Beyond MC/DC Coverage Testing

Hans J. Holberg
SVP Marketing & Sales, BTC Embedded Systems AG
Gerhard-Stalling-Straße 19, 26135 Oldenburg, Germany
hans.j.holberg@btc-es.de

Dr.-Ing Stefan Häusler
Product Manager, BTC Embedded Systems AG
Gerhard-Stalling-Straße 19, 26135 Oldenburg, Germany
stefan.haeusler@btc-es.de

Abstract: In the last 5 years, the Back-to-Back testing approach became very popular in the automotive domain and could be applied successfully. One reason is the trend to subsume all development and test activities to a higher abstraction layer, the model level. Another reason is the introduction of the ISO 26262 standard, which recommends the back-to-back testing approach to assure equal behavior between model and code for functional safety reasons.

Several coverage criteria like Decision Coverage (DC) and Modified Condition/Decision Coverage (MC/DC) are state-of-the art for back-to-back testing. But due to the fact that not all internal signals are available for comparison due to missing visibility, pure code coverage criteria are not enough to uncover all potential system errors. A vector set fulfilling even strong metrics like MC/DC may be inadequate to uncover all possible differences between model and code. One reason is that MC/DC is inadequate to complete describe the behavior of specific blocks. Another reason is the so called masking effect leading to situations, that integral errors could not be observed at the SUTs interface right in the moment of the testing time frame.

In this paper, we provide a solution to address both problems by introducing model based test properties and by extending the MC/DC coverage criterion with necessary system data conditions. This will guarantee the observation of internal errors at the observable interface and thus achieves a huge quality improvement. The resulting new test case definitions could become very complicated for human beings thus it can take long time to achieve high coverage manually. To automate this task, we have extended our existing test vector generation approach targeting 100% MC/DC coverage to generate vectors covering all additional test goals.

The presented approach has been introduced in an existing test and verification environment within DENSO.

1 Introduction

In the last 5 years, the Back-to-Back testing approach became very popular in the automotive domain and could be applied successfully in model based development processes. It is a recommended method by the ISO26262 [ISO26262] to find differences between model and code.

Code Coverage criteria as Decision Coverage (DC) or Modified Condition/Decision Coverage (MC/DC) are widely used to measure test quality and are defined as follows:

Decision Coverage (“DC”): In order to reach Decision Coverage, it requires two test cases: true and false outcome of the Decision evaluation during execution of the corresponding test case in the testing environment of the SUT.

Condition Coverage (“CC”): Condition Coverage is given if each Condition in a Decision takes on all possible outcomes at least once.

Modified Condition/Decision Coverage (“MC/DC”) requires that each condition is shown to independently affect the outcome of the decision. This ensures that the effect of each condition is tested relative to the other conditions.

Unfortunately, these structural code coverage criteria are not sufficient to detect all kinds of possible difference between model and code.

Depending on the used blocks in the model, additional test cases are needed to uncover potential specific implementation related defects. One example is the relational operator (<, <=, >, >=, ==, !=) for which any kind of branch coverage criteria does not guarantee the quality of the implementation. MC/DC coverage does not take all relevant boundary values into account in order to find differences e.g. between (i>5) and (i>=5).

A complete MC/DC test coverage alone, would not necessarily uncover the wrong implementation (i>=5) on the code side as shown in the following table.

Test i	Model (FLP) i>5.0	Implementation (FXP: 2^0) i>=5.0
0.0	FALSE	FALSE
10.0	TRUE	TRUE

Because of that, an extension of the test case with 3 additional test properties has been defined in order to find any kind of defect for this specific function/block.

- (a) - (b) = 1
- (a) - (b) = 0
- (a) - (b) = -1

The following table shows how these three additional test properties for the relational operator [a (REL.OP) b] fix this test gap on the implementation side.

