Sound Verification Techniques for Developing High-Integrity Medical Device Software

Jay Abraham
The MathWorks

Paul Jones
FDA / CDRH

Raoul Jetley
FDA / CDRH

Abstract

Embedded software in medical devices is increasing in content and complexity. Traditional software verification and testing methods may not provide the optimum solution. This paper discusses the application of sound verification techniques in the development of high integrity medical device software. Specifically, this paper will explore the application of formal methods based *Abstract Interpretation* techniques to mathematically prove the absence of a defined set of run-time errors. The verification solution is then compared and contrasted to other software analysis and testing methods, such as code review, static analysis and dynamic testing.

Introduction

The sophistication and complexity of embedded software contained within medical devices is increasing. State of the art pacemakers may contain up to 80,000 lines of code, while infusion pumps may have over 170,000 lines of code. These devices must operate with utmost concern for safety and reliability [1].

Historically, the typical options available for verifying medical device software have been code reviews, static analysis, and dynamic testing. Code reviews rely solely on the expertise of the reviewer and may not be efficient for large code bases. Traditional static analysis techniques rely mainly on a pattern-matching approach to detect unsafe code patterns, but cannot prove the absence of run-time errors. Lastly, with the increasing complexity of device software, it is virtually impossible to dynamically test for all types of operating conditions.

Medical Device Software

Medical devices address a continuum of diagnosis and treatment applications varying in complexity from digital thermometers, insulin pumps, pacemakers, cardiac monitors, to anesthesia machines, large ultrasound imaging systems, chemistry analyzers and proton beam therapy systems. As science and technology advances, so does the complexity and capability of next generation devices. The easily identifiable signature of these devices is that they are intended to directly or indirectly affect the public health.

Software is becoming ubiquitous in medical devices and contributing to their complexity [5]. As a result, various worldwide regulatory organizations include language in their directives to explicitly address device software [2]. Recent studies are pointing to increasing failure rates of medical devices due to software coding errors. A study conducted by Bliznakov et al finds that over the period of 1999-2005 11.3% of FDA initiated recalls were attributable to software coding errors [6].

The FDA increased its involvement in reviewing the development of medical device software in the mid 1980s when software coding errors in a radiation therapy device contributed to the lethal overdose of a number of patients [2]. FDA has published several guidance documents and recognized several standards addressing good software development processes. More recently, the FDA has established a software laboratory within the Center for Devices and Radiological Health (CDRH) Office of Science and Engineering Laboratories (OSEL) to identify software coding errors in devices under recall investigation [1]. As part of its investigation the software laboratory examines the source code to understand and identify the root cause of a device failure.

Examination of the software verification processes in other industries may be instructive in the development of high integrity embedded software. For example, significant rigor is applied to the verification of software for aerospace and automotive applications [3]. These

© 2009

industries have adopted various analysis and verification tools to improve the quality of software. The FAA (Federal Aviation Administration) requires certification of software to the DO-178B (Software Considerations in Airborne Systems and Equipment Certification) standard as part of an aircraft air worthiness certification process. The standard establishes levels of flight failure condition categories such that the greater the severity of a failure condition category the greater the rigor in design process, verification, and assurance required. The automotive industry adheres to software standards and often complies with the IEC-61508 international standard (specifying the "Functional Safety of Electrical, Electronic, and Programmable Electronic Safety-Related Systems").

Embedded Device Failures

A study of embedded device failures can be instructive to help avoid repeating mistakes of the past. We cite three examples of device failures illustrated by Ganssle [11].

1. Therac 25 – A radiation therapy device that overdosed patients with radiation. The real-time operating system (RTOS) for the device did not support a message passing scheme for threads. Therefore global variables were used instead. Insufficient protection of the global variables resulted in incorrect data being used during device operation. A second issue was an overflow condition on an 8 bit integer counter. These software coding errors resulted in overdosing patients with as much as 30 times the prescribed dose.

