

# **AUTOMATED TESTING IN MODEL BASED DESIGN– AN IMPLEMENTATION FOR CRUISE CONTROL MANAGER**

Johannes Slettengren  
Scania CV AB, Systems and Software Department, Sweden

## **Abstract**

During the model based development of a cruise control application, the need for an automated, flexible and expandable test environment was recognized. The application inputs are analog, digital and CAN-signals. Outputs are all CAN signals. It is a part of an ECU containing other applications, all written in C.

The demands on the test environment were to be able to integrate testing efficiently in development process implying a need for quick execution. Test cases should be written in an “easy-to-read” manner to minimize duplicate specifications. The tests should be used to provide a base-line for regression testing. Hence, repeatability is a key property. The tests should be able to execute in several environments, e.g. Simulink-simulation, generated-code running on host or target. Test cases should carry a minimum of environment-specific details. Test cases should be defined using a hierarchical approach (test-set, -case, -act, -step).

The solution is a three-step procedure. First step is a stand-alone test definition. Second step is the test execution and the third step is test result evaluation and display/report generation. With this layout, the first and third step is independent of environment and is fully implemented in Matlab. The second step will be entirely environment-dependent.

## **Keywords**

### **1. INTRODUCTION**

During the model based development of a cruise control manager for heavy duty vehicles, the need for a strategy for environment independent testing was recognized. A few standpoints were decided upon from the beginning:

- Test cases should be written in m-files and should contain all necessary information to execute tests in the Simulink environment. This includes setting input signals, criteria for passing tests and settings for the test engine.
- Test scripts should be easy to read and, together with high level functional

descriptions, serve as a complete test description/specification.

- A test object class should be created containing methods for setting and testing signals.
- The test engine should support many test execution environments, simulation being the natural/basic test method.

#### **1.1. Goal System: Scania Cruise Control Manager**

Scania has had several cruise control applications distributed over several ECUs for quite some time. They consist of (regular) Cruise Control (CC), Adaptive Cruise Control (ACC), Downhill Speed Control (DHSC) and Eco Cruise (ECC). Being similar in

functionality, it was a natural decision that these were all to be implemented in a single ECU, namely the Coordinator (COO).

The Coordinator is (among other things) a central hub in the electrical system connected to all three of the main CAN-buses (named 'Red', 'Yellow' and 'Green'). The software in the ECU is all written in C, except from the Cruise Control applications which are generated from Simulink using Real-time Workshop Embedded Coder. Figure 1 shows a schematic picture of the CAN-architecture and some ECUs that are involved in the Cruise Control functionality.

The ACC was already implemented using Simulink/Stateflow (Eriksson et al, 2007) but on another ECU. The other cruise controls were hand written and had to be implemented again in Simulink/Stateflow.

## 2. TEST ENGINE DESCRIPTION

The test engine consists of three main steps: Test definition, test execution and test evaluation. This chapter will discuss the whole setup in detail. Figure 2 shows the complete setup schematically.

### 2.1. User interface

The test engine is started from the Matlab command prompt. There is no graphical user interface connected to the test engine. The reasoning for this is based many arguments, the main being that a GUI I too hard to maintain. And besides, the users all have good computer skills and would surely try to go around the GUI and script everything anyway.

Typically, all user will create their own m-file, say 'startTesting.m', which includes the proper function calls to invoke the test engine.

