Leveraging Formal Methods – Based Software Verification to Prove Code Quality & Achieve MISRA compliance

Prashant Mathapati
Senior Application Engineer
MATLAB EXPO

The problem

Complex systems can fail … with drastic consequences

- Ariane-5, expendable launch system
  - Overflow error
  - Resulted in destruction of the launch vehicle

- USS Yorktown, Ticonderoga class ship
  - Divide by zero error
  - Caused ship's propulsion system to fail

- Therac-25, radiation therapy machine
  - Race condition and overflow error
  - Casualties due to overdosing of patients

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

Analyzing and proving embedded software

- Good design and testing
  - Helps eliminate functional errors

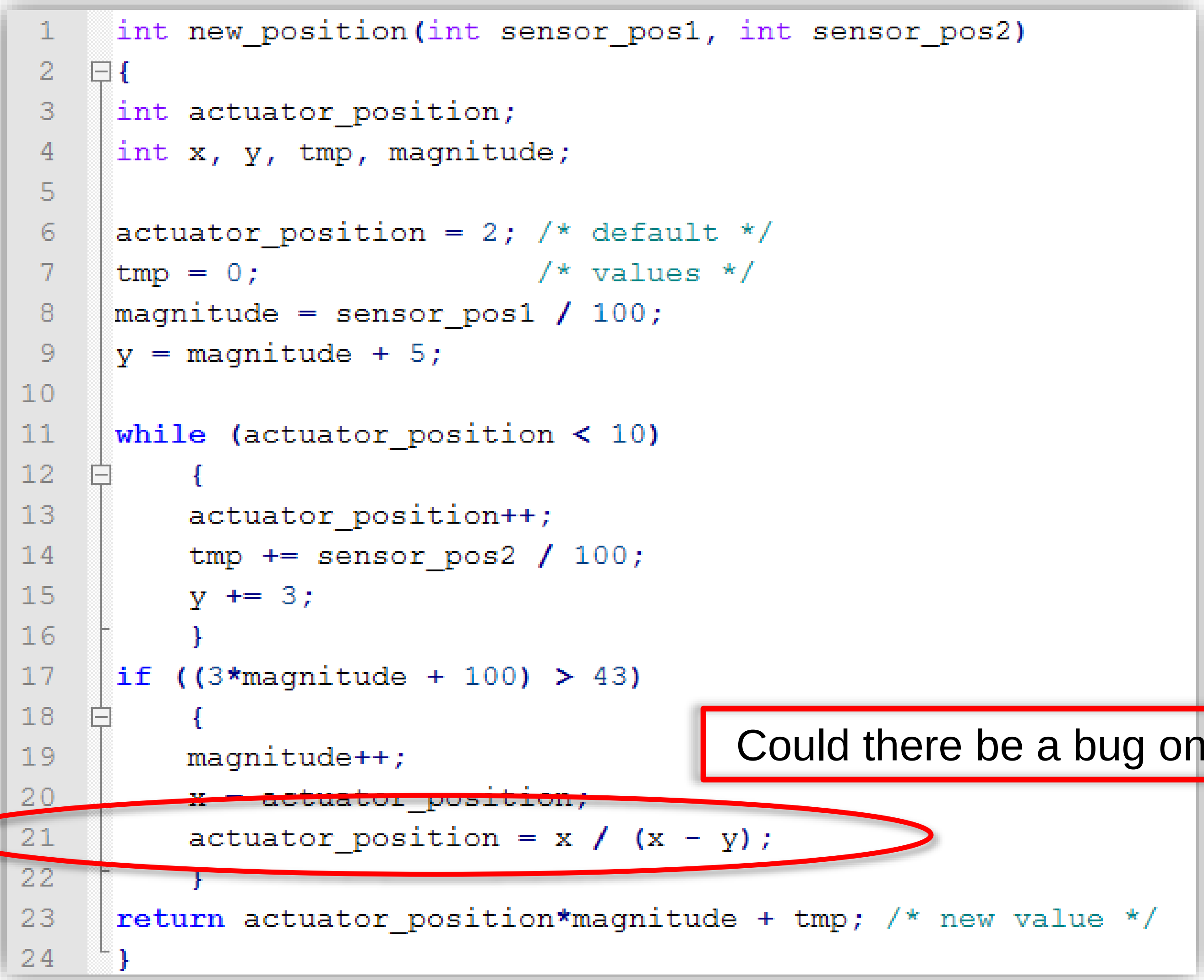
- But, robustness concerns may still exist
  - Undetected run-time errors will cause catastrophic failure

- *Polyspace: static code analysis using formal methods*
  - *Address robustness concerns*
  - *Ensures safe and dependable software*

How does *Polyspace* help you?

