Best Practices for Verification and Validation
62% Cost Savings Using Model-Based Design

- Total Savings: $3,720,000
- Total Investments: $592,000
- ROI: 528%

* Averaged data from Aerospace customers
Model Verification and Validation

Key Motivators:
- Exchange of executable specifications with customers
- Large project – collaboration between multiple engineers
- Code generation
- Industry standards / customer request

Benefits:
- Fewer production bugs, higher design quality
- Reduction of cost of testing and validation
- Process compliant with the latest industry standards (CMMI, DO, ISO)
Verification Activities in MBD

- **Model Verification**
  - Simulation
  - Requirement Traceability
  - Modeling Standard Guidelines
  - Model Coverage Analysis
  - Formal Verification

- **Code Verification**
  - SIL (Software-In-the-Loop) and PIL (Processor)
  - Code Metrics and Coding Rule Checking
  - Formal Verification (Abstract Interpretation)
Best Practices for Model Verification and Validation

- Simulation
- Requirement Traceability
- Modeling Standard Guidelines
- Model Coverage analysis
- Formal Verification
Simulation: Functional Verification

- Executable specification
- Stimulate with test scenarios
- Compare data from multiple runs

The Lunar Module Digital Autopilot Design

How it Would be Done Today!
Best Practices for Model Verification and Validation

- Simulation
- Requirement Traceability
- Modeling Standard Guidelines
- Model Coverage analysis
- Formal Verification
Requirements Traceability

- Traceability analysis of models relative to:
  - System requirements
  - Design descriptions
  - Interface specifications
  - Change requests
  - Defect reports

- Standards and Certification:
  - CMMI / SPICE
  - DO-178B
  - IEC 61508 / ISO 26262

- In product:
  - Link editor
  - Support for DOORS, Word, Excel, PDF, URL
  - Model highlighting
  - Reporting
  - Customization API
Compare & Merge XML Files Exported from Simulink Models

- Examine what changed
- Comparison artifact reports
- Merge changes identified
Best Practices for Model Verification and Validation

- Simulation
- Requirement Traceability
- Modeling Standard Guidelines
- Model Coverage analysis
- Formal Verification
Increasing Model and Process Complexity

- Millions of blocks
- Teams of developers
- Many versions, years of development cycle
- OEM / supplier hand-off

Need common modeling standards and review process.

Solution: modeling guidelines and automated model checking
Example of MAAB Guideline

http://www.mathworks.co.kr/automotive/standards/maab.html
Modeling Standards Checking

- Automates checking of
  - Guideline conformance
  - Readability
  - Performance
  - Efficiency
  - Potential errors

- Supports
  - MAAB Guidelines
  - High Integrity Guidelines
  - Code Generation Guidelines
  - ...

Model Advisor Interface
Best Practices for Model Verification and Validation

- Simulation
- Requirement Traceability
- Modeling Standard Guidelines
- Model Coverage analysis
- Formal Verification
Model Coverage Analysis

- Dynamic - data collected during simulation
- Results in model and html report
- Types of Model Coverage
  - Cyclomatic Complexity
  - Decision Coverage
  - Condition Coverage
  - Modified Condition/Decision Coverage
  - Lookup Table Coverage
  - Saturate on Integer Overflow Coverage
  - Signal Range Coverage
  - Signal Size Coverage
  - Simulink Design Verifier Coverage
Best Practices for Model Verification and Validation

- Simulation
- Requirement Traceability
- Modeling Standard Guidelines
- Model Coverage analysis
- Formal Verification
Model Coverage Testing

Import Tests

Manual Test Input

Test with Formal Methods

Simulink Design Verifier

Measure Coverage in Simulation

Simulink Verification and Validation
Identifying Design Errors Early

- Static Run-Time Error Detection
  - Automatic identification of hard-to-find design inconsistencies in the model without running simulation
- Supported detecting types of errors
  - Dead logic
  - Integer overflow
  - Division by zero
  - Range violation
  - Assertion violation
  - Out of bound array access
Best Practices for Code Validation