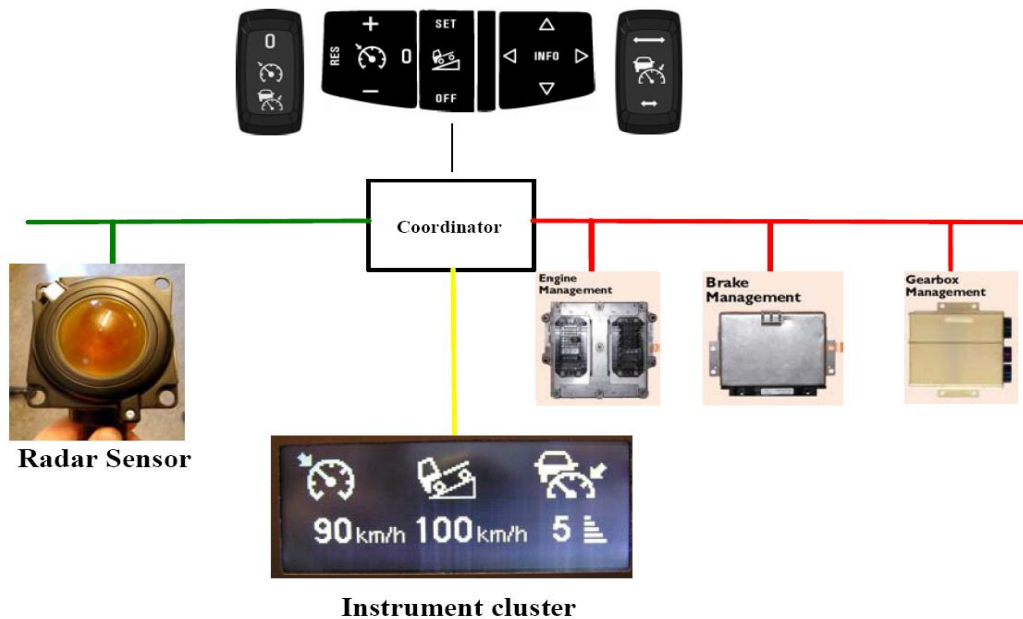
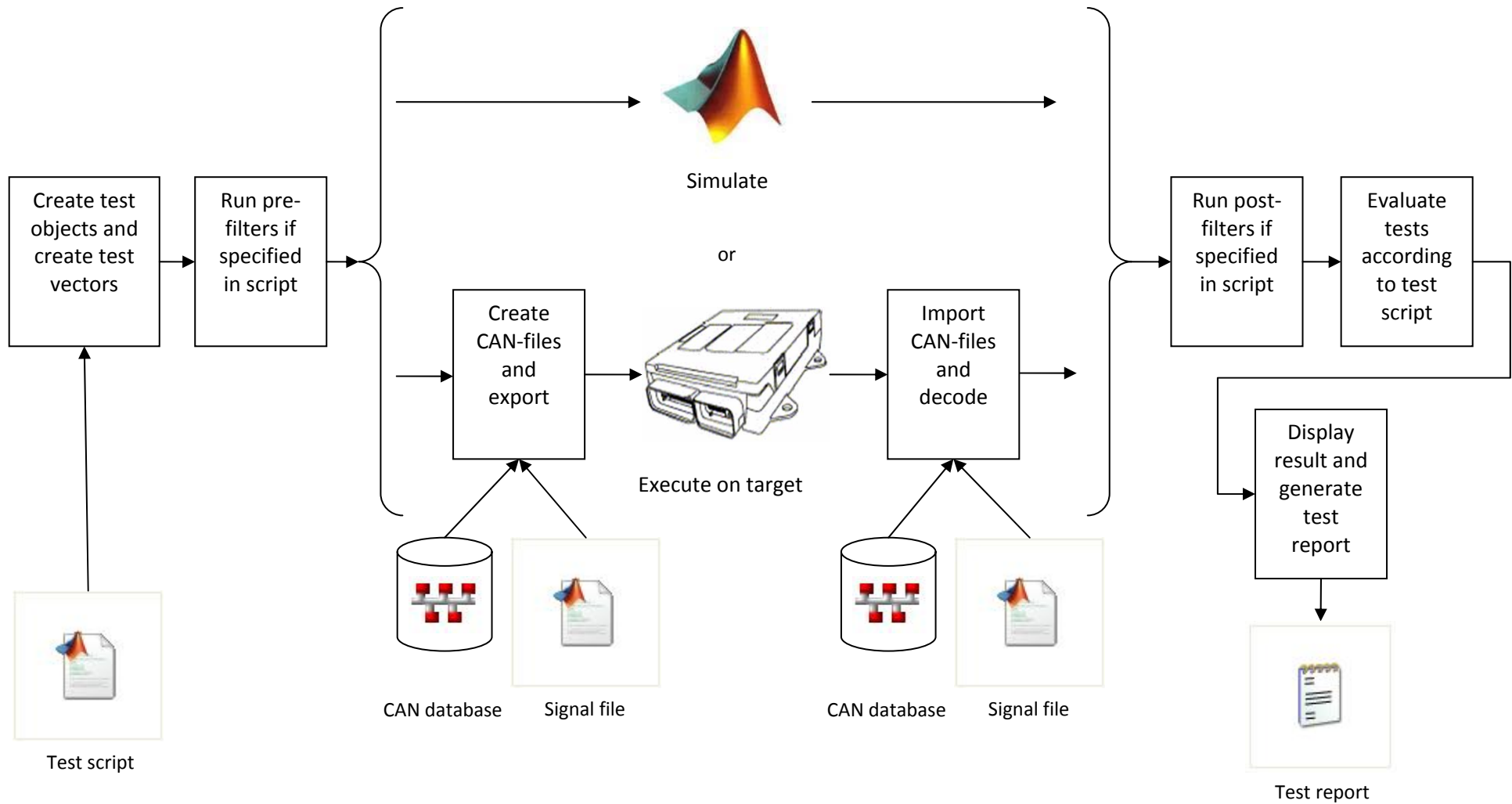


