MATLAB EXPO 2017
KOREA
4월 27일, 서울
등록 하기 matlabexpo.co.kr
Verification Techniques in Model-Based Design for High Integrity System

Young Joon Lee
Principal Application Engineer
Key Takeaways

1. Find bugs early, develop high quality software

2. Replace manual verification tasks with workflow automation

3. Learn about reference workflow that conforms to safety standards

“Reduce costs and project risk through early verification, shorten time to market on a certified system, and deliver high-quality production code that was first-time right”  Michael Schwarz, ITK Engineering
Safety of Electronic Systems

- Critical functionality in industries such as Automotive, Aerospace, Medical, Industrial Automation
- Real-time operation
  - Compute time lag cannot be tolerated
- Predictable behavior
  - No unintended functionality
- Must be robust
  - Program crash or reboot not allowed
Role of Certification Standards

- ISO 26262 (Automotive)
  - Defines functional safety for automotive electronic systems
  - Automotive Safety Integrity Level ASIL QM, A to D (least to most; derived from severity, controllability, probability)
  - ISO 26262-6 pertains to software development, verification, and validation

- DO-178 (Avionics)
  - Guidelines for the safety of software in certain airborne systems
  - Level A to E (most critical to least)
  - Verification activities include review of requirements and code, testing of software, code coverage

- IEC 62304 (Medical Device)
  - Describes software development and maintenance processes for medical device software
  - Safety levels Class A to C (least critical to most)
  - Identifies various verification and testing activities
Traditional Development Process

- Start with a paper design
- Manually determine system architecture
- Identify algorithms for the application
- Start writing code for the algorithms
- Develop testing platform to unit test algorithms
- Manually unit test the code with the testing platform
- Test the design with the real hardware and code
- Find bugs, fix bugs, repeat … very painful !!!
Problems with Traditional Development Process

- Errors introduced during development
- They are detected late in the process
- Latent errors remain in the software

- Develop some tests
- Unit test on PC/Desktop
- Integration test on hardware or in the field
Addressing Design and Development Challenges

It is easier and less expensive to fix design errors early in the process when they happen.

Model-Based Design enables:

1. Early testing to increase confidence in your design
2. Delivery of higher quality software for production use
3. Credits and artifacts for certification to satisfy safety standards
Model Based Design

- Modeling
  - Model algorithms and environment
  - Explore design alternatives and options

- Simulation
  - Design exploration with simulation
  - Find issues early, on your desktop PC

- Production code
  - Code generated automatically from model
  - Early verification for high quality code
Reference Verification and Validation Workflow
Reference Verification and Validation Workflow

- Certifiable Model-Based Design Workflow to develop critical embedded software
- Reviewed and approved by TÜV SÜD certification authority
Reference Verification and Validation Workflow

Requirements for system or software component

Textual Requirements

Executable Specification

Modelling

Model used for production code generation

Code Generation

Generated C/C++ code

Compilation and Linking

Object code
Reference Verification and Validation Workflow

- Predict dynamic system behavior by simulation
  - System & environment models
  - Precision with floating point
- Use of simulation results for system design
  - Fast What-If studies
  - Short iteration cycles

Textual Requirements → Executable Specification → Modelling → Code Generation → Compilation and Linking → Object code

Model used for production code generation → Generated C/C++ code
Reference Verification and Validation Workflow

- Model tuned for target processor
  - Fixed point mathematics, real-time behavior
- Configure for production use
  - Support for standards (AUTOSAR, ASAP2)
Reference Verification and Validation Workflow

- Automatically generated code for target processor
  - Optimized, efficient C/C++ code
- Fine grain control of generated code
  - Files, functions, data

Textual Requirements

Executable Specification

Modelling

Model used for production code generation

Generated C/C++ code

Code Generation

Compilation and Linking

Object code
Reference Verification and Validation Workflow

1. **Textual Requirements**
2. **Executable Specification**
3. **Modelling**
4. **Model used for production code generation**
5. **Code Generation**
6. **Generated C/C++ code**
7. **Object code**
8. **Compilation and Linking**

Object code linked with system software and flashed to ECU
Verification and Validation
Tasks and Activities
Verification and Validation Tasks and Activities

- Find missing or incomplete requirements
- Are requirements sufficiently specified
- Identify inconsistent requirements
- **Product: Simulink Verification & Validation**

