WHITE PAPER

10 Reasons to Use Static Analysis for Embedded Software Development

MathWorks®

Overview

Software is in everything. And in many embedded systems like flight control, medical devices, and powertrains, quality and reliability are critical. But developers sometimes have to make tradeoffs in how much verification to do on a tight deadline.

This paper presents a unique way to solve this problem. It introduces static analysis using formal methods to more thoroughly verify software quality, reliability, and security.

Edsger Dijkstra, a pioneer in computer science, once stated, "Program testing can be used to show the presence of bugs, but never to show their absence!" And yet, many projects interpret the absence of test failures as proof of quality.

Using static analysis with Polyspace® products, you can prove the absence of run-time errors. You can identify potential bugs in source code, even at the component level, without having to integrate with the system and test it on hardware, or execute any code.

There are 10 good reasons to use static analysis with Polyspace to verify software:

1. Polyspace provides immediate feedback to the developer, with detailed insights such as run-time variable range information.
2. Polyspace reduces and guides unit testing efforts, by proving the absence of defects across all possible inputs.
3. Polyspace finds dead code and provides code metrics—important for an effective code review.
4. Polyspace detects complex defects, like proving the absence of race conditions in a multi-threaded application and tracing defects to the source.
5. Polyspace helps enforce coding guidelines such as MISRA®—both decidable and undecidable rules and directives.
6. Polyspace helps you identify and avoid security vulnerabilities and comply with security standards such as CERT C, ISO 17961, and CWE.
7. Polyspace provides detailed control and data flow information with exhaustive and sound semantic analysis.
8. Polyspace provides a framework to manage code quality to achieve software quality objectives.
9. Polyspace provides artifacts to meet certification standards—ISO 26262, IEC 61598, IEC 62304, and DO-178B/C.
10. Polyspace is part of the toolchain for Model-Based Design—with traceability to and from a Simulink model.

A Quick Example

Before we get into the details of the top 10, consider the following example function with two inputs:

```
1   int new_position(int sensor_pos1, int sensor_pos2)
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 value */
24  }
25
```

How would you manually review this short piece of code, especially if you want to ensure the absence of defects? 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 analysis would consider all possible values for each of the inputs.

Static Code Analysis

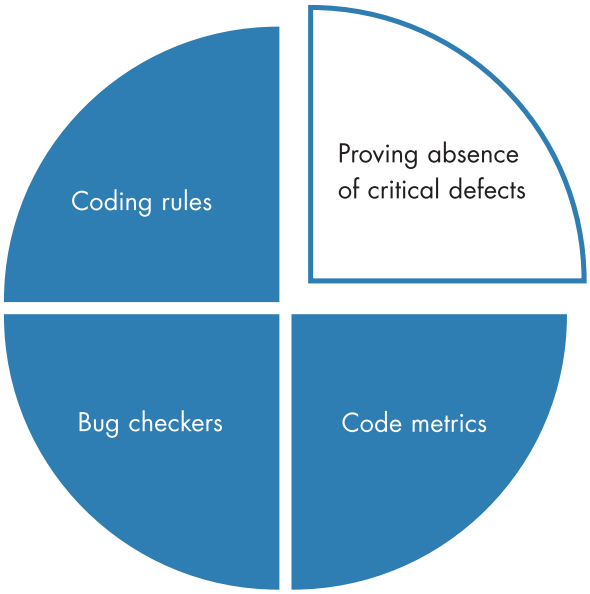
This is where static analysis comes in. Static code analysis refers to any technique to analyze source code without executing the software. Without the overhead of writing and running test cases or instrumenting your code, static code analysis automates manual verification tasks.

There are multiple forms of static analysis, and some of them are built into compilers and IDEs, or provided in well-known tools such as lint:

- Style checkers that can enforce certain coding standards and best practices such as style guides
- Tools that can enforce coding guidelines such as MISRA
- Tools that can help detect defects based on heuristics

Polyspace static analysis products include these, and also adds a unique approach by using formal methods to prove the absence of errors and verify run-time behavior. It goes beyond the use of heuristics to find bugs or implement coding guidelines. Polyspace uses proof-based techniques such as abstract interpretation to prove that the software is safe under all run-time conditions.