Figure 1. Schematic view of some of the components that are involved in the cruise control applications



**Figure 2. Complete test system. The output of the system is a test report specifying failed and passed test acts.**

## 2.2. Test Script Signal Representation

When designing a test system, the question of how to represent the signals in the test scripts will have to be dealt with sooner or later. There are ambiguities with signal naming between the Simulink model, the corresponding CAN-signal and I/O in the platform code. Furthermore, signals in the model can have several realizations depending of parameterization.

The solution for this test system is to have a signal list with aliases. Even though it is encouraged to use the model names in the test scripts, the author of the test script can choose to use the CAN-representation of the signal (“`Bus:MsgName:[optional MsgExtension:]SignalName`”).

A danger with allowing the user to specify signals using different aliases is that – depending on parameterization – a signal may either be realized as a CAN-signal or analog or digital I/O. Using the model representation, this is dealt with during conversion to the target test file and when in simulation test mode it is handled in the model. But with the extensions to the test engine (see Extensions chapter), not all tested signals will have a model representation.

## 2.3. Load Signals from Test Scripts

The test scripts, written as m-files, create objects containing all information needed for the test execution. The main methods that constitute the test object class are *setSignal*, *testSignal* and *wait*.

The test scripts are written sequential with instructions to set a signal’s value, wait (hold the values for a given time), and test that a signal is within a given range. This method of defining tests gives tests that are easy to write and read, but is not suitable for testing dynamic events. To include some more dynamics more methods are included. For instance, to set a ramp using the ‘setSignal’ command would require lots of setSignals intervened with wait-commands for the duration of the ramp. Instead, a new method, *setRamp*, was created which will produce the same result.

To further simplify the writing of test scripts, the user has the option to specify one or more

prefilters. For example, the driver will activate a certain gear and push the accelerator pedal. This will give a certain engine speed and vehicle speed. The engine speed and vehicle speed can then be modeled in a prefilter which can be either a Simulink model or an m-function. There is no plausibility check of signals.

Regardless of the chosen test execution method (model or target), the first phase of the execution of the test engine produces all signals in the Matlab workspace.

## 2.4. Test Execution

Test execution can be carried out on host or on target. Each test case specify whether it is able to run in simulation mode, target mode or both. Ideally, all test scripts should support both methods.

### 2.4.1. Model Test Execution

With signals created in the Matlab workspace, executing tests in the Simulink environment is straight forward. After loading the model and its objects, a simple ‘run’ command from the test engine simulates the model.

When simulation is finished, the output signals are available in the Matlab workspace for further evaluation.

The model used for testing is the same used for generating production code. No test harness or other modification is needed for simulation. This is made possible with input and output blocks written as s-functions with the property to read signals from the Matlab workspace during simulation. The code generated from these blocks will be function calls to the platform code.

### 2.4.2. Target Test Execution

Executing tests on target consist of three parts. Firstly, the signals need to be converted into another format/representation. Secondly, these signals must be fed to the ECU running the application software. Thirdly, the outputs of the ECU need to be fed back to Matlab for evaluation.

The format chosen for the export is ascii text files with CAN traffic to be outputted to the four CAN-buses included in the Target test system (see Figure 3). The target system outputs the CAN-frames onto the CAN-buses; three of them connected directly to the ECU, and one connected to a control system for analog and digital I/O.

