# Verifying Code When Software Reliability Is Critical

By Paul Barnard, Marc Lalo, and Jim Tung

For many embedded software projects, the primary verification goal is to find as many bugs as possible, as quickly as possible. The static analysis tools commonly used for this purpose are good at detecting flaws, but they do not prove that source code is free of fatal run-time errors. As a result, these tools do not prevent endless debugging loops and lengthy code-checking procedures. More seriously, they may leave potentially catastrophic defects undetected—unacceptable for applications that require high reliability. PolySpace™ code verificaton products provide a different approach—one that proves the absence of certain types of run-time errors.

To accomplish this, a code verification tool must exhaustively investigate every piece of code and verify its reliability against all possible data values. It must often perform tasks of a mathematical sophistication that exceeds the capabilities of a standard error-detection tool. For example, it must:

- Solve numerical algorithms
- Interpret pointers
- Mathematically model programming constructions, such as loops, if-then-else, and undecidable conditions
- Read all language-specific constructions

In addition, the tool must provide diagnostics in a format that is simple and accessible to any user (Figure 1).

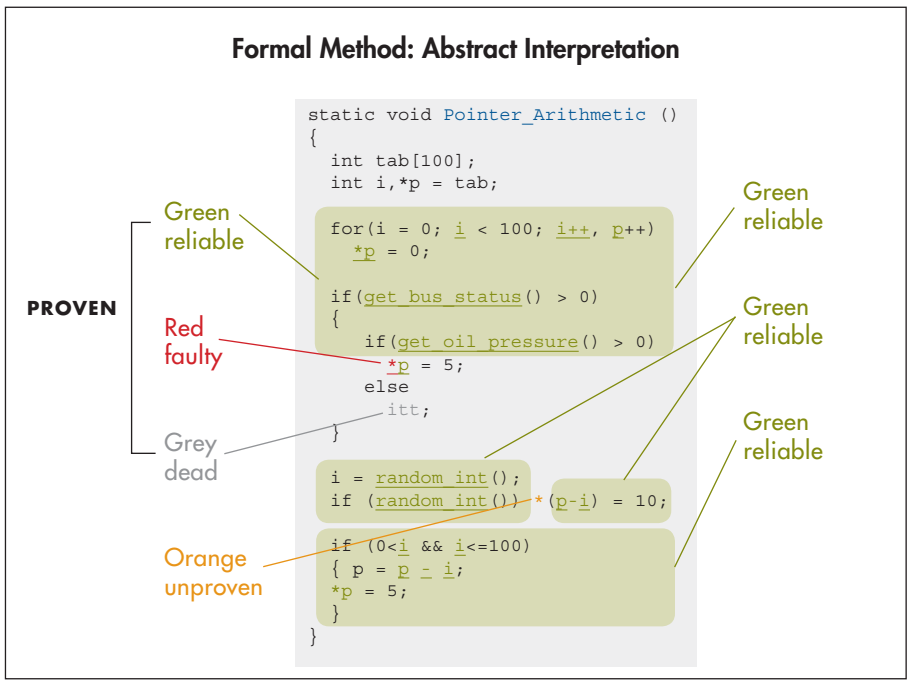


Figure 1. *Using simple color-coding to show code verification results.*

## Solving Numerical Algorithms

Among the arithmetic errors that can occur in the following code, we need to prove the absence of a division by zero at the highlighted line. The problem can be reformulated

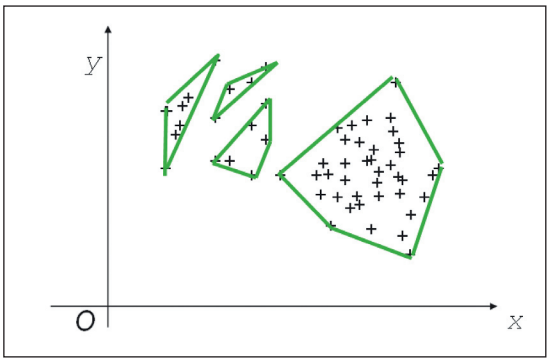


FIGURE 2. *A cloud of data points approximated by complex polyhedrons, used internally by the verification tool.*

as the following question: Can x and y be equal at the same time?

```
int where_are_the_errors (int input)
{
    int x,y,k,l;
    k = input / 100;
    x = 2;
    y = k + 5;
    while (x < 10)
    {
        x++;
        y = y + 3;
    }

    if ((3*k + 100) > 43)
    {
        y++;
        x = x / (x - y);
    }

    return x;
}
```

To answer that question, the verification tool must statically compute x and y, starting at line 1 and going through the code line by line, an exhaustive procedure that would be virtually impossible to perform accurately during a visual code review.

To solve these kinds of numerical problems, code verification tools must understand and model data closely—for example, by using complex polyhedrons or other geometrical shapes (Figure 2).

## Reading Pointers (Aliasing)

When data refers to other data—for example, by means of pointers in the C language—the verification tool must recognize the original data and the referenced data. It must then perform pointer analysis to determine how the original data and the referenced data affect each other.

