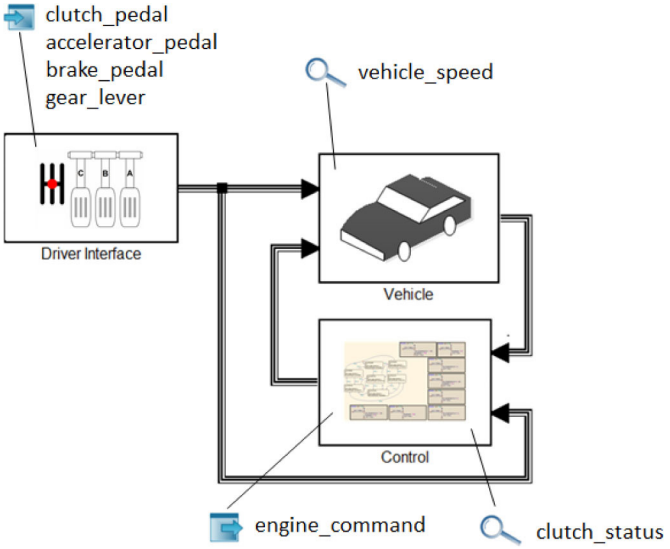

Figure 18. Closed-loop model

To enable the specification of functional test cases in Arttest a test project and a harness model need to be generated. From the harness model internal signal are selected that are required to be overwritten or measured. With this information the interface signals together with the parameters and constants can be defined with their initial values.

```
Initial Conditions:

    Inputs

    Outputs

    Internal Signals
        <key_value>          [0] override
        <clutch_pedal>       [0] override
        <brake_pedal>        [0] override
        <accelerator_pedal>  [0] override
        <gear_lever>         [0] override

        <vehicle_speed>      [0]
        <clutch_state>       [RELEASED]

        <engine_command>     [0] (TUBE;-0.1;0.1;1;0.5;0.1;0)

    Calibratables
        <STOP_INHIBIT_SPEED> {ws} |STOP_INHIBIT_SPEED.Value|
        <MAX_STOP_SPEED>     {ws} |MAX_STOP_SPEED.Value|
        <STOP_DELAY>         {ws} |STOP_DELAY.Value|

    Constants
        <TRUE>       [1]
        <FALSE>      [0]
        <OFF>        [0]
        <ON>         [1]
        <TUBE>       [2]
        <SEQ>        [3]
        <RELEASED>   [0]
        <PRESSED>    [1]
        <STOP>       [1]
        <NEUTRAL>    [0]
```

Listing 4. Test Case - Initial Conditions

The initial conditions file for the test case in Listing 4 defines that the internal signals *key_value, clutch_pedal, brake _pedal, accelerator_ pedal* and *gear_lever* will be overridden and are initialized with the value 0. The internal signals *vehicle_speed* and *clutch state* will be

observed only. The *engine command* signal will be evaluated using the tubular check *(TUBE)* and the initial value of the expected output will be 0. The calibration values *STOP_INHIBIT_SPEED, MAX_STOP_SPEED* and *STOP_DELAY* are taken from the corresponding values in the Matlab® workspace (e.g. *STOP_DELAY. Value*). The variables defined in the *Constants* section only exist and can be used in the test specification context. The file with the initial conditions can be reused for different test cases. In a test case only those initial values need to be set that differ from this default configuration.

After defining the initial conditions, a test case can be specified that accelerates the vehicle to the speed required to reset the stop inhibitor. The following test case sequence (see Listing 5) modifies the accelerator and clutch pedals and changes the gear values from first to second gear accelerating the vehicle unless the vehicle speed is higher than the threshold *STOP_INHIBIT_SPEED*. When the threshold is reached, the accelerator pedal is changed to a lower value to keep the vehicle speed at a constant value.

```
Initial Conditions:

Test:

    Step 1:
        simultaneously (
            after {0} seconds <key_value> set to [1]
            after {1} seconds <key_value> set to [2]
            after {2} seconds <key_value> set to [1]
            after {3} seconds
                <clutch_pedal> ramps to [100;100;0]
                    within {0.5;0.5;1.5} seconds
            after {4} seconds <gear_lever> set to [1]
            after {4} seconds
                <accelerator_pedal> ramps to [60;60;0]
                    within {1;3;0.5} seconds
            after {8.5} seconds
                <clutch_pedal> ramps to [100;100;0]
                    within {0.5;0.5;1.5} seconds
            after {9} seconds <gear_lever> set to [2]
            after {10} seconds
                <accelerator_pedal> ramps to [80]
                  within {1} seconds
            after {10} seconds
                when [vehicle_speed>STOP_INHIBIT_SPEED]
                    during {3} seconds then (
                <accelerator_pedal> ramps to [35]
                    within {1} seconds
            )
        )
        wait for {5} seconds

Acceptance:

    Criterion 1:
        % no change for engine_command
```