Test i	Model (FLP) i>5.0	Implementation (FXP: 2^0) i>=5.0
4.0	FALSE	FALSE
5.0	FALSE	***TRUE***
6.0	TRUE	TRUE

Even if this aspect is considered when creating tests, there exists another problem. Not all internal signals of a system under test (SUT) can be observed during testing activities. In many cases, the output of one block is an internal signal that is not directly observable. A decision whether a property holds for a given set of test vector is not trivial. Reason is that some blocks can mask signals (and potential errors occurred during code generation), which can cause difference of functional behavior, between model and code. This is the so called masking effect leading to situations, that integral errors could not be observed at the SUTs interface right in the moment of the testing time frame.
The following simplified example in Figure 1 explains the problem: it shows a switch block connected to a min block. A difference is indicated between the model where signal2 is checked to be larger than three and the code where Signal2 is checked to be less than three.

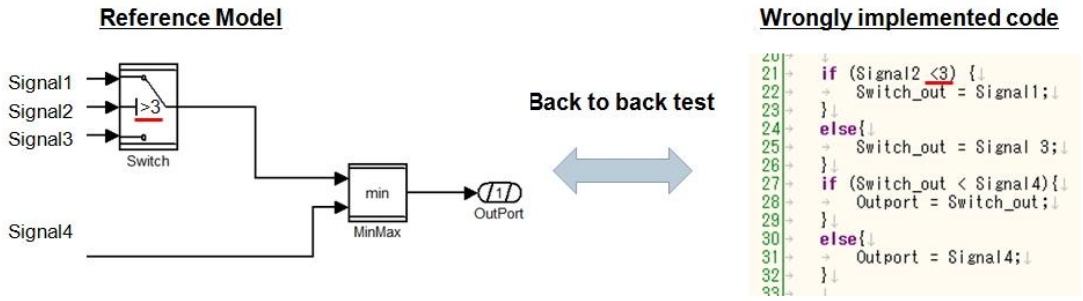


Figure 1: Example Model and Code with wrong implementation

The following table shows a vector set with 100% MC/DC coverage for the C-Code example. Both outport results show the same result on the model and on the code. The error is masked and not detected, because an insufficient set of vectors is used.

Vector name	**Signal 1**	**Signal 2**	**Signal 3**	**Signal 4**	**Reference Model**		**Wrongly Implemented Code**	
					Switch	**Outport**	**Switch_out**	**Outport**
Vector 1	10	2	10	9	10	9	10	9
Vector 2	10	3	10	10	10	10	10	10
Vector 3	10	4	10	11	10	10	10	10

There are two reasons why the difference between model and code is not found for this set of vectors:

1. If Signal1 and Signal3 have always the same value as in the example. It does not matter whether the “if” or “else” block is executed. Always the same value is observable at the output. For the switch block, the same statement like for relational operator block holds: additional test properties are needed.
2. Even if output value of Switch block were different (see table below) between Model and Code, min block could filter the difference, if the vector set is not optimal. Note that the error is visible now on the outport of the switch, but the switch is not part of the observable interface during testing.

Vector name	Signal 1	Signal 2	Signal 3	Signal 4	Reference Model		Wrongly Implemented Code	
					Switch	Outport	Switch_out	Outport
Vector 1	10	2	**11**	10	11	10	10	10
Vector 2	10	3	**11**	12	11	11	11	11
Vector 3	10	4	**11**	10	10	10	11	10

To increase test quality by preventing this masking effect, DENSO developed a methodology to solve the mentioned two issues:
a) assure correct stimuli values at block inputs to prevent situations like the one described for the relational operator or switch block.
b) assure errors are always propagated to visible interface objects.
The developed methodology is model based as it is unintuitive to find conditions which can keep propagation by code analysis.
The remainder of the paper is structured as follows: In section 2, we will demonstrate an approach addressing issue a) using custom defined test properties for blocks of models. Section 3 describes how we enable observability of these defined test properties on the visible interface, therefore addressing issue b). Section 4 describes the implementation of this methodology within BTC EmbeddedTester based upon C-Observer and Automatic Vector Generation technology to automatically generate stimuli vectors covering all test properties and assuring observability on the visible interface.

2 Test Properties for Blocks of a Model

In the previous section, we showed an example that MC/DC Code Coverage is not enough to capture all possible combinations…
In the example, a vector set was used where MC/DC was fulfilled, but an error was not visible due to the fact that Signal1 and Signal3 where always identical.