- Finds bugs
- Checks coding rule conformance (MISRA/JSF/Custom)
- 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, …

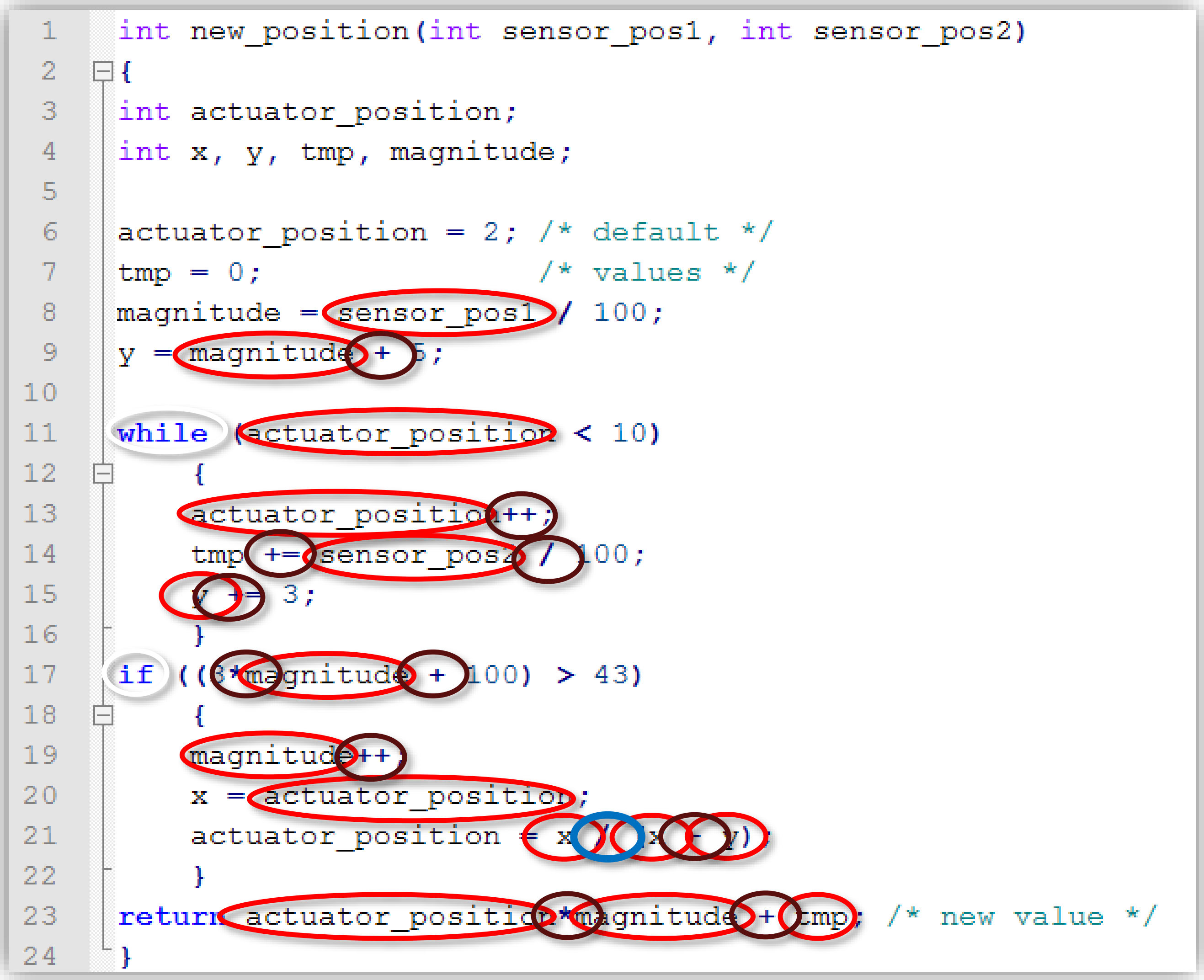
Can you find a bug?



```
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 */
}
```

Could there be a bug on this line?

Other potential run-time errors to consider



```
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 */
}
```

