Comprehensive Static Analysis of Embedded Software (C/C++ and Ada) Using Polyspace Products

Prashant Mathapati
Senior Application Engineer
MathWorks India

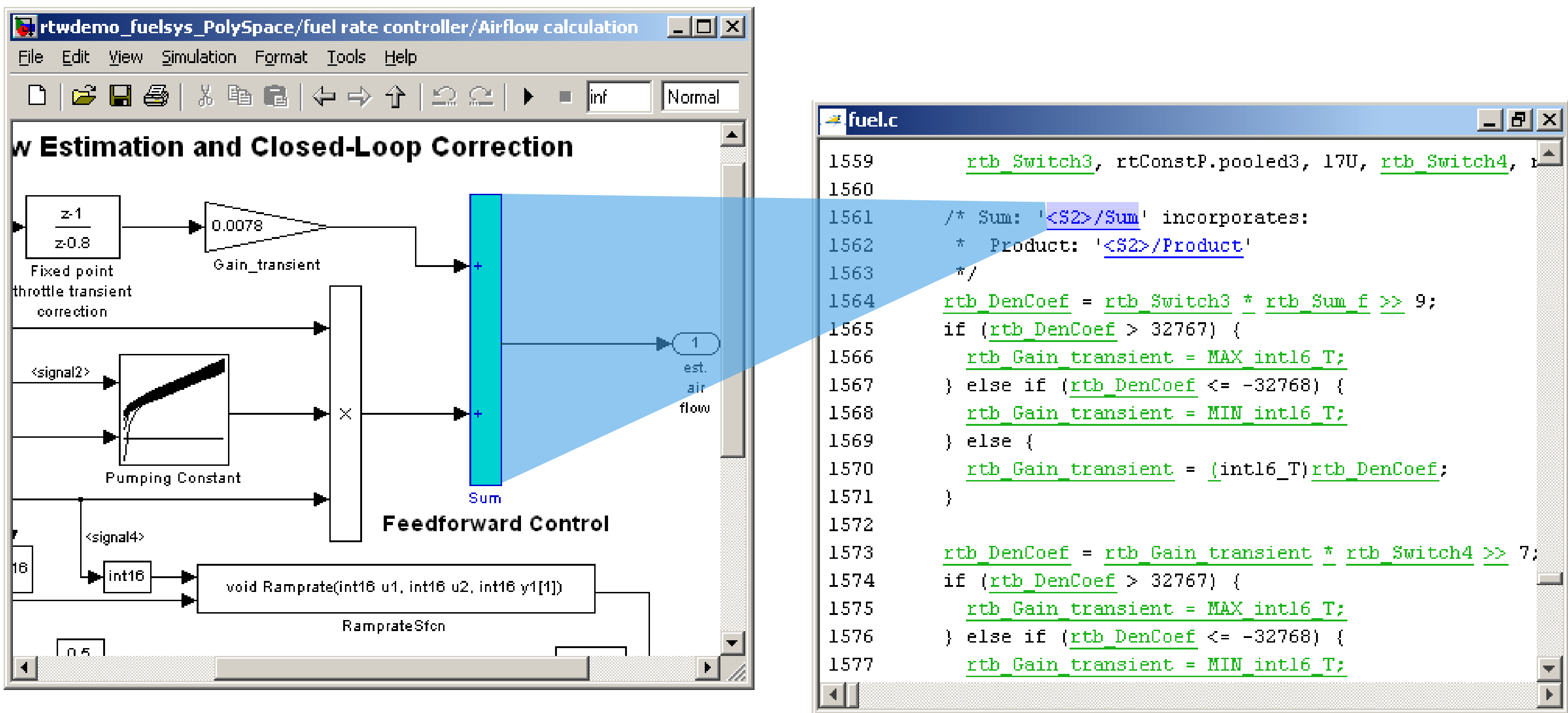
What's the value of verifying mixed generated and hand-written Code?

