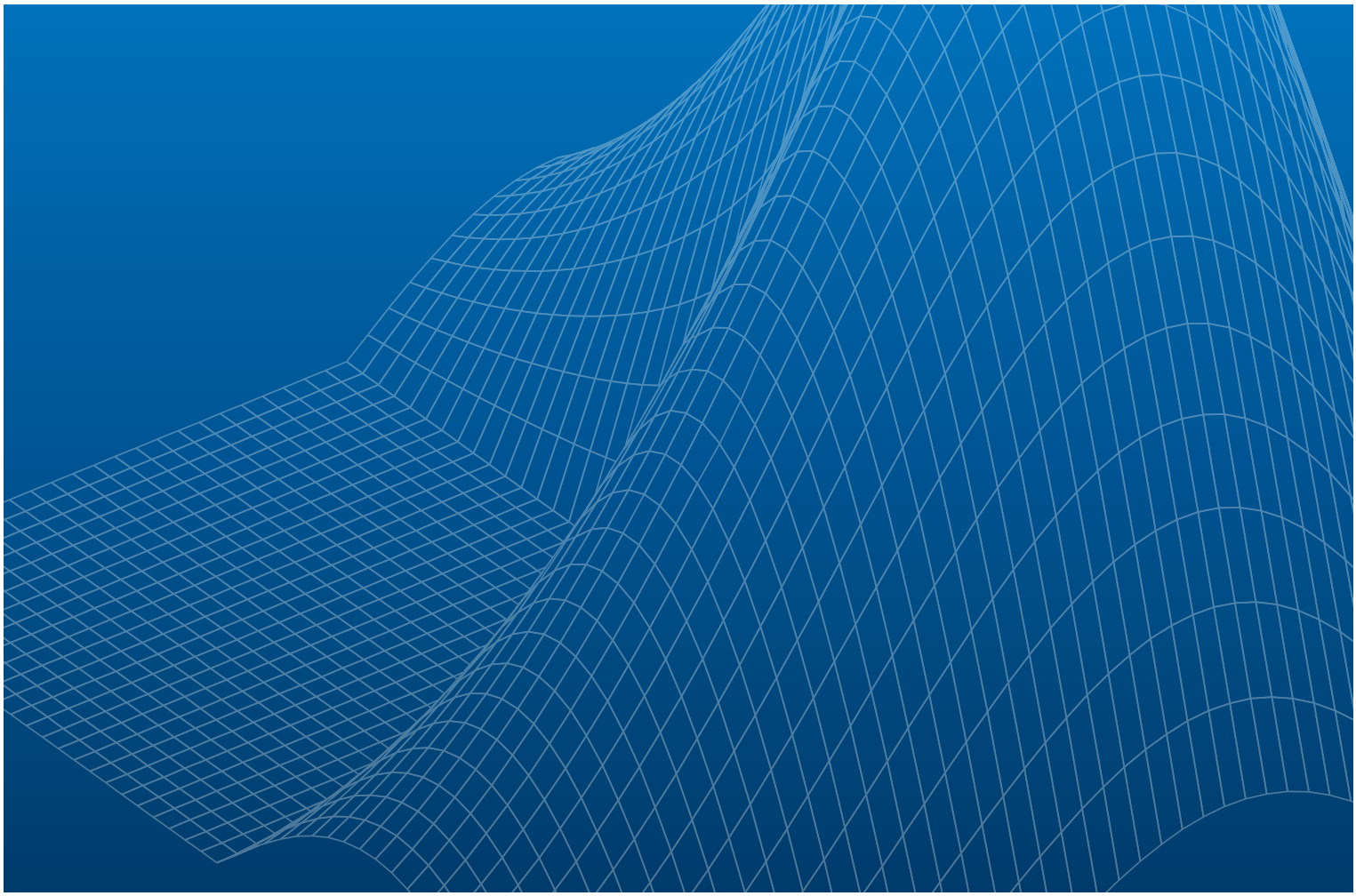

Comprehensive Static Analysis Using Polyspace Products

A Solution to Today's Embedded Software Verification Challenges

WHITE PAPER

MathWorks®

Introduction

Verification of embedded software is a difficult task, made more challenging by time pressure and the increasing complexity of embedded systems. Most software development teams rely on manual or non-exhaustive techniques to detect bugs and run-time errors in software. Code reviews are labor-intensive and often impractical for large, complex applications. Dynamic, or white-box, testing requires engineers to write and execute numerous test cases. When tests fail, additional time is required to find the cause of the problem through an uncertain debugging process. Testing is not exhaustive and cannot alone be relied on to produce safe software.

Polyspace static analysis products take a different approach. They find bugs in embedded software and use proof-based techniques such as abstract interpretation to prove that the software is safe. Polyspace Bug Finder™ identifies run-time errors, data flow problems, and other defects in C and C++ embedded software. Using static analysis, Polyspace Bug Finder analyzes software control, data flow, and interprocedural behavior. It lets you triage and fix bugs early in the development process.

Polyspace Code Prover™ provides a much deeper analysis, which proves the absence of overflow, divide-by-zero, out-of-bounds array access, and other critical run-time errors in C and C++ source code. It uses formal method techniques such as abstract interpretation to prove code correctness. In Polyspace Code Prover run-time verification results, each operation is color-coded to indicate whether it is free of run-time errors, proven to fail, unreachable, or unproven.

With abstract interpretation, Polyspace Code Prover automatically verifies important dynamic properties of programs, including proving the presence or absence of run-time errors. It combines the pinpoint accuracy of code reviews with automation that enables early detection of errors and proof of code robustness. By verifying the dynamic properties of embedded applications, abstract interpretation encompasses all possible behaviors of the software and all possible variations of input data, including how software can fail. It gives proof of code correctness, thereby providing strong assurance of code reliability.