2. Ariane 5 – The very first launch of this rocket resulted in the destruction of the launcher with a loss of the payload. The cause was an overflow in a pair of redundant Inertial Reference Systems which determined the rocket's attitude and position. The overflow was caused by converting a 64 bit floating point number to a 16 bit integer. The presence of a redundant system did not help, because the backup system implemented the same behavior.

3. Space Shuttle Simulator – During an abort procedure all four main shuttle computers crashed. Examination of the code identified a problem in the fuel management software where counters were not correctly re-initialized after the first of a series of fuel dumps were initiated. The result was that the code would jump to random sections of memory causing the computers to crash.

For further specific examples of medical device issues, the FDA's CDRH Manufacturer and User Facility Device Experience Database (MAUDE) [21] catalogs anomalous behavior and adverse events in an easy to find manner. Searching the database for the keyword "firmware" yields a number of medical devices that the FDA is tracking with respect to reported adverse events.

Causes of Device Failures

Code defects, resulting in run-time errors are the major cause for embedded device failures described above. Run-time errors are a specific class of software errors considered as latent faults. The faults may exist in code, but unless very specific tests are run under particular conditions, the faults may not be realized in the system. Therefore, the code may appear to function normally, but may result in unexpected system failures, sometimes with fatal consequences. A few causes of run-time errors are given below (this list is not exhaustive):

1. Non-initialized data – If variables are not initialized, they may be set to an unknown value. When using these variables the code may sometimes function as desired, but under certain conditions the outcome becomes unpredictable.

2. Out of bounds array access – An out of bounds array access occurs when data is written or read beyond the boundary of allocated memory. This error may result in code vulnerabilities and unpredictable results during execution.

3. Null pointer dereference – A Null pointer dereference occurs when attempting to reference memory with a pointer that is NULL. Since this a "non-value" for memory, any dereference of that pointer leads to an immediate system crash.

4. Incorrect computation – This error is caused by an arithmetic error due to an overflow, underflow, divide by zero, or when taking a square root of a negative number. An incorrect computation error can result in a program crash or erroneous result.

© 2009

5. Concurrent access to shared data – This error is caused by two or more variables across different threads try to access the same memory location. This could lead to a potential race condition and may result in data corruption as a result of multiple threads accessing shared data in an unprotected fashion.

6. Illegal type conversions – Illegal type conversions may result in corruption of data and produce unintended consequences.

7. Dead code – Although dead code (i.e. code that will never execute) may not directly cause a run-time failure, it may be important to understand why the programmer wrote such code. Dead code may also signal poor coding practices or lost design specifications.

8. Non-terminating loops – These errors are caused by incorrect guard conditions on program loop operations (e.g. for, while, etc.) and may result in a system hangs or halts.

Software Verification and Testing

Traditional software verification and testing consists of code reviews and dynamic testing. In [15] Fagan discusses how code inspections and reviews can reduce coding errors. Experience has shown that while this can be a relatively effective way of verifying code, the process needs to be complemented with other methods. One such method is static analysis. This is a somewhat new method that largely automates the software verification process [16]. This technique attempts to identify errors in the code; but does not necessarily prove their absence. The next sections discuss these methods and introduces the application of formal methods based Abstract Interpretation to code verification.

Code Reviews

Fagan discusses the process of code reviews and inspections quite extensively in [15]. The process includes a description of the team that will perform the review (moderator, designer, coder, and tester), the preparation process (which involves the creation of a checklist), and the inspection itself. The stated objective is to find errors in code. Processes on conclusion of the review are also described. These include rework to address errors found and follow-up to ensure that issues and concerns raised during inspection are resolved. Another aspect of code reviews can involve checking compliance to certain coding standards such as MISRA-C or JSF++ (for C and C++).

Detecting subtle run-time errors can be a difficult proposition. For example, an overflow or underflow due to complex mathematical operations that involve programmatic control could easily be missed during a code review.