Variables may not be initialized

Overflow potential

Division by zero potential

Dead code potential

Exhaustive testing

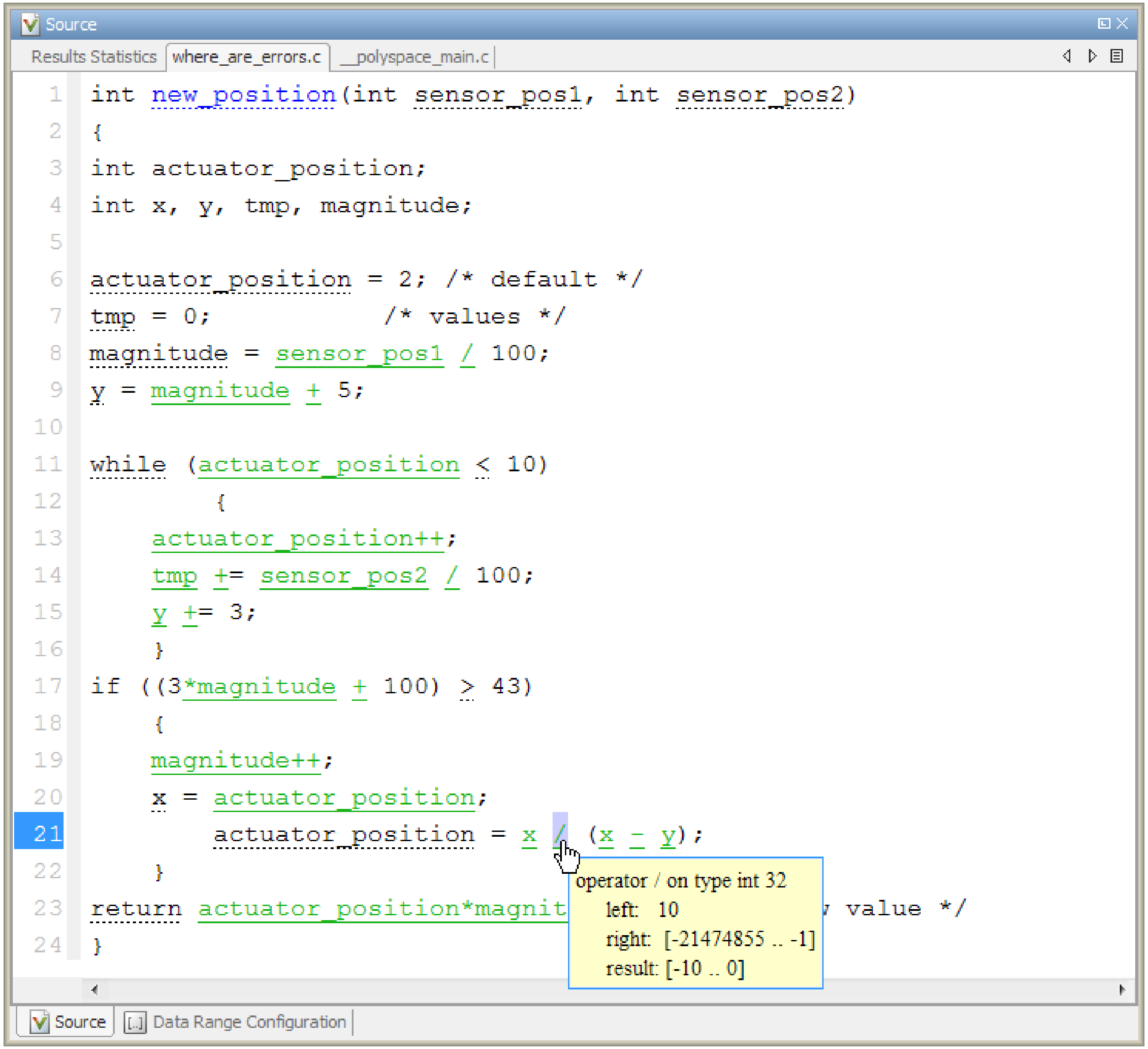
- If both inputs are signed int32
  - Full range inputs: `-2`^{31}`-1 .. +2`^{31}`-1`
  - All combinations of two inputs: 4.61X10^{18} test-cases

- 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

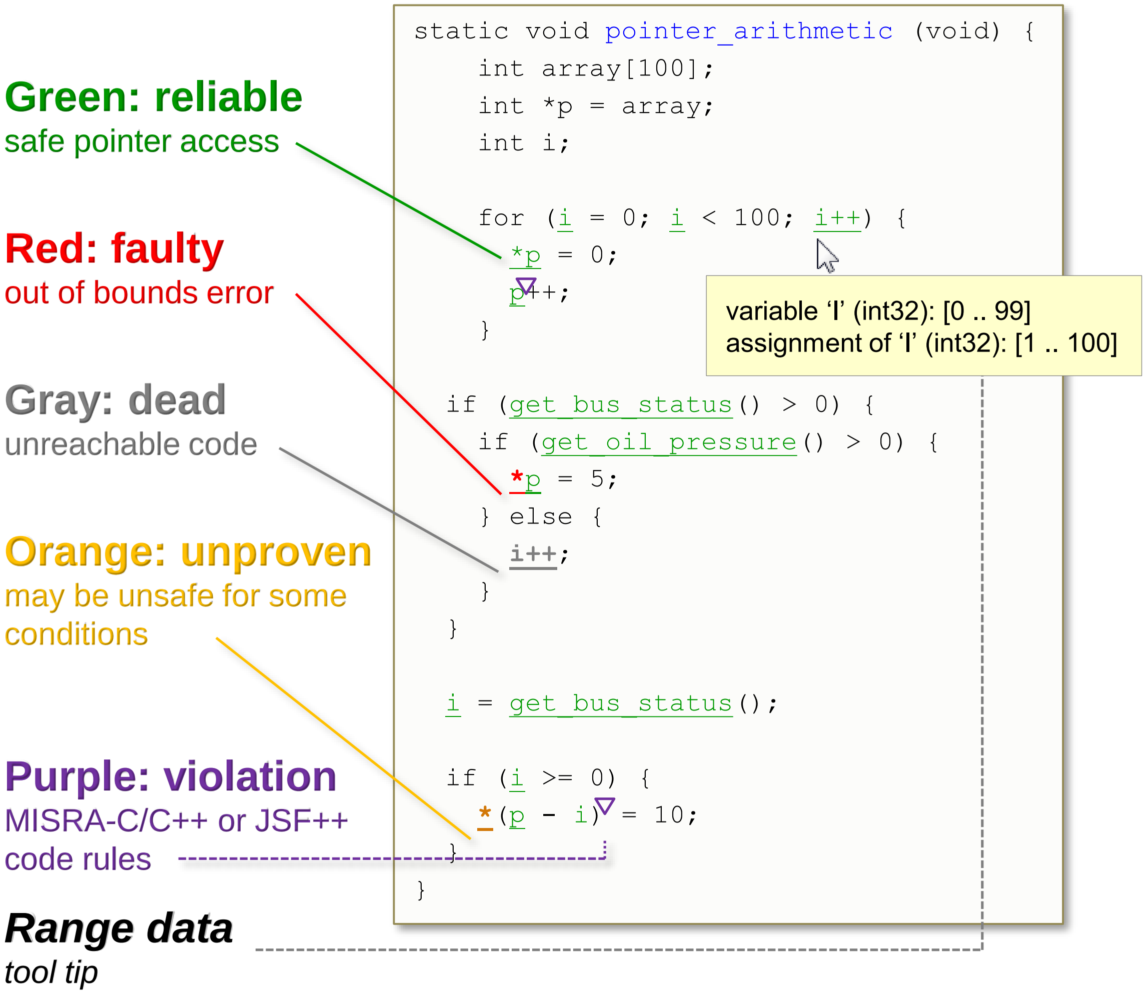
Results from Polyspace



```
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*magnit
}
```

```
operator / on type int 32
  left:  10
  right:  [-21474855 .. -1]
  result: [-10 .. 0]
```