- 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 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

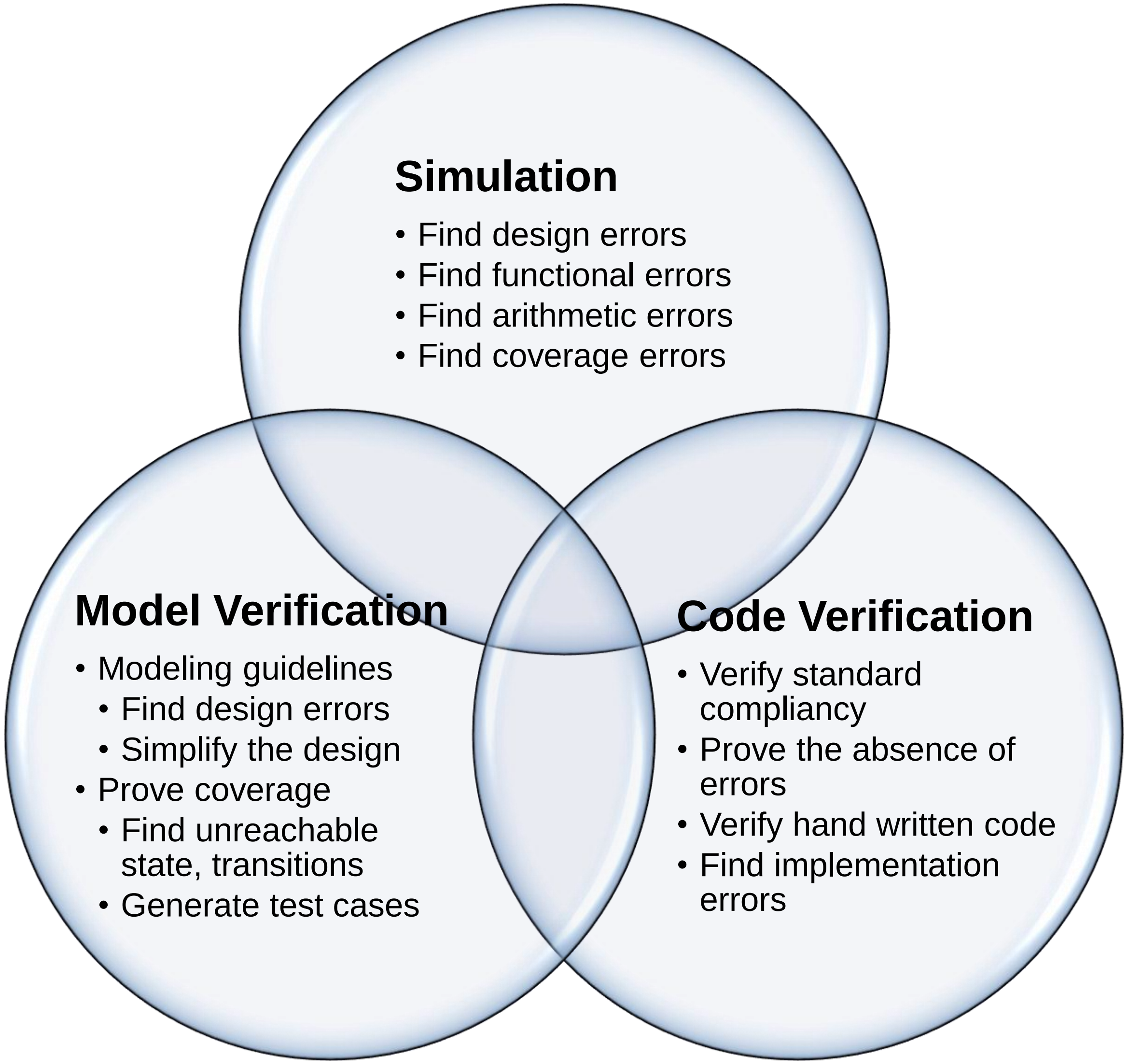
Polyspace results on generated code are traced back to the model



Examples of Run-Time Errors Found in Legacy Code, Mixed Workflow, and/or the Design

