Model-Based Real-Time Testing of Embedded Automotive Systems

Pawel Skruch and Gabriel Buchala
Delphi Automotive

ABSTRACT

The paper presents a model-based approach to testing embedded automotive software systems in a real-time. Model-based testing approach relates to a process of creating test artifacts using various kinds of models. Real-time testing involves the use of a real-time environment to implement test application. Engineers shall use real-time testing techniques to achieve greater reliability and/or determinism in a test system. The paper contains an instruction how to achieve these objectives by proper definition, implementation, execution, and evaluation of test cases. The test cases are defined and implemented in a modeling environment. The execution and evaluation of test results is made in a real-time machine. The paper is concluded with results obtained from the initial deployment of the approach on a large scale in production stream projects.


INTRODUCTION

The extensive use of electronics and software within the last decades has revolutionized the implementation and scope of modern embedded systems in the automotive industry. The embedded systems have become increasingly sophisticated and their software content has grown rapidly. Model-driven development (MDD) and model-based testing (MBT) methodologies have become the preferred approaches for the development of such systems. MDD describes a software development approach in which models of software systems are used for application design and implementation. MBT relates to a process of creating test artifacts from various kinds of models by application of a number of sophisticated methods. The results of the study [2] show that with the right (i.e., correctly applied) approach MDD and MBT can bring significant cost savings and also improvement in the quality of the final system.

Real-time testing involves the use of a real-time environment to evaluate the system at its normal operating frequency, speed, or timing. Engineers shall use real-time testing techniques to achieve greater reliability and/or determinism in a test system. Real-time testing shall enable to process all signals simultaneously (i.e., respecting precise timing requirements) and evaluate test results online during test execution. Online test evaluation provides results immediately (once a test case is executed) and without test engineers spending additional time and effort on the offline analysis of the logs. Additionally, an online evaluation testing strategy can be adapted depending on the results as they occur to make the tests more efficient.

The paper presents an integrated model-based approach for testing embedded automotive software systems in a real-time. The presented approach concerns mainly black-box testing and includes definition and implementation of test cases in a modeling environment and execution and evaluation of test results in a real-time hardware. The system is organized as follows. In the next section, possible scenarios of using models in system development process are reminded. Following section deals with a representation of the system under test (SUT) under which the presented approach is based on. The sections called ‘Test Notation’ and ‘Test Implementation’ describe in details definition and implementation aspects of test cases in MATLAB®/Simulink® environment. The next sections called ‘Test Evaluation’ and ‘Test Execution’ are devoted to test execution aspects in dSPACE™ real-time hardware. Conclusions complete the paper.

SYSTEM DEVELOPMENT USING MODEL-BASED APPROACH

The model-based design process, which includes both the code development and test development processes, is detailed in Figure 1 and Table 1. In this process, software code development and test development are performed concurrently. Very important for those processes are system requirements which form an input for both software engineering and test engineering activities.
Models can be used in engineering disciplines to specify the system's behavior in a clear and unambiguous form. In such case, the models are built based on the requirements defined by customer and often delivered in the form of written specification. Before the requirements are modeled or specified they must be gathered through an elicitation process. Requirements elicitation is a part of the requirement engineering process that is in scope of systems engineering competency. The model of the developed system can reside at different levels of abstraction and it is usually created for different purposes. An implementation model is designed to meet the requirements from a software engineering point of view and it is often used together with a code generator. The implementation model takes into consideration the various target system implementation constraints such as real-time operating system, scheduling details, signal representations, fixed-point vs. floating-point number specification, etc. Models used to generate test cases are called test models. Such models usually capture the intended behavior of the SUT and contain test implementation and execution constraints imposed by test execution platform. Once the code is compiled and integrated with other software and hardware components, the entire system is being tested. Testing process is conducted through the test harness that allows executing of test cases and generating associated test reports. The primary goals of the testing process are to find defects and to show the system's compliance to its requirements. If the tests fail, then either the implementation model needs to be redesigned (because of software error) or the test model has to be changed (because of testing error). The system is not released until all tests pass.

The model can be also used as an oracle that is a source to determine expected results to compare with the actual result of the SUT [1]. The approach (Figure 2) stipulates that the same inputs are applied to both the SUT and to the model. The signals are physical in case of the SUT and virtual in case of the model. The judgment whether the result of a test is in conformance with the model is delegated to a test comparator. The test comparator is usually a tool that compares the actual output produced by the SUT with the expected output produced by the model.

In this paper, models are used to create test environment and test harness where the tests cases can be defined and implemented and then executed and evaluated in a real-time execution platform.