Results from Polyspace



Green: reliable
safe pointer access

Red: faulty
out of bounds error

Gray: dead
unreachable code

Orange: unproven
may be unsafe for some conditions

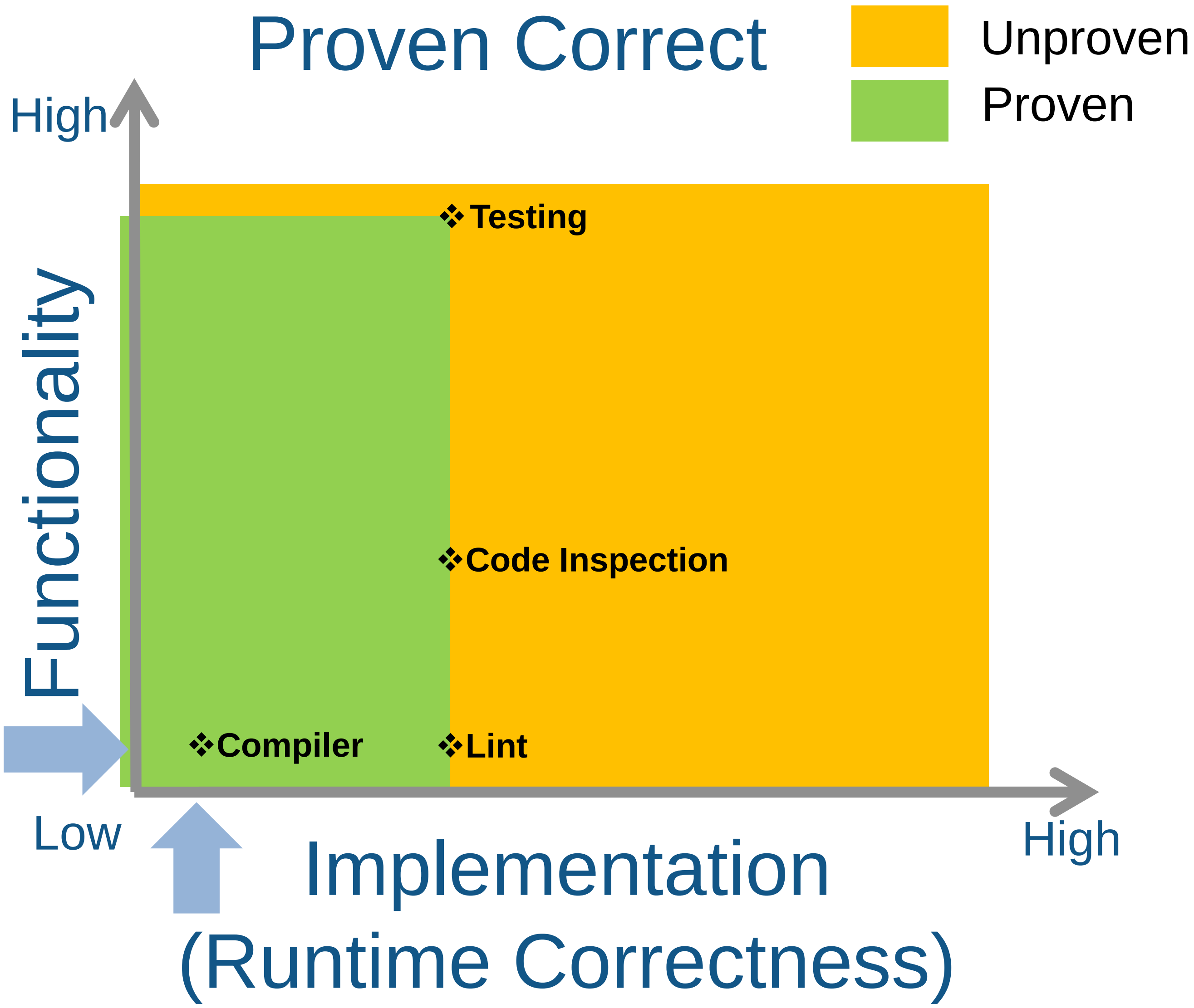
Purple: violation
MISRA-C/C++ or JSF++ code rules