**Requirements Traceability**
Text ↔ Models

- Textual Requirements
- Executable Specification
- Model used for production code generation
- Generated C/C++ code
- Object code

**Code Generation**
**Compilation and Linking**
Bi-directionally Trace Requirements

Textual Requirements

1. Identify lane markers (yellow/white lines, solid vs. broken lines)
2. Determine when car is in-between lane
3. Warn when there is Right Lane or Left Lane departure

Design Model
Verification and Validation Tasks and Activities

- Functional Testing
  - Does design meet requirements
  - Confirm correct design behavior
  - Verify no unintended behavior
  - *Product: Simulink Test, Simulink Design Verifier*

Textual Requirements → Executable Specification → Model used for production code generation → Generated C/C++ code → Object code

- Modelling
- Code Generation
- Compilation and Linking
Functional Testing

- **Functional testing process**
  - Author test-cases (derived from requirements)
  - Use formal verification to auto generate tests (more on this next)
  - Execute tests across design environments (with test iterations)
  - Monitor test verdicts (pass/fail)

- **Product: Simulink Test**
  - Test harness to isolate component under test
  - Author complex test scenarios with Test Sequence
  - Manage tests, execution, and results
Verification and Validation Tasks and Activities

- Prove design meets requirements
  - Formally verify requirements and safety
  - Test case generation for functional testing
- Prove that the design is robust
  - Check that the design does not contain errors such as overflow, divide by zero, dead logic, ...
- Product: Simulink Design Verifier
Motivation for Formal Verification (Formal Methods)

- “Program testing can be used to show the presence of bugs, but never to show their absence” (Dijkstra)

- “Given that we cannot really show there are no more errors in the program, when do we stop testing?” (Hailpern)

_Dijkstra, “Notes On Structured Programming”, 1972_
Formal Methods Technique – Model Checking

- **Given**
  - Design model
  - Requirement specification

- **Prove that**
  - Design meets the requirement specification, or
  - Does not meet the requirement and automatically generate test-case proving requirement not met
Prove That Design Meets Requirements
With Model Checking

Checks that design meets requirements
- Gear 2 *always* engages when speed > 50
- Gear 2 *never* engages when speed < 5

Expected behavior of design

Behavior that design should not exhibit
Test Case Generation for Functional Testing

With Model Checking

- Specify functional test objectives
  - Define custom objectives that signals must satisfy in test cases

- Specify functional test conditions
  - Define constraints on signal values to constrain test generator
Formal Methods Technique – Abstract Interpretation

- Consider multiplication of three integers
  
  \(-4586 \times 34985 \times 2389 = ?\)

- Quickly compute the final value by hand
  - What is the final answer?
  - What about the sign?

- The sign result
  - Could be positive, negative (or zero)
  - Per math rules, we know it is negative

- We abstracted complex details
  - Provably know precisely the sign
Prove That Design is Robust

With Abstract Interpretation

Design can suffer from overflows, divide by zero, and other robustness errors

- Proven that overflow does NOT occur
- Proven that overflow DOES occur
Verification Task

Coverage Analysis

- Coverage metric
  - Measure of how much software has been tested
- Identify testing gaps to find
  - Untested design elements
  - Dead logic and unreachable states
- Identify requirement issues
  - Missing or inconsistent functional requirements
  - Discover requirement problems early

Product: Simulink Verification & Validation
Coverage Concepts

- **Types of coverage**
  - Statement: each statement in the code executed
  - Decision: has every branch of control statements executed
  - Condition: Boolean sub-expression evaluated for both true and false
  - Modified Condition Decision Coverage (MCDC)

- **MCDC explained**
  - All entry/exit points invoked
  - Condition in decisions and conditions taken all possible outcomes
  - Each condition in a decision independently affects decision outcome

```c
if (X && Y)
    Z = 1;
else
    Z = -1;
end
```
Verification and Validation Tasks and Activities

- Test Generation for Coverage
  - Automate manual task of writing test-cases and test inputs
    - Intelligent determination of input combinations for high coverage
  - Formal methods based test generation
    - Analyze design, states, logic paths in the design model
  - *Product: Simulink Design Verifier*

![Diagram showing the process from Textual Requirements to Executable Specification, Modelling, Code Generation, Generation of C/C++ code, and finally to Object code with Compilation and Linking.](image-url)
Addressing Missing Coverage