By using bug finding and code proving tools, businesses can reduce costs while accelerating the delivery of reliable embedded systems. This paper describes how to use Polyspace Bug Finder and Polyspace Code Prover and the technique of abstract interpretation to overcome the limitations of conventional techniques for verifying embedded software.

Challenges in Testing Embedded Software

As processing power and memory costs have decreased, embedded software applications have become ubiquitous. Over the past several years, strong demand for complete, multipurpose software applications has led to larger, more complex embedded systems. In some industries, the quantity of embedded software doubles every 18 months.

Because the number and complexity of these applications continues to grow, the safety-, mission-, life- or business-critical aspects of many embedded applications increasingly demands higher levels of reliability and robustness.

Market pressure affects all software development. An application under development can be subject to constant changes, including requirement modifications, specification updates, and design changes.

Development organizations must often meet the conflicting business objectives of delivering higher quality embedded software while reducing time to market. The challenge is increased by the complexity of the applications being developed and the frequent shortage of engineering resources.

One solution is to improve efficiency by using software tools for code generation, design, and code instrumentation. These tools have enabled embedded software development teams to do more in less time. However, testing and debugging tools have not kept pace with the increases in embedded software size and complexity. As a result, the cost of testing an embedded system today can be more than half of the total development costs.

Early code verification is an effective way to relieve time and cost pressures. Errors detected in the late phases of the development cycle are more difficult to debug because they must be traced back to their source among thousands or millions of lines of code, usually by testers who did not write the code themselves. The cost of fixing problems found late in testing is 10 to 20 times higher[1] than the cost of fixing the same errors during coding.

While early code verification offers clear benefits, it is still the exception rather than the rule. For many teams, this means that testing is done close to the project's deadline, when time pressure is at its highest. In this environment, run-time errors—which are typically among the most difficult to discover, diagnose, and fix—can be missed altogether.