Dynamic Testing

Dynamic testing verifies the execution flow of software; e.g. decision paths, inputs and outputs. Wagner describes the methodology and explains the application of this testing philosophy according to the dimensions of type (functional and structural) and granularity (unit, integration, and system) [16]. Dynamic testing involves the creation of test-cases and test-vectors and execution of the software against these tests. Comparison of the results to expected or known correct behavior of the software is then performed. Wagner also includes a summary of various statistics compiled on the effectiveness of dynamic testing. His analysis shows that the mean effectiveness of dynamic testing is only about 47%. In other words, over half of potential errors on average are not detected with dynamic testing.

Hailpern in [9] and Dijkstra in [10] sum up the challenges of software testing as *"given that we cannot really show there are no more errors in the program, when do we stop testing?"* [9] and *"program testing can be used to show the presence of bugs, but never to show their absence"* [10]. Butler and Finnelli further explain that life testing of ultra-reliability software is not feasible. Life testing is referred to as actual testing of the device under all possible conditions For example, to quantify 10^{-8}/hour failure rate will require more than 10^{8} hours of testing [8].

Static Analysis

Static analysis is a technique used to identify potential and actual defects in source code. The static analysis process utilizes heuristics and statistics and does not require code execution or the development of test-cases. The types of errors found could be thought of as strong compiler type checks (e.g. checking if variables are always initialized or used) to more sophisticated dataflow based analysis.

© 2009

As described in [16] and [17], these tools can certainly find errors in code, but there is a high false positive rate. The term false positive refers to the identification of an error that is not real. The time and energy spent on tracking a false positive can lead to frustration on the part of software engineers [17]. Wagner [16] presents a summary of findings with respect to false positive rates. The average number of false positives detected in some static analysis tools is 66%.

In addition to false positives, it is also important to understand false negatives. A false negative occurs when the static analysis tool fails to identify an error [18]. In [17] there is an extensive discussion of false negatives concluding that decreasing the probability of false negatives will increase the probability of false positives. The use of static analysis can provide a certain amount of automation in the verification process, but this advantage must be weighed carefully against the capability of these tools to generate false positives and false negatives.

Formal Methods

The term *formal methods* has typically been applied to proof based verification of a system with respect to its specification. This term can also be applied to a mathematical rigorous approach of proving correctness of code [13]. This approach may help reduce false negatives, i.e. the inability to conclusively state that the code is free of certain types of run-time errors. The next section describes the use of Abstraction Interpretation as a formal methods based verification solution that can be applied to software programs.

Abstract Interpretation

The explanation of Abstract Interpretation is best accomplished by studying a simple example. Consider the multiplication of three large integers:

$$-4586 \times 34985 \times 2389 = ?$$

For the mathematical problem defined it is difficult to quickly compute the final value by hand. However, if we abstract the result of the computation to the sign domain (i.e., either positive or negative), it is easy to understand that the sign of the computation will be negative. Determining the sign for this mathematical computation is an application of Abstract Interpretation. The technique enables us to know precisely some properties of the final result, in this example, the sign, without having to multiply the integers fully. We also know that the sign will never be positive for this computation. In fact Abstract Interpretation will prove that the sign of the operation will always be negative and never positive.

Let us now consider a simplified application of the formal mathematics of Abstract Interpretation to software programs. The semantics of a programming language is represented by the concrete domain *S*. Let *A* represent the abstraction of the semantics. The abstraction function α maps from the concrete domain to the abstract domain. The concretization function γ maps from the abstract domain *A* to the concrete domain *S*. α and γ form a Galois connection and are monotonic [22]. Certain proof properties of the software can be performed on the abstract domain *A*. It is a simpler problem to perform the proof on the abstract domain *A* versus the concrete domain *S*.

The concept of soundness is important in context of a discussion on Abstract Interpretation. Soundness implies that when assertions are made about a property, those assertions are proven to be correct. The results from Abstract Interpretation are considered sound because it can be mathematically proven with structural induction that abstraction will predict the correct outcome. When applied to software programs, Abstract Interpretation can be used to prove certain properties of software, e.g., to prove that the software will not exhibit certain run-time errors [20].