Range data
tool tip

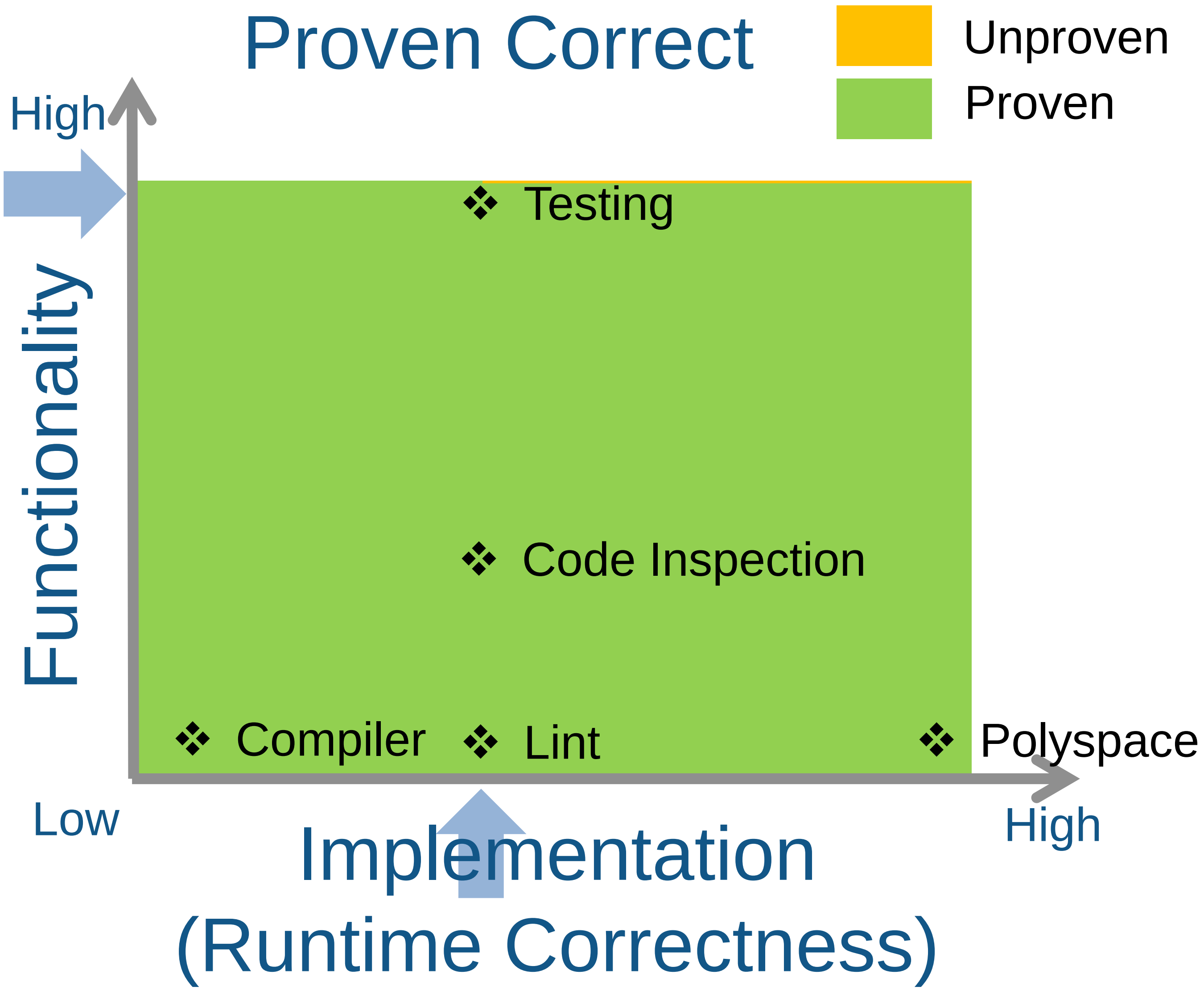
```
static void pointer_arithmetic (void) {
    int array[100];
    int *p = array;
    int i;

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

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

  i = get_bus_status();

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

variable ‘I’ (int32): [0 .. 99]
assignment of ‘I’ (int32): [1 .. 100]

Validation and Verification



Validation and Verification



Proven Correct

🟨 Unproven

🟩 Proven

High

Functionality

- ❖ Testing
- ❖ Code Inspection
- ❖ Compiler
- ❖ Lint
- ❖ Polyspace

Low

High

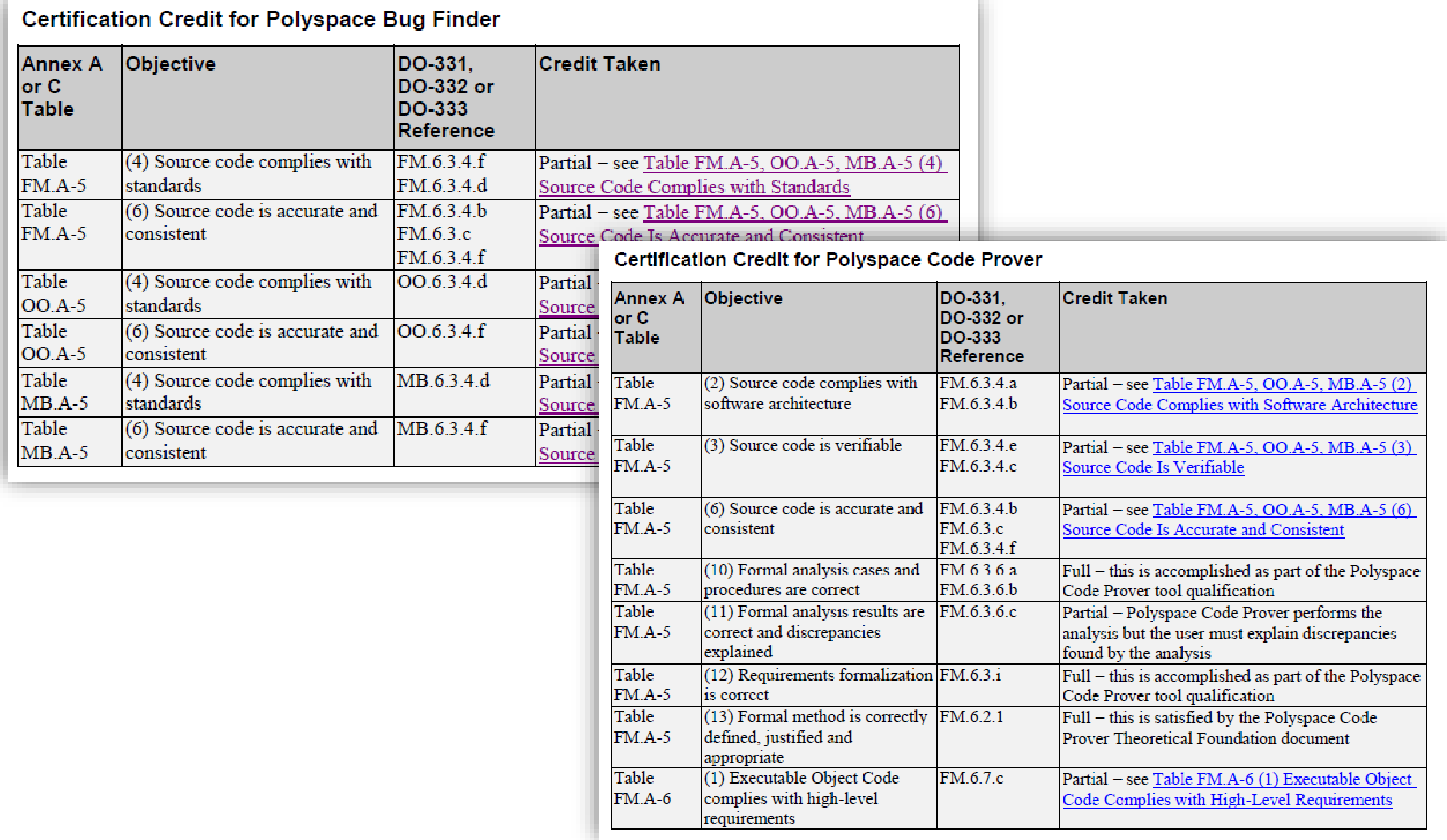
Implementation
(Runtime Correctness)

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

DO-178 certification credit



Certification Credit for Polyspace Bug Finder

Annex A or C Table	Objective	DO-331, DO-332 or DO-333 Reference	Credit Taken
Table FM.A-5	(4) Source code complies with standards	FM.6.3.4.f FM.6.3.4.d	Partial – see Table FM.A-5, OO.A-5, MB.A-5 (4) Source Code Complies with Standards
Table FM.A-5	(6) Source code is accurate and consistent	FM.6.3.4.b FM.6.3.c FM.6.3.4.f	Partial – see Table FM.A-5, OO.A-5, MB.A-5 (6) Source Code Is Accurate and Consistent
Table OO.A-5	(4) Source code complies with standards	OO.6.3.4.d	Partial Source
Table OO.A-5	(6) Source code is accurate and consistent	OO.6.3.4.f	Partial Source
Table MB.A-5	(4) Source code complies with standards	MB.6.3.4.d	Partial Source
Table MB.A-5	(6) Source code is accurate and consistent	MB.6.3.4.f	Partial Source

Certification Credit for Polyspace Code Prover

Annex A or C Table	Objective	DO-331, DO-332 or DO-333 Reference	Credit Taken