These errors, caused by arithmetic and other anomalies in the code, are also known as latent faults because they are not readily apparent under normal operation. Examples of such errors include division by zero, out-of-bounds array accesses, illegal dereferencing of pointers, and reading non-initialized data. Software developers find that 20-50% of bugs detected in software applications during maintenance are run-time errors.[2] These errors may cause non-deterministic behavior, incorrect computations, such as integer overflow, or processor halt. They all have unpredictable, sometimes dramatic, consequences on application reliability.

Limitations of Common Run-time Error Detection Techniques

The methods and tools conventionally used to detect and debug run-time errors rely on technology that is several decades old. The approaches fall into two broad categories: manual code reviews and dynamic testing. These techniques focus on finding some and not all errors. They are unable to verify that there are no more run time errors in the software.

Manual code review can be an effective technique for finding run-time errors in relatively small applications that have 10,000-30,000 lines of code. For larger systems, manual review is labor intensive. It requires experienced engineers to review source code samples and report dangerous or erroneous constructs, an activity that is complex, non-exhaustive, non-repeatable, and costly. Proving the absence of run-time errors is a complex operation that is not manageable by code review.

---

1 Basilli, V. and B. Boehm. "Software Defect Reduction Top 10 List." Computer 34 (1), January 2001

2 Sullivan, M. AND R. Chillarge. "Software Defects and Their Impact on System Availability." Proceedings of the 21th International Symposium on Fault-Tolerant Computing (FTCS-21), Montreal, 1991, 2-9. IEEE Press

Dynamic testing requires engineers to write and execute test cases. Many thousands of test cases may be required for the typical embedded application. After executing a test case, the test engineer must review the results and trace errors back to their root cause. This testing methodology has not improved much in the last four decades. While engineers can take other steps to improve the chances of detecting anomalies, such as code instrumentation and code coverage, dynamic testing is based on a trial-and-error approach that provides only partial coverage of all the possible combinations of values that can be encountered at run time. Like manual code reviews, the process is resource-intensive. Time spent writing the test cases and waiting for an executable system to be developed often forces dynamic testing to be delayed until the end of the development cycle, when errors are most expensive to fix.

Challenges of Using Dynamic Testing to Find Run-time Errors in Embedded Applications

Testing costs increase exponentially with the size of the application. Since the number of errors tends to be constant for a fixed number of lines of code—a conservative estimate is 2 to 10 errors per 1,000 lines—the chance of finding these errors decreases as the total number of lines in a software application increases. When the size of source code is doubled, the testing effort generally must be multiplied by four or more to obtain the same level of confidence as before.

Dynamic testing identifies only symptoms of the error, not the cause. As a result, additional debugging time must be spent to reproduce each error and then localize its cause after it has been detected during program execution. Tracing an anomaly back to its root cause in the code can be extremely time-consuming when the defect is detected late in development. An anomaly detected in validation can take 100 hours to trace, while one caught in unit testing may be localized in an hour or less. While some testing activities can be automated, debugging cannot. For example, if every 1,000 lines of new code include 2 to 10 errors, a 50,000 line application would contain a minimum of 100 errors. A developer spending an average of 10 hours debugging each error would need 1,000 hours to debug the application.

Dynamic testing frequently leads engineers to instrument their code, so that anomalies can be observed more easily during program execution. But code instrumentation takes time, adds execution overhead, and can even mask errors, such as memory violation and access conflicts on shared data. Methods based on code instrumentation detect errors only if the test cases that are executed raise an anomaly.

The effectiveness of run-time error detection depends on the ability of the test cases to analyze the combinations of values and conditions that can be encountered at run-time. Test cases generally cover only a fraction of all possible combinations. This leaves a large number of untested combinations, including some that may cause a run-time error.

Finding Bugs Early in the Development with Polyspace Bug Finder

When code is being developed, it is recommended to check it for bugs early in the development process. Software engineers want quick and efficient methods for identifying bugs in software. Polyspace Bug Finder is a static code analysis tool for analyzing code components or entire embedded software

projects. Polyspace Bug Finder uses fast, static code analysis techniques including formal methods with low false positive rates to pinpoint numerical, dataflow, programming, static memory, dynamic memory, and other bugs in source code.

