Formal Verification and Automatic Testing for Model-based Development

in compliance with ISO 26262
Is your software safe?
Do you have evidence?
Focus of this session

To answer to those questions,

This session focuses on

Software Verification and Testing

with respect to ISO 26262 Part 6 and 8.
Verification on Design Phases
ISO 26262 requires...

To ensure model or software code comply with requirements

"In the design phases, verification is the evaluation of the work products, such as requirement specification, architectural design, models, or software code, thus ensuring that they comply with previously established requirements for correctness, completeness and consistency" [ISO 26262-8, clause 9]
Several methods are recommended in ISO 26262-6. Our solution is Formal Verification.

ISO 26262-6, Clause 8: Table 9—Methods for the verification of software unit design and implementation

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1a Walk-through(^a)</td>
<td>++</td>
</tr>
<tr>
<td>1b Inspection(^a)</td>
<td>+</td>
</tr>
<tr>
<td>1c Semi-formal verification</td>
<td>+</td>
</tr>
<tr>
<td>1d Formal verification</td>
<td>O</td>
</tr>
<tr>
<td>1e Control flow analysis(^b, c)</td>
<td>+</td>
</tr>
<tr>
<td>1f Data flow analysis(^b, c)</td>
<td>+</td>
</tr>
<tr>
<td>1g Static code analysis</td>
<td>+</td>
</tr>
<tr>
<td>1h Semantic code analysis(^d)</td>
<td>+</td>
</tr>
</tbody>
</table>

**NOTE:** Formal verification is rated only "o" or "+". However, it's usage is same as semi-formal verification, and it's more complete and highly automated. BTC-ES supports both of these verification technologies.

++ = highly recommended
+  = recommended
o  = no recommendation or against its usage
Formal Verification (Model Checking)

- **Model (or C-Code)**
- **Software Requirements**
- **Analysis Engine**
- **Formal Requirements**

**Design** → **Software Requirements** → **Analysis Engine** → **Formal Requirements**

- **Requirements violation found**: BAD Model
- **Requirements ALWAYS fulfilled**: GOOD Model

Analyzes model and reports result
Advantage of Formal Verification

Mathematically COMPLETE verification is possible

Simulation
- Initially limited runs can be evaluated
- Requirement violations may be missed

Formal Verification
- All possible runs can be evaluated
- All requirement violations can be detected
Let's look an example of a software requirement for Adaptive Cruise Control (ACC).

Requirement 1:
When the brake- or acceleration-pedal is pressed, the ACC system shall be deactivated in 20ms in any situation.
Advantage of Formal Verification: Example

Verification target is complex.
How do you make sure that requirement is always fulfilled by the model? Do you make it by simulation?

### Test Case 1

<table>
<thead>
<tr>
<th>Interfaces / Time</th>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4</th>
<th>Step 5</th>
<th>Step 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Input] Brake Pedal</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>[Input] Accel Pedal</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[Input] Vehicle Speed</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>[Input] Distance</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[Output] ACC Active</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* 1 Step = 10 ms

Simulation → Model

ACC deactivated → GOOD

Record outputs
Advantage of Formal Verification: Example

What is the other situations?

- ACC was active for 10 hours.
- Both pedals are pressed at the same time.
- ACC start button is kept pressed.
- Gear is set to Neutral.
- Acceleration Pedal is pressed very short time.

Can we cover all possible situations? → **NO**

It's **impossible** by simulation, but **possible** by Formal Verification.
Advantage of Formal Verification: Example

Requirement 2:
Absolute acceleration value shall never exceed 2.5km/h per second.

Can you imagine test case for simulation?

→ NO

REASONS:
• Evaluation of infinite time is impossible by simulation.
• This requirement just prohibits wrong behavior, specific situation is not given. We cannot define any specific inputs to the model.

Formal Verification is Possible without specific Test case.
Formal Verification requires Formal Requirements

Our technology enables intuitive requirements formalization which enables your PC understands your requirements.

