ISO 26262 compliant software verification using highly automated test technologies
Introduction

Requirement-based Testing

Back-to-back Testing

Formal Specification and Formal Verification

Conclusion
1. Do you want to test your model or your code?

2. How much time do you spend with writing and executing test cases?

3. What if your PC could understand your requirements?
BTC EmbeddedTester is certified as „Fit for purpose“ for ISO26262

Reference Workflow for TargetLink + BTC EmbeddedTester available
Process and Tools for Model-based Development

Informal Requirement
- EmbeddedTester Base
  - Test Definition

MathWorks Simulink
- EmbeddedSpecifier
  - Formal Notation

Modeling
- EmbeddedTester Base
  - Requirement Based Testing

Model
- EmbeddedValidator
  - Formal Verification

Formal Requirement
- EmbeddedTester Base
  - Requirement Based Testing

Production Code
- EmbeddedTester Base
  - Requirement Based Testing

Automatic Code Generation
- EmbeddedValidator
  - Formal Verification

Back-to-Back Testing
- EmbeddedTester
  - Back-to-Back Testing

TargetLink
- EmbeddedSpecifier
  - Formal Notation
Agenda

- Introduction
- **Requirement-based Testing**
- Back-to-back Testing
- Formal Specification and Formal Verification
- Conclusion
Too many defects being introduced!

Defects being detected too late!

Problem: 80% of development costs are spent identifying and fixing defects

Solution: Systematic and efficient unit test allows to discover errors earlier
Model-based Testing - Challenges

- **Traceability between Tests and Requirements**
- **Test execution on Model and Production Code**
- **Model Coverage**
- **Code Coverage**
- **Handling of Calibration Parameters**
- **Reporting**
- **Debugging**

**Problem:** A collection of not well integrated tools and scripts

**Solution:** Integrated Test Environment as „one-stop“ solution for model and code
Requirement-Based Testing with BTC EmbeddedTester

**Debugging:**
- Simulink
- TargetLink
- Visual Studio

**BTC EmbeddedTester**
- Import Requirements
- Write Test Results
- Import Test Architecture
- Execute Test Vectors
- DOORS/PTC Integrity/Excel/Word

**Reporting:**
- Test Results
- Requirements Coverage
- Model Coverage
- Code Coverage

MIL  .c/.h  .obj
SIL  PIL
Before creating or importing test cases, requirements can be imported in order to link and trace test cases to requirements.

Direct access to DOORS or PTC Integrity Databases.
Embedded Tester – Requirement-based Testing

Create Test Cases

Execute Test Cases

Calculate Coverage

Create or import/export functional tests
Embedded Tester – Requirement-based Testing

Create Test Cases ➔ Execute Test Cases ➔ Calculate Coverage

Model Refinement

Model

Simulink-MIL ➔ TargetLink-MIL

Automatic Code Generation

C-Code

SIL

Compilation

Obj-Code

PIL

Compare to reference

Model

Test Cases

Import

Simulation

?
1. Strong hierarchical approach

- Automatic analysis of model hierarchy
- Easily test and debug sub-functions in model and code
- No need to extract system under test manually

2. Grey Box

- Automatic detection of interface variables on all hierarchy levels
- Possibility to treat DISP variables as output
- Possibility to treat CAL variables as input
Embedded Tester – Test Execution

- Test Report Generation (HTML, PDF).
- Automatic comparison and setting of PASSED/FAILED.
- Direct link to test management for view or export for debugging.

<table>
<thead>
<tr>
<th>Name</th>
<th>Length</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>tv_ss3_tl_mil5</td>
<td>1</td>
<td>PASSED</td>
</tr>
<tr>
<td>tv_ss3_tl_mil6</td>
<td>2</td>
<td>FAILED</td>
</tr>
</tbody>
</table>

Tolerance Definitions

<table>
<thead>
<tr>
<th>Check</th>
<th>Name</th>
<th>Kind</th>
<th>Data Type</th>
<th>LSB</th>
<th>Offset</th>
<th>Unit</th>
<th>Relative Tolerance</th>
<th>Absolute Tolerance</th>
<th>Relative Path</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>f2[#1]</td>
<td>Outpt</td>
<td>Int16</td>
<td>1</td>
<td>0</td>
<td>n.a.</td>
<td>0%</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Testvector comparisons