**SYSTEM REPRESENTATION**

We consider the SUT as a set of inputs $u(t) \in U \subset \mathbb{R}^n$, state variables $x(t) \in X \subset \mathbb{R}^m$, and outputs $y(t) \in Y \subset \mathbb{R}^m$ that are expressed as vectors (Figure 3). Here, $U$ is called the input state space, $X$ is called the internal state space, $Y$ is the output state space, $\mathbb{R}^n$, $\mathbb{R}^m$ are real vector spaces of column vectors, the independent variable $t > 0$ is time, $x(0) \in \mathbb{R}^n$ is the given initial condition $r$, $n$, $m$ are positive integers that determine numbers of inputs, state variables, and outputs, respectively. The relationship between input, state, and output variables is determined by system requirements and can have different forms (e.g., logical formulas, state machines, algebraic equations, differential equations, etc.).
The system representation in the form of input/state/output notation (called also state space notation) provides a convenient, compact, and universal way to model and analyze systems with multiple inputs and outputs.

**TEST NOTATION**

The system specification in the form of an input/state/output representation can be used to describe tests in the way to be independent from test methods, test implementation, and test execution environment. The industry standard \[4\] defines a test case as a set of inputs, execution preconditions, and expected outcomes developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. Following this definition, a single test case \(T_{\text{case}}^{(j)}\) can be specified as

\[
T_{\text{case}}^{(j)} = \{T^{(j)}, x_0^{(j)}, u^{(j)}(\cdot), y^{(j)}(\cdot)\},
\]

in case of black-box testing \[3\], or

\[
T_{\text{case}}^{(j)} = \{T^{(j)}, x_0^{(j)}, u^{(j)}(\cdot), x^{(j)}(\cdot), y^{(j)}(\cdot)\},
\]

in case of grey-box testing \[6\]. Here, \(u^{(j)}(\cdot)\) is an input function applied to the tested system, \(x^{(j)}(\cdot)\) is an expected state function, \(y^{(j)}(\cdot)\) is an expected output function within the execution time window \([0, T]\) when the system starts from the initial state \(x_0^{(j)}\), \(j = 1, 2, ..., N\) is a label to indicate different test cases. A collection of one or more test cases forms a test suite

\[
T_{\text{suite}} = \{T_{\text{case}}^{(1)}, T_{\text{case}}^{(2)}, ..., T_{\text{case}}^{(N)}\}.
\]

The test cases (1) and (2) are defined on the basis of continuous-time signals. In the implementation these signals have to be approximated by discrete-time signals. In the presented approach, a continuous-time signal is approximated by a set of discrete points with linear interpolation between them. Figures 4 and 5 illustrate this implementation for digital and analog signals.

Formally, these representations can be described using vectors for time and value coordinates, that is,

\[
s_{\text{time}} = [t_1, t_2, t_3, t_4, t_5, t_6],
\]

(3)

\[
s_{\text{value}} = [v_1, v_1, v_2, v_2, v_1, v_1, v_1],
\]

(4)

in case of the digital signals plotted on Figure 4, and

\[
s_{\text{time}} = [t_1, t_2, t_3, t_4, t_5, t_6],
\]

(5)

\[
s_{\text{value}} = [v_1, v_2, v_2, v_1, v_1, v_1],
\]

(6)

in case of the analog signal plotted on Figure 5. The implemented representation of a continuous-time signal is then characterized by a pair \(s_{\text{time}}, s_{\text{value}}\), where \(s_{\text{time}}\) stands for the time coordinate and \(s_{\text{value}}\) stands for the value coordinate of the represented signal.

It shall be noticed that the approximation of the continuous-time functions in the form (3), (4) and (5), (6) is perfectly and fully supported by MATLAB®/Simulink® environment.

**TEST EVALUATION**

Evaluation and test comparison mechanism are necessary parts of any successful test design as they are used for determining whether a test passes or fails. The comparison mechanism is often named in software engineering as test comparator \[5\]. The test comparator (see also Figure 2) is usually a combination of software and hardware components as its intention is to compare signals of different nature (i.e., physical vs. virtual). It shall be underlined that the comparison of signals significantly differs from the comparison of single values and it can be realized either in time or frequency domain. Following the concept presented in Figure 2, the test...
comparator shall compare the actual output $y_s(t)$ produced by the SUT with the expected output $y(t)$ produced by the model or calculated based on requirements. A possible practical realization of the comparison function in time domain for a given test case $T^{(j)}_{case}$ is presented below:

$$
\begin{align*}
    z(T^{(j)}_{case}) &= \begin{cases} 
    0 & \text{if } y_s(t) \in [y_l(t), y_u(t)] \\
    1 & \text{otherwise}
    \end{cases} \quad \| y^{(j)}(t) - y_s^{(j)}(t) \| < \varepsilon, \\
\end{align*}
$$