Cousot and Cousot [12, 13] describe the application and success of Abstract Interpretation to static program analysis. Deutsch describes the application of this technique to a commercially available solution in [19]. The application of Abstract Interpretation involves computing approximate semantics of the software code with the abstraction function α such that it can be verified in the abstract domain. This produces equations or constraints whose solution is a computer representation of the program's abstract semantics.

© 2009

Lattices are used to represent variable values. For the sign example described earlier, the lattice shown in Fig 1 can be used to propagate abstract values in a program (starting at the bottom and working to the top). Arriving at any given node in the lattice proves a certain property. Arriving at the top of the lattice indicates that a certain property is unproven and is indicative that under some conditions the property is proven correct and other conditions proven incorrect.

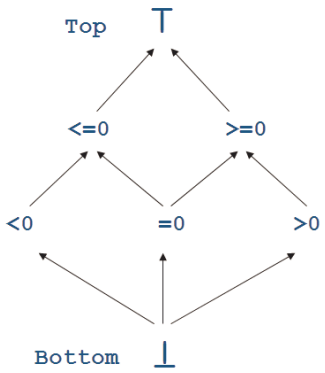

Fig 1: Lattice representation of variables

Over approximation is applied to all possible execution paths in a program. Techniques such as widening and narrowing [22] and iteration with a solver are used to solve the equations and constraints to prove the existence of or the absence of run-time errors in source code.

Using Abstract Interpretation for Code Verification

The identification and proving the absence of dynamic errors such as run-time errors that can occur during code execution can be accomplished by defining a strongest global invariant *SGI(k)*. Where *SGI(k)* is the set of all possible states that are reachable at point *k* in a program *P*. A run-time error is triggered when *SGI(k)* intersects a forbidden zone. *SGI(k)* is the result of formal proof methods and can be expressed as least fixed-points of a monotonic operator on the lattice of a set of states [19].

To see the application of Abstract Interpretation to code verification, consider the following operation in code:

$$X = X/(X-Y)$$

Several issues could cause a potential run-time error. These run-time errors include:

1. Variables are not initialized.
2. An overflow or underflow on the subtraction operation *(X-Y)*.
3. An overflow or underflow on the division operation.
4. If *X* is equal to *Y*, then a divide by zero will occur.
5. The assignment to *X* could result in an overflow or underflow.

Let us examine condition #4 (divide by zero). Plotting *X* and *Y* as shown in Fig 2, one can see that the 45° line comprising of *X=Y* would result in a run-time error. The scatter plot shows all possible values of *X* and *Y* when the program executes this line of code (designated with +).

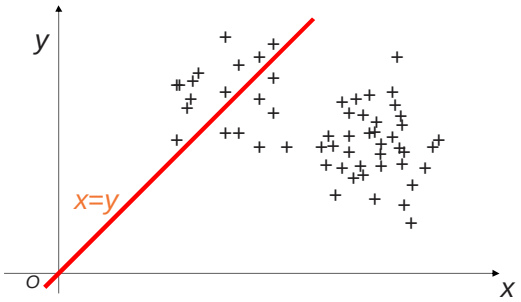

Fig 2: Plot of data for X and Y

Dynamic testing would utilize enumeration over various combinations of *X* and *Y* to determine if there will be a failure. However, given the large number of tests that would have to be run, this type of testing may not detect or prove the absence of the divide by zero run-time error.

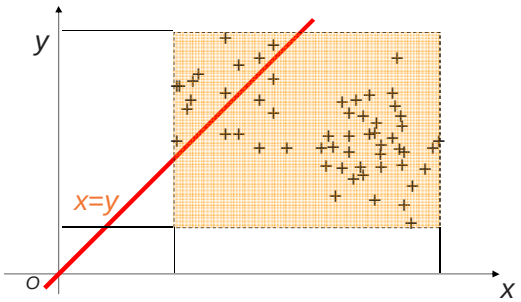

Fig 3: Type analysis

Another methodology would be to apply type analysis to examine the range of *X* and *Y* in context of the run-time error condition (i.e. *X=Y*). In Fig 3, note the bounding box created by type analysis. If the bounding box intersects *X=Y*,

© 2009

then there is a potential for failure. Some static analysis tools apply this technique. However, type analysis in this case is too pessimistic, since it includes unrealistic values for *X* and *Y*.