<table>
<thead>
<tr>
<th>Comparison #1</th>
<th>Result</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference Vector</td>
<td>tv_ss3_tl_mil5</td>
<td>length: 1</td>
</tr>
<tr>
<td>Simulated Vector</td>
<td>n.a.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Comparison #2</th>
<th>Result</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference Vector</td>
<td>tv_ss3_tl_mil5</td>
<td>length: 2</td>
</tr>
<tr>
<td>Simulated Vector</td>
<td>n.a.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deviations</th>
<th>Name</th>
<th>State</th>
<th>Reference Value</th>
<th>Simulated Value</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>f2[#1]</td>
<td>0</td>
<td>-49654.0</td>
<td>15882.0</td>
<td>65536.0</td>
<td></td>
</tr>
<tr>
<td>f2[#1]</td>
<td>1</td>
<td>144196.0</td>
<td>13124.0</td>
<td>-131072.0</td>
<td></td>
</tr>
</tbody>
</table>
Debugging Environment can be created for any hierarchy level!
9.4.4  To evaluate the completeness of test cases and to demonstrate that there is no unintended functionality, the coverage of requirements at the software unit level shall be determined and the structural coverage shall be measured in accordance with the metrics listed in Table 14. If necessary, additional test cases shall be specified or a rationale shall be available.

Table 14 — Structural coverage metrics at the software unit level

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1a</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Statement coverage</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1b</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>Branch coverage</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1c</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>MC/DC (Modified Condition/Decision Coverage)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE 2  In the case of model-based development, software unit testing may be moved to the model level using analogous structural coverage metrics for models.
Bi-directional traceability between requirements and test cases

Identify untested requirements

Identify violated requirements
- Coverage information based on Simulink V&V Toolbox
- Coverage is cumulated for runs on different model hierarchy levels
- Intuitive Graphical Colouring of Simulink and Stateflow charts.
Reporting: Code Coverage Analyse Report

- Global Code Coverage (Coverage Statistics, Condition, Decision, C/DC, MC/DC, Switch and Function Coverage)
- Detailed Code Coverage (UID for test properties, links to the code and model parts)
- Coloured Code Coverage (Source Code with Coloured Coverage Indication)

Create Test Cases  
Execute Test Cases  
Calculate Coverage

- **Tests**  
  - Statement Coverage: 16 (% 88.9%)  
  - Decision Coverage: 4 (75%)  
  - Condition Coverage: 0 (n.a.)  
  - Condition/Decision Coverage: 7 (85.7%)  
  - Modified Condition/Decision Coverage: 7 (85.7%)

- **Unreachable**  
  - (n/inf)  
  - 0  
  - 0  
  - 0  
  - 0  

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique ID</td>
<td>TC10</td>
</tr>
<tr>
<td>File</td>
<td>C:\DOKUME<del>1\avalea\LOKALE</del>1\Temp\tgcv\PROF_1~1\static\codegen\TLProj\Module\Module.c</td>
</tr>
<tr>
<td>Line</td>
<td>212</td>
</tr>
<tr>
<td>Shared With Functions</td>
<td>SimpleMinMax/TL/Module/Subsystem/Module/F1</td>
</tr>
<tr>
<td>Target/Link Blocks</td>
<td>(Module/F1/min)</td>
</tr>
<tr>
<td>Expression</td>
<td>((Sa_max)&lt;(Sa1_In3))</td>
</tr>
</tbody>
</table>
| Properties | D:10:0 decision became false  
D:10:1 decision became true  
unknown | covered | import |

```c
208  /* Requirements: Compute
209  Description: REQ_2
210  Document: Simple
211  Location: B1
212  MinMax: Module/F1/mi
213  if (Sa2_max) < (Sa1_In
214  /* TL_AUTO_COMMENT B
215  Sa2_min = Sa2_max;
216  }
217  else {
218  /* TL_AUTO_COMMENT B
219  Sa2_min = Sa1_In3;
220  }
221  }
```
Agenda

- Introduction
- Requirement-based Testing
- Back-to-back Testing
- Formal Specification and Formal Verification
- Conclusion
Back-2-Back Test - Motivation

### Problem:
- High quality test requires appropriate test scenarios. Standards like ISO 26262 define corresponding coverage objectives for the unit test of embedded software.
- Objectives like MC/DC are extremely difficult to achieve with manually written tests.