Listing 5. Test Case – Enable Stop Requests

The input values that are calculated based on the settings in the test case together with the resulting vehicle speed are shown below in Figure 19.

The test specification file has the name EnableStopSequence.txt. This file can be reused in the next test case, shown in Listing 6, where the requirement to command an engine shut down is verified. In *Step 1* the pre-defined test sequence is imported and all the input signal changes are taken from the imported test case. In *Step 2* neutral is selected and the brake pedal is pressed until the vehicle is in standstill. In *Criterion 1* acceptance conditions are also imported. The acceptance conditions of *Criterion 2* specify the expected engine stop request. According to the requirement, the engine command shall be set to *STOP* when the conditions to shut down the engine are fulfilled and a time *STOP_DELAY* has elapsed. As two calibration sets with different values for *STOP_DELAY* are defined, the execution of this test specification will result in two simulation runs with different values for the acceptance criteria.

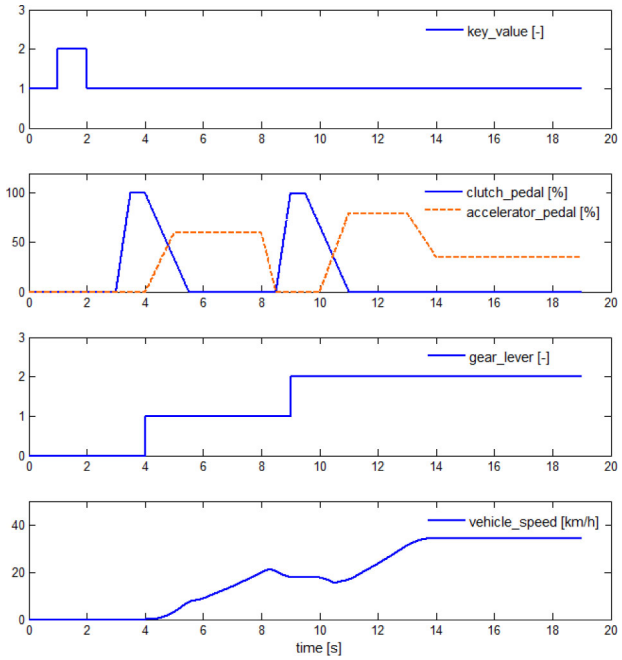

Figure 19. Simulation Results – Enable Stop Requests

```
Initial Conditions:

    Calibratables 1
        <STOP_DELAY> [1] {ws} |STOP_DELAY.Value|

    Calibratables 2
        <STOP_DELAY> [0] {ws} |STOP_DELAY.Value|

Test:

    Step 1:
        Import from "EnableStopSequence.txt"

    Step 2:
        simultaneously (
            after {0} seconds
                <accelerator_pedal> ramps to [0]
                    within {1} seconds
            after {1} seconds
                <clutch_pedal> ramps to [100]
                    within {0.5} seconds
            after {2} seconds
                <gear_lever> set to [NEUTRAL]
            after {2} seconds
                <clutch_pedal> ramps to [0]
                    within {1.5} seconds
            after {4} seconds
                <brake_pedal> ramps to [30]
                    within {1} seconds
        )
        wait for {5} seconds

Acceptance:

    Criterion 1:
        Import from "EnableStopSequence.txt"

    Criterion 2:
        when [vehicle_speed<MAX_STOP_SPEED &
            gear_lever==NEUTRAL &
                clutch_state==RELEASED]
                    during {8} seconds then (
        after {STOP_DELAY} seconds
            <engine_command> set to [STOP]
)
```