So it is not sufficient to just evaluate the relational operator decision (result of a switch block in the code) to true and false, but to test it in more detail. The same holds basically for any other kind of model block type.
Custom test properties define the behavior of a block, how it should be translated. A set of defined custom test properties for a used block set results in an own Model Coverage definition.

Example Switch block:

We know that errors can be masked, if the used vector set does not assure that Signal1 and Signal3 are always different. Furthermore, we should test both cases where the comparison of Signal2 against the switch boundary value results to true and false. Therefore, the following test properties are of importance for all switch blocks within a model:

in2 > threshold && in1 !=in3
in2 < threshold && in1 !=in3

Example MinMax block

For a MinMax block, the set of needed test properties is straightforward and very similar to what is needed to reach MC/DC coverage.

in1 > in2
in1 < in2

The additional test property definition is done for each block type of your supported block set.
Within the proposed solution, defined test properties are the main intellectual property of each software development company. It has major influence on resulting vectors and therefore on resulting test quality. The technical approach developed by BTC Embedded Systems (as described in section 4) enables a customer to come to an own Model Coverage definition fulfilling the specific test goals and guidelines within a company.

3 Towards Enhanced Observability on Model Level

Once additional test properties are defined, we can assure to test exactly the needed properties. But as described in section 1, we still cannot be sure that a failure visible at a block is still visible on the observable interface.
For this reason, we introduce three additional concepts:

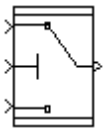
Observability Condition

An Observability Condition is defined on block type level. It is defined for a block input and defines the condition that assures that the input value is visible at the output. Example: a MIN block with two inputs has two observability conditions. One condition to make in1 visible at the output (in1 > in2) and one condition to make in2 visible at the output (in1 < in2).

Custom Test Objective

It must be made sure that each test property for each block within a model is observable. This is model specific. For each block within a model, a test property is combined with observability conditions lying on a so called Observability Path. An Observability Path starts from the block under test and ends on an observable interface. This together will form a Custom Test Objective. A Test Property has as many Custom Test Objectives as Observability Paths exist. Always all custom test objectives are considered as it increases the probability to find a test for at least one of these custom test objectives as not each one may be reachable.

The following example from section 1 looks as follows. It shows test properties for each block type as well as Observability Conditions for each block type. It contains one Observability Path from the **Switch** block over **Min** block to **OutPort.**



Test Properties for Switch block

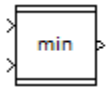
swT1: in2 > threshold && in1 != in3

swT2: in2 < threshold && in1 != in3

Observability Conditions

swC1: in2 > threshold

swC2: in2 <= threshold



Test Proprties for Min block

minT1: in1 > in2

minT2: in1 < in2

Observability Conditions

minC1: in1 < in2

minC2: in1 > in2

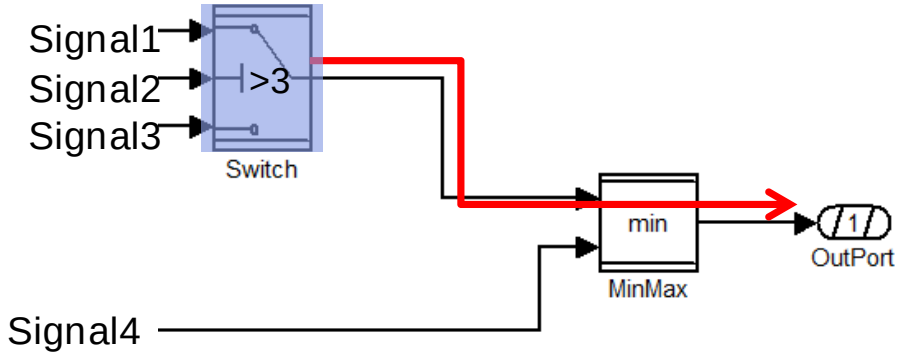


Figure 2: Observability Path from Switch Block to visible OutPort

To enable testing of **swT1** and **swT2** for a switch block, these test properties must be combined with observability condition minC1. The **Switch** output is only visible at **OutPort**, if the value of the first input is smaller than the second input value. Based on this, the following Custom Test Objectives can be derived for the Switch Block shown in Figure 2. Each signal placeholder within the conditions are replaced by concrete signal names of the model:

