
Christian Guß
Application Engineer
The MathWorks GmbH
Agenda today

- Engineering challenges today
- Quality question
- Uniqueness of Polyspace
- How to comply with Safety Standards?
- Software Quality Objectives
High-Integrity Applications are complex

Software-based systems that are designed and maintained such that they have a high probability of carrying out their intended function

The Cost of Failure…

Ariane 5: Overflow Error

$7,500,000,000

Rocket & payload lost
The Cost of Failure...

Recall

Due to ECU software bug
The Cost of Failure…

Therac-25: Race Condition, Overflow

6 Casualties due to radiation overdose
DEPARTMENT OF TRANSPORTATION
Federal Aviation Administration
14 CFR Part 39
RIN 2120–AA64
Airworthiness Directives; The Boeing Company Airplanes
AGENCY: Federal Aviation Administration (FAA), DOT.
ACTION: Final rule; request for comments.
SUMMARY: We are adopting a new airworthiness directive (AD) for all The Boeing Company Model 787 airplanes. This AD requires a repetitive maintenance task for electrical power deactivation on Model 787 airplanes. This AD was prompted by the determination that a Model 787 airplane that has been powered continuously for 248 days can lose all alternating current (AC) electrical power due to the generator control units (GCUs) simultaneously going into failsafe mode.

This condition is caused by a software counter internal to the GCUs that will overflow after 248 days of continuous power.

We are issuing this AD to prevent loss of all AC electrical power, which could result in loss of control of the airplane.
DATES: This AD is effective May 1, 2015. The Director of the Federal Register approved the incorporation by reference of certain publications listed in this AD as of May 1, 2015.
We must receive comments on this AD by June 15, 2015.

https://www.federalregister.gov/articles/2015/05/01/2015-10066/airworthiness-directives-the-boeing-company-airplanes
How to avoid this? Engineering challenges today...

- Check MISRA
- Comply with ISO, IEC safety requirements
- Optimize Design and Architecture
- Deliver Bug free software
- Measure Code Quality
- Reduce time and efforts for software testing
Typical Example...about the quality question

Final release... It's done... Here is your software!

Great! Is it tested?

Engineer

OEM
Typical Example…about the quality question

Yes! Everything is tested! No errors!

Please prove it to me!

Engineer

OEM
Typical Example…about the quality question

Please prove it to me!
Do You Know Which Quality Is Your Code Today?

Your source code

Testing

0% proven correct

0.000000...1% executions verified

0.000000...2% executions verified

0.000000...3% executions verified

Many executions for functional testing... do not prove code correctness
Problem: Tests aren’t exhaustive

“Program testing can be used to show the presence of bugs, but never to show their absence” (Dijkstra [1])

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

```c
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;
    }
}
```

**Colors:**
- 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
Difference of semantic analysis & testing

Your source code

T0

T0 + 3 months

T0 + 6 months

Polyspace

Testing

0% proven correct

0.000000...1% executions verified

0.000000...2% executions verified

0.000000...3% executions verified

More and more is proven correct

Many executions for functional testing...

... do not prove code correctness
Typical Example…about the quality question

Please prove it to me!
Examples of software safety requirements

- Initiation
- Architecture
- Design/Implementation
- Unit Testing
- Integration Testing

**System validation/acceptance Tests**

**User Acceptance Testing**
- Complete Integration & Test

**System Tests**
- Run system tests on integrated controller

**Component Tests**
- Run component tests on target

**Code Verification and Validation**
- Subsystem Integration & Test

**IMPLEMETATION**
- C, C++
- VHDL, Verilog
- Structured Text
- MCU, DSP, FPGA, ASIC, PLC

**REQUIREMENTS**
- Architectural Design
  - Environment Models
  - Physical Components
  - Algorithms

**Subsysytem Design**
- Test vectors and expected outputs

**Component Design**
- Test vectors and expected responses

**System-Level Specification**
Examples of software safety requirements

Table 1 – Topics to be covered by modelling and coding guidelines

<table>
<thead>
<tr>
<th>Topics</th>
<th>ASIL A</th>
<th>ASIL B</th>
<th>ASIL C</th>
<th>ASIL D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a    Enforcement of low complexity</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1b    Use of language subsets</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1c    Enforcement of strong typing</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1d    Use of defensive implementation techniques</td>
<td>o</td>
<td>+</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1e    Use of established design principles</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>1f    Use of unambiguous graphical representation</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1g    Use of style guides</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1h    Use of naming conventions</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
</tbody>
</table>