With Abstract Interpretation, a more accurate representation of the data ranges of *X* and *Y* are created. Since various programming constructs could influence the values of *X* and *Y* (e.g., pointer arithmetic, loops, if-then-else, multi-tasking, etc.) an abstract lattice is defined [19]. A simplified representation of this concept is to consider the grouping of the data as polyhedron as shown in Fig 4. Since the polyhedron does not intersect *X=Y* we can conclusively say that a division by zero will not occur.

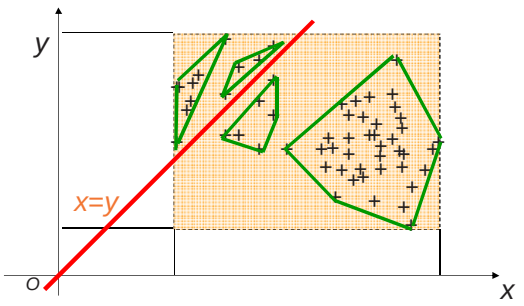

Fig 4: Abstract interpretation

As described earlier, Abstract Interpretation is a sound verification technique. With Abstract Interpretation, assertions regarding the specified run-time aspects of the software are proven to be correct.

A Code Verifier based on Abstract Interpretation

The Abstract Interpretation concept can be generalized as a tool set that can be used to detect a wide range of run-time errors in software. In this paper, we use PolySpace® as an example to explain how such a tool can help detect and prove the absence of run-time errors such as overflows, divide by zero, out of bound array access, etc. PolySpace is a code verification product from The MathWorks® that utilizes Abstract Interpretation. The input to PolySpace is C, C++, or Ada source code. The output is source code painted in four colors. As described in Table 1 and as shown in Fig 5 PolySpace informs the user of the quality of the code using this four color coded scheme.

Green	*Indicates code is reliable*
Red	*Run-time error due to faulty code*
Gray	*Shows dead or unreachable code*
Orange	*Unproven code*

Table 1: Explanation of colors used in PolySpace

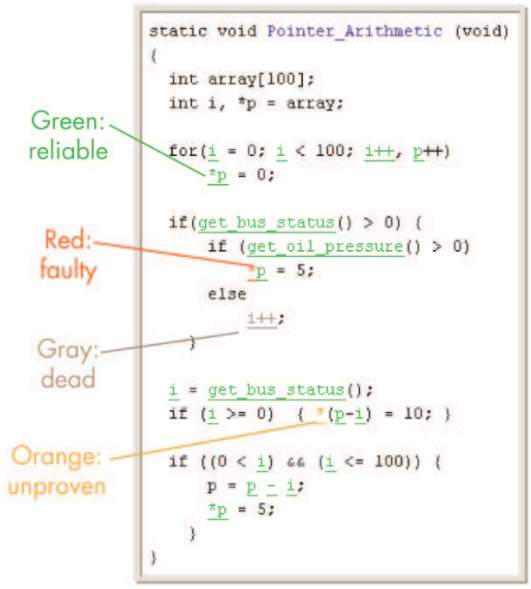

Figure 5: PolySpace color scheme

Demonstrating the effective of application of Abstract Interpretation using PolySpace is explained with the following exercise. Consider the function:

```
1       int where_are_errors(int input)
2       {
3       int x, y, k;
4
5       k = input / 100;
6       x = 2;
7       y = k + 5;
8       while (x < 10)
9               {
10              x++;
11              y = y + 3;
12              }
13
14      if ((3*k + 100) > 43)
15              {
16              y++;
17              x = x / (x - y);
18              }
19
20      return x;
21      }
```

The goal is to identify run time errors in the function *where_are_errors()*. The function performs various mathematical computations, includes a *while* loop, and an *if* statement. Note that all variables are initialized and used. On line 17 a potential divide by zero could occur if *x=y*.