This provides a complete verification approach for software developers.



10 Reasons to Use Static Analysis with Polyspace Products

Polyspace helps software development teams and quality and test teams in a number of ways.

1. **Polyspace provides immediate feedback to the developer—with detailed insights such as run-time variable range information.**
   A significant number of defects are introduced during the coding stage. They propagate downstream, detected at different stages and need to be fixed later—causing rework and delays.

   ☐ Polyspace lets you see and fix issues while coding, before they propagate any further.

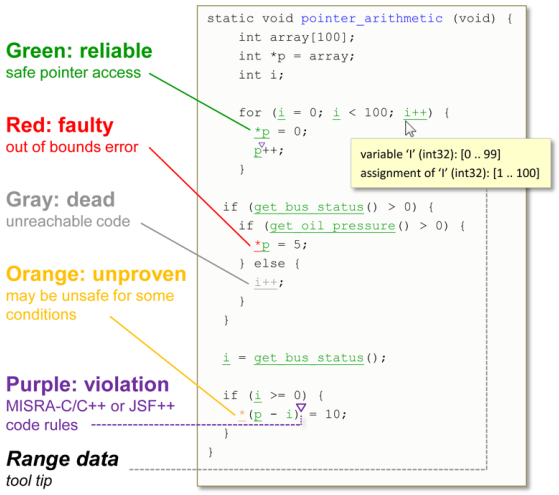
   ☐ Polyspace enables you to write better quality code with integration into your development environment.

"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. **Polyspace reduces and guides unit testing efforts—by proving the absence of defects across all possible inputs.**
   Unit tests are used to verify the robustness of components. This requires writing test cases that can highlight run-time errors. Writing unit tests manually can be challenging because of the guesswork and assumptions.

   ☐ 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.



Run-time error attributes are color-coded.

3. **Polyspace identifies dead code and provides code metrics—valuable insights for an effective code review.**
   Unreachable or dead code leads to unknown gaps in coverage. More test cases to increase coverage fail and lead to redundancies. Alternatively, you discover in a code review that an implementation is too complex and needs to be refactored.

   ☐ Code metrics such as cyclomatic complexity and function coupling in Polyspace help assess the design and implementation architecture.

   ☐ Polyspace identifies dead code at the time of implementation to avoid inefficiencies.

"Polyspace Code Prover identified dead code in our generated code as well as 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

4. **Polyspace detects complex defects—such as proving the absence of race conditions in your multi-threaded applications.**
   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 you check 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

5. **Polyspace helps enforce coding guidelines such as MISRA—both decidable and undecidable rules and directives.**
   Coding guidelines help avoid unsafe constructs for safety-critical applications. But manual checking for compliance to coding rules is laborious. Delegating this to the quality assurance team downstream creates late stage feedback to modify and reverify the code.

   ☐ Polyspace automates the enforcement of standards – MISRA C 2004, MISRA AC AGC, MISRA C 2012, or custom internal coding guidelines such as naming conventions. This helps your development team write code that is consistent, readable, and maintainable.

"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

6. **Polyspace helps you identify and avoid security vulnerabilities and comply with security standards such as CERT C, ISO 17961, and CWE.**
   With increasing connectivity of embedded systems such as 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 V&V.

   ☐ Polyspace can help avoid security vulnerabilities by detecting violations of secure coding rules like CERT C, ISO 17961, CWE.