Detect and correct areas with high complexity

Polyspace Metrics
Examples of software safety requirements

Table 1 – Topics to be covered by modelling and coding guidelines

Enforcement of coding guidelines

<table>
<thead>
<tr>
<th>Verification</th>
<th>Coding Rules</th>
<th>Software Quality Objectives</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Confirmed</td>
<td>Justified</td>
</tr>
<tr>
<td></td>
<td>Defects</td>
<td>%</td>
</tr>
<tr>
<td>concurrency.c</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>datatflow.c</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>dynamicmemory.c</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>numeric.c</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>other.c</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>programming.c</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>staticmemory.c</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

Polyspace Metrics
Justify MISRA violations with Polyspace formal method

```c
float risk_of_floatdivisionbyzero(int p)
{
    float i;
    float j = 1.0;
    float tmp;

    tmp = j / p;
    if (tmp < 0.0) {
        i = -1.0;
    } else if (tmp > 1.19209290e-7F) {
        i = 1024.0 / tmp;
    } else {
        i = 0.0;
    }

    return i;
}
```

MISRA Violation

Proven save (no Overflow)

A violation

Polyspace Code Prover
Justify MISRA violations with Polyspace formal method
Examples of software safety requirements

Initiation | Architecture | Design/Implementation | Unit Testing | Integration Testing

**System validation/acceptance Tests**

- **User Acceptance Testing**
- **Complete Integration & Test**
- **System-Level Integration & Test**

**Architectural Design**
- Environment Models
- Physical Components
- Algorithms

**Subsystem Design**
- Component Design
  - Test vectors and expected responses

**Component Tests**
- Component tests in simulation
- Run component tests on target

**Code Verification and Validation**
- Subsystem Integration & Test

**IMPLEMENTATION**
- C, C++
- VHDL, Verilog
- Structured Text
- MCU, DSP, FPGA, ASIC, PLC

**System Tests**
- System tests in simulation
- Run system tests on integrated controller

**Requirements**
- System-Level Specification
- Subsystem Design
Examples of software safety requirements

- Initiation
- Architecture
- Design/Implementation
- Unit Testing
- Integration Testing

Minimize size in balance with other design considerations

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restricted size of software components</td>
<td>++</td>
</tr>
</tbody>
</table>
Examples of software safety requirements

| Initiation | Architecture | Design/Implementation | Unit Testing | Integration Testing |

**Requirements**
- System-Level Specification
- Architectural Design
  - Environment Models
  - Physical Components
  - Algorithms

**System Tests**
- Run system tests in simulation
- System tests in simulation
- Run component tests on integrated controller

**Component Tests**
- Test vectors and expected responses
- Test vectors and expected outputs
- Component tests in simulation
- Run component tests on target

**Code Verification and Validation**
- Run system tests on integrated controller

**System Validation/Acceptance Tests**
- User Acceptance Testing

**Implementation**
- C, C++
- VHDL, Verilog
- Structured Text
- MCU, DSP, FPGA, ASIC, PLC

**Complete Integration & Test**
- System-Level Integration & Test
- Subsystem Integration & Test
- Subsystem Implementation
Examples of software safety requirements

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1c: Control flow analysis</td>
<td>+   +   ++  ++</td>
</tr>
</tbody>
</table>

Detect poor and potentially incorrect program structures

```c
int roll() {  
    return(rand() % 6 + 1);  
}  

void operation(int);  

void main() {  
    srand(time(NULL));  
    int die = roll();  
    if(die >= 1 && die <= 6)  
        /*Unreachable code*/  
        operation(die);  
}
```
Examples of software safety requirements

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control flow analysis</td>
<td>++</td>
</tr>
</tbody>
</table>

Detect poor and potentially incorrect program structures

```c
int orderregulate(void)
{
    int tmp, X;
    Increase_PowerLevel();
    SHR4.A = 22;
    tmp = PowerLevel SHR4.A;
    X = tmp;
    return X;
}
```
Examples of software safety requirements

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1f</td>
<td>++</td>
</tr>
</tbody>
</table>

Detect poor and potentially incorrect program structures

```
static int SHR5 = 5;
```

<table>
<thead>
<tr>
<th>Event</th>
<th>File</th>
<th>Scope</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Written value: 5</td>
<td>tasks1.c</td>
<td>_init Globals()</td>
<td>29</td>
</tr>
<tr>
<td>Written value: 28</td>
<td>tasks1.c</td>
<td>proc1()</td>
<td>103</td>
</tr>
<tr>
<td>Read value: 5</td>
<td>tasks1.c</td>
<td>proc1()</td>
<td>103</td>
</tr>
<tr>
<td>Read value: 5 or 28</td>
<td>tasks1.c</td>
<td>proc2()</td>
<td>112</td>
</tr>
</tbody>
</table>
Examples of software safety requirements

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1d</td>
<td>0</td>
</tr>
</tbody>
</table>

**Initiation**

- Architecture
- Design/Implementation
- Unit Testing
- Integration Testing

Detect poor and potentially incorrect program structures

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
Semantic Analysis Example

Read A INT;
A = A + 2 ;
...