	Model constructions	Code constructions
Arithmetic errors	▪ Scaling ▪ Unknown calibrations ▪ Untested data ranges	▪ Overflows, division by zero, bit-shifts, square root of negative numbers
Memory corruption	▪ Array manipulation in Stateflow ▪ Handwritten lookup table functions	▪ Out-of-bounds array indexes ▪ Pointer arithmetic
Data truncation	▪ Unexpected data flow	▪ Overflows, wrap around
Coding errors	▪ Unreachable states, transitions	▪ Noninitialized data ▪ Dead code

Example of Workflow

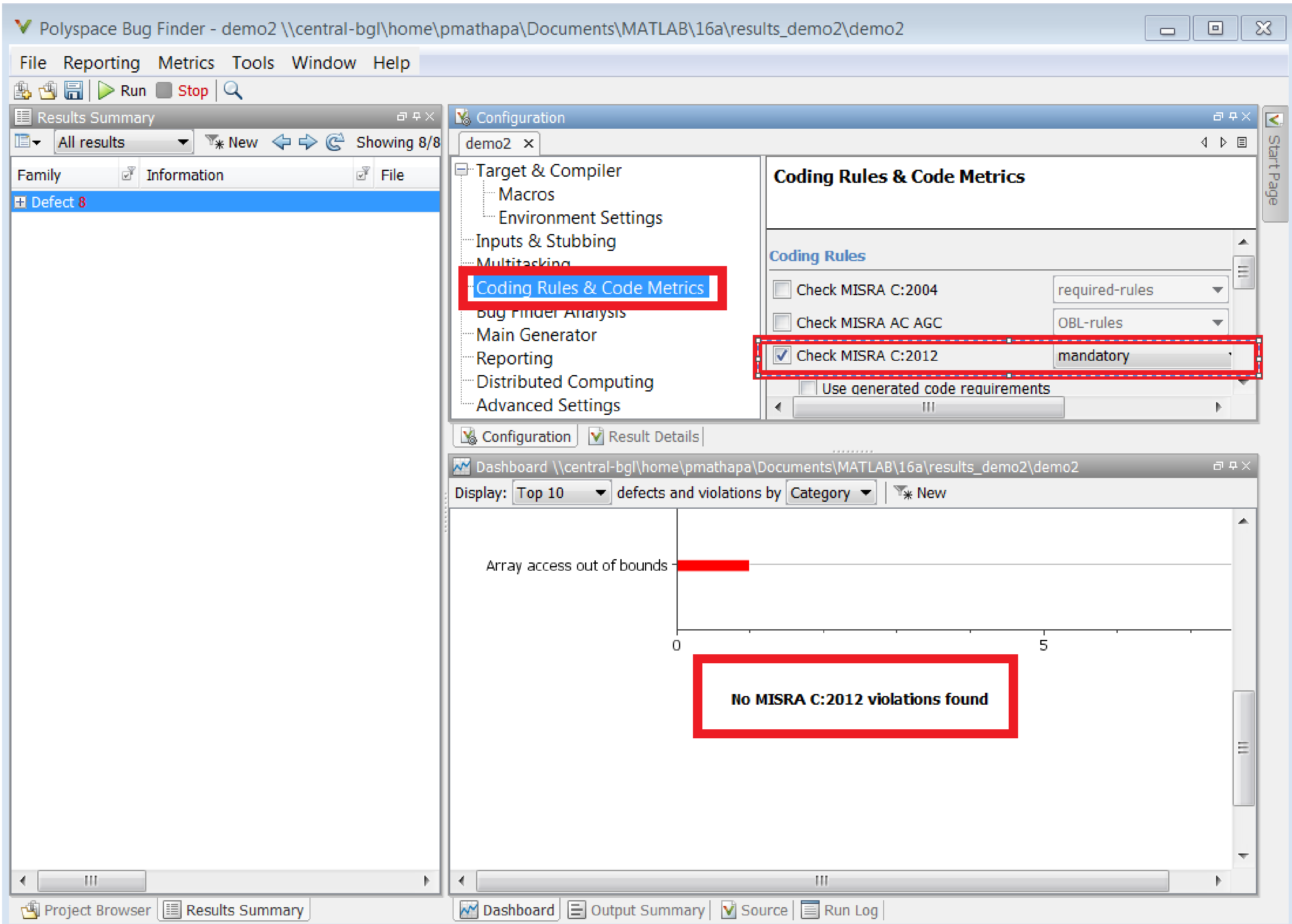


- Every tool chain has redundancy
- The best win is to do early Verification

Demo

Zero – MISRA C 2012 Mandatory Violations

Auto Code by Embedded Coder

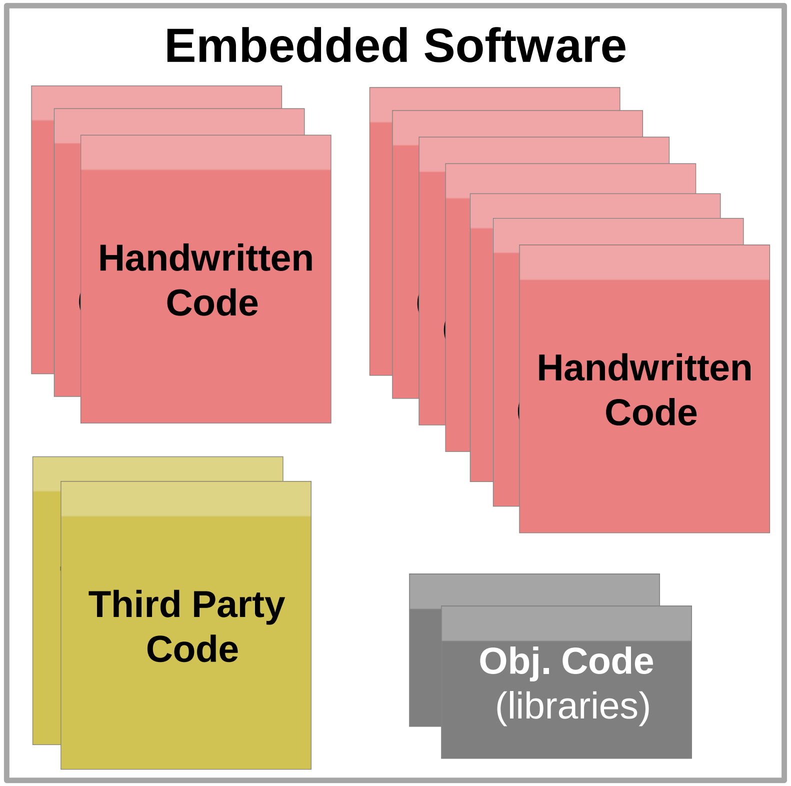


Practical Use of Polyspace

Three Real World Scenarios

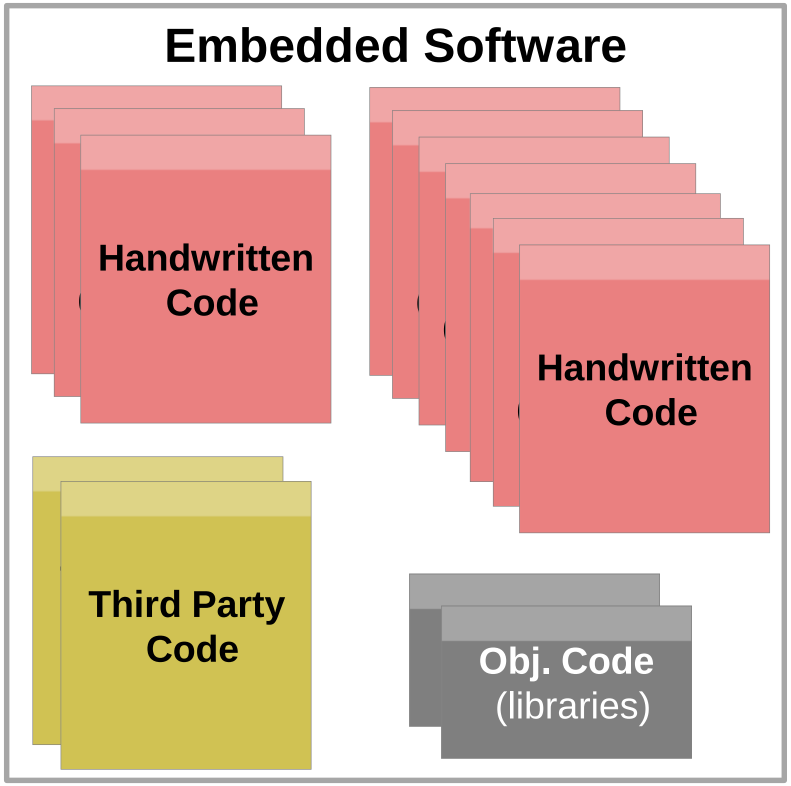
- Scenario #1
  - All handwritten code
- Scenario #2
  - Handwritten code inside generated code (Embedded Coder)
- Scenario #3
  - Generated code inside handwritten code