Setting parameters directly in the test scripts is also supported by the test engine. The target test system writes the parameters to the ECU using KWP2000.

Since the only output of the tested ECU is CAN-traffic, it is pretty straight forward to log it into the same ascii format that was used for the inputs. This is then sent back to the host computer where the signals are decoded into vectors in the Matlab workspace.

### 2.5. Test Evaluation

The test evaluation phase of the test engine also works independent of the test execution method. A *testSignal* triggers a comparison of the specified signal's value and the value specified in the test script. It is possible to test if the value is greater than, smaller than, equal to or lies within a specified range. All of this is done at a specific point in time, determined by the *wait* commands in the test script.

There is no immediate support for testing the

dynamics of a signal, e.g. to see if a signal is increasing or decreasing. This is solved by supporting post filters that act on the output signals producing new, virtual ones.

Assume that we are interested in testing that the brake request is increasing. A post filter that act on the signal *brakeRequest*, producing the virtual signal *brakeRequestGrad*, is designed in Simulink and specified in the test script. The test engine will now run the filter after test execution and before test evaluation. The command

```
ts.testSignal('brakeRequestGrad', ...
'MORETHAN', 0);
```

will now fulfill our intentions of testing that the brake request is increasing.

After evaluation, the user may choose to either plot all signals (zoomed in to the points in time where tests have been done) or only plot the failed test cases or just produce a test report. A simple click on the plot of a failed test act opens a new figure plotting all signals involved in that particular test act.

### 2.6. Parameters

The vehicle industry in general has to deal with their products being built in many variants. This also becomes obvious in the embedded systems and in the embedded applications. One way to approach this is to have multiple software

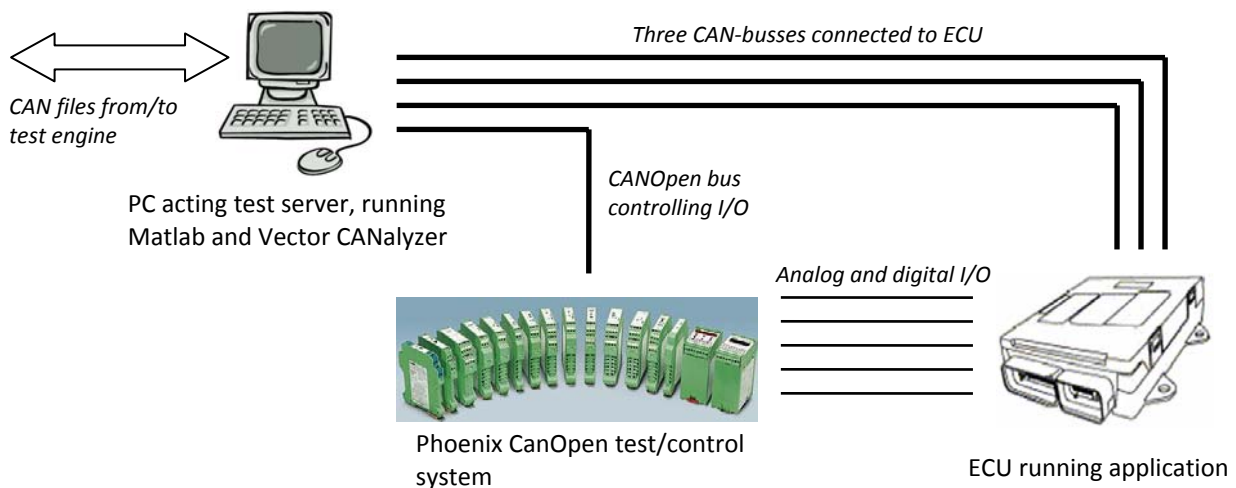
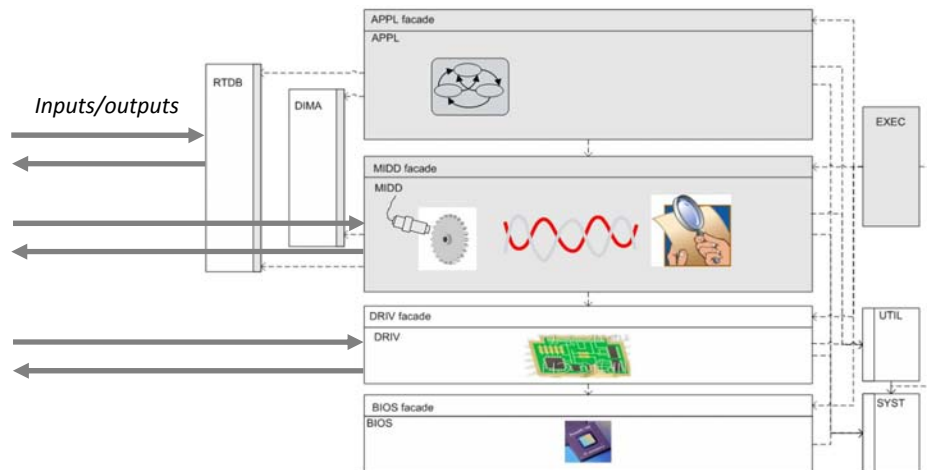


