Mit Polyspace die Qualität des generierten und handgeschriebenen Codes effektiv verbessern
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: **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…
Many executions for functional testing…
… do not prove code correctness

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
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” (C. Guss, today)

How is Polyspace code verification unique?

- 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;
    }
}
```
Many executions for functional testing…

… do not prove code correctness

More and more is proven correct

Which Quality Is Your Code Today?

Your source code

T0  T0 + 3 months  T0 + 6 months

0% proven correct

0,000000...1% executions verified

0,000000...2% executions verified

0,000000...3% executions verified
STATIC CODE ANALYSIS
Polyspace covers different approaches:

- Compiler Warnings
- Code Metrics, and Coding Rules
- Formal Methods (full proven)
- Bug Findings (Fast, more categories)
Polyspace product family for C/C++

- **Polyspace Bug Finder**
  - Quickly find bugs in embedded software
  - Identifies errors such as run-time, programming, dataflow, static and dynamic errors
  - Check code compliance for MISRA
  - Intended for *every day use* by software engineers

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

## 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>1a Walk-through</td>
<td>++</td>
<td>+ o o</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)
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
What does Polyspace help you with?

1. UNCOVER BUGS
   - Bug free code and efficient code reviews

2. SAVE TIME
   - Reduce the amount of robustness testing

3. CERTIFICATION DOCUMENTATION
   - Certification credits and automated reports