- SIL(Software-In-the_Loop) and PIL(Processor-)
- Code Metrics including HIS Metrics
- Coding Rules
  - MISRA-C 2004
  - MISRA AC AGC 1.0
  - MISRA C++
  - JSF++
  - Custom Coding Rule (Naming Convention)
- Formal Verification
Software-in-the-Loop (SIL) Testing

- Execution
  - Host/Host
  - Nonreal-time
Processor-in-the-Loop Testing

- **Execution**
  - Host/Target
  - Nonreal-time

**Diagram Description**
- **Test Suite**
- **HW Drivers**
- **Plant Model**
- **Verification**

**Production Processor**
Best Practices for Code Validation

- SIL (Software-In-the_Loop) and PIL (Processor-)
- Code Metrics including HIS Metrics
- Coding Rules
  - MISRA-C 2004
  - MISRA AC AGC 1.0
  - MISRA C++
  - JSF++
  - Custom Coding Rule (Naming Convention)
- Formal Verification
Best Practices for Code Validation

- SIL (Software-In-the-Loop) and PIL (Processor-)
- Code Metrics including HIS Metrics
- Coding Rules
  - MISRA-C 2004
  - MISRA AC AGC 1.0
  - MISRA C++
  - JSF++
  - Custom Coding Rule (Naming Convention)

- Formal Verification
Formal Methods based Static Code Analysis

- You Can Prove That
  - \( c = a + b; \)
    - will never overflow
  - \( j = \text{arr}[i]; \)
    - \( i \) will always be within array bounds
  - \( *\text{ptr} = 12; \)
    - will never be an illegal dereference
  - \( w = x / (y + z); \)
    - \( y \) never equal to \(-z\) or visa versa (divide by zero)

- And many more …
**Polyspace**
*Formal Methods based Static Code Analysis*

- Exhaustively verify code
  - Detect and prove absence of runtime errors
  - Precisely determines and propagates variable ranges

- Languages supported
  - C, C++, and Ada

- Verify SW robustness
  - Analyze for full range operating conditions
  - Specified ranges of parameters and inputs
Example: Proving Absence of Run-time Errors

```c
int new_position(int sensor_pos1, int sensor_pos2)
{
    int actuator_position;
    int x, y, tmp_pos, magnitude;

    actuator_position = 2; /* default */
    tmp_pos = 0; /* values */
    magnitude = sensor_pos1 / 100;
    y = magnitude + 5;
    x = actuator_position;

    while (actuator_position <= 10)
    {
        actuator_position++;
        tmp_pos = sensor_pos2 / 100;
        y = 3;
    }

    if ((3*(magnitude + 100) >= 43)
    {
        magnitude++;
        x = actuator_position;
        actuator_position = x / (x - y);
    }

    return actuator_position + tmp_pos; /* new value */
}
```
Example: Proving Absence of Run-time Errors

```c
16     x := 0;
17     }
18     if ((3*magnitude + 100) > 43)
19     {
20         magnitude++;
21         x = actuator_position;
22         actuator_position = x / (x - y);
23     }
24     return actuator_position + tmp_pos; /* new value */
25     }
```
Example: Proving Absence of Run-time Errors

```
16     x = 3;
17     }
18     if ((3*magnitude + 100) > 43)
19       {
20       magnitude++;
21       x = actuator_position;
22       actuator_position = x / (x - y);
23       }
24     return actuator_position + tmp_pos;
25     }
```

variable ‘x’ (int 32): 10
Example: Proving Absence of Run-time Errors

```c
x = y;
if ((3 * magnitude + 100) > 43)
{
    magnitude++;
    x = actuator_position;
    actuator_position = x / (x - y);
}
return actuator_position;
```

