Leveraging Formal Methods for Verifying Models and Embedded Code

Prashant Mathapati
Application Engineering Group
The Cost of Failure...

News reports:

Recall

Due to ECU software bug
The Cost of Failure…

USS Yorktown: Divide-by-zero Error

0 Knots

Top speed
The Cost of Failure…

Therac-25: Race Condition, Overflow

6 Casualties
due to radiation overdose
What do all these systems have in common?

- Complex software developed to rigorous standards
- Extensively reviewed, analyzed and tested
- Yet still succumbed to costly failure
When is it safe to ship?

40% of all bugs are runtime errors  
- IBM Study

33% of all medical devices sold in U.S. between ’99 and ’05 recalled for software failures  
- U. of Patras (Greece) Study
Reasons for Failure

- **Cause of failure:**
  - Design or Code defects resulting in run-time errors

- **What are run-time errors?**
  - Also known as “latent faults”
  - Rarely manifest directly and frequently

- **Effects include**
  - Software crashes
  - Unexpected software behavior

- Exhaustive testing is out of reach, allowing errors to remain in the design and code.

- 30-40% of errors found in software are run-time errors.
Early errors found late are costly.

Relative Cost to Fix Defects per Phase Found

Engineers didn’t understand the problem to solve!

Engineers got the problem, but the solution doesn’t work!

The solution works, but the implementation has errors!

Source: Return on Investment for Independent Verification & Validation, NASA, 2004
Let’s look at the problem we are trying to solve…

- What is unique about control logic as oppose to dataflow algorithm?
Difficulties with Complex Logic

- Multiple entry/exit transitions
- Simultaneous transitions
- Execution order
Difficulties with Complex Logic

- Complex conditional statements
How do you test your models today?

- Simulation Based Testing

  Develop test inputs
  Apply the test inputs to model
  Analyze the results with Expected output
How do you ensure a test case exercised all simulation pathways?

Model Coverage…
Simulation Based Testing

- The quality of your verification activity is based on the input vectors you created. Error or design defect can only be detected if the proper stimulus exists.
Gaps in Simulation Based Testing

- The method itself is inefficient in being exhaustive or complete.
  - A set of functional test case that meets MC/DC coverage objective is only a “minimum” set of test cases.
    - Test cases can’t cover every possible combination of different scenarios.
    - Inputs are defined based on what we already knew.

- What about the unknown?
  - Priority, synchronization, timeout…etc

- We are performing **Acceptance** test.
Formal Verification

- Formal verification is...
  - “…mathematically rigorous procedures to search through the possible execution paths of your model for design errors, test cases and counterexamples.” (Simulink Design Verifier homepage)
  - “…act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics.” (Wikipedia)
Example application of formal verification:

- Error detection in model and code

**Model:**
- **Simulink Design Verifier**

**Code:**
- **Polyspace Code Verifier**
Simulink Design Verifier
use cases –

- finding design errors

  - Determine unreachable or dead transitions/states without manually generating any test vectors.

  - Identify missing use cases or missing requirements.

  - Prove correctness of your design
    - Use of condition and objective
    - Use of requirements modeling
Determine Unreachable or Dead Transitions/States Without Manually Generating Test Vectors

- Ensure algorithmic logic is structurally correct *during* design phase.
  - Early detection

**Transition expression can never be True**
Identify Missing Use Cases or Missing Requirements

- Search for hard to find and/or missing functional test cases.
  - User can use test cases generated by Simulink Design Verifier and reverse engineer missing requirements.
    - Understand missing functional test cases associated with missing requirements or functionalities.

Why do we have missing coverage?

Simulink Design Verifier
Generate Test Vector for Missing Coverage
Automatically Generate Test Cases for Equivalence Testing

- Quickly create a set of test cases that can be used for meeting the equivalence testing criteria in high-integrity application standards
  - ISO26262
    - Software-in-the-Loop
    - Processor-in-the-Loop
Formal Methods based Static Code Analysis

- Find run-time errors
  - In legacy or hand code
  - In the model caused by mixed code integration
  - In the design - when missed by the workflow

- Prove the absence of run-time errors
  - Prove code is free of run-time errors
  - Check MISRA or MISRA AC AGC compliance
  - Prepare for independent code verification (DO-178B, IEC 61508, …)

- Check workflow integrity, including mixed environments
  - Browse code-model level to verify the implementation
  - Catch defects missed by the workflow
  - Find implementation errors
Augmenting Static Code Analysis with Formal Methods

- Static code analysis – scan source code to automate verification
  - Range from *unsound* methods to *sound* techniques

**low**

Compiler warnings
  - Incompatible type detection, etc.