Table FM.A-5	(2) Source code complies with software architecture	FM.6.3.4.a FM.6.3.4.b	Partial – see Table FM.A-5, OO.A-5, MB.A-5 (2) Source Code Complies with Software Architecture
Table FM.A-5	(3) Source code is verifiable	FM.6.3.4.e FM.6.3.4.c	Partial – see Table FM.A-5, OO.A-5, MB.A-5 (3) Source Code Is Verifiable
Table FM.A-5	(6) Source code is accurate and consistent	FM.6.3.4.b FM.6.3.c FM.6.3.4.f	Partial – see Table FM.A-5, OO.A-5, MB.A-5 (6) Source Code Is Accurate and Consistent
Table FM.A-5	(10) Formal analysis cases and procedures are correct	FM.6.3.6.a FM.6.3.6.b	Full – this is accomplished as part of the Polyspace Code Prover tool qualification
Table FM.A-5	(11) Formal analysis results are correct and discrepancies explained	FM.6.3.6.c	Partial – Polyspace Code Prover performs the analysis but the user must explain discrepancies found by the analysis
Table FM.A-5	(12) Requirements formalization is correct	FM.6.3.i	Full – this is accomplished as part of the Polyspace Code Prover tool qualification
Table FM.A-5	(13) Formal method is correctly defined, justified and appropriate	FM.6.2.1	Full – this is satisfied by the Polyspace Code Prover Theoretical Foundation document
Table FM.A-6	(1) Executable Object Code complies with high-level requirements	FM.6.7.c	Partial – see Table FM.A-6 (1) Executable Object Code Complies with High-Level Requirements

Applicability to ISO 26262

ISO 26262-6 Software unit design and implementation

Methods		ASIL				Applicable Tools / Processes
		A	B	C	D	
1a	Walk-through	++	+	o	o	Polyspace Bug Finder, Polyspace Code Prover
1b	Inspection	+	++	++	++	
1c	Semi-formal verification	+	+	++	++	
1d	Formal verification	o	o	+	+	Polyspace Code Prover