(7)

If the actual output is within a predefined tolerance range $\varepsilon$ relative to the expected output, then this test is qualified as pass ($z = 0$, system ok), otherwise the test is qualified as fail ($z = 1$, system error). It should be noticed that the tolerance range can be defined in different ways depending on the system being tested. It can be symmetrical, as in the formula (7), allowing the system to produce a few percent more or less than the expected value. It is also possible to define different upper and lower bounds of tolerance, for instance, allowing the system to produce 2% more, and 5% less than expected.

The next step is to adapt the general idea for test results comparison, as described above, to the implemented representation of signals, as illustrated on Figures 4 and 5. The idea is to define two kinds of tolerances (upper and lower tolerances) for analog signals and four kind of tolerances (upper, lower, left, and right tolerances) for digital signals (see Figures 6 and 7).

Figure 6. Implemented comparison mechanism for digital signals.

Figure 7. Implemented comparison mechanism for analog signals.

The upper tolerance is calculated as maximum allowed difference if the actual output signal is higher than the expected one. The lower tolerance is defined as maximum allowed difference if the actual output signal is lower than the expected one. In case of digital signals, the left tolerance means maximum allowed difference in time if the step on the actual output signal appears before the step on the expected one. The right tolerance stands for maximum allowed difference in time if the step on the actual output signal appears after the step on the reference signal.

TEST IMPLEMENTATION

Based on the information provided in the previous sections, it is possible now to define, implement, and evaluate a test case that includes both magnitude-continuous and time-continuous signals. Figures 8 and 9 illustrate a simple test case defined with the help of Signal Builder block from the Simulink® library. The test case contains two inputs and one output (expected) signals. As the expected signal is digital, it is natural to set to zero the upper and lower tolerances. The left and right tolerances are set to 100 ms, what means that the signal change is expected maximum 100 ms before or after the specified point in time.

Figure 8. Graphical representation of a simple test case.

```matlab
% INPUTS
PowerMode_time = [0 1 1 7 7 13];
PowerMode_value = [0 0 4 4 0 0];
HornSwitch_time = [0 2 2 5 5 8 8 11 11 13];
HornSwitch_value = [0 0 1 1 0 0 1 1 0 0];
% OUTPUTS
HornRelay_time = [0 2 2 5 5 8 8 11 11 13];
HornRelay_value = [0 2 2 5 5 8 8 11 11 13];
HornRelay_uppertol = [0];
HornRelay_lowertol = [0];
HornRelay_lefttol = [0.1];
HornRelay_righttol = [0.1];
```

Figure 9. Textual (m-file) representation of a simple test case.

The following part of this section describes the configuration of the test application that has been designed using Simulink tool in the MATLAB® environment. Besides standard Simulink® blocks dSPACE™ Real-Time Interface (RTI) library has been used to link test application software with test system hardware.

Figures 10 and 11 presents exampled flow graphs for signals that are generated by the test system and then directed through RTI interface to the corresponding inputs of the system being tested. Three types of flow graphs have been shown in these figures. They correspond to digital, analog, and resistance signals (these signals are inputs for the tested system and outputs for the test system) and can be freely multiplied. When IN_Signal_X_Enable flag is unset, then corresponding signal is excluded from the experiment which is
started when *TestEnable* flag is set. In such situation this signal can be changed in manual mode only. When *TestEnable* flag is set, then a waveform defined in SIGNAL_TestBlock will be applied to the corresponding input of the SUT.

**Figure 10.** The flow graph created in Simulink® for the input signals applied to the SUT.

**Figure 11.** The flow graph created in Simulink® for the input signals applied to the SUT (view of IN_Signal_X_Test subsystem).

Figures 12 and 13 present exampled flow graphs for signals that are measured by the test system and are available through RTI interface in the Simulink® model. When OUT_Signal_X_Enable flag is unset, then this signal is excluded and not evaluated in the experiment which is started when *TestEnable* flag is set. When *TestEnable* flag is set, then the measured signal is compared with the reference signal waveform defined by SIGNAL_TestBlock. The *TestEnable* flag is then a synchronization trigger for all enabled input and output flow graphs.

**Figure 12.** The flow graph created in Simulink® for the output signals measured by the test system from the SUT.

**Figure 13.** The flow graph created in Simulink® for the output signals measured by the test system from the SUT (view of OUT_Signal_X_Test subsystem).