These defects are identified in the source code, with trace-back information to help identify the cause of the defect. Coding rules violations (MISRA C/C++ and JSF++) are identified directly in the source code, with informational messages about the rule violation. This information can be used to iteratively debug and fix code early in the development process. Polyspace Bug Finder supports command line invocation, standalone graphical user interface and use with the popular Eclipse™ IDE. It can be integrated into build environments for automated use. It can also be integrated into automatic code generation environments provided by Simulink® or TargetLink® or IBM® Rational® Rhapsody®.

For each defect detected, Polyspace Bug Finder provides detailed information regarding the cause of the defect. For example, in situations where an integer overflow occurs, it traces all line numbers in the code that lead to the overflow condition. Software developers can use this information to determine how best to fix the code. Quality engineers can use this information to classify the defect for further action.

Proving Code with Polyspace Code Prover

Polyspace Code Prover uses a mature and sound formal methods technique known as abstract interpretation. This technique bridges the gap between conventional static analysis techniques and dynamic testing by verifying the dynamic properties of software applications at compilation time. Without executing the program itself, abstract interpretation investigates all possible behaviors of a program—that is, all possible combinations of inputs and all possible execution sequences—in a single pass to determine how and under which conditions the program can fail.

How Abstract Interpretation Works

Abstract interpretation relies on a broad base of mathematical theorems that define rules for analyzing complex dynamic systems such as software applications. Instead of proceeding with the enumerative analysis of each state of a program, abstract interpretation represents these states in a more general form and provides the rules to manipulate them. Abstract interpretation not only produces a mathematical abstraction, it also interprets the abstraction.

To produce a mathematical abstraction of program states, abstract interpretation thoroughly analyzes all variables of the code. The substantial computing power required for this analysis has not been readily available in the past. Abstract interpretation, when combined with non-exponential algorithms and today's increased processing power, is a practical solution to complex testing challenges.

When applied to the detection of run-time errors, abstract interpretation performs a comprehensive verification of all risky operations and automatically diagnoses each operation as proven, failed, unreachable, or unproven. Engineers can use abstract interpretation to obtain results at compilation time, the earliest stage of the testing. See Appendix B for more information on applying abstract interpretation and Appendix C for sample code.

Benefits of Abstract Interpretation to Prove Code

Abstract interpretation is an efficient, cost-effective way to ensure delivery of reliable embedded systems. This proof-based capability provides four benefits: assurance of code reliability, increased efficiency, reduced overhead, and simplified debugging.

Assurance of Code Reliability

Through its exhaustive code review, abstract interpretation not only enables run-time error detection, but also proves code correctness. This is especially important in safety-critical applications in which system failures can be catastrophic. Traditional debugging tools are tuned to detect errors but do not verify the robustness of the remaining code. Abstract interpretation makes it possible to identify code that will never cause a software fault, thereby removing any uncertainty about the software's reliability.

Increased Efficiency

By verifying the dynamics of applications, abstract interpretation lets embedded developers and testers identify the code sections in their program that are free of run-time errors from the ones that will lead to a reliability breach. Because errors can be identified before code is compiled, abstract interpretation helps teams realize substantial time and cost savings by finding and eliminating run-time errors when they are easiest to fix.

Reduced Overhead

Abstract interpretation requires no execution of the software, so it produces thorough results without the overhead of writing and executing test cases. There is no need to instrument code and then strip out the instrumentation before shipping software. And abstract interpretation can be implemented in ongoing projects without changing existing development processes.

Simplified Debugging

Abstract interpretation streamlines debugging because it directly identifies the source of each error, not just its symptom. It eliminates time wasted in tracking crashes and data corruption errors back to their source, as well as the time previously spent trying to reproduce sporadic bugs. Abstract interpretation is repeatable and exhaustive. Each operation in the code is automatically identified, analyzed, and checked against all possible combinations of input values.