© 2009

Given the control structures and mathematical operations on *x* and *y*, could *x=y*?

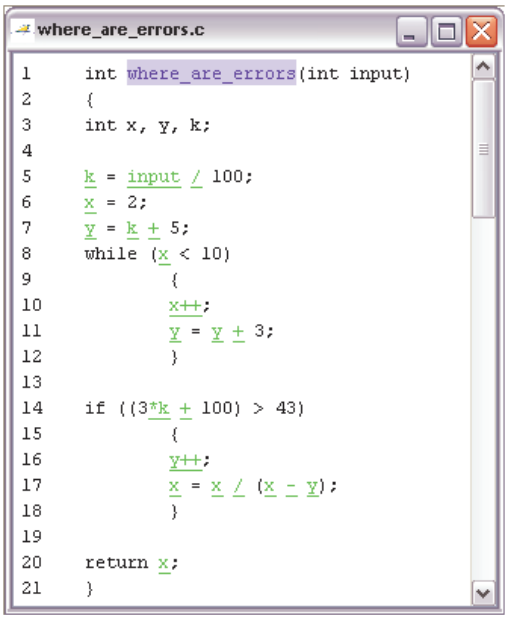

Fig 6: PolySpace results on safe code

As shown in Fig 6, PolySpace has proven there are no run-time errors in this code. This is because line 17 is executed only when the condition *(3*k + 100 > 43)* evaluates to true. Moreover, since the value of y is dependent on that of *k*, PolySpace determines that at line 17, while *x* is equal to *10*, *y* will always have a value greater than *10*. As a result, there cannot be a divide-by-zero error at this line.

This result is determined efficiently without the need to execute code, write test-cases, add instrumentation in source code, or debug the code. PolySpace also identifies all aspects of code that could potentially have a run-time error. These are underlined (see Fig 6). For this example, since there are no run-time errors, the underlined code is painted in green. For example, on line 17 the underlined green on the division '/' operator proves safety for overflow as well as division by zero.

Code Verification and Medical Device Software

The use of Abstract Interpretation based code verification to ensure high quality and reliability of critical software is not new. Brat and Klemm document the use of PolySpace for mission critical code used on the Mars Exploration Rover's flight software [14]. Best practices applied in industries such as aerospace can be easily applied to software in medical devices as well [1]. And not surprisingly, the FDA's software laboratory (housed within the Office of Science and Engineering Laboratories) uses code verification tools to examine source code in medical devices in which the embedded software may have caused adverse effects [1]. This article also states that these tools have helped find issues in the code as well as help clear the software of faults in product recall situations.

Applying this verification methodology to medical device software has several advantages. Using tools based on this technique, software development and quality teams are able to efficiently show that their code is run-time error free (within the degree of soundness of the tools used). The next section describes some additional run-time error conditions that can be determined using Abstract Interpretation. For purpose of illustration, the verification was performed using PolySpace and the results displayed are as they would be with PolySpace.

Example: Out of Bounds Pointer Dereferencing

```
1       void out_of_bounds_pointer(void)
2       {
3
4       int ar[100];
5       int *p = ar;
6       int i;
7
8       for (i = 0; i < 100; i++)
9               {
10              *p = 0;
11              p++;
12              }
13
14      *p = 5;
15
16      }
```

In the example code shown, note that the array *ar* has been allocated for 100 elements. Thru pointer aliasing, the pointer *p* points to the first element of array *ar*. The *for* loop with counter *i* will increment from 0 to 99. The pointer *p* is also incremented. On exit of the *for* loop the index will point to element 100 of the array *ar*. Attempting to store data at this location will cause a run-time error.

By using Abstract Interpretation, a lattice containing the variable for pointer *p* is created. Then by iteration and solving, it can be proven if it is possible to exceed the allocated range for *p*

© 2009

such that an out of bound array access occurs. The graphic in Fig 7 shows the PolySpace code verification results. Notice the green underlined statements (line 8, 10, 11, and partially on 14). These are PolySpace checks indicating that no run-time failure will occur at these sections of code. Also notice that PolySpace has identified line 14 where data written to memory with pointer *p* is a coding error (colored in red) which will result in a run-time error.