**Solution:** A push-button solution for automatic test case generation and execution

---

Table 14 — Structural coverage metrics at the software unit level

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1a Statement coverage</td>
<td>++</td>
</tr>
<tr>
<td>1b Branch coverage</td>
<td>+</td>
</tr>
<tr>
<td>1c MC/DC (Modified Condition/Decision Coverage)</td>
<td>+</td>
</tr>
</tbody>
</table>

Source: ISO 26262
Automatic Testcase Generation – Random vs. Model Checking

Problem:
- Random algorithms typically generate a large set of long and redundant test cases
- Result:
  - Test execution and debugging is time consuming
  - No information available about the parts of Model or Code that are not covered

Solution:
- Model Checking performs a complete mathematical analysis of the system under test
- Result:
  - Short and non-redundant test cases for maximal coverage
  - Mathematical proof, that uncovered parts are unreachable
EmbeddedTester – Workflow for Back-to-Back Testing

Model

Simulink-MIL

Model Refinement

Model

TargetLink-MIL

Automatic Code Generation

C-Code

Compilation

SIL

Stimuli Vectors

~100% Coverage

Generate automatically

Stimuli Vector Generation

Record Reference Outputs

Back to Back Testing

Back to Back Testing

Obj-Code

PIL
BTC EmbeddedTester - Test Goals

- **Structural Coverage Goals**
  - Statement Coverage
  - Condition Coverage
  - Decision Coverage
  - Switch-Case-Coverage
  - Function-Call-Coverage
  - Modified Condition/Decision Coverage
  - Domain Coverage

- **Robustness Analyse**
  - Relational Operators
  - Division-by-Zero
  - Down-casting
  - Range Violation
  - Unreachable

### Example

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Relational Operator (GEQ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique ID</td>
<td>TC92</td>
</tr>
<tr>
<td>File</td>
<td>CL_Controller.c</td>
</tr>
<tr>
<td>Line</td>
<td>1032</td>
</tr>
<tr>
<td>Shared With Functions</td>
<td>CentralLockingTL_OK/CL_Controller/Subsystem/CL_Controller</td>
</tr>
<tr>
<td>TargetLink Blocks</td>
<td>(CL_Controller/Aggregate_Behavior/Motor_DriverDoor)</td>
</tr>
<tr>
<td>Expression</td>
<td>Ch1_Motor_DriverDoor_ctr&gt;=3</td>
</tr>
<tr>
<td>Properties</td>
<td>RO:92:1 operation became true</td>
</tr>
<tr>
<td></td>
<td>RO:92:2 operation became false</td>
</tr>
<tr>
<td></td>
<td>RO:92:3 (left-right) == 1</td>
</tr>
<tr>
<td></td>
<td>RO:92:4 (left-right) == 0</td>
</tr>
<tr>
<td></td>
<td>RO:92:5 (left-right) == 1</td>
</tr>
</tbody>
</table>
1. Model vs. Code
   - Typical use case in order to compare Simulink Model (MIL) to C-Code (SIL/PIL)

2. Simulink Model vs. TargetLink Model
   - Useful in case an original Simulink Model (e.g. provided by a customer) is modified to become a TargetLink Model (e.g. because of unsupported blocks or for optimization reasons)

3. Old Target vs. New Target
   - In case existing code needs to be used on a new Target Processor or in case the Target Processor changes during the project, a Back-2-Back Test PIL vs. PIL can show that this has no impact on the behavior of a software unit

4. Current Model Version vs. Previous Model Version
   - Automatic regression test between model versions

5. Current Matlab/TargetLink Version vs. Previous Version
   - The EmbeddedTester Migration Suite allows to verify automatically, that the migration to a new Matlab and/or TargetLink version does not change the behavior of models and production code.
Agenda

• Introduction
• Requirement-based Testing
• Back-to-back Testing
• **Formal Specification and Formal Verification**
• Conclusion
A Hierarchy of Notation Methods is defined

The more safety critical a function is, the more formal the notation and verification is recommended
Challenges when specifying requirements in a formal way

- **Problem 1**: Some languages that might be used to express requirements are not formal

- **Problem 2**: Formal methods are often considered to be too mathematical and too difficult to learn

- **Solution**: A tool and a method that allows engineers to take their textual requirements and intuitively derive semi-formal and formal notations

Example of a formal specification in LTL

\[
S \rightarrow x.((\Diamond_{x+n,x+n} \text{TRUE}) \rightarrow \Box (y.(\Diamond \text{TSE} \rightarrow (\text{TC} U_{[y+\min,y+\max]} \text{TEE}) \rightarrow z.(\Diamond_{z+\min,z+\max} \text{ASE} \wedge (\Diamond_{z+\min,z+\max} \text{ASE} \rightarrow r.(\text{AC} U_{[r+\min,r+\max]} \text{AEE}))))))
\]
Formal Specification with Patterns

- Intuitive formalization process thanks to Pattern library in EmbeddedSpecifier
- Non-ambiguous representation helps to improve the quality of requirements
- Formalized requirements are later used in Formal Verification process
- “Proven in use” in Automotive and Avionics Industry
Motivation for simulation-based formal verification

**Problem:**
- Testcases are typically created per requirement
- It might stay undetected, if e.g. Test3 violates Requirement1

**Solution:**
- Use a Requirement Observer to automatically observe the status of each requirement during the complete test process.