Conclusion

Polyspace Code Prover uses abstract interpretation for static analysis to verify code. Along with Polyspace Bug Finder, these products offer an end-to-end software verification capability for early stage development use, spanning bug-finding, coding rules checking, and proof of run-time errors. This capability ensures the reliability of embedded software that must operate at the highest levels of quality and safety.

Appendix A: An Analogy from the Physical World

An engineer who needs to predict the trajectory of a projectile in midair has three options:

1. Make an exhaustive inventory of the different particles that will be on the projectile's path, study their properties, and determine how impact with each particle will affect its trajectory. This approach is impractical due to the huge number and variety of particles encountered during flight. Even if it were possible to know in advance all the conditions that could be encountered at any time, such as wind speed and cloud drops, these would change for every new flight. This means a thorough analysis would need to be run again before every launch.
2. Launch many projectiles to derive empirical laws of motion and related error margins. These findings can be used to estimate the trajectories within a certain confidence interval. This approach is both costly and time-consuming, however, and each attempt will change the conclusions. Furthermore, exhaustively testing the projectile under every possible combination of conditions is all but impossible.
3. Use the laws of physics and known values (force of gravity, air braking coefficient, initial speed, and so on) to transform the problem into a set of equations that may be solved by mathematical rules, either formal or numeric. This approach produces solutions for a wide range of conditions that become parameters in the mathematical model, enabling the engineer to predict projectile behavior under a variety of conditions.

Abstract interpretation is like the mathematical modeling approach. It derives the dynamic properties of data from the software source code—that is, equations between variables—and applies them to the verification of specific dynamic properties.

Appendix B: Applying Abstract Interpretation

To better understand how abstract interpretation works, consider a program, P, that uses two variables, X and Y. It performs the operation:

X=X/(X-Y)

To check this program for run-time errors, we identify all possible causes for error on the operation:

- X and Y may not be initialized
- X-Y may overflow or underflow
- X and Y may be equal and cause a division by zero
- X/(X–Y) may overflow or underflow

While any of these conditions could cause a run-time error, the following steps focus on the possibility of division by zero.

We can represent all possible values of X and Y in program P on a diagram. The red line in Figure 1 represents the set of (X, Y) values that would lead to a division by zero.

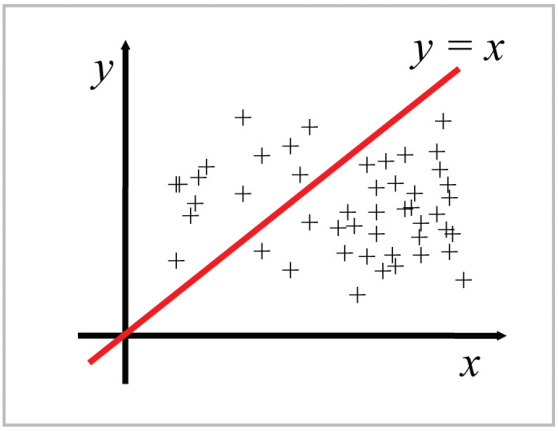

Figure 1. All possible values of X and Y in program P.

An obvious way to check for division by zero would be to enumerate each state and determine whether it is on the red line. This is the approach taken by conventional white-box testing, but it has fundamental limitations. First, the number of possible states in a real-world application is typically very large because there are many variables in use. It would take years to enumerate all possible states, making an exhaustive check all but impossible.

In contrast to the brute force approach of enumerating all possible values, abstract interpretation establishes generalized rules to manipulate the whole set of states. It builds an abstraction of the program that can be used to prove properties.