**high**

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

Formal methods
  - Sound proof based techniques, applied to source code
  - Can show SW is robust for wide operating conditions
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
    OR
  - Specified ranges of parameters and inputs

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

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
Polyspace in Model Based Design

Polyspace results on generated code are traced back to the model...

Available for Embedded Coder, dSPACE TargetLink, and Telelogic Rhapsody
Proving Absence of Runtime Errors

You Can Prove That:

\[
c = a + b; \quad \text{will never overflow}
\]

\[
j = arr[i]; \quad i \text{ will always be within array bounds}
\]

\[
*ptr = 12; \quad \text{will never be an illegal dereference}
\]

\[
w = x / (y + z); \quad y \text{ never equal to } -z \text{ or visa versa (divide by zero)}
\]

And many more …
Example: Proving Absence of Run-time Errors

- There are none.
- Proven

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

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

    while (actuator_position < 20)
    {
        actuator_position++;
        tap_pos += sensor_pos2 / 100;
        y += 63;
    }

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

    return actuator_position + tap_pos; /* new value */
}
```
Why prove the absence of run-time errors?
Implications of verification, static analysis & unit testing

Number of operations \( x \) input values

Verification

Static analysis

Testing

0% proven reliable

T0

+3 months

+6 months

Improvement is possible and measurable

One-time improvement, but nothing measurably proven

Required for functional testing. Not suitable to prove code correctness

Improvement is possible and measurable
Formal Methods Supplement to DO-178C and DO-278A

- Allows the use of Formal Methods Analysis for verification of:
  - Software requirements
  - Software design
  - Source code
  - Executable object code
- Adds an objective that the formal analysis method must be shown to be sound
- What does this mean
  - Polyspace is a formal methods tool for analysis of source code
  - As a Criteria 2 tool, partial credit may be taken for analysis of executable object code and this is included in the DO Qualification Kit
  - DO Qualification Kit already includes a Theoretical Foundation document to justify the soundness of Abstract Interpretation
ISO 26262 Certification of MathWorks Products

- TÜV SÜD certified:
  - Real-Time Workshop Embedded Coder
  - Simulink Design Verifier
  - Simulink Verification and Validation
  - Polyspace products for C/C++
- For use in development processes which need to comply with IEC 61508, ISO 26262, or EN 50128

Note: The products listed above were not developed using certified processes
TRW Automotive Develops and Tests Electric Parking Brake Using Simulink and Simulink Design Verifier

Challenge
Design tests for an electric parking brake control system

Solution
Use Simulink Design Verifier to automatically generate tests that maximize model coverage and enable systematic design verification

Results
- Test development time reduced from days to hours
- 100% model coverage achieved
- Formal testing begun two months into the project

“Everyone knows that errors are much less expensive to fix when you find them early. With Simulink Design Verifier, we build on the advantages of Model-Based Design by performing formal testing in the first phases of development.”

Christoph Hellwig
TRW
Nissan Increases Software Reliability with Polyspace Products for C/C++

**Challenge**
Identify hard-to-find run-time errors to improve software quality

**Solution**
Use MathWorks tools to exhaustively analyse Nissan and supplier code

**Results**
- Suppliers' bugs detected and measured
- Software reliability improved
- Polyspace products for C/C++ adopted by Nissan suppliers

"Polyspace products for C/C++ can ensure a level of software reliability that is unmatched by any tools in the industry."

Mitsuhiko Kikuchi, Nissan
Key Takeaways

- **Simulink Design Verifier:**
  - Design Error Detection
    - Dead Logic, Divide by Zero etc…
  - Test Case Generation
    - Decision, Condition and MC/DC Coverage ….  
  - Property Proving
    - To prove or disprove specified property

- **Polyspace:**
  - Prove code correctness:
    - Provide sustainable quality measures
    - Identify which part of the code is reliable
  - Detect run-time errors:
    - Clean functional tests of run-time errors
    - Detect software errors quickly and early
Thank You