Language for requirements formalization is

**Simplified Universal Pattern(SUP)**
Verification on Testing Phases
ISO 26262 requires...

To ensure software comply with requirements within a test environment

"In the test phases, verification is the evaluation of the work products within a test environment to ensure that they comply with their requirements. " [ISO 26262-8, clause 9]

The requirement is very similar to verification on design phases. What is the difference?
Verification on Testing Phases

ISO 26262 says...

Test environment shall be as closely as possible to the target environment

"The test environment for software unit testing shall correspond as closely as possible to the target environment." [ISO 26262-6, clause 9] "The test environment for software integration testing shall correspond as closely as possible to the target environment." [ISO 26262-6, clause 10]

and MILs, SILs, PILs, HILs are recommended.

They mean,

- Some of test targets can be black box,
- and Execution of test target is required. in this case,

→ Formal Verification is infeasible.
Different verification method from design phases are recommended for testing phases. It's Requirements-based test.

ISO 26262-6, Clause 9/10: Table 10/Table 13—Methods for software unit/integration testing

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>D</td>
</tr>
<tr>
<td><strong>1a Requirements-based test</strong></td>
<td>++</td>
</tr>
<tr>
<td></td>
<td>++</td>
</tr>
<tr>
<td></td>
<td>++</td>
</tr>
<tr>
<td></td>
<td>++</td>
</tr>
<tr>
<td><strong>1b Interface test</strong></td>
<td>++</td>
</tr>
<tr>
<td></td>
<td>++</td>
</tr>
<tr>
<td></td>
<td>++</td>
</tr>
<tr>
<td></td>
<td>++</td>
</tr>
<tr>
<td><strong>1c Fault injection test</strong></td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>/+++</td>
</tr>
<tr>
<td></td>
<td>++</td>
</tr>
<tr>
<td><strong>1d Resource usage test</strong></td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>++</td>
</tr>
<tr>
<td><strong>1e Back-to-back comparison test between model and code</strong>,</td>
<td>+</td>
</tr>
<tr>
<td>if applicable</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>++</td>
</tr>
<tr>
<td></td>
<td>++</td>
</tr>
</tbody>
</table>

“++” = highly recommended  
“+” = recommended  
“o” = no recommendation or against its usage

Our solution is...

enhanced Requirements-based test by Formal methods, named

**Formal Test**
Formal Test (Enhanced Requirements-based test)

- Test cases
- Analysis
- Software Requirements
- Requirements fulfilled
- Requirements violated
- Test FAILED
- Test PASSED
- Model
- C-Code
- Object Code
- ECU
- Design
- Formalization
- Verdict
Formal Test is Flexible and Scalable.

Supported by Formal Test
Automatic cross check between Formal Requirements and Test cases is possible.

Test Case 1

Test Case 2

Test Case 3

Test Case N

Requirement 1

Requirement 2

Requirement 3

Requirement M
But the issue regarding completeness of test cases comes back.

Simulation based testing can never be complete.

Simulation

initial state

requirement violation

Test cases

Only limited runs can be evaluated

→ Requirement violations may be missed
Completeness of Test cases

ISO 26262 recommends measurement of structural coverages to evaluate completeness of test cases.

### ISO 26262-6, Clause 9: Table 12 — 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>

“++” = highly recommended
“+” = recommended
“o” = no recommendation or against its usage

### ISO 26262-6, Clause 10: Table 15 — Structural coverage metrics at the software architectural level

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1a Function coverage&lt;sup&gt;a&lt;/sup&gt;</td>
<td>+</td>
</tr>
<tr>
<td>1b Call coverage&lt;sup&gt;b&lt;/sup&gt;</td>
<td>+</td>
</tr>
</tbody>
</table>
High Coverage Test cases

However, coverage may not be measurable on test targets.

Our technology enables automatic test cases generation with respect to coverages on the left side of V, and execution on the right side of V.
Highly automated and high completeness verification technologies in compliance with ISO 26262 are available.

- **Formal Verification** for Design Phases
- **Formal Test** for Testing Phases

They provide evidence of your verification regarding functional safety requirements.

**Why not BTC EmbeddedPlatform?**
Thank you.

MDS테크놀로지(주) www.mdstec.com
SDS사업부 전자제어팀 정보철 과장
031-627-3114 / 010-5469-1327
bochul@mdstec.com