One example of such an abstraction, called type analysis, is shown in Figure 2. This type of abstraction is used by compilers, linkers, and basic static analyzers by applying type inference rules. In type analysis, we project the set of points on the X and Y axes, get the minimum and maximum values for X and Y, and draw the corresponding rectangle. Since the rectangle includes all possible values of X and Y, if a property is proven for the rectangle, it will be valid for the program. In this case, we are interested in the intersection between the red line and the rectangle, because if the intersection is empty, there will never be a division by zero.

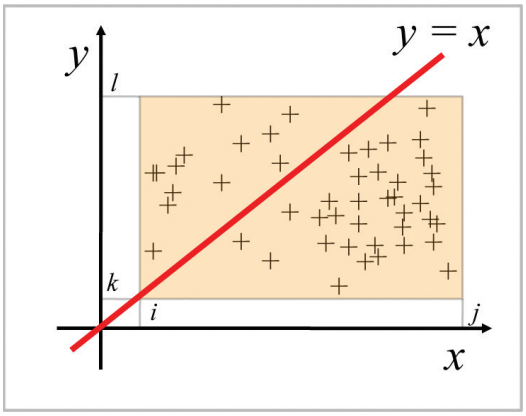

Figure 2. Type analysis projects all values of X and Y onto the axes.

The key drawback of type analysis is that the rectangle includes too many unrealistic values of X and Y. This yields poor, imprecise results and generates a large number of warning messages that often go unread, if the engineer does not switch them off altogether.

Instead of large rectangles, abstract interpretation establishes rules to build more precise shapes, as in Figure 3. It uses techniques based on integer lattices, union of polyhedra, and Groebner bases to represent relationships between data (X and Y) that take into account control structures (such as if-then-else, for-loops and while-loops, and switch), inter-procedural operations (function calls), and multitask analysis.

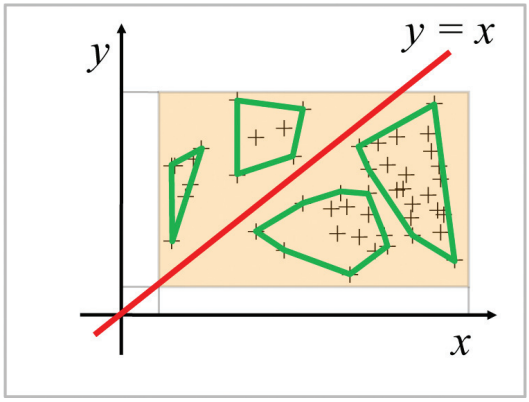

Figure 3. Abstract interpretation establishes rules to build more precise shapes representing relationships between X and Y.

Unlike compilers and other static analyzers, abstract interpretation does not rely solely on the idea of computing relationships between data types and constant values. Instead, it derives these relationships from the semantics of each operation and operand in the program, and uses them as guides to inspect the source code.

With abstract interpretation the following program elements are interpreted in new ways:

- An index in a for-loop is no longer an integer but a monotonic increasing discrete function from lower to upper limit.
- A parameter passed to a function is no longer a variable or a constant, but a set of values that may be used to constrain the local data used in the function.
- Any global shared data may change at any time in a multitask program, except when protection mechanisms, such as memory locks or critical sections, have been applied
- A pointer is a data type that may create aliases between explicit data and generate side effects and hidden concurrent accesses on shared data in multitasking applications.
- A variable not only has a type and a range of values, but also a set of equations, including control flow sensitive relationships that build it.
- Ultimately, run-time errors are equations, also called correctness conditions that abstract interpretation can solve, using the equations that tie variables together.

Appendix C: Examples of Abstract Interpretation

The following examples are code constructs that produce run-time errors. Polyspace Code Prover is capable of detecting these run-time errors by abstract interpretation.

