WHITE PAPER

7 Ways to Make Embedded Software Safe and Secure

with Static Analysis and Formal Methods

MathWorks®

Edsger Dijkstra, a pioneer in computer science, once said, "Program testing can be used to show the presence of bugs, but never to show their absence!" Yet, many embedded projects mistakenly interpret the absence of test failures as proof of quality. This white paper presents a unique way to address this problem.

Using modern static analysis tools that apply **formal methods**, development teams ***can*** *prove the absence of bugs*—without having to integrate the system, test it on hardware, or execute any code.

Polyspace® static analysis software provides formal methods technology in two products:

- ***Polyspace Bug Finder***™ identifies probable run-time errors and hundreds of other classes of bugs in source code, checks compliance with coding rules and security standards, and produces code metrics reports.
- ***Polyspace Code Prover***™ formally proves the absence of critical run-time errors in code units and integrated code, without test cases or code execution. Polyspace Code Prover uses formal methods (abstract interpretation) to consider all possible inputs, execution paths, and variable values, with no false negatives.

Polyspace is certified by TÜV SÜD for use in development processes that are required to comply with functional safety standards such as ISO 26262, IEC 61508, and IEC 62304, and it is qualifiable for use with DO-178B/C.

7 Ways to Make Embedded Software Safe and Secure with Polyspace Static Analysis and Formal Methods

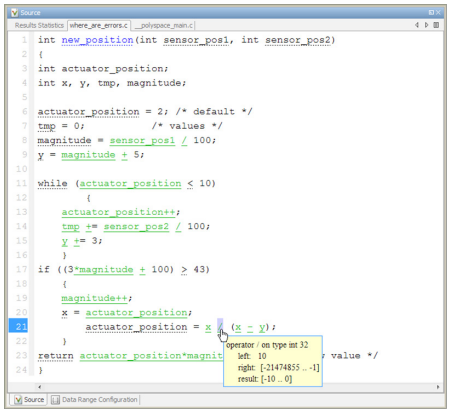
Development teams at Nissan, Airbus, Delphi, and other organizations using Polyspace report 7 key advantages:

1. **Immediate feedback for developers.** Polyspace provides detail such as run-time *variable range information for every point in the code,* potential overflow/underflow bug conditions, and dead code, and it helps enforce coding guidelines such as MISRA®.
2. **Focused unit test strategies.** By proving the *absence of critcal defects* ***across all possible inputs***, Polyspace reduces and guides unit test development efforts.
3. **Concurrency defect detection.** Polyspace *proves the absence of race conditions* in a multithreaded application and traces defects to the source.
4. **Documented flow information.** Polyspace provides *detailed control and data flow information* with exhaustive and sound semantic analysis.
5. **Security compliance**. Polyspace uses bug finding, code proving, and standards checking to *help identify and avoid security vulnerabilities* and comply with standards such as CERT C, ISO 17961, and CWE. It *can* ***prove*** *the absence of security vulnerabilities* that hackers exploit—such as buffer overflow, illegal pointer dereference, and uninitialized variables.
6. **Artifacts for certification standards.** Polyspace is *certified by TÜV SÜD for use in development processes that are required to comply with functional safety standards* such as ISO 26262, IEC 61508, and IEC 62304, and it is qualifiable for use with DO-178B/C. You can get certification credits—for example, for formal methods, control/data flow, and range checking—that you cannot get with other tools.
7. **Integration with Model-Based Design**. Polyspace is part of the toolchain for Model-Based Design—with *traceability to Simulink® and Stateflow® models.*

Testing vs. Formal Methods: A Quick Illustration

```
1  int new_position(int sensor_pos1, int sensor_pos
2  {
3  int actuator_position;
4  int x, y, tmp, magnitude;
5
6  actuator_position = 2; /* default */
7  tmp = 0;              /* values */
8  magnitude = sensor_pos1 / 100;
9  y = magnitude + 5;
10
11 while (actuator_position < 10)
12     {
13     actuator_position++;
14     tmp += sensor_pos2 / 100;
15     y += 3;
16     }
17 if ((3*magnitude + 100) > 43)
18     {
19     magnitude++;
20     x = actuator_position;
21     actuator_position = x / (x - y);
22     }
23 return actuator_position*magnitude + tmp; /* new
24 }
25
```

Example source code.



Polyspace analysis results.

Consider the function in the example, which has two inputs: How would you manually review this short piece of code, especially if you want to ensure the absence of defects?

Testing approach. If you consider testing, you would only need two test cases for full modified condition/decision coverage (MCDC). Is that sufficient to ensure robustness? *An exhaustive test would consider all possible values* for each of the inputs.

Static analysis. Static code analysis is often used to supplement test cases. Static analysis, which automates many manual verification tasks such as enforcing coding standards and style guides, can also find defects based on heuristics.