![Diagram showing the relationship between requirements and test cases](image-url)
Simulation-based Formal Verification

Solution 1: Online-Verification

BTC EmbeddedSpecifier
- Export Formal Specification

Test Environment

Requirement Observer
- Test Cases
- System Under Test
- Observe
- Requirement Status
  - Fullfilled / Violated

Solution 2: Offline-Verification

BTC EmbeddedValidator BASE
- Import
- Test Data
- Formal Specification

Test case XY violated
- Requirement 5
- Requirement fullfilled
Model Checking vs. Testing

**Problem:**
- A testcase only represents one possible path through the system
- It is impossible to cover all paths with test cases

**Solution:**
- Model checking analyses all possible paths and guarantees a bug-free system
EmbeddedValidator - Method

Formal Specification

Formal Verification

Safety Requirements

BTC EmbeddedSpecifier

BTC EmbeddedValidator

dSPACE TargetLink

TargetLink Code

BTC Formal Specification

BTC Formal Verification

Simulation-based

Complete Analysis

TargetLink Code

BTC EmbeddedValidator

Safety Requirements

Code does not fulfill the requirement

Counter Example

Code fulfills requirement
• About BTC
• Introduction
• Requirement-based Testing
• Structural Testing (Back-to-back Testing)
• Formal Specification and Formal Verification
• Conclusion
Conclusion – 3 Test Methods

- **Requirements-based testing** usually finds about 20-40% of the problems.
- Additional 30-40% of the software problems can be directly found by using **structural testing** and **back-to-back** comparison (Small effort due to automatic test case generation and test execution).
- **Formal verification** is especially relevant for testing of safety-critical software.

- Combination of test methods is recommended to achieve high quality
- ISO 26262 provided guidelines on the test methods to be used for each ASIL
Conclusion

- Requirement-Based-Testing for Simulink/TargetLink-Models and C-Code
- Highly integrated with dSPACE TargetLink
- Connection to Requirements-Management Tools like DOORS
- Automatic test execution (MIL/SIL/PIL) on any hierarchy level
- Automatic generation of debugging environments
- Integrated coverage measurement (requirements coverage, model coverage, code coverage)
Efficiency Improvements

Customer experiences show a decreasing overall test effort by 50 to 70%

- Test case generation effort for maximal structural coverage could be minimized by 90%!
- Effort savings of up to 70% during test execution and test evaluation phases
- Half of the debugging effort could be saved by using interactive and automatic debugging tool support

Quality Improvements

Improved „Maturity Gates“

- In average the MC/DC Coverage rates are 30% higher in contrast to manual test approaches

Process Improvement

ISO26262 Certified Tool-Chain with BTC EmbeddedTester supports Standard conform development and test process
Can be intuitively used by requirements engineers or function developers to create semi-formal and formal requirements

(Semi-)Formal and unambiguous representation helps to improve requirements quality

Import from (and traceability to) RM-tools like DOORS and PTC Integrity

Automatic translation into machine readable specification in order to use requirements in the verification process

Directly supports safety standards like IEC 61508 & ISO 26262

Highly recommended for Safety Critical Applications
✓ Simulation Based Formal Verification allow crosschecking of all test cases against all requirements

✓ Integrated reporting with full traceability to original requirements

✓ Flexible concept for supporting different test environments:
  ✓ Support for Simulink models and TargetLink models available
  ✓ Support for dSPACE VEOS and dSPACE HIL-Systems planned for 2014
  ✓ Other test architectures can be easily integrated
Mathematical proof that Code can never violate a requirement

Check requirements on any hierarchy level of a model

For requirements that can be violated, a counter example and a debugging model is generated automatically
Thank you.

Hans J. Holberg
Senior Vice President Marketing & Sales
Gerhard-Stalling-Str. 19, 26135 Oldenburg, GERMANY
Tel.: +49 441 969 738 14