Custom Test Objectives for switch block in the model

CTO1: **Signal2 > 3 && Signal1 != Signal3 && Switch < Signal4**

CTO2: **Signal2 < 3 && Signal1 != Signal3 && Switch < Signal 4**

If a set of vectors fulfills these custom test objectives, it is assured that errors are not masked. The following table shows the vector set from section 1 and a new vector set fulfilling both custom test objectives. For each CTO there is one vector that covers it. Within a Back – to – Back Test, the outport of the model and the code are different.

Vector name	Signal 1	Signal 2	Signal 3	Signal 4	Reference Model		Wrongly Implemented Code	
					Switch	Outport	Switch_out	Outport
Vector 1	10	2	11	10	11	**10**	10	**10**
Vector 2	10	3	11	12	11	**11**	11	**11**
Vector 3	10	4	11	10	10	**10**	11	**10**
CTO1	10	5	11	12	11	**10**	10	**11**
CTO2	10	0	11	12	10	**11**	11	**10**

4 Automatic Test Vector Generation for Custom Test Objectives

Custom test objective derivation and test vector creation for such objectives may become very time consuming and error prone when done manually. Therefore, BTC Embedded Systems AG implemented a solution based on BTC EmbeddedTester [BTCEW2010]. The tool internally uses a so called Virtual Verification Platform (“VVP”) as a semantic basis for any kind of analysis techniques, like automatic test vector generation algorithms/engines. In this case, the behaviour description of the system under test (“SUT”), the environment of the SUT and the target property specification is given as C-Code within the VVP-Architecture (see Figure 3), which can be seen in the following figure. C-Code as a semantic basis for test- and verification activities has a lot of advantages in practice as c-code is a de facto standard in the development of embedded systems in the automotive domain. So any given C-Code of the SUT or the Environment Specification can be re-used in this approach.

The base technology of the existing testing environment can use self-contained C-Code in order to automatically analyse the SUT regarding any given test- and check property. The SUT with its software architecture (functions and its wiring) is given as self-contained C-Code automatically generated by the auto code generator of the functional or implementation model. The environment of the SUT is also given as C-Code, which can be reused from any plant model descriptions or can be synthesized from given environment high-level specifications.

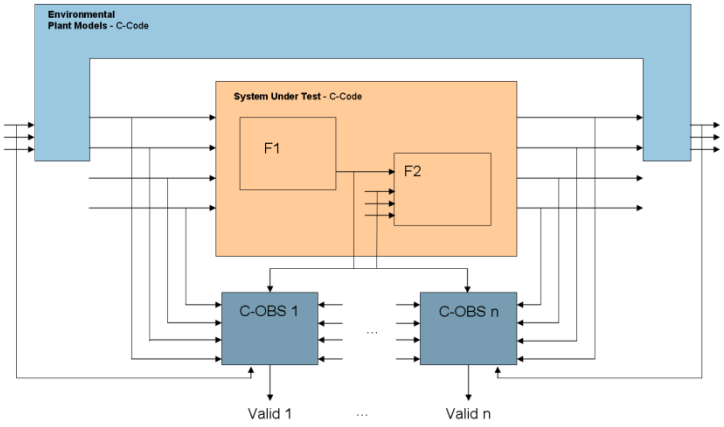


Figure 3: Virtual Verification Environment enhanced by C-Code-Observers (see also [BTCEW2011])

Any kind of system properties are represented by the so called C-Observers (C-OBS1..n). These observers are in general small c-programs which are running in parallel to the SUT during any test or analysis step in order to observe the correctness of the behavior of the SUT in respect to the described requirements or the purpose of automatically generate desired test scenarios for property

coverage. The C-Observer Functions returns the so called valid-signal (Valid1..n), which indicated accepted behavior with a TRUE (1) or error states with a FALSE (0). This allows automating the test generation and validation totally, if the properties are fully represented by such observers.