Listing 6. Test Case – Command Stop Request

The actual simulation results after the test execution are shown in Figure 20 and Figure 21. The results show that the engine command is set to *STOP* a delay time after the vehicle speed is below *MAX_STOP_SPEED*

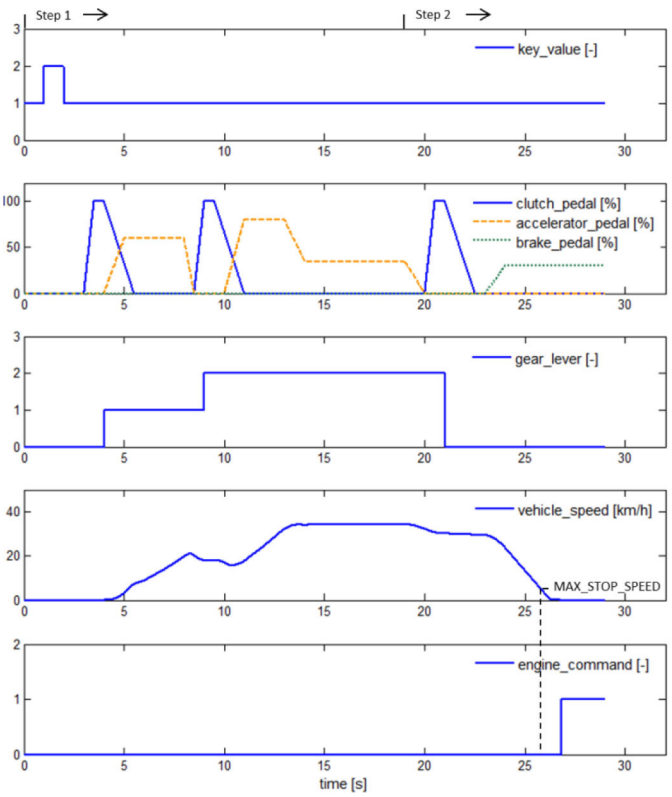

Figure 20. Simulation Results – Command Stop Request (Calibratables 1)

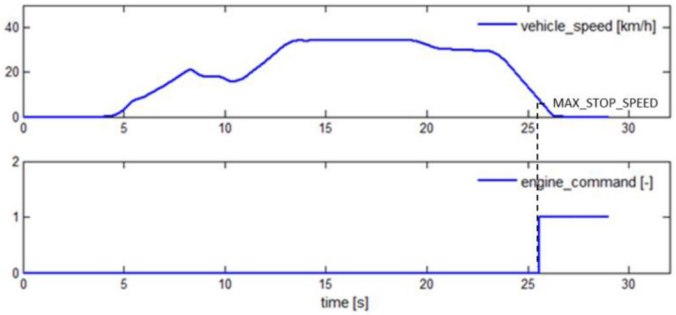

Figure 21. Simulation Results – Command Stop Request (Calibratables 2)

The expected output and the specified reference values are different for some executions steps, but as the signal changes within the defined time and value tolerances, the test is passed (see Figure 22).

Displayed Object: engine_command (Sequential)

Time	Signal Value	Reference Signal Value
26.805	0	0
26.81	0	0
26.815	0	1
26.82	0	1
26.825	0	1
26.83	0	1
26.835	0	1
26.84	0	1
26.845	0	1
26.85	1	1
26.855	1	1
26.86	1	1

Figure 22. Difference of the Simulated and Reference Signal

The generated HTML test report is shown below. The results of each executed test case with the corresponding evaluation results can be reviewed (see Figure 23).