Control Structure Analysis: Out-of-bounds pointer dereferencing after a for-loop

```
10: int ar[100];
11: int *p = ar;
12: int i;
13: for (i = 0; i < 100; i++; p++)
14:     { *p = 0;}
15: *p = 5;
```

In this example, p is a pointer that can be abstracted as a discrete increasing function varying by 1 from the beginning of the array ar. Upon exiting the for-loop when i equals 100, the pointer p is also increased to 100. This cause pointer p to be out-of- bounds at line 15, as the array index ranges from 0 to 99. Abstract interpretation would prove this piece of code reliable and would identify line 15 as the source of a run-time error.

Control Structure Analysis: Out-of-bounds array access within two nested for-loops

```
20: int ar[10];
21: int i,j;
22: for (i = 0; i < 10; i++)
23: {
24:     for (j = 0; j < 10; j++)
25:     {
26:             ar[i - j] = i + j;
27:     }
28: }
```

Both i and j are variables that are monotonically increasing by 1 from 0 to 9.

The operation i–j used as an index for array ar will eventually return a negative value. Abstract interpretation of this code would prove this code reliable and would identify the out-of-bounds array access at line 26.

Note that the run-time errors in these examples often lead to corruption of the data stored near array ar. Depending on how and when this corrupted data is used elsewhere in the program, debugging this kind of error without abstract interpretation can take considerable effort.

Inter-procedural analysis: Division by zero

```
30: void foo (int* depth)
31: {
32:     float advance;
33:     *depth = *depth + 1;
34:     advance = 1.0/(float)(*depth);
35:     if (*depth < 50)
36:             foo (depth);
37: }
38:
39: void bug _ in _ recursive ()
40: {
41:     int x;
42:     if (random _ int())
43: {
44:     x = -4;
45:     foo ( &x );
46: }
47:     else
48:     {
49:             x = 10;
50:             foo ( &x );
51:     }
52: }
```

In this function, depth is an integer that is first increased by 1. It is then used as a denominator to determine the value of advance and thereafter is recursively passed to the function foo. Checking whether the division operation at line 34 will cause a division by zero requires an interprocedural analysis to determine which values will be passed to the function foo (see bug_in_recursive, lines 45 and 50), as it will determine the value of 'depth.

In the preceding code, the function foo can be called in two different circumstances in the function bug_in_recursive. When the if statement at line 42 is false, foo is called with a value of 10 (line 50).

Therefore, *depth becomes a monotonic discrete increasing function varying by 1 from 11 to 49. The equation at line 34 will not return a division by zero.

However, when the if statement at line 42 is true, then foo is called with a value of -4. Therefore, *depth becomes a monotonic discrete increasing function varying by 1 from -3 to 49. Eventually, *depth will be equal to 0, causing the equation at line 34 to return a division by zero.

A simple syntax check will not detect this error, nor will all test cases. Abstract interpretation will prove all the code reliable except line 45. This illustrates the unique ability of abstract interpretation to perform inter-procedural analysis and distinguish problematic function calls from acceptable ones. If not fixed, a division by zero error will cause processor halt. This kind of error can also require significant debugging time due to the recursive constructs in use.

Multitask analysis: Concurrent access to shared data

Abstract interpretation handles control and data flow analysis and is capable of checking multitasking applications. A key concern with such applications is ensuring that shared data and critical resources have no unexplained concurrent access. Data aliasing and task interleaving make it difficult to find this type of concurrent access problem.

With data aliasing, pointers are used for shared memory access. This approach can create hidden or implicit relationships between variables, so that one variable may be unexpectedly modified during program execution through pointers, causing sporadic anomalies. The analysis of this problem requires very efficient pointer analysis algorithms. With abstract interpretation these can be implemented to provide a list of shared data and the list of read and write accesses by functions and tasks along with the related concurrent access graph.

Tasks interleaving makes multitask applications problematic to debug because bugs are very difficult to reproduce when they depend on a sequence of tasks being executed in a specific order in real time. Abstract interpretation takes every possible interleaving of tasks into account, for a complete control-flow analysis. Constructing these results or the concurrent access graph by hand would be exceptionally difficult.

91517v02 09/13