Scenario #1: All Handwritten Code



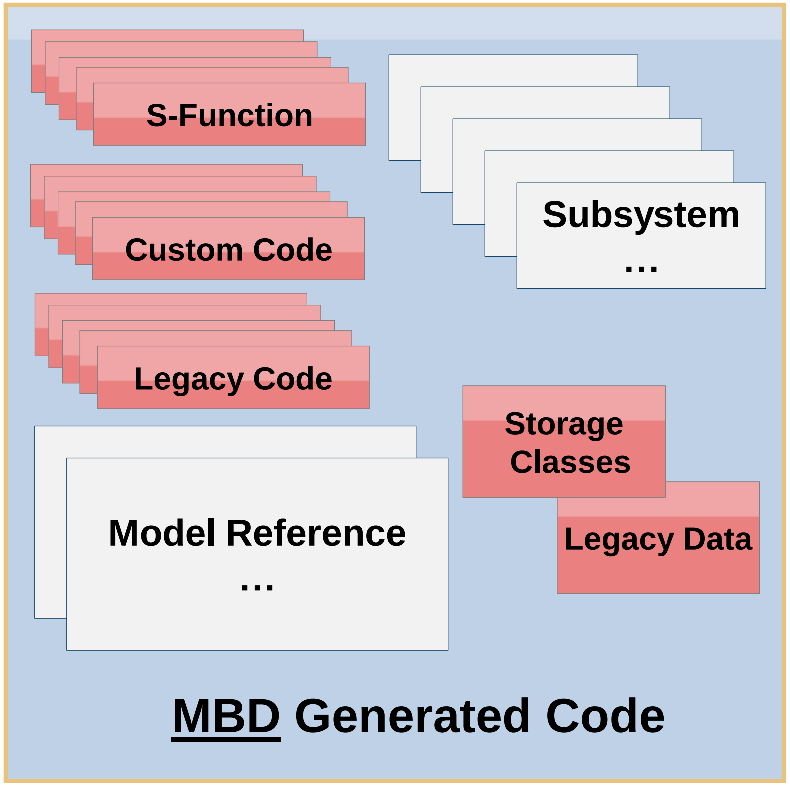
- Embedded software components
  - Complete system 100s of KLOC
  - Comprise of many functions and tasks
  - All integrated with handwritten code
- Problems encountered
  - Runtime bugs in the handwritten and third party code (*inadequate unit or component verification*)
  - How to verify at the interface level
  - Assuring that the entire system is robust