For this example, a static analysis tool could broaden test coverage by checking many possible values for the variables. But as a side effect it will produce many false warnings for values that will never occur.

Formal methods. Polyspace static analysis products cover these basic methods, but also use formal methods to verify run-time behavior and prove the absence of errors.

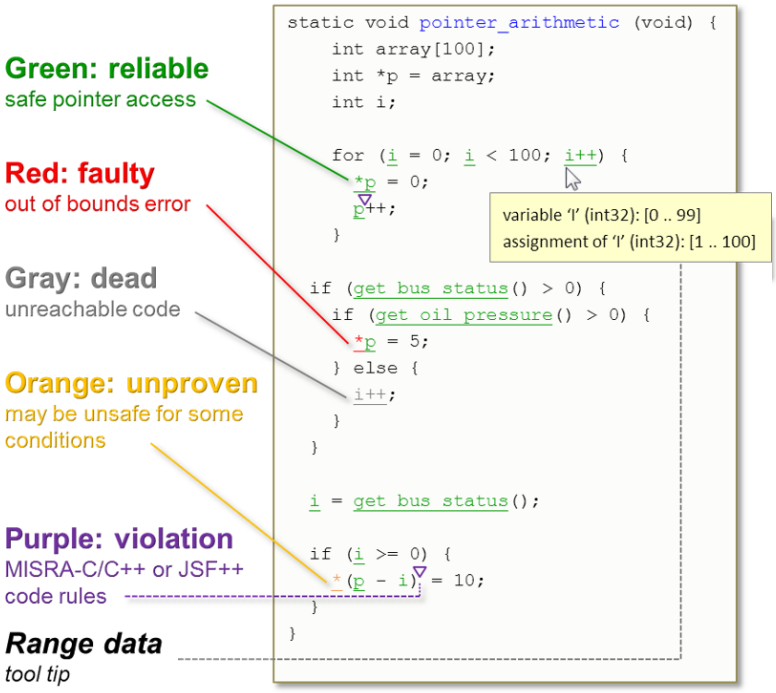
Polyspace uses proof-based techniques such as abstract interpretation to prove that the software is safe under all run-time conditions. It also identifies variable range information for every point in the code, and maps control and data flow.

Solution to the example. In the example code, Polyspace has proven that the code is free from critical run-time errors. Consider line 21, where a division-by-zero might be suspected. With formal methods, Polyspace has determined that the range of the variables x and y are such that they can never be equal, and therefore a divide-by-zero cannot occur on this line.

A Closer Look: 7 ways to make embedded code safe and secure

1. Immediate feedback for developers.

In embedded systems, a significant number of run-time defects are introduced during the coding stage. These errors are often missed during code review, and they cannot be detected by conventional static analysis. They propagate downstream and are often detected later, during hardware testing, causing rework and delays.



Run-time error attributes are color coded.

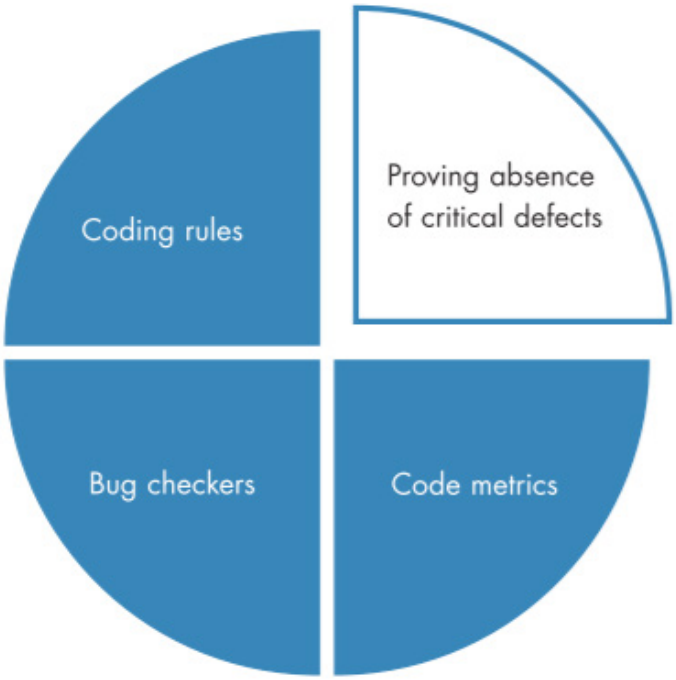
Polyspace addresses this by providing static analysis that includes run-time detail, such as ***variable range information for every point in the code***, potential overflow/underflow bug conditions, and dead code. It also helps enforce coding guidelines such as MISRA. By applying static analysis and formal methods, developers can see and fix run-time issues while coding, before they propagate any further.