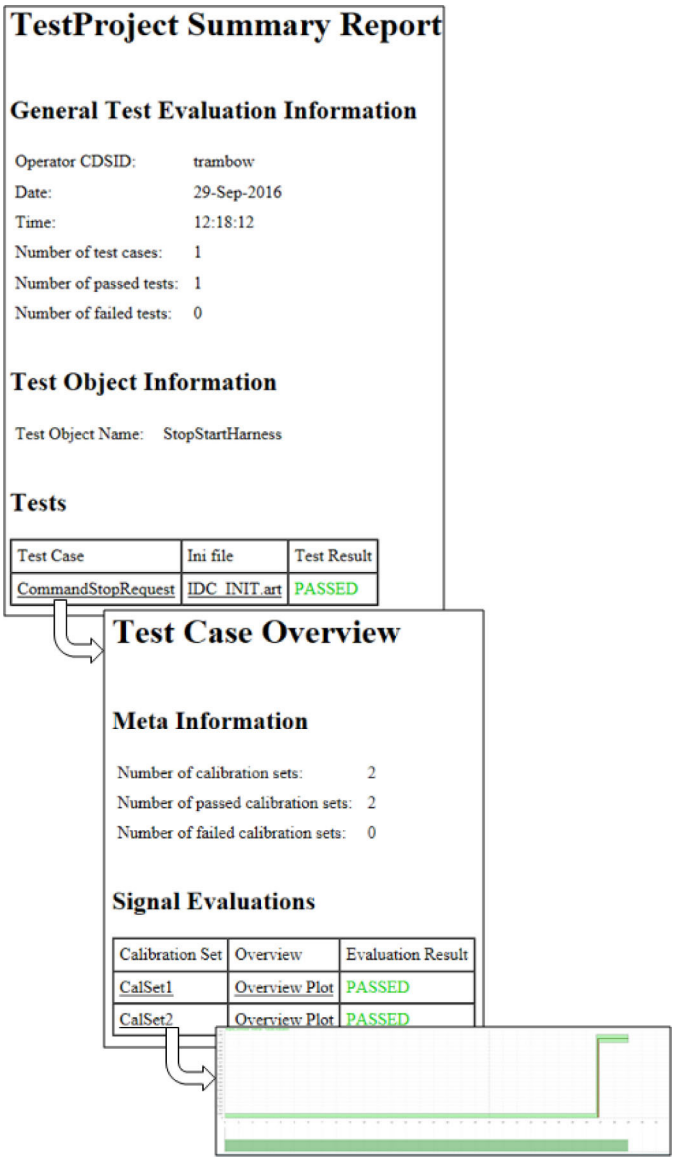

Figure 23. Test Report

Summary

This paper presents the tool Arttest which addresses the introduced requirements for requirement-based functional testing of signal-based software systems. Arttest automates activities such as test preparation, execution, evaluation and report generation and thus allows the tester to focus on the time-consuming task of test case specification. To further ease the test specification, a new text based specification language is introduced. The tool support of the specification language includes syntax checks, syntax highlighting, content completion and supporting features such as model interface detection, visual previews of specified signals including acceptance regions as well as access to Matlab® objects and values. Furthermore, Arttest enables the tester to react on events based on conditions and to manipulate stimuli based on arbitrary other signal values.

Moreover, it allows to override internal signals with arbitrary signal values during test execution. Thus, extensive closed-loop testing including fault-injection for robustness testing is enabled. To cope with the ISO 26262, Arttest supports the import of requirement data and integrates into the Ford process to provide traceability of test related data. We show the suitability of Arttest and the test specification language for functional closed-loop testing on model level presenting an industrial case study.

References

1. Broy, M., Krcmar, H., Zimmermann, J. et al., "Economical impact of model-based development of embedded software systems in cars", ATZ Autotechnology (2011) 11: 54., 2011, doi:10.1365/s35595-011-0026-3
2. IEEE Standards Board, “Standard Glossary of Software Engineering Technology”, IEEE Std - 610.12-1990
3. ISO, “Internation Organization for Standardization” http://www.iso.org, accessed Oct. 2016
4. dSPACE, “AutomationDesk”, http://www.dpsace.com, accessed Oct. 2016
5. Zuras, D. et al. "IEEE standard for floating-point arithmetic", IEEE Std 754-2008, 2008, 1–70
6. Zhang, P. et al., "WCOMP: Waveform Comparison Tool for Mixed-signal Validation Regression in Memory Design", 2007 Asia and South Pacific Design Automation Conference, IEEE, 2007

The Engineering Meetings Board has approved this paper for publication. It has successfully completed SAE’s peer review process under the supervision of the session organizer. The process requires a minimum of three (3) reviews by industry experts.

ISSN 0148-7191

http://papers.sae.org/2017-01-0004