Using Polyspace for Scenario #1



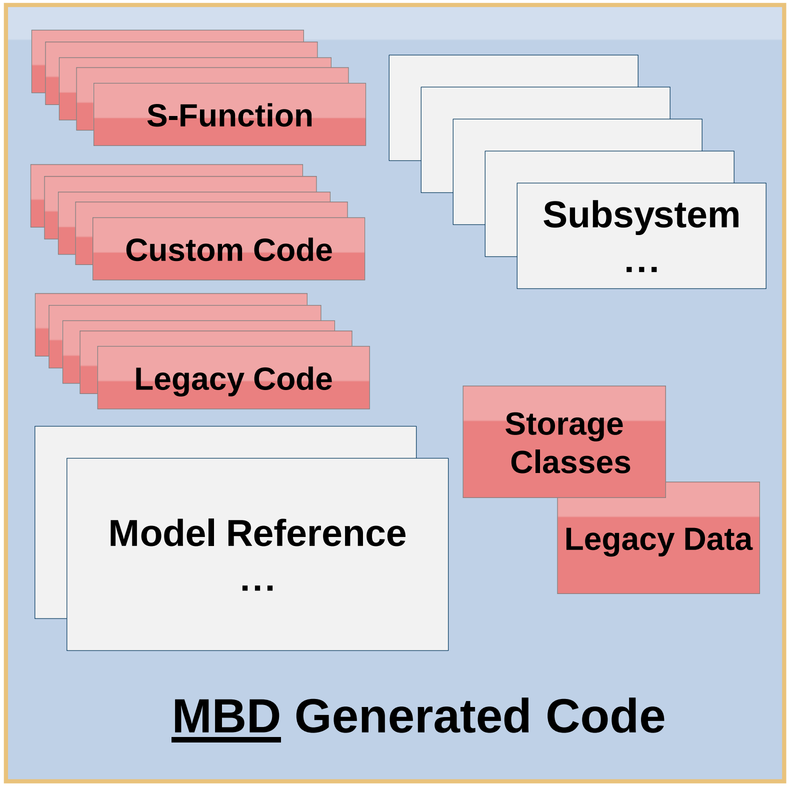
- Modular or component verification
  - Run Polyspace on each function
  - *Robustness*: full-range or worst-case conditions, or
  - *Contextual*: apply range limits on interfaces
- Integration level verification
  - Run Polyspace on integrated code
  - Practical limits depending on code complexity and LOC

Scenario #2: Handwritten Code Inside MBD



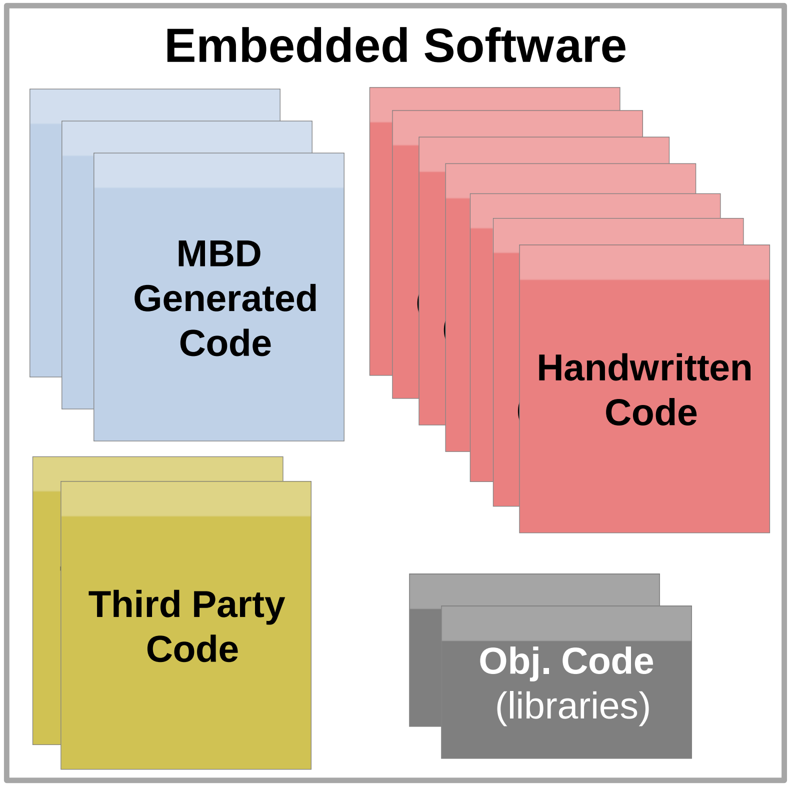
- Generated code for model component
  - Consists of subsystems and model references
- Often includes handwritten code
  - In the form of S-Functions and legacy code
  - Individually, small in size (100s LOC)
  - May be automatically repeated many time within the MBD generated code
- Problems with integration
  - Handwritten code fails (*robustness issue*), or causes generated code to fail
  - Generated code may cause handwritten code to fail (*Interface related failures*)
  - Handwritten code treated as blackbox by Simulink