Figure 3. Target test system



**Figure 4. Emulation test execution for handwritten applications. This test mode can input and read data to different layers of the software architecture.**

releases for each ECU, all with unique article numbers. However, with several hundred, if not thousands, variants and software testing being a fundamental part of development, this approach is not feasible.

Instead, Scania has chosen to have one software release per system no matter what the vehicle is equipped with. The functionality is instead determined purely using end-of-line parameterization. This approach opens up for many possibilities for an automatic test system. In principal, it would be possible to have the same test run for all possible combinations of parameters.

In the model test, it is very easy to set the parameters before executing a test act. In the target test mode, this is solved by having the target test system set the parameters in the ECU using KWP2000. This solution and many others in the target test system are non-generic in many ways, making the test setup for target testing harder to adapt to another system. But having said that, the interface is generic and stable which will make designing an automated test rig for any other ECU possible.

### 3. EXTENSIONS

The test implementation described in this paper has evolved further and is continuing to do so as

this is written. It is now being used to test the hand written applications in the Coordinator. The option of model testing in Simulink is of course not available for the hand written code, but an analog test execution method has been created: emulation on host computer. The emulation software is written in C++ and was deigned long before the test engine. The emulator has the ability to input and output data to different layers of the software. Hence, tests can incorporate everything from CAN and I/O decoding or just the application layer. The concept is shown in Figure 4. Hence, the test execution is invoked from Matlab, but performed by another tool.

The test engine is also used on other ECUs, e.g. the Gear Management System (GMS). The modifications needed to have full functionality (model and target tests) for this ECU is the target test execution part, and mostly include adaptation of the test rig.

### 4. CONCLUSION

To design a test engine of this magnitude requires a lot of planning. It is not hard to implement something that works most of the time, but to cover all situations can be a hassle. The approach of choosing Matlab as the tool to realize the test engine proved to be a success in many ways. The idea of having an easy way of

performing regression testing during development is important and while it took quite some time to set up a working hardware test rig, simulation tests were done from a very early stage.

There are other important points to make regarding the test environment. For instance, analyzing failed test acts using automatic plotting scripts is very helpful. The possibility to look not only at the signal whose value was tested to be wrong, but also to have the option of plotting all signals included in the test act is important.

Furthermore, using object oriented programming for the test scripts has been a major advantage. The importance of controlling what can be typed in test scripts should not be underestimated.

The choice of not creating a GUI for the test engine has been good from several aspects. Firstly, it has saved development time. Secondly, it has made managing the test engine less cumbersome. And thirdly, it has forced the users to get a better understanding of the system and thereby contribute more if bugs appear.

With the effort that was put into designing the test engine to be non-proprietary (for Scania ECUs), extending the test environment to test applications implemented for other systems was easily done. Of course, it is recommended that the applications are somewhat similar in design, but they need not be similar in functionality.

A great advantage with having a clean interface between test definition, test execution and test evaluation is that it leaves room for further extensions. It is simple to add new drivers for coding or decoding the test vectors into other formats that can be run on whatever target system.

## 5. REFERENCES

1. Eriksson, M., Lindqvist, K., and Andersson, H., (2007). "Development of a Production Adaptive Cruise Controller for Heavy Trucks Using Model-Based Design and Production Code Generation", Mathworks Automotive Advisory Board 2007.