variable ‘y’ (int 32): 11 .. 21474865
Example: Proving Absence of Run-time Errors

```c
int new_position(int sensor_pos1, int sensor_pos2)
{
    int actuator_position;  // parameter sensor_pos1 (int 32): full-range [-2^{31} .. 2^{31} - 1]
    int x, y, tmp_pos, magnitude;

    actuator_position = 2; /* default */
    tmp_pos = 0; /* values */
    magnitude = sensor_pos1 / 100;
    y = magnitude + 5;
    x = actuator_position;

    while (actuator_position <= 10)
    {
        actuator_position++;
        tmp_pos = sensor_pos2 / 100;
        y *= 3;
    }

    if (3*magnitude + 100) > 43)
    {
        magnitude++;
        x = actuator_position;
        actuator_position = x / (x - y);
    }

    return actuator_position + tmp_pos; /* new value */
}
```
Example: Proving Absence of Run-time Errors

```c
    int new_position(int sensor_pos1, int sensor_pos2)
    {
        int actuator_position;
        int x, y, tmp_pos, magnitude;

        actuator_position = 2; /* default */
        tmp_pos = 0; /* values */
        magnitude = sensor_pos1 / 100;
        y = magnitude + 5;
        x = actuator_position;

        while (actuator_position <= 10)
        {
            actuator_position++;
            tmp_pos += sensor_pos2 / 100;
            y *= 3;
        }

        if ((3*magnitude + 100) >= 43)
        {
            magnitude++;
            x = actuator_position;
            actuator_position = x / (x - y);
        }

        return actuator_position + tmp_pos; /* new value */
    }
```
Example: Finding a Run-Time Error

```c
int array[100];
int i, *p = array;

for(i = 0; i < 100; i++)
{
    *p = 0;
    p++;
}

if(get_bus_status() > 0)
{
    if(get_oil_pressure() > 0)
    {
        p = 5; /* Out of bounds */
    }
    else
    {
        /* dereference of variable 'p' (pointer to int 32, size: 32 bits):
        pointer is not null
        points to 4 bytes at offset 400 in buffer of 400 bytes, so is outside bounds
        may point to variable or field of variable in: {Pointer_Arithmetic:array} */
    }
}
```
Why verify generated code?

- You have hand code interacting with generated code in a way that may induce ripple effects/bugs (e.g., global data)

- Your customer or cert standard requires source code verification techniques supported by Polyspace (e.g., MISRA AC AGC)

- You need to have confidence to know that your production code is independently verified (e.g., all green with Polyspace)
Why use Polyspace on generated code

- Generated code will interface with other code (libraries, handwritten code, etc.)
- Simulink models may contain handwritten code, e.g. in the form of S-Functions
- Software quality processes, certification standards require the use of code standards such as MISRA
- Polyspace provides traceability from the code to Simulink models
- Polyspace can help identify range related failures in the generated code
- Polyspace is a good complement to Simulink Design Verifier as it can work on the generated code irrespective of the modeling paradigms
Benefits of using Polyspace on generated code

- Ensure robustness of production code, by verifying interface between handwritten and generated code

- Satisfy code rule compliance by checking generated code for code rule standards such as MISRA-AC-AGC

- Speed debug and reporting by tracing code verification results from code to Simulink models

- Prove that run-time errors are not caused by invalid or missing min/max ranges

- Gray code can identify blocks that are not needed
Verification Activities in MBD

- Model Verification
  - Simulation
  - Requirement Traceability
  - Modeling Standard Guidelines
  - Model Coverage Analysis
  - Formal Verification

- Code Verification
  - SIL and PIL
  - Code Metrics and Coding Rule Checking
  - Formal Verification (Abstract Interpretation)
Best Practices – Customer Examples

- **Simulation**
  - *Bell Helicopter* designs civilian tiltrotor

- **Requirement Traceability**
  - *Continental* tests heavy-duty trucks

- **Model Coverage analysis**
  - *TRW* generates tests for parking brake

- **Formal verification methods**
  - *Nissan* increases software reliability