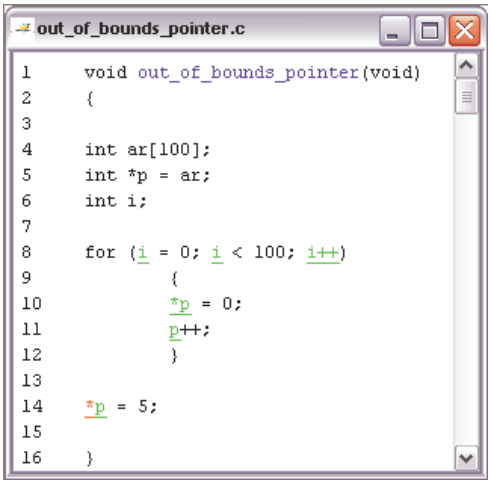

Fig 7: PolySpace results for out of bounds pointer

Example: Inter-procedural Calls

```
1       void comp (int* d)
2       {
3
4       float advance;
5       *d = *d + 1;
6       advance = 1.0/(float)(*d);
7
8       if (*d < 50)
9       comp (d);
10
11      }
12
13      void bug_in_recursive (void)
14      {
15      int x;
16
17      x = 10;
18      comp ( &x );
19
20      x = -4;
21      comp ( &x );
22      }
```

In the example above, *d* is an integer in function *comp()* that is incremented by 1. It is then used as the denominator to determine the value of the variable *advance* and after that it is recursively passed to the same function. Checking whether the division operation at line 6 will cause a division by zero requires an inter-procedural verification to determine which values will be passed to the function *comp()*.

In the code example, two values are passed to the function *comp()*. When called with 10, **d* becomes a monotonic discrete variable increasing from 11 to 49. Line 6 will not result in a division by zero. However, when *comp()* is called with a value of -4, **d* increases from -3 to 49. Eventually, **d* will be equal to 0, causing line 6 to return a division by zero.

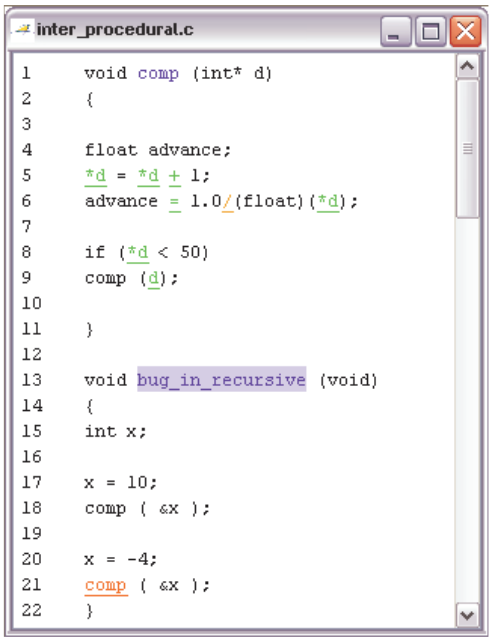

Fig 8: PolySpace results on inter-procedural code

A simple syntax check will not detect this run-time error. Abstract Interpretation with PolySpace as shown in Fig 8 proves that all code is free of run-time errors except at line 6 and 21. Note that at line 18, the function call to *comp()* succeeds, but fails on line 21. The division by zero is reflected with an orange color, because when *comp()* is called on line 18, there is no division by zero, but a division by zero will occur with the call on line 21. This example illustrates the unique ability of Abstract Interpretation to perform inter-procedural analysis with pointer aliasing to distinguish problematic function calls from acceptable ones.