In the following code, the verification tool must recognize that aliasing is subject to a conditional reassignment at the first highlighted line.

```
# define RANGE (X,MIN,MAX) \
X=random; while ((X<MIN) ||
(X>MAX)) X = random;
volatile int random;
int x,y;
int f(int *ptr)
{
    int results;
    if (random) ptr = &y;
    *ptr = *ptr + 1;
    results = x + y + *ptr;
    return results;
}
void main(void)
{
    int tmp;
    RANGE(x, 0, 10);
    RANGE(y, -10, 10);
    tmp = f(&X);
    tmp = (2*tmp*tmp + 3*tmp + 1) /
    (tmp + 19);
    tmp++;
}
```

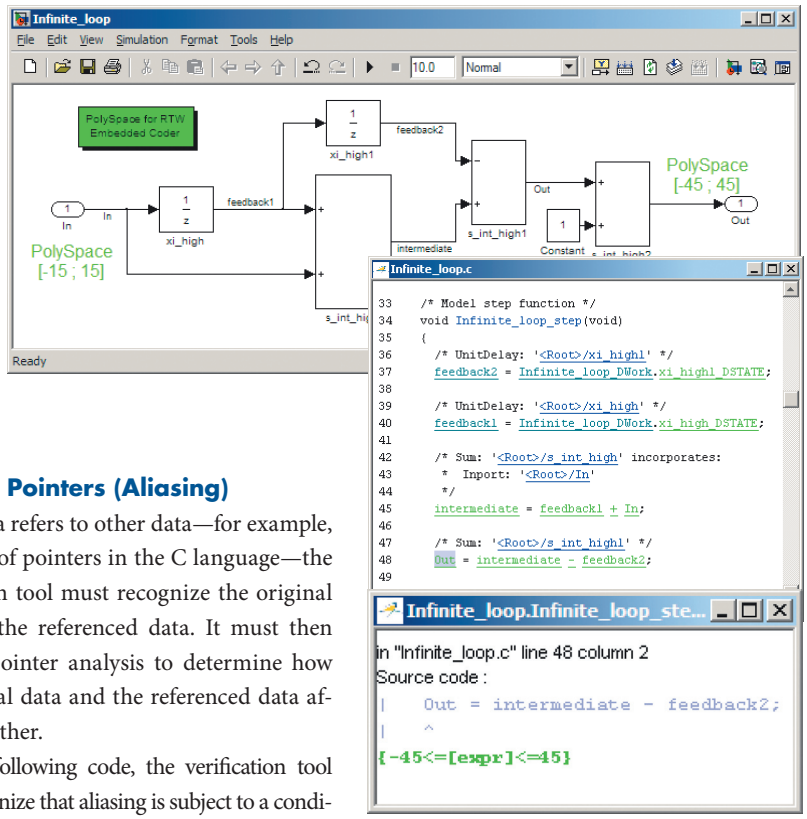


FIGURE 3. *Top: Model invoking two-step memorization of variables. Bottom: Code generated from the model, showing how data ranges can be propagated throughout code called within an infinite loop.*

At the end of this statement, ptr points to either x or y, resulting in two potential paths for the results. In both cases, tmp is greater than -19, making the division by zero on the second highlighted line impossible.

## Modeling Programming Constructions

The verification tool must understand the underlying syntax of programming constructions such as for-loops, if-then-else, and indeterminate conditions. For instance, it must mathematically compute what happens within an infinite loop without executing it (Figure 3).

## Reading Language-Specific Constructions

In certain languages, including C++, a massive amount of computation can be hidden behind a single line of code. With object-

## Verifying Code with PolySpace Products

**Delphi Diesel Systems** develops diesel injector technology to enable global OEM automotive customers to reduce noise levels, pollutant emissions, fuel consumption, and torque. Delphi uses PolySpace products to analyze software modules as soon as they are available—before functional unit tests.

**CSEE Transport,** a leading developer of signaling and control command systems for high-speed rail transportation, uses PolySpace products to verify the security software for their signaling system. Some run-time errors in the hand-written Ada code were undetectable by classical verification. CSEE Transport used PolySpace products to validate individual modules before final integration.

**NATO HAWK Management Office** (NHMO) maintains a range of complex, mission-critical applications for the HAWK surface-to-air missile system. To meet reliability standards, the NHMO team must identify and eliminate run-time errors. NHMO uses PolySpace products to perform a detailed analysis of application dynamics, generating information that helps set code review priorities.

**GlucoLight Corporation** developed a noninvasive, continuous glucose monitoring system that uses imaging technology and requires no manual intervention. To improve the reliability of supplier-provided embedded software and to prepare for FDA certification, GlucoLight used PolySpace products to verify new or changed C++ code on a class-by-class basis. PolySpace products highlighted error-prone structures, enabling developers to avoid using these structures in later versions of the code.

oriented constructs such as inheritance, polymorphism, and templates, verifying C++ code can quickly become complicated. The code verification tool must mathematically determine which instructions are being executed at each line of the program.

### Making an Intensive Task Practical

Proving the absence of run-time errors such as overflows, divide by zero, and out-of-bounds array access in source code is an intensive task requiring the use of formal mathematics and full knowledge of language semantics. PolySpace products make this task practical because they verify source code to prove the absence of certain run-time errors without the need to compile and run the code. Since the analysis is performed on the source code, they can be used in applications involving hand-written code, automatically generated code, or a combination of the two.

In addition to performing code verification, PolySpace tools test for MISRA-C® compliance. For applications in which code is automatically generated from Simulink® models or UML diagrams, PolySpace link products connect the verification results back to the original model.

PolySpace tools have been applied effectively throughout the software development and verification process (see sidebar). They have been used as a quality gate for code written by different development teams. They have been integrated with the "submit" and "build" steps for individual software components, providing more immediate feedback to the software developer. With back-annotation links available to modeling tools such as Simulink and Rhapsody®, they have been used in the design phase to help the system engineer or algorithm developer detect design weaknesses in the design by analyzing the automatically generated code. ■

### Resources

**POLYSPACE PRODUCTS**
*www.mathworks.com/nn8/polyspace*

**WHITE PAPER: Code Verification and Run-Time Error Detection Through Abstract Interpretation**
*www.mathworks.com/nn8/abstract_interpretation*

91607v00 10/08