**SIGNAL_TestBlock** (Figure 14) is a Simulink® block that compares two signals named as *input* and *reference* in every time step with a given tolerance. *input* signal shall be connected to the output of the SUT, *reference* is defined by time and value vectors delivered to the block as parameters (*Time vector* and *Reference vector* on Figure 14). The user has to define four kinds of tolerances: upper tolerance (maximum difference if the *input* signal is higher than the *reference*), lower tolerance (maximum difference if the *input* signal is lower than the *reference*), left tolerance (maximum difference if the step on the *input* signal appears before the step on the *reference* signal), and right tolerance (maximum difference if the step on the *input* signal appears after the step on the *reference* signal).

**Figure 14.** SIGNAL_TestBlock and its parameters.
**SIGNAL_TestBlock** can be used in few scenarios. In first scenario, two signals are compared with fixed values of tolerances. In this case, all tolerances are defined as one element vector and set up from the *Block Parameters* window. Those values will be used during the whole experiment, because they are the last elements of the parameter vectors. In second scenario, two signals are compared with dynamic values of tolerances. In this scenario user can define tolerances with the *Block Parameters* window as a vector with different values for each time step. The last value will be used for each time step of the experiment if the number of time steps exceeds the number of the vector elements.

The values of *SIGNAL_TestBlock* can be defined by **Signal Builder** (please remember that the signal configuration can be also imported to **Signal Builder** from MATLAB® Workspace from external tools such as csv files or MS Excel).

**TEST EXECUTION**

When the configuration model of the test application includes all inputs and outputs of the tested system, then it shall be compiled and downloaded to a real-time machine. The approach has been verified on dSPACE™ modular hardware with DS1006 Processor Board, DS2211 HIL I/O Board, DS4004 HIL Digital I/O Board, and DS4330 LIN Interface Board. The SUT was a body controller unit with more than 200 I/O and 1000 CAN (Controller Area Network) and LIN (Local Interconnect Network) signals. dSPACE™ AutomationDesk® has been used as a test sequencer tool. In this case, the test sequencer is an environment intended to set up the parameters of test cases, trigger start and stop of test execution, read the test results and prepare the test report. Figures 15, 16, 17, 18 illustrate the key elements of an exampled test case designed in AutomationDesk® environment. It is a good evidence that the test cases can be created in an effective and efficient way. The entire execution and test evaluation is done on the real-time machine. With such approach hundreds of signals of different nature (in this case digital, analog, PWM (Pulse-Width Modulation), CAN, LIN) are processed in the real-time and simultaneously. Evaluation of test results is made online during test execution. Online test evaluation provides results immediately (once a test case is executed) and without test engineers spending additional time and effort on the offline analysis of the logs. Additionally, an online evaluation testing strategy can be adapted depending on the results as they occur to make the tests more efficient.

![Figure 15. General structure of a test case in AutomationDesk®.](image1)

![Figure 16. Initialization part of an exampled test case.](image2)
CONCLUSIONS

In this paper, a model-based approach has been presented that can help test engineers in testing embedded systems in a real-time. The approach has been developed and implemented by test engineers from DELPHI Technical Center Krakow, Poland. It is an initial deployment on a large scale in production stream projects in the area of embedded software testing in the automotive industry. Although the presented approach has been applied to projects coming from the automotive industry, but the obtained results give a good perspective on the applicability of this approach for other industrial projects.

It is worth mentioning that the presented approach can be integrated with other test artifacts created from various kinds of models. When test cases generated automatically from models have proper format, they can be easily included to the presented tool chain. For example, generated test vectors from Simulink Design Verifier™ toolbox from MathWorks™ can be used, practically without any modification, to verify the system behavior in the presented test configuration. When model of the system being tested is available, then it can be included to the test configuration and can act as a block that automatically produces expected results. Additionally, the approach can be also used without real-time environment, that is, it can be applied in model-in-the-loop (MIL), software-in-the-loop (SIL), and processor-in-the-loop (PIL) test configurations.

REFERENCES


CONTACT INFORMATION

Pawel Skruch
Engineering Group Manager
Electronic Controls
Delphi Technical Center Krakow
ul. Podgorki Tynieckie 2, 30-399 Krakow, Poland
Office: +48.12.252.1730
Mobile: +48.668.691.040
pawel.skruch@delphi.com
ACKNOWLEDGMENTS

We would like to thank Dr. Marcin Piatek from Technika Obliczeniowa in Krakow for the help in designing and implementation of SIGNAL_TestBlock. Special thanks to Dr. J. Richert and M. Sanf from dSPACE for their valuable remarks concerning optimal usage of dSPACE real-time environment during their visit at Technical Center Krakow.