A good design workflow leads to a good design, but verification *proves* it!
LEAR CORPORATION
The 100-day design cycle with MATLAB and Simulink
Model-Based Design and a Testing and Proving Workflow

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

Verification & Validation

Modelling → Code Generation → Compilation and Linking

Design
Start with Requirements

Requirements for system or software component

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

Modelling → Code Generation → Compilation and Linking
Transform Requirements into Executable Specifications

- Simulink models for continuous or discrete time behavior
  - Signal processing filters
  - Control algorithms
- Stateflow for logic and discrete events control
  - Start-up behavior, health checking
  - Supervisory control

Requirements Traceability

Textual Requirements → Executable Specification

Model used for production code generation → Generated C/C++ code → Object code

Compilation and Linking

Code Generation

Modelling
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 in Simulink
Test Early in Simulation

- **Textual Requirements**
- **Executable Specification**
- **Model used for production code generation**

- 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

Component and system testing
Functional Testing

- Author test-cases that are derived from requirements
  - Use test harness to isolate component under test
  - Test Sequence to create complex test scenarios

- Manage tests, execution, results
  - Re-use tests for regression
  - Automate in Continuous Integration systems such as Jenkins
Formal Verification: Proving Requirements

Checks that design meets requirements
• Condition 1: Gear 2 *always* engages
• Condition 2: Gear 2 *never* engages
Formal Verification: Test Case Generation

Automatically generate test cases for:
• Functional Requirements Testing
• Model Coverage Analysis

- The **Test Objective** block defines the values of a signal that a test case must satisfy.
- The **Test Condition** block constrains the values of a signal during analysis.
Formal Verification: Proving Robustness

Detect overflows, divide by zero, and other robustness errors

- Proven that overflow does NOT occur
- Proven that overflow DOES occur
Coverage Analysis

Model Coverage

- Measure how much has been tested
  - Find untested design elements
  - Find dead logic and unreachable states
- Identify requirement issues early
  - Missing functional requirements
  - Inconsistent functional requirements

Code Coverage

Textual Requirements ➔ Executable Specification ➔ Modelling ➔ Model used for production code generation ➔ Code Generation ➔ Other code ➔ Compilation and Linking ➔ Object code

- Generated C/C++ code
Coverage Analysis: also for self-written C/C++ in S-functions

S-Function block "sldemo_sfun_counterbus"

Parent: sldemo_lct_bus/TestCounter
Uncovered Links:

Metric Coverage
Cyclomatic Complexity 3
Condition 67% (4/6) condition outcomes
Decision 75% (3/4) decision outcomes
MCDC 50% (1/2) conditions reversed the outcome

Detailed Report: sldemo_lct_bus_sldemo_sfun_counterbus_instance_1_cov.html

<table>
<thead>
<tr>
<th>File Contents</th>
<th>Complexity</th>
<th>Decision</th>
<th>Condition</th>
<th>MCDC</th>
<th>Stmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>counterbus.c</td>
<td>3</td>
<td>75%</td>
<td>67%</td>
<td>50%</td>
<td>90%</td>
</tr>
<tr>
<td>...counterbusFcn</td>
<td>3</td>
<td>75%</td>
<td>67%</td>
<td>50%</td>
<td>90%</td>
</tr>
</tbody>
</table>
Static Code Analysis

- Code metrics and standards
  - Comment density, cyclomatic complexity,…
  - MISRA and security standards compliance
  - Custom check authoring
- Bug Finding
  - Data and control flow
  - CERT C check for security vulnerabilities
- Code Proving
  - Formal Methods / Abstract Interpretation
  - No false negatives
Green implies absence of the most important classes of run-time errors: Formally Proven

**Green: reliable**
safe pointer access

**Red: faulty**
out of bounds error

**Gray: dead**
unreachable code

**Orange: unproven**
may be unsafe for some conditions

**Purple: violation**
MISRA-C/C++ or JSF++ code rules

**Range data**
tool tip
Equivalence Testing (Back to Back Testing)

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

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

Model used for production code generation

Generated C/C++ code

Object code

Code Generation

Compilation and Linking

Textual Requirements

Executable Specification

Modelling
Software In the Loop (SIL) Testing

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

Test Vectors

Model → Generated Code

Desktop Simulation (on PC) → Embedded Coder

Object Code Execution (on PC)

Compare

Results

PC Compiler

Object File

Embedded Coder

Generated Code

Show equivalence, model to code
Assess code execution time
Collect code coverage
Processor In the Loop (PIL) Testing

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

Model

Desktop Simulation (on PC)

Results

Test Vectors

Generated Code

Object File

Object Code Execution (on target)

Compare

== ?

Embedded Coder

Cross Compiler

Results
Model-Based Design Reference Workflow (IEC 61508-3)

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

Modelling → Code Generation → Compilation and Linking

Automotive (ISO 26262) → Medical (IEC 62304) → Aerospace (DO-178) → Rail (EN 50128) → Industrial (IEC 61508)
Training

- Verification and Validation of Simulink Models
- Testing Generated Code in Simulink
- Polyspace for C/C++ Code Verification
- Polyspace Bug Finder for C/C++ Code Analysis

Public

On-Site
Key Takeaway

A good design workflow leads to a good design, but verification *proves* it!