Intro to Proving Absence of Errors in C/C++ Code

Develop high quality embedded software

Kristian Lindqvist
Senior Pilot Engineer
MathWorks
The Cost of Failure…

Ariane 5:
Overflow Error

$7,500,000,000

Rocket & payload lost
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 failed
Examples of software bugs and errors

- Run-time errors
- Concurrency issues
- Programming errors
- Dead or unreachable code
- Static and dynamic memory errors
Testing is not enough!

- Good design and testing
  - Helps eliminate functional errors

- Implementation correctness - Robustness
  - Undetected run-time errors will cause catastrophic failure
Can you find a bug?

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

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

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

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

    return actuator_position*magnitude + tmp; /* new value */
}
```
Can you find a bug?

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

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

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

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

    return actuator_position*magnitude + tmp; /* new value */
}
```
Consider the operation: \( x / (x - y) \)

Potential run-time errors
- Variables \( x \) and \( y \) may not be initialized
- An overflow on subtraction
- If \( x == y \), then a divide by zero will occur

How to prove that run-time errors \textit{do} or \textit{do not} exist?
Exhaustive testing

- If both inputs are signed int32
  - Full range inputs: $-2^{31} - 1 \ldots +2^{31} - 1$
  - All combinations of two inputs: $4.61 \times 10^{18}$ test-cases

- Test time on a Windows host machine
  - 2.2GHz T7500 Intel processor
  - 4 million test-cases took 9.284 seconds
  - Exhaustive testing time: 339,413 years

Exhaustive Testing is Impossible
Polyspace demonstration

where_are_errors
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 ++;
        }
    } else {
        i = get_bus_status();
        if (i >= 0) {
            *(p - i) = 10;
        }
    }
}
How is Polyspace code verification unique?

Statically verifies all possible executions of your code (considering all possible inputs, paths, variable values)

- Proves when code will not fail under any runtime conditions
- Finds runtime errors, boundary conditions and unreachable code without exhaustive testing
- Gives insight into runtime behavior and data ranges
- Mathematically sound – has no false negatives
Detailed Polyspace demonstration

demo_c
Why verify code in Model-Based Design?

- May contain S-Functions (handwritten code)
- Generated code may interface with legacy or driver code
- Interface may cause downstream run-time errors
- Inadequate model verification to eliminate constructional errors
- Certification may require verification at code level
Benefits of running Polyspace from Simulink

- Find bugs in S-Functions in isolation
- Check compliance for MISRA (or MISRA-AC-AGC)
- Annotate models to justify code rule violations
- Trace code verification results back to Simulink models
- Qualify integrated code (generated code and handwritten code)
- Independent verification of generated code
- Easily produce reports and artifacts for certification
Traceability from code to models

Polyspace Bug Finder and Polyspace Code Prover verification results, including MISRA analysis can be traced from code to model
Polyspace product family for C/C++

- **Polyspace Code Prover**
  - Proves code to be safe and dependable
  - Deep verification of software components
  - Perform QA signoff for production ready code

- **Polyspace Bug Finder**
  - Quickly find bugs in embedded software
  - Check code compliance for MISRA and JSF
  - Intended for every day use by software engineers

Ada language also supported for proving code
How does Polyspace help you?

- Finds bugs and security vulnerabilities
- Checks coding rule conformance (MISRA/JSF/Custom/CWE*)
- Provides metrics (Cyclomatic complexity etc)
- Proves the existence and absence of errors
- Indicates when you’ve reached the desired quality level
- Certification help for DO-178 C, ISO 26262, EN 50128, IEC 62304, IEC 61508
Thank You!