1e	Control flow analysis	+	+	++	++	Polyspace Bug Finder, Polyspace Code Prover
1f	Data flow analysis	+	+	++	++	
1g	Static code analysis	+	++	++	++	
1h	Semantic code analysis*	+	+	+	+	Polyspace Code Prover

Table 9 – Methods for the verification of the software unit design and implementation

* … is used for mathematical analysis of source code by use of an abstract representation of possible values for the variables. For this it is not necessary to translate and execute the source code.
(ISO 26262-6, table 9, Method 1h)

Why verify code in MBD?

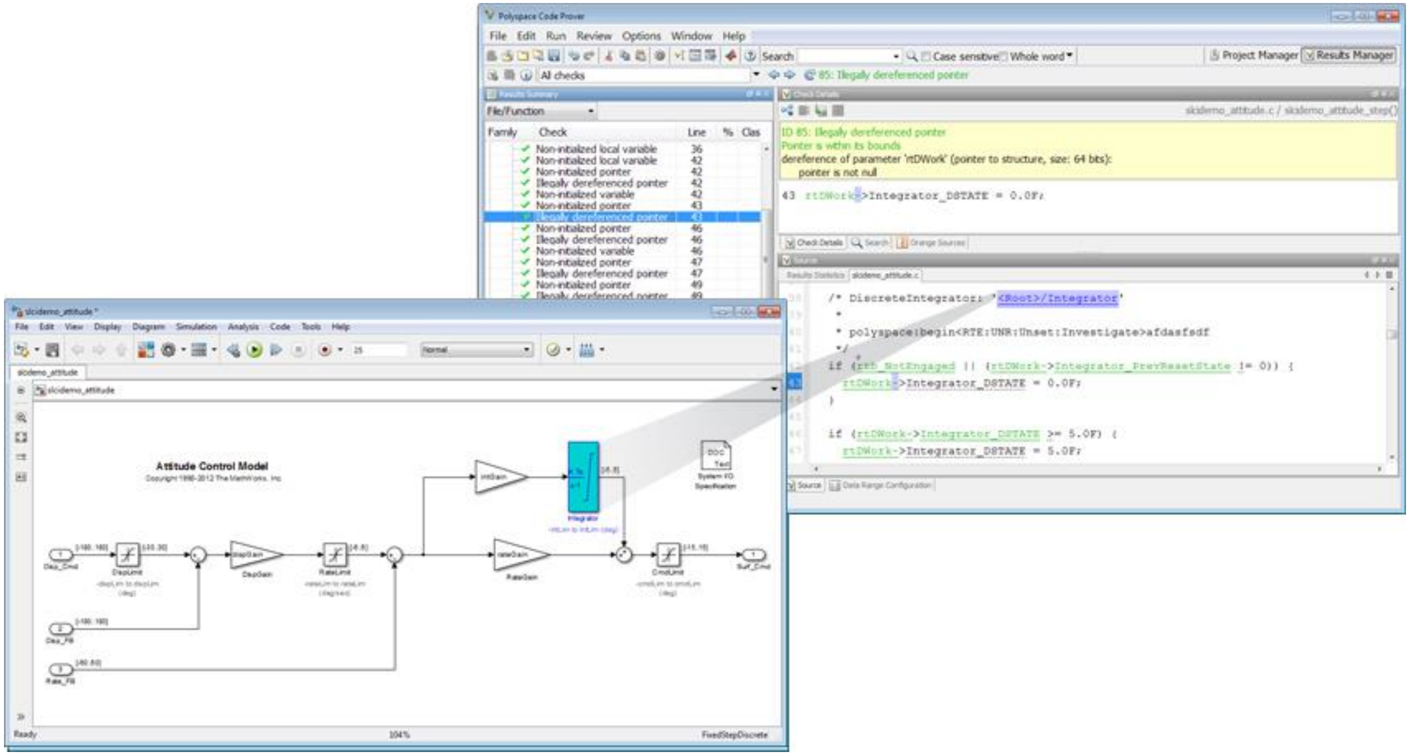
- May contain S-Functions (handwritten code)
- Generated code may interface with legacy or driver code
- Interface may cause downstream run-time errors
- Inadequate model verification to eliminate constructional errors
- Certification may require verification at code level

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 support available for TargetLink and UML Rhapsody

Traceability from code to models



Family	Check	Line	%	Clas
	✓ Non-initialized local variable	36		
	✓ Non-initialized local variable	42		
	✓ Non-initialized pointer	42		
	✓ Illegally dereferenced pointer	42		
	✓ Non-initialized variable	42		
	✓ Non-initialized pointer	43		
	✓ Illegally dereferenced pointer	43		
	✓ Non-initialized pointer	46		
	✓ Illegally dereferenced pointer	46		
	✓ Non-initialized variable	46		
	✓ Non-initialized pointer	47		
	✓ Illegally dereferenced pointer	47		
	✓ Non-initialized pointer	49		
	✓ Illegally dereferenced pointer	49		

ID 85: Illegally dereferenced pointer
Pointer is within its bounds
dereference of parameter 'rtDWork' (pointer to structure, size: 64 bits):
pointer is not null

