# Static Analysis of Dynamic Properties - Automatic Program Verification to Prove the Absence of Dynamic Runtime Errors

Klaus Wissing

PolySpace Technologies GmbH
Argelsrieder Feld 22
82234 Wessling-Oberpfaffenhofen
Klaus.Wissing@PolySpace.com

**Abstract:** This paper introduces formal verification techniques applied by PolySpace Verifier as a static approach to measure dynamic software quality attributes. It is proving the correctness of atomic operations in the source code in regards to run-time errors. PolySpace is unique in assessing dynamic properties with a static analysis of the source code. The document outlines the use of the results during maintenance, re-engineering and also development of software. It also gives a short tool description and an overview about used methods and techniques, supported programming languages and requirements.

## 1 Introduction

In V-Model style development processes, more then 50% of the development effort is spent for verification and validation, primarily done by means of testing. During structural/whitebox testing, coverage metrics (e.g. C0, C1, MCDC) are measured and often used as quality attributes monitored by QA. With functional/blackbox testing compliance with the expected functionality should be established.

However, Dijkstra already identified in the seventies that "Program testing can be used to show the presence of bugs, but never to show their absence". He also stated “The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness.”

Such proof also helps development engineers during maintenance and development. We thus propose to adopt a new approach to software verification: the use of Abstract Interpretation done by PolySpace. PolySpace implements this method of applied mathematics to prove the absence of semantic runtime errors (RTEs, runtime errors according to the programming language’s rules for defining data [types, structures…] and valid operations on the data; e.g. divide by zero, overflows, dereferencing a pointer into out-of-bounds areas beyond arrays, etc.). Faults caused by RTEs may lead to failure such as inconsistent data, non-determinism or system halt and might influence the

functionality of a program. According to a study on "Software defects and their impact on system availability", 26% of all observed software faults and more than 57% of all critical failures are caused by RTEs [SUL91].

Getting a mathematical proof for absence of semantic RTEs along with data dictionaries and call trees can help in source code reviews, software changes and re-engineering.

## 2 Static Formal Verification of Dynamic Program Property RTE

The tool PolySpace Verifier refers to an extension of static data-flow analysis aiming at verifying dynamic properties of programs using the mathematical framework of formal semantics [Deu04]. Pioneered by Wegbreit and Kildall, those techniques were first widely used by modern compilers to solve code optimization problems [Kil73], [Weg75].

Formal semantics is a part of theoretical computer sciences that engages in proving correctness of computer programs (verification) by the use of mathematic methods. PolySpace Verifier applies such formal methods to express the semantics of a source program (P) and to compute exhaustively, statically (without any specific input data) and automatically an abstract model (P') of the dynamic runtime behaviour of P.

Taken the model P', PolySpace Verifier proves (verifies and falsifies) source code properties against a formal specification (E), which is the runtime error behaviour as defined in the programming language used.

P = set of states of a program
P' = approximated superset of P
E = set of erroneous states defined by the programming language

Between P' and E four predications can exist:

| | |
|---|---|
| P' ∩ E is empty | => operation verified (no run-time error) |
| P' in E contained | => operation falsified (run-time error) |
| P' is Ø | => unreachable, dead Code |
| P' ∩ E is not empty | => unproved operation (potential run-time error) |

Thanks to increasing processor performance and new, very effective methods to statically represent dynamic control structures (e.g.: for loop, switch, if-then-else), elaborated data types (pointer aliasing, array and structures) and intricate control flows (e.g.: function calls), static verification can now be used to automatically verify or falsify the correctness of results of operations under all operating conditions at unit, module or integration level of programs in a fast way.

PolySpace Verifier, as an exhaustive approach, issues checks to prove the result of each and every operation in the source code considering the variation domain for the variables involved in the respective operation. The computed variation domain of any program variable is always equal to it's real variation domain or a superset of it. The direct consequence is that PolySpace Verifier never will report a wrong proof for any operation checked in an analysis.

PolySpace Verifier currently supports the following programming languages:

| | | | |
|---|---|---|---|
| C: | ISO/IEC 9899:1990 | C++: | ISO/IEC 14882:1998 |
| ADA95: | ISO/IEC 8652:1995 | ADA83: | ANSI/MIL-STD-1815A-1983 |

## 3 Applying PolySpace Verifier

PolySpace Verifier takes as input the source code of an application (at function-, unit- or integration-level) and produces result as tabulated text as well as a color-coded source where each operation is classified according to the RTE properties if it were executed. There are four categories:

- Green : the operation will never trigger a run-time error for all possible executions of the program (verified, proven free of runtime errors)
- Red : the operation will always (i.e. at each execution of the program) generate a run-time error (falsified)
- Grey : the operation cannot be executed – it is a piece of dead code
- Orange : unproved operation – there may be an error depending on the specific calling context of the function that contains the operation