Figure 4 shows a simplified BTC Embedded Tester Back-to-Back Testing workflow that fully automates the methodology described in the previous sections based on ATG and C-Observer technology.

As input, BTC EmbeddedTester gets a dSPACE TargetLink model, its corresponding generated C-Code and an XML file holding all defined test properties and Observability conditions. Compared to standard Back-to-Back Testing workflow, only the mentioned XML file is an additional user input. All subsequent steps are done automatically by the developed solution based on BTC EmbeddedTester.

In a first step, we have to assure that a C-Observer implementing custom test objectives is able to “see” each single block within the C-Code. For this purpose, an internal TL model is created and annotated based on the original model. Afterwards, C-Code is generated using TargetLink. This C-Code assures the needed visibility for the model.

In a next step, C-Observers are automatically generated based on the given XML file to represent derived Custom Test Objectives for the model under test. These C-Observers together with the internally generated C-Code are the basis for the ATG engines of BTC EmbeddedTester. Now, we are able to automatically generate vectors for the defined custom test objectives or even formally verify that a specific custom test objective is unreachable. The vector generation produces a set of vectors to be used for Back-to-Back test in step 4. Furthermore, a Model Coverage Report based on the defined block test properties within the XML file and the concrete model is created.

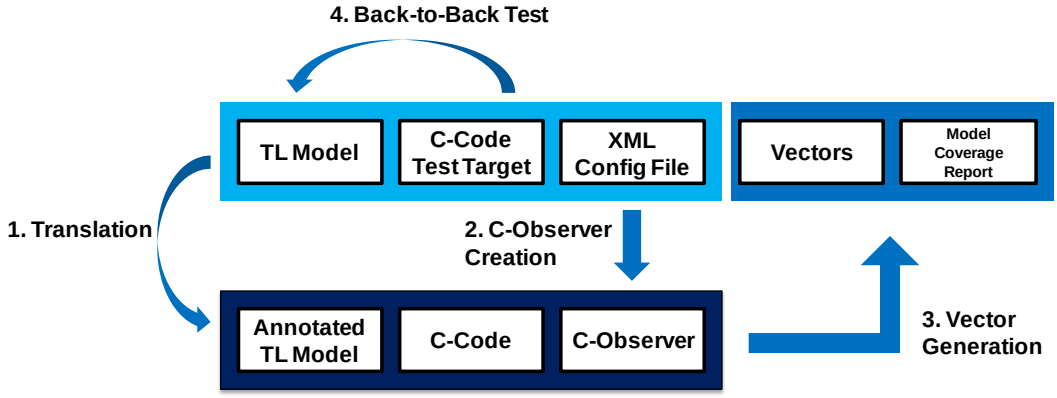


Figure 4: Simplified Workflow within BTC EmbeddedTester

5 Conclusion

In this paper we introduced a solution developed together with the Japanese tier one supplier DENSO to increase test quality for ISO26262 compliant structural Back-to-Back testing.
We introduced a methodology that solves the drawbacks of existing code coverage criteria. It allows to describe an own model coverage definition using custom defined test properties for blocks of models. These model based test properties assure that a block's behavior will be correctly tested.
Furthermore, the methodology not only assures that tests cover all defined test properties. It also guarantees that possible errors are propagated to the visible interface.
We implemented the methodology within BTC EmbeddedTester based upon C-Observer and Automatic Vector Generation technology to simplify the application. The solution automatically generate stimuli vectors covering all test properties and assuring observability on the visible interface.

Bibliography

[ISO26262] Road vehicles – Functional Safety, International Organization for Standardization, ISO 26262, 2011

[BTCEW2010] Fully Automated Back-to-Back Testing to support ISO 26262 Compliant Software Development, Hans Holberg and Dr. Brockmeyer, Embedded World Conference 2010

[BTCEW2011] ISO 26262 compliant verification of functional requirements in the model-based software development process, Hans Holberg and Dr. Brockmeyer, Embedded World Conference 2011