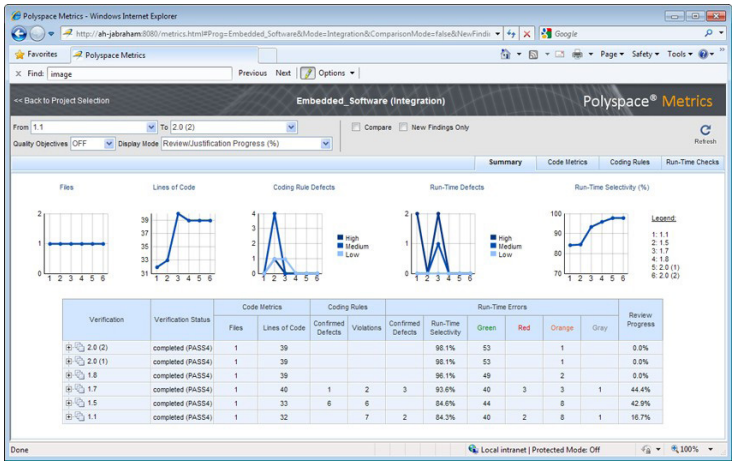
7. **Polyspace provides detailed control and data flow information—with exhaustive and sound semantic analysis.**
   Control and data flow information is very useful while designing and debugging your code. But a manual approach can be daunting.

   ☐ 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."
— EADS Launch Vehicles

8. **Polyspace provides a framework to manage code quality—and achieve software quality objectives.**
   Although teams may measure and document various quality metrics, the metrics are seldom help to guide the development process.

   ☐ Polyspace provides a framework to use a combination of standard metrics to determine the software quality level and track it through the development cycle.



You can define a centralized quality model to track run-time errors, code complexity, and coding rules violations, and track your progress toward predefined software quality objectives.

MathWorks®

9. **Polyspace provides artifacts to meet certification objectives of standards – credits for ISO 26262, IEC 61598, DO-178B/C, and IEC 62304.**
   Certification activities have very specific V&V objectives.

   ☐ Using Polyspace, you can automate and produce artifacts to meet certification requirements.

Alenia Aermacchi used Polyspace® static analysis tools to check the 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 and Validation using DO Qualification Kit for DO-178.

10. **Polyspace is part of the toolchain for Model-Based Design – with traceability to and from a Simulink model.**

    ☐ You can run static analysis on generated code either from Simulink® models or dSPACE® TargetLink® blocks. You can launch the analysis from within Simulink and trace the results back to the model.

On average, Polyspace can help software developers and quality engineers reduce development time by 12%.

ROI of Static Analysis with Polyspace Products

The ROI is strong for using Polyspace static analysis in your software development process. Users routinely report that Polyspace is a game changer.

- **40% time savings in manual code reviews**
  Reduces the engineering time spent on manual code reviews by providing artifacts that capture detailed control and data flow analysis
- **20% reduction in testing efforts—especially in robustness testing**
  Reduces the cost of debugging and fixing late stage defects and the related project delays by identifying defects early
- **300:1—Average engineering hours saved by fixing a bug before it escapes into the field**
  Avoids failure in the field from missed run-time errors by exhaustively proving run-time behavior without false negatives.

Polyspace static analysis provides an efficient, cost-effective solution that lets you deliver reliable embedded systems, especially those that must operate at the highest levels of quality, safety, and security.

Polyspace Static Analysis Products

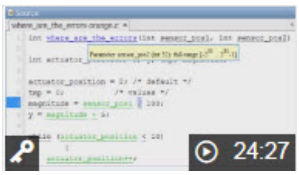
Polyspace static analysis products include Polyspace Bug Finder™ and Polyspace Code Prover™.

Polyspace Bug Finder identifies run-time errors, concurrency issues, security vulnerabilities, and other defects in C and C++ embedded software. Using static analysis, including semantic analysis, Polyspace Bug Finder analyzes software control, data flow, and interprocedural behavior. By highlighting defects as soon as they are detected, it lets you triage and fix bugs early in the development process.

Polyspace Code Prover is a sound static analysis tool that proves the absence of overflow, divide-by-zero, out-of-bounds array access, and certain other run-time errors in C and C++ source code. It produces results without requiring program execution, code instrumentation, or test cases. Polyspace Code Prover uses semantic analysis and abstract interpretation based on formal methods to verify software interprocedural, control, and data flow behavior. You can use it on handwritten code, generated code, or a combination of the two. Each operation is color-coded to indicate whether it is free of run-time errors, proven to fail, unreachable, or unproven.

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

93127v00 05/17