Vom Prototyp bis zur Zertifizierung
Die Qualität von sicherheitskritischer Software mit Polyspace sicherstellen

Stefan David
Application Engineering
Today’s topics

- Introduction to the Polyspace approach
  - The cost of failure
  - Comparison with traditional techniques
- Workflows and Use Cases
  - Detecting SW defects and dead code
  - Automating robustness tests and code reviews
  - Checking coding rules and producing code complexity metrics
- Covering functional safety aspects
- Q&A
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
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
Examples of software bugs and errors

- Requirements based (functional) errors
- Run-time errors
- Concurrency issues
- Programming errors
- Dead or unreachable code
- Static and dynamic memory errors
Testing and verifying embedded software

- Good design, functional tests and simulation
  - Helps eliminate functional errors

- But, robustness and quality concerns may still exist
  - Undetected errors might cause catastrophic failures

- Polyspace: static code analysis using formal methods
  - Address robustness concerns
  - Ensures safe, portable and dependable software
STATIC CODE ANALYSIS
Polyspace covers many different approaches:

- Finds bugs
- Checks coding rule conformance
- Provides metrics
- Formally proves correctness and absence of errors

- Indicates when you’ve reached the desired quality level
- Certification help for DO-178, ISO 26262, IEC 61508, …
Examples: **Run-Time Errors** and **Programming Defects**

- Non-initialized data
- Out of bound array access
- Null pointer dereference
- Incorrect computation
- Concurrent access to shared data
- Illegal type conversion
- Unreachable states or code
- Overflows
- Non-terminating loops
- And lots more…

- Invalid use of = operator
- Invalid use of == operator
- Write without further read
- Uncalled function
- Missing null in string array
- Qualifier removed in conversion
- Race condition
- Invalid use of other standard library routine
- Memory leak
- And lots more…
Which problem is addressed by Formal Proof Methods?

Problem: Tests aren’t exhaustive

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

“Given that we cannot really show there are no more errors in the program, when do we stop testing?” (Hailpern [2])

“Imagine how much time is used debugging and reviewing correct software” (Stefan David, today)

---

Can you find a run-time error?

```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 run-time error?

```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 */
}
```

Could there be a bug on this line?
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: **339413 years**

Exhaustive Testing is Impossible
Other potential run-time errors to consider

```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 + 3;

    while (actuator_position < 10)
    {
        actuator_position++;
        tmp += sensor_pos2 / 200;
        i = 3;
    }

    if ((magnitude + 100) > 43)
    {
        magnitude++;
        x = actuator_position;
        actuator_position = x + 100;
    }