```
43 rtDWork->Integrator_DSTATE = 0.0F;
```

```
/* DiscreteIntegrator: '<Root>/Integrator'
 *
 * polyspace:begin<RTE:UNR:Unset:Investigate>afdasfsdf
 */
if (rtb_NotEngaged || (rtDWork->Integrator_PrevResetState != 0)) {
  rtDWork->Integrator_DSTATE = 0.0F;
}

if (rtDWork->Integrator_DSTATE >= 5.0F) {
  rtDWork->Integrator_DSTATE = 5.0F;
```

Polyspace Bug Finder and *Polyspace Code Prover* verification results, including MISRA analysis can be traced from code to model

EADS Ensures Launch Vehicle Dependability with Polyspace Products for Ada

Ariane 5 launcher taking off.

Challenge

To automate the identification of run-time errors in mission-critical software for launch vehicles

Solution

Use Polyspace products to analyze 100,000 lines of Ada code developed in-house and by third-party contractors

Results

- Development time reduced
- Subcontractor code verified
- Exhaustive tests streamlined

> **"The Polyspace solution is unique - it detects run-time errors without execution and has the advantage of being exhaustive."**
>
> **EADS Engineer**

[Link to user story](#)

Nissan Motor Company Increases Software Reliability with Polyspace Products for C/C++


Nissan Fairlady Z.

Challenge

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

Solution

Use MathWorks tools to exhaustively analyze 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 Motor Company**

Link to user story

ROI Analysis

- **Earlier discovery of hard-to-find run-time errors**
  - *66%* more error detection in earlier development phases.
- **More bugs found before release**
  - *31%* more bugs found compared to manual reviews and testing.
- **Shorter quality assurance cycles**
  - Reduce development time by as much as *39%* by finding run-time errors faster and earlier.
- **More thorough validation of code**
  - Continuously improve software quality resulting in *fewer* run-time errors found during later development stages where they are most expensive to find and fix.
- **Reduced development costs**
  - *42%* reduction in total cost of development resulting in savings of approximately *$1,000* per day.

Polyspace Impact in Software Development

Finding runtime errors that might have been missed

→ Improves quality and safety

Finding runtime errors earlier, when quicker/cheaper to fix

→ Saves time, saves money

Knowing how data will behave, and which code is risky

→ Improves code

→ Improves code reviews

Proving reliability and robustness without exhaustive testing

→ Shortens verification cycle

→ Focuses testing where it’s more effective

→ Lets you know when you’re done