Using Polyspace for Scenario #2



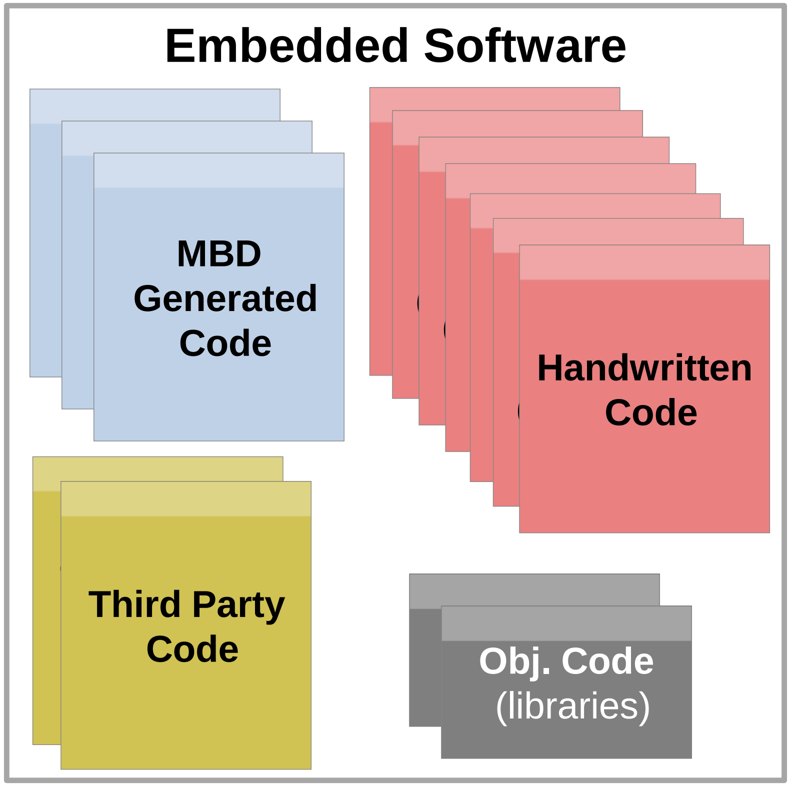
- Modular verification of S-Functions or legacy code
  - *Robustness*: full-range or worst-case conditions, or
  - *Contextual*: apply range limits on interfaces
- Verification of mixed handwritten and generated code
  - Can perform robustness and contextual verification on interfaces of the generated code, including global data
  - *Polyspace* product traces code level defects back to the *Simulink* model
  - Handwritten code treated as whitebox by Polyspace

Scenario #3: Generated code inside handwritten code



- Code integration outside MBD
  - Generated code integrated together with handwritten code
  - All components integrated into embedded software with handwritten code
- Problems with integration
  - Runtime bugs in the handwritten and third party code (*inadequate unit or component verification*)
  - Verifying generated code especially at interface level
  - How to project relevant problems back to the model?
  - Assuring that the entire system is robust

Using Polyspace for Scenario #3



- Modular verification of handwritten or generated code
  - Run Polyspace on each function or file
  - *Robustness*: full-range worst-case conditions, or
  - *Contextual*: apply range limits on interfaces
- Integration level verification
  - Run Polyspace on integrated code
  - *Polyspace* products traces code level defects back to the *Simulink* model
  - Practical limits depending on code complexity and LOC

Verify mixed generated and hand-code

Prove the absence of run-time errors in source code

```
static void Pointer_Arithmetic (void)
{
  int array[100];
  int i, *p = array;

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

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

  i = get_bus_status();
  if (i >= 0)  { *(p-i) = 10; }

  if ((0 < i) && (i <= 100)) {
      p = p - i;
      *p = 5;
    }
}
```

Green: reliable

Red: faulty

Grey: dead

Orange: unproven

Proven

Quality improvement

- Prove the absence of errors
- No compilation, no execution, no test cases
- Early verification of C/C++ or Ada

Thank You