    return actuator_position * magnitude + tmp;  /* new value */
}
```

- **Variables may not be initialized**
- **Overflow potential**
- **Division by zero potential**
- **Dead code potential**
Results from Polyspace Code Prover
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;
    }
}
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

Polyspace verification on generated code
Defects detected by Polyspace Bug Finder

**Numerical**
Zero divide, Overflow, Shift Integer and Float conversion Overflow
Invalid use of std. library math routine, …

**Static Memory**
Array access out of bounds, Null pointer access, …

**Dynamic Memory**
Use of previously freed pointer, Unprotected dynamic memory allocation, …

**Programming Defects**
Invalid use of = or == operator, Declaration mismatch, …

**Dataflow**
Write without further read, Non-initialized variable, …

**Other**
Race Condition, …

For C and C++
Polyspace support for code rules compliance

- MISRA C Checker
- MISRA AC AGC -- application of MISRA-C for generated code
- MISRA C++ Checker
- JSF++ Checker
- Customization
  - Turn rules off / warning / error
- Custom Coding rules
  - Define custom naming conventions
  - Mark violations as reviewed or indicate future action
Metrics measured by Polyspace Bug Finder

**Files**
- Lines
- Lines of code
- Comment density
- Estimated function coupling

**Function**
- Lines within body
- Executable lines
- Cyclomatic complexity
- Language scope
- Paths
- Calling functions
- Called functions
- Call occurrences
- Instructions
- Call levels
- Function parameters
- Goto statements
- Return points

**Project**
- Files
- Header files
- Recursions
- Protected shared variables
- Non-protected shared variables

**Software quality objectives**
- Define custom levels with thresholds to measure achievement of a quality level
- Support for HIS (Hersteller Initiative Software) metrics
- Support for SQO (Software Quality Objectives) metrics

Polyspace web dashboard
Where is Polyspace used?
High Integrity Embedded Systems

Requirements

Architectural Design

Acceptance Test

SW Integration Test

SW Unit Design

SW Unit Test

Implementation

CP or BF

CP or BF

CP or BF

CP..Code Prover

BF..Bug Finder
Certification support
Functional Safety Standards: e.g. IEC 61508 and derivatives

Generic Safety Standard

IEC 61508 → EN 61508 European Norm

Sector / Product Specific Derivates

- Railway
- Medical
- Nuclear
- Process Industry
- Automotive

- EN 5012x
- IEC 60601
- IEC 62304
- IEC 61513
- IEC 61511
- ISO 26262

ISO … International Organization for Standardization
IEC … International Electrotechnical Commission
# Applicability to ISO 26262

ISO 26262-6 Software unit design and implementation

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
<th>Applicable Tools / Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1a Walk-through</td>
<td>++</td>
<td>+</td>
</tr>
<tr>
<td>1b Inspection</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>1c Semi-formal verification</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>1d Formal verification</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>1e Control flow analysis</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>1f Data flow analysis</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>1g Static code analysis</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>1h Semantic code analysis*</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

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)
### Methods

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
<th>Applicable Tools / Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a One entry and one exit point in subprograms and functions</td>
<td>++ ++ ++ ++</td>
<td>Polyspace Bug Finder – Coding Guidelines</td>
</tr>
<tr>
<td>1b No dynamic objects or variables, or else online test during their creation</td>
<td>+ ++ ++ ++</td>
<td>Polyspace Bug Finder – Coding Guidelines</td>
</tr>
<tr>
<td>1c Initialisation of variables</td>
<td>++ ++ ++ ++</td>
<td>Polyspace Code Prover – Code Verification</td>
</tr>
<tr>
<td>1d No multiple use of variables names</td>
<td>+ ++ ++ ++</td>
<td>Polyspace Bug Finder - Diagnostics</td>
</tr>
</tbody>
</table>

**Table 8 – Design principles for software unit design and implementation (1/2)**
**Applicable Tools / Processes for ISO 26262**

Example: Software unit design and implementation

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
<th>Applicable Tools / Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1e</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>Avoid global variables or else justify their usage</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>1f</td>
<td>0</td>
<td>+</td>
</tr>
<tr>
<td>Limited use of pointers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1g</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>No implicit data type conversions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1h</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>No hidden data flow or control flow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1i</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>No unconditional jumps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1j</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>No recursions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8 – Design principles for software unit design and implementation (2/2)
### Article by Elektrobit on use of Polyspace

**Removing Run-Time Errors from AUTOSAR Components Using Polyspace Code Verifiers**

Polyspace code verifiers enable us to demonstrate conclusively that the software we deliver is free of certain run-time errors. More importantly, they enable us to do so faster, more thoroughly, and with less manual review than was previously possible.

Alexander Much, Elektrobit
Medical Device Customer feedback

- ~200 defects in 15kloc, 2-3 critical found with Bug Finder
- “In my opinion every developer should use it from tomorrow” (SW Team Lead, Infusion Pumps)
- Note:
  - Found in workflow with existing static analysis tools in place.
  - Systems already certified to be compliant with functional safety standards
3 things to remember about Polyspace tools

- Helps to prevent and detect a large set of bugs early
- Provides value to Model-Based design and hand-code workflows
- Streamlines review, testing and certification tasks
Thank you!
Thank you!
## Capabilities of Polyspace Bug Finder and Polyspace Code Prover

<table>
<thead>
<tr>
<th>Capability</th>
<th>Polyspace Bug Finder</th>
<th>Polyspace Code Prover</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Bug detection, enforce code rules, review results*</td>
<td>Prove code correctness</td>
</tr>
<tr>
<td>When to use</td>
<td>Early during development</td>
<td>Early use and for QA signoff</td>
</tr>
<tr>
<td>Technology</td>
<td>Fast formal methods based static code analyzer for efficient bug finding</td>
<td>Deep formal methods based abstract interpretation engine for code proving</td>
</tr>
<tr>
<td>Color coded proof based results</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>(red, green, ...)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Obtain variable range data</td>
<td>Limited</td>
<td>Yes, detailed</td>
</tr>
<tr>
<td>Code rule checking (MISRA/JSF)</td>
<td>Yes</td>
<td>Yes, including XML output</td>
</tr>
<tr>
<td>Code size Limit</td>
<td>No limit</td>
<td>Practical limit based on computer hardware</td>
</tr>
<tr>
<td>Queue and Distribute Jobs</td>
<td>Yes (with PCT/MDCS)</td>
<td>Yes (with PCT/MDCS)</td>
</tr>
</tbody>
</table>

* Polyspace Bug Finder license can be used to view results from Polyspace Code Prover