> *"Dynamic testing merely allowed us to detect the symptoms of the errors. Polyspace code verification pinpoints their root cause, saving us significant debugging efforts."*
>
> — **Frédéric Retailleau, Delphi Diesel Systems**

2. Focused unit test strategies.

Unit testing can verify the robustness of components, but it requires writing test cases that can highlight run-time errors. Writing unit tests manually can be challenging, and adding test cases to increase coverage can still miss critical errors and/or lead to redundancies in the code.

Polyspace ***proves that code is safe from run-time failures,*** eliminating the need for robustness tests and identifying the specific unproven operations that need further analysis or tests. By proving the absence of critical defects ***across all possible inputs***, Polyspace reduces and guides unit test development efforts.



"Polyspace Code Prover identified issues in our handwritten code. It also identified code that was free from errors, and code that needed our close attention. The results enabled us to perform a targeted evaluation of the code during our formal inspection process."

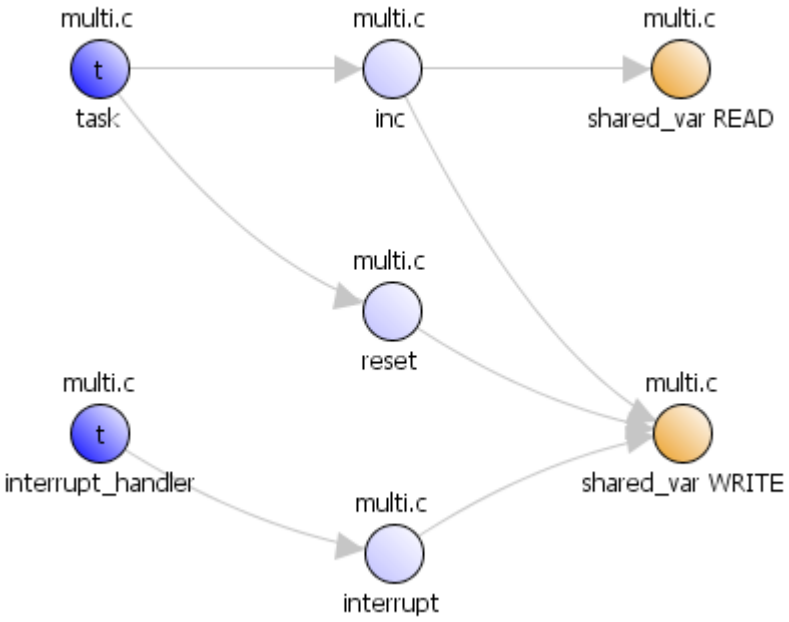
— Dr. Karen Gundy-Burlet, NASA Ames Research Center

3. Concurrency defect detection.

Polyspace ***proves the absence of race conditions and other concurrency issues*** in a multithreaded application and traces defects to the source.

Complex defects such as static memory issues, concurrency issues, and subtle run-time errors are hard to detect, and may slip into production. They require the right test cases and are difficult to reproduce.

Polyspace can help detect these defects before checking code into your source code repository, thereby avoiding test and debug of hidden bugs. With an event trace to step through in the debugging process, Polyspace reduces debugging effort for complex defects.



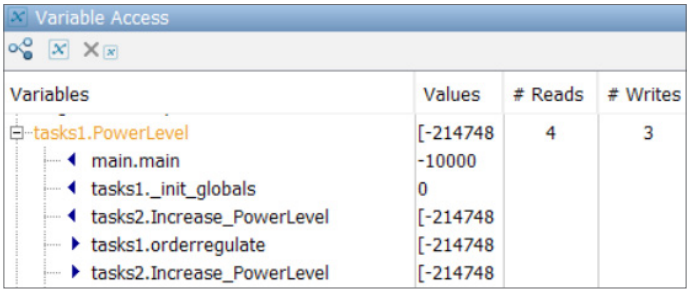
"Polyspace products not only find which operations can experience run-time errors, they also identify those that will never have one, no matter the operating conditions. Furthermore, they can do so during coding, thus before unit testing. This is of tremendous value to our suppliers."

— Mitsuhiko Kikuchi, Nissan

4. Documented flow information.

Polyspace provides ***detailed control and data flow information*** with exhaustive and sound semantic analysis. Control and data flow information is very useful while designing and debugging code, but it's difficult to trace with a manual approach.

Polyspace provides this information as function call trees, a global data dictionary, and variable data ranges that help in debugging complex run-time edge cases.