If (A = 3) or (A = 7) or (A > 11)  // improve robustness in regards of specification
then
    B = C / (A - 8);
    ...
else
    ...
Endif ;

No test cases to write
No execution
No simulation
Examples of software safety requirements

<table>
<thead>
<tr>
<th>Initiation</th>
<th>Architecture</th>
<th>Design/Implementation</th>
<th>Unit Testing</th>
<th>Integration Testing</th>
</tr>
</thead>
</table>

**Requirements**
- System-Level Specification
- Subsystem Design
- Architectural Design
  - Environment Models
  - Physical Components
  - Algorithms

**Implementation**
- Component Design
  - Test vectors and expected responses
- Component Tests
  - Component tests in simulation
  - Run component tests on target
- System Tests
  - System tests in simulation
  - Run system tests on integrated controller
- Code Verification and Validation
  - Run system tests on integrated controller
- Subsystem Implementation
- Subsystem Integration & Test
- User Acceptance Testing
  - Complete Integration & Test
- System-Level Integration & Test
Examples of software safety requirements

Use Polyspace to assist with analysis of equivalence classes
- Analysis of interface data into the unit
- Analysis of control modes within the unit

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1b Generation and analysis of equivalence</td>
<td>++</td>
</tr>
<tr>
<td>classes</td>
<td>++</td>
</tr>
<tr>
<td></td>
<td>++</td>
</tr>
</tbody>
</table>

SW Unit 1

Input Data

SW Unit 3

invalid

valid

invalid
Example: Equivalence class testing at unit level

Testing

```c
int test_add( int a, int b )
{
    int c = Add1(a, b);
    if ( a >= 0 && b >= 0 && c < 0 )
    {
        fprintf ( stderr, "Overflow!\n" );
    }
    if ( a < 0 && b < 0 && c >= 0 )
    {
        fprintf ( stderr, "Underflow!\n" );
    }
    return c;
}
```

Fast and reusable

Exhaustive

No Testcases

No Execution

No Simulation
Examples of software safety requirements

<table>
<thead>
<tr>
<th>Initiation</th>
<th>Architecture</th>
<th>Design/Implementation</th>
<th>Unit Testing</th>
<th>Integration Testing</th>
</tr>
</thead>
</table>

**REQUIREMENTS**
- System-Level Specification
  - Architectural Design
    - Environment Models
    - Physical Components
    - Algorithms
- Subsystem Design
  - Component Design
    - Test vectors and expected responses
- Implementation
  - Code Verification and Validation
    - Run component tests on target
  - System Tests
    - System tests in simulation
  - Component Tests
    - Component tests in simulation
  - System validation/acceptance Tests
    - Run system tests on integrated controller
- User Acceptance Testing
  - Complete Integration & Test
  - System-Level Integration & Test
Examples of software safety requirements

<table>
<thead>
<tr>
<th>Initiation</th>
<th>Architecture</th>
<th>Design/Implementation</th>
<th>Unit Testing</th>
<th>Integration Testing</th>
</tr>
</thead>
</table>

**Methods**

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis of boundary values</td>
<td>A</td>
</tr>
<tr>
<td>1c</td>
<td>+</td>
</tr>
</tbody>
</table>

Use Polyspace to help with identification and analysis of boundaries

- Identification of Zero divisor
- Range analysis of boundaries

Testing at the edge of equivalence partition
Example: Optimize Design and Architecture decision

Non Robust Module

External code

Potential Runtime Error inside!!!
Example: Optimize Design and Architecture decision

Non Robust Module

Additional Range-Limiting Code

Free from Runtime Errors
Leverage an integrated tool chain
For Model-Based Design and automatic code generation and hand-written code

- Use traceability back to the model
- Use available context information for inputs and parameters
Resume: Scope of the metrics - Key Categories