Improving the Reliability of Medical Devices

Abstract Interpretation based code verification is a useful tool that can be applied to improve the

© 2009

quality and reliability of embedded software in medical devices. By employing an exhaustive mathematical proof based process, Abstract Interpretation not only detects run-time errors, but also proves the absence of these run-time errors in source code. Traditional verification tools are tuned to find bugs and do not verify the reliability of the remaining code. Abstract Interpretation based code verification makes it possible to identify software code that will or will not cause a software fault.

By using code verification tools early in the development phase, software teams may realize substantial time and cost savings by finding and eliminating run-time errors when they are easiest and cheapest to fix. Since Abstract Interpretation operates at the source code level, it does not require execution of the software to determine specified coding errors. Finally, Abstract Interpretation streamlines the debugging process by identifying the source of run-time errors directly in source code, not just the symptom. The solution eliminates time wasted tracing system crashes and data corruption errors back to source, as well as time spent attempting to reproduce sporadic bugs.

In conclusion, a code verification solution that includes Abstract Interpretation can be instrumental in assuring software safety and a good quality process. It is a sound verification process that enables the achievement of high integrity in embedded devices. Regulatory bodies like the FDA and some segments of industry recognize the value of sound verification principles and are using tools based on these principles.

Acknowledgements

The authors are thankful to Brett Murphy, Gael Mulat, Jeff Chapple, Marc Lalo, Parasar Kodati, Patrick Munier, Paul Barnard, and Tony Lennon of The MathWorks for their help in developing and reviewing this paper.

References

1. Taft, “CDRH Software Forensics Lab: Applying Rocket Science To Device Analysis”, The Gray Sheet, 2007

2. Fries, “Reliable Design of Medical Devices”, 2006

3. Knight, “Safety Critical Systems, Challenges and Directions”, International Conference on Software Engineering, 2002

4. Ganssle, “When Disaster Strikes”, Embedded Systems Design, 2004

5. Lee, et al., “High-Confidence Medical Device Software and Systems,” IEEE Computer, 2006

6. Bliznakov, Mitalas, Pallikarakis, “Analysis and Classification of Medical Device Recalls”, World Congress on Medical Physics and Biomedical Engineering, 2006

7. Wallace, Kuhn, “Failure Modes in Medical Device Software”, International Journal of Reliability, Quality and Safety Engineering, 2001

8. Butler, Finelli “The Infeasibility of Quantifying the Reliability of Life-Critical Real Time Software”, IEEE Transactions on Software Engineering, 1993

9. Hailpern, Santhanam, “Software Debugging, Testing, and Verification”, IBM Systems Journal, 2002

10. Dijkstra, “Notes On Structured Programming”, 1972

11. Ganssle, “Learning From Disaster”, Embedded Systems Conference Boston, 2008

12. Cousot, “Abstract Interpretation: Theory and Practice”, International SPIN Workshop on Model Checking of Software, 2002

13. Cousot, “Abstract Interpretation Based Formal Methods and Future Challenges”, Informatics. 10 Years Back. 10 Years Ahead, 2001

14. Brat, Klemm , “Static Analysis of the Mars Exploration Rover Flight Software”, First International Space Mission Challenges for Information Technology, 2003

© 2009

15. Fagan, “Design and Code Inspections to Reduce Errors in Program Development”, IBM Systems Journal, 1976

16. Wagner, “A Literature Survey of the Software Quality Economics of Defect Detection Techniques”, ACM/IEEE International Symposium on Empirical Software Engineering, 2006

17. Engler et al, “Weird Things That Surprise Academics Trying to Commercialize a Static Checking Tool”, Static Analysis Summit, 2006

18. Chapman, “Language Design for Verification”, Static Analysis Summit, 2006

19. Deutsch, “Static Verification of Dynamic Properties, SIGDA, 2003

20. Cousot, “Abstract Interpretation”, ACM Computing Surveys, 1996

21. FDA, www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfMAUDE/TextSearch.cfm, 2009

22. Cousot and Cousot, “Comparing the Galois Connection and Widening / Narrowing Approaches to Abstract Interpretation”, Symposium on Programming Language Implementation and Logic Programming, 1992

© 2009