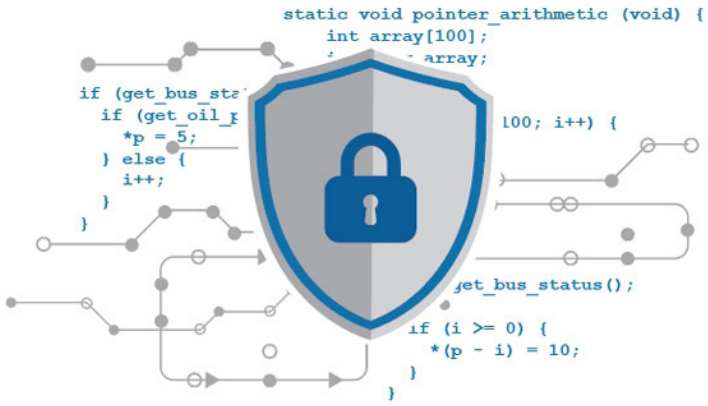
"The Polyspace solution is unique—it detects run-time errors without execution and has the advantage of being exhaustive."

— Airbus

5. Security compliance.

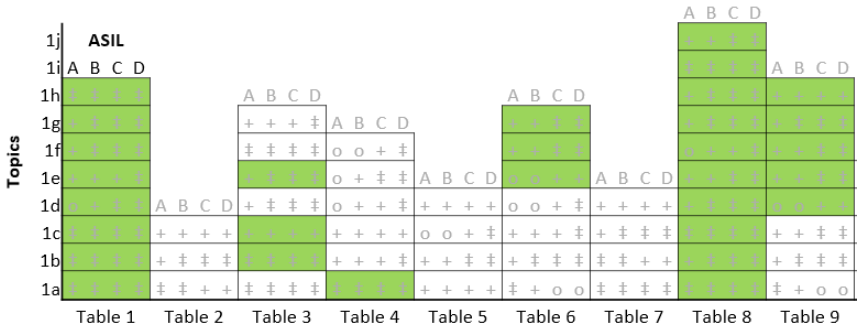
With increasing connectivity of embedded systems such as advanced driver assistance systems (ADAS), security is a growing concern, especially when it affects safety. A holistic approach to security spans the entire development cycle and therefore requires additional verification and validation.

Polyspace uses bug finding, code proving, and standards checking to help *identify and avoid security vulnerabilities* and comply with standards such as CERT C, ISO 17961, and CWE. It can ***prove the absence of security vulnerabilities*** that hackers exploit—such as buffer overflow, illegal pointer dereference, and uninitialized variables.



6. Artifacts for certification standards.

Polyspace is *certified by TÜV SÜD for use in development processes that are required to comply with functional safety standards* such as ISO 26262, IEC 61508, and IEC 62304, and it is qualifiable for use with DO-178B/C. You can get certification credits—for example for formal methods, control/data flow, and range checking—that you cannot get with other tools.

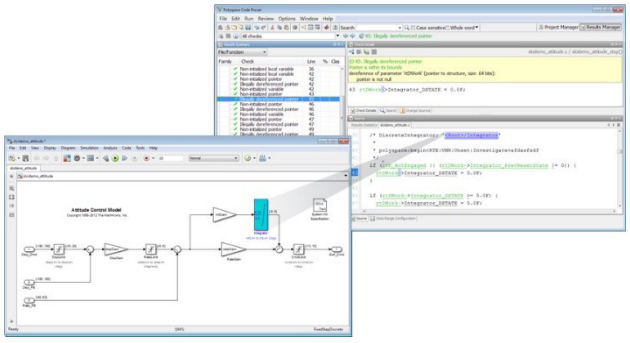


Credits for ISO 26262

> *"As we seek premarket approval status from the FDA, Polyspace Code Prover is central to our efforts to demonstrate that we have done our utmost to prove code correctness and ensure code quality."*
>
> **— Lars Schiemanck, Miracor Medical Systems**

7. Integration with Model-Based Design.

Polyspace is part of the toolchain for Model-Based Design—with *traceability to Simulink models*. You can run static analysis on generated code from either Simulink models or dSPACE® TargetLink® blocks. You can launch the analysis from within Simulink and trace the results back to the model.

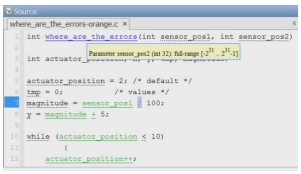


Example: Alenia Aermacchi used Polyspace to check code for run-time errors, ensure compliance with MISRA C coding standards, and create artifacts for certification credit. They qualified Polyspace code verifiers and Simulink verification products using DO Qualification Kit for DO-178.

Learn More

Solar Impulse Uses Polyspace Static Analysis for Solar Airplane

Using Polyspace products to find and eliminate problems earlier and faster helped Solar Impulse save 1–2 engineer-years of development time and satisfy objectives to ensure DO-178 compliance.



Debunking Misconceptions About Static Analysis

Debunk misconceptions about static analysis. These include statements like, “I don’t need it because I do sufficient testing,” or, “Static analysis is only necessary if you’re meeting certification.”

Request a Trial | Speak to an Expert