Step 1
- Functional Tests
- Design Model
- Coverage Analysis
  - Coverage Report
    - Partial Coverage (less than 100%)

Step 2
- Test Generator (formal methods)
- Additional Tests
  - Coverage Report
    - Full Coverage (100%)
Verification and Validation Tasks and Activities

- Checks conformance to coding standards
  - MISRA (Motor Industry Software Reliability Association)

- Finds bugs and proves absence of run-time errors
  - In the integrated code

- *Products: Polyspace Bug Finder, Code Prover*
Motivation for Static Code Analysis

- The Generated Code is integrated with other Handwritten Code
- Impossible to exhaustively test the integrated code for bugs
- Certification standards require checking code for coding standards
- Critical run-time errors can cause un-intended behavior
Static Code Analysis Techniques

- Compiler warnings
  - Incompatible type detection, etc.

- Code metrics and standards
  - Comment density, cyclomatic complexity, MISRA C/C++

- Bug finding
  - Pattern matching, heuristics, data/control flow

- Code proving
  - Formal methods with abstract interpretation
  - No false negatives

Results from Polyspace Code Prover

static void pointer_arithmetic (void) {
   int array[100];
   int *p = array;
   int i;
   for (i = 0; i < 100; i++) {
      p = 0;
      p++;
   }

   if (get_bus_status() > 0) {
      if (get_oil_pressure() > 0) {
         *p = 5;
      } else {
         i++;
      }
   }
   i = get_bus_status();

   if (i >= 0) {
      (p - i)^2 - 10;
   }
}
Verification and Validation Tasks and Activities

Equivalence Testing

SIL – Software in the Loop (prevention of unintended functionality)

PIL – Processor in the Loop (back to back testing)

Textual Requirements → Executable Specification → Modelling → Model used for production code generation → Generated C/C++ code → Object code

Code Generation → Compilation and Linking

Reference: ISO 26262 www.iso.org
Software In the Loop (SIL) Testing

- Show equivalence, model to code
- Assess code execution time
- Collect code coverage

Test Vectors

Model Generator

Desktop Simulation (on PC)

Generated Code

PC Compiler

Object File

Object Code Execution (on PC)

Compare

== ?

Results

Results
Processor In the Loop (PIL) Testing

- Verify numerical equivalence
- Assess target execution time
- Collect on target code coverage

- Process
  - Model
  - Desktop Simulation (on PC)
  - Results
  - Code Generator
  - Generated Code
  - Cross Compiler
  - Object File
  - Object Code Execution (on target)
  - Results

- Compare
  - == ?
# MathWorks Solution Summary

<table>
<thead>
<tr>
<th>Requirements Traceability</th>
<th>Simulink Verification and Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing</td>
<td>Simulink Test, Simulink Design Verifier</td>
</tr>
<tr>
<td>Formal Verification</td>
<td>Simulink Design Verifier, Polyspace Code Prover</td>
</tr>
<tr>
<td>Coverage Analysis</td>
<td>Simulink Verification and Validation</td>
</tr>
<tr>
<td>Static Code Analysis</td>
<td>Polyspace Bug Finder, Polyspace Code Prover</td>
</tr>
<tr>
<td>SIL, PIL</td>
<td>Simulink Test</td>
</tr>
</tbody>
</table>
Key Takeaways

1. Find bugs early, develop high quality software
2. Replace manual verification tasks with workflow automation
3. Learn about reference workflow that conforms to safety standards

UAV Flight Control Software Development and Verification … “development effort reduced by 60%” Jungho Moon, KAL
Additional Customer References and Applications

**Airbus Helicopters Accelerates Development of DO-178B Certified Software with Model-Based Design**
Airbus Helicopters Accelerates Development of DO-178B Certified Software with Model-Based Design

**Baker Hughes Improves Precision of Oil and Gas Drilling Equipment**
Baker Hughes Improves Precision of Oil and Gas Drilling Equipment

**Continental Develops Electronically Controlled Air Suspension for Heavy-Duty Trucks**
Continental AG used MathWorks software to design a level and roll control system for heavy-duty, 40-ton trucks.

**Lear Delivers Quality Body Control Electronics Faster Using Model-Based Design**
Lear Delivers Quality Body Control Electronics Faster Using Model-Based Design