- Quality Objectives
- Detailed Design Description
- Code Metrics
- MISRA-2004 Rules
- Un-reachable Branches
- Non-terminating Constructs
- Run-time Errors
- Dataflow Analysis
- Quality Plan
- SQO

Software Quality Objectives
A quality model called SQO

- **Key Categories**
  - List of suggested quality objectives
  - Thresholds to reach quality requirements
  - Built on metrics from HIS*

*) Hersteller Initiative Software (HIS)

Example of Software Quality Requirement defined in SQO:

3.7. **Systematic runtime errors**

SQR-200 The supplier shall demonstrate that for all files within a module a review of systematic runtime errors has been performed and that errors which have not been corrected are justified, for the following categories:

- Out-of-bounds array access
- Division by zero
- Read access to non-initialized data
Introducing the Software Quality Objectives (SQO) project

- ISO-26262 doesn’t detail how to set up OEM/suppliers interface

- OEM and suppliers have to agree on:
  - A quality model and a process guide
  - Compliant to ISO-26262

- One example is “SQO”:
  - OEM and suppliers created a working group*
  - They published a document named “Software Quality Objectives for Source Code”

* MathWorks acted as a facilitator in the group
Software Quality Objectives: Incremental Quality

**SQO Step 1**
- Quality Plan & Detailed Design
- Code Metrics
- 1st MISRA-2004 rules subset

**SQO Step 2**
- Systematic run-time errors
- Non terminating constructs

**SQO Step 3**
- Unreachable branches

**SQO Step 4**
- 1st subset of potential run-time errors
- 2nd MISRA-2004 rules subset
- 2nd subset of potential run-time errors

**SQO Step 5**
- 3rd subset of potential run-time errors
- Dataflow Analysis

**SQO Step 6**
- Start of Design Prototyping Testing Start of Production
SQO is compliant with ISO 26262

IEC certification process **requires** to define a quality model and its application

<table>
<thead>
<tr>
<th>ISO 26262 objective</th>
<th>SQO level</th>
<th>No/Partial/ Full coverage</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a Enforcement of low complexity</td>
<td>SQ0-1</td>
<td>Full</td>
<td></td>
</tr>
<tr>
<td>1b Use of languages subsets</td>
<td>SQ0-5</td>
<td>Full</td>
<td></td>
</tr>
<tr>
<td>1c Enforcement of strong typing</td>
<td>SQ0-5</td>
<td>Full</td>
<td></td>
</tr>
<tr>
<td>1d Use of defensive implementation techniques</td>
<td>SQ0-5</td>
<td>Partial</td>
<td>MISRA rules about if and switch statements</td>
</tr>
<tr>
<td>1e Use of established design principles</td>
<td>SQ0-5</td>
<td>Partial</td>
<td>Some design guidelines are checked at the code level with MISRA</td>
</tr>
<tr>
<td>1f Use of unambiguous graphical representation</td>
<td>N/A</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>1g Use of style guides</td>
<td>N/A</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>1h Use of naming conventions</td>
<td>SQ0-5</td>
<td>Partial</td>
<td>Naming of Global Variables</td>
</tr>
</tbody>
</table>

**It’s not** an additional task of the certification process
Using tools to implement SQO
Example SQR-150: Code complexity
Coverage: Full

Tools exist!

- To create reports and automate ISO 26262 AND SQO activities
- PASS/FAIL status for each objective
- Progress monitoring over time
- Clear list remaining items to be reviewed/fixed
- Pre-qualified for all ISO 26262 ASIL

<table>
<thead>
<tr>
<th>Files</th>
<th>Header Files</th>
<th>Lines</th>
<th>Lines of Code</th>
<th>Components Counting</th>
<th>Exeutable Lines</th>
<th>Cyclomatic Complexity</th>
<th>File Functions</th>
<th>Called Functions</th>
<th>Call Occurrence</th>
<th>Instructions/Call Lengt</th>
<th>Function Parameters</th>
<th>Code Statements</th>
<th>Revert Points</th>
<th>Quality Status</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Legend:
1. version_04_RTE_Rload
2. version_05_Adress_Error
3. version_06_Address_Changes
4. version_07_Address_Overlaps
5. version_08_SensorInfo_Added
6. version_09_DataRange_Added
Summary: Typical engineering challenges …

- Check MISRA
- Optimize Design and Architecture
- Measure Code Quality
- Comply with ISO, IEC safety requirements
- Deliver Bug free software
- Reduce time and efforts for software testing
Thanks!

Interested to reduce your ISO/IEC compliance effort?