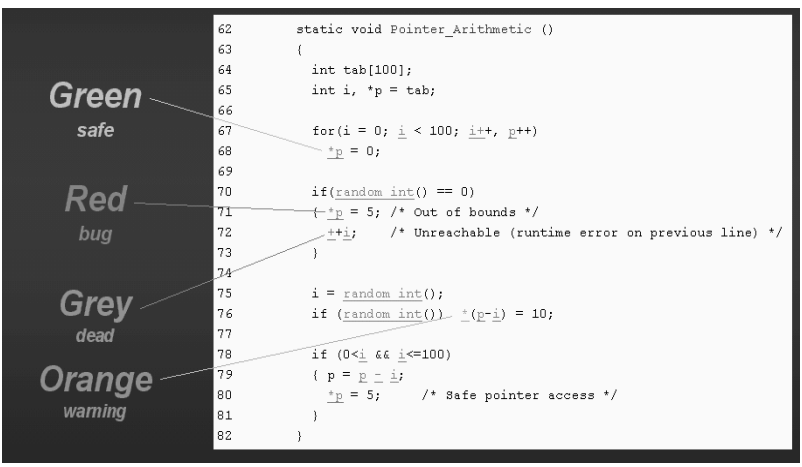


Figure 1: example of a color-coded source code provided by PolySpace Verifier

Run-time errors checked by PolySpace Verifier include:

- Dereferencing through null
- Out-of-bounds pointers

- Out-of-bounds array accesses
- Read access to a non-initialized data
- Access conflicts on shared data (multithreaded apps and/or interrupt routines)
- Invalid arithmetic operations: division by zero, square root of a negative number,…
- Overflow and underflow on integers and floating-point numbers
- Unreachable (dead) code

Regarding control and data flow documentation and understanding, PolySpace Verifier builds the global data dictionary, a concurrent access graph for each shared variable of the program and a call tree for the program. All these results of a verification can be interactively browsed and help understanding the semantics of sources.

## 4. Using Verification Results to Assess Status and Improvement Requirements

That said, proving the correctness of an atomic operation, taking into account all possible operand values in any combination (incl. worst-case) represents the robustness of this operation. In theoretical computer science, correctness of an algorithm is asserted when it is said that the algorithm is correct with respect to a specification. PolySpace Verifier proves the correct implementation of a program according to the formalized programming language specification. The PolySpace metric for robustness of a program is the amount of verified operations within a program (at function-, unit-, or integration-level) compared to all evaluated operations:

$$\text{Robustness} = \frac{\sum \text{verified (correct) operations}}{\sum \text{evaluated operations}}$$

Knowing the robustness of a source under development or maintenance helps to schedule further analysis and/or development by development engineers and QA. Focussed functional testing in case of low robustness measures also can be planned and scheduled so that the risk in re-using existing modules is understood.

Robustness also can be automatically and objectively measured during initial exploration of unknown sources. If the development has been outsourced, the basic quality of the results easily can be assessed during acceptance tests. Quality thresholds can be set and monitored to establish the extend of improvements accomplished.

Other measures like semantic code-density can be derived from the verification results. They may help - along with other source attributes like time to understand a function - to decide which parts of the software might be re-used or if a redesign should be considered.

Furthermore developers or QA can detect resource wasting anomalies, such as dynamically dead code or unused variables. In case of slim resources commonly found in embedded applications, appropriate types of variables can be determined by observing the values of variables for the complete program-flow.

A PolySpace verification also can be used to experiment and understand different execution scenarios for the software without the need to actually execute it in a respective context. Running applications in different scenarios sometimes can get very expensive (e.g. if significant amount of external input needs to be simulated or generated by special hardware) or may not even be feasible (e.g. if the software is to control security functions). In such cases verification also is a good means to prepare an actual test: if there is a proof that no runtime error will happen, testers do not need to worry about detection of robustness issues during the test. If unexpected dead code, potential robustness issues or even certain errors are identified by the verification, the test campaign will not be started prior to fixing the issues.

This approach also applies for design flaws in a model-based design environment with automatic code-generation.

For example, after the verification of the generated sources a model developer will exactly know where a saturation block is needed and where useless blocks may be put. This can gain a high level of robustness and execution efficiency at the same time.

## 5 Conclusion

Static analysis to prove the absence of run-time errors, once the domain of theoretical researchers, now is commercially available with PolySpace verifier. Verification is a repeatable technique that may be used at any time, without any prior knowledge of the code to be analyzed. PolySpace's semantic, interactive browsing of sources is well suited for use in maintenance and re-engineering projects as well as in all stages of ongoing development prior to functional testing and validation.

PolySpace also provides a strong improvement in reliability and robustness as it is exhaustive by design and provides objective robustness measures.

## References


[Deu04] Deutsch, Dr. A. Next generation testing tools for Embedded Applications. White paper, PolySpace Technologies, www.polyspace.com, 2004.

[Kil73] Kildall, G. A unified approach to global program optimization. Proceedings of the ACM Symposium on principles of programming languages, 194-206. 1973

[Sul91] Sullivan, M. and Chillarege, R. Software defects and their impact on system availability. Proc. 21th International Symposium on Fault-Tolerant Computing (FTCS- 21), Montreal, 1991, 2-9, IEEE Press.

[Weg75] Wegbreit, B. Property extraction in well-founded property sets. IEEE Transactions on software engineering, 1(3), 270-285. 1975