Building Confidence in the Quality and Reliability of Critical Software

Jay Abraham, MathWorks
Jon Friedman, MathWorks

Abstract. Software in critical civilian and military aerospace applications, including avionics and other systems in which quality and reliability are imperative, continues to become both more common and more complex. The embedded software development organizations that build these systems must meet stringent quality objectives that are mandated by their organizations or required by customers or governments. For engineering teams to meet these objectives, and to ideally deliver high quality software, state of the art testing and verification solutions are needed. This article examines formal methods based software verification and testing approaches that have been applied to critical software projects in civil and military aerospace and defense projects. Examples are provided to illustrate how these verification techniques can be deployed in practice to improve the quality and reliability of complex avionics systems.

1. The Components of Avionics Embedded Software

Avionics software implemented in critical aerospace applications consists of special purpose embedded software. This software often operates in real time and is responsible for critical operations. Examples include digital flight control systems, full authority digital engine control, guidance navigation control, and similar systems. The embedded software responsible for these systems will consist of multiple components, including automatically generated, handwritten, and third-party code as well as libraries (see Figure 1).

Generated code: Generated code is synthesized from models that are used to describe and analyze the behavior of complex systems and algorithms.

Handwritten code: Handwritten code may include interfaces to hardware (for example, driver software for a cockpit display system, airspeed sensor, or another hardware subsystem), or it may be translated manually from specification documents or models.

Third-party code: Third-party code may be delivered by suppliers or it may be required as part of larger software system (for example, to interface with the real-time operating system).

Libraries: Object code is part of the application code that exists as a library or as compiled legacy code. By definition this software is delivered or is only available in the form of object code (binary files).

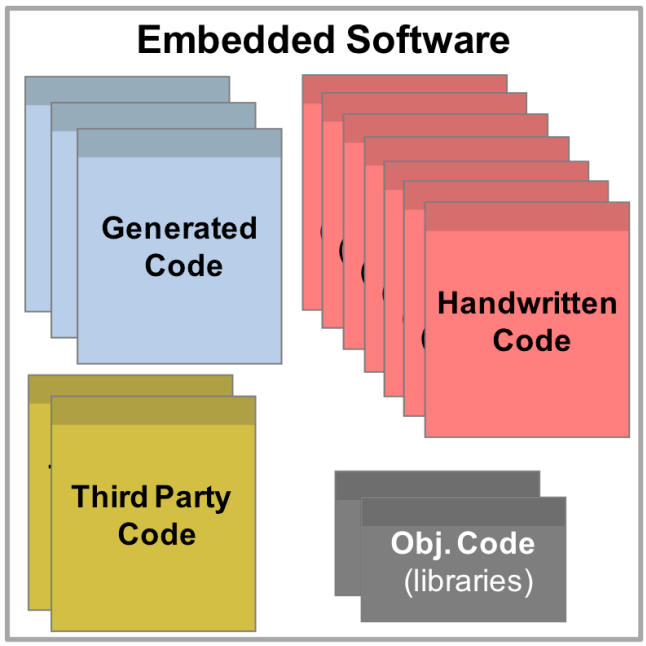

Figure 1: Components of embedded software.

2. Embedded Software Design, Implementation, and Verification

Typically, embedded software design starts by gathering system and software requirements. The code is then written or generated to implement the software. Verification processes focus on confirming the software requirements are implemented correctly and completely, and that they are traceable to the system requirements. The software must be tested and analyzed to ensure that it not only performs as required, but does not include any unintended operations. Additional tests are performed as the software is integrated with the hardware and validated at the system level. The design and verification process described above is often referred to as the V diagram process (see Figure 2).

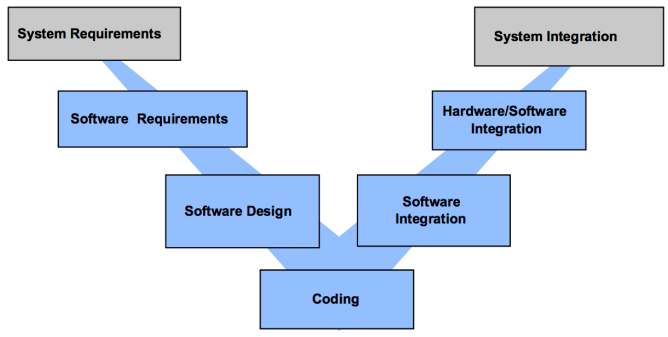

Figure 2: Embedded software design, implementation, and verification (V Diagram).

Even with robust verification processes, complex systems can fail. Causes of failure include insufficient specification, design errors, software coding errors or defects, and other issues unrelated to software. Ideally design and coding errors should be detected on the right-hand side of the V diagram during software, hardware, and aircraft testing and integration processes.

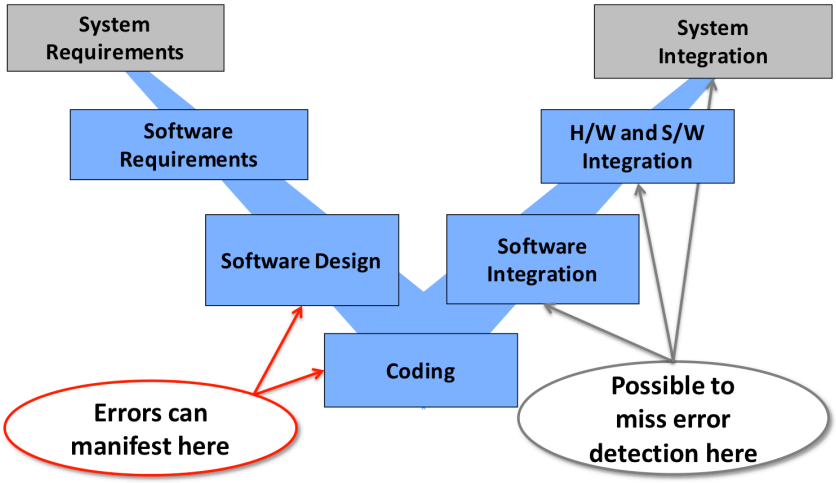

Figure 3: Errors often manifest in the design and coding phases.

Unfortunately, testing phases may fail to detect some errors unless exhaustive testing is employed. A study by the SEI found that for every 100 defects injected in the left-hand side of the V diagram, on average 21 latent defects remain in the system after the testing and verification processes were completed (see Figure 3) [1]. These bugs remain because exhaustive testing is generally not practical. Other techniques must be used to eliminate remaining defects; however, in a report on the use of static analysis to improve quality of code, the authors found that early detection of defects was important, but challenging to accomplish [2]. The difficulty was primarily due to human factors such as the inability to triage results from the tool to identify where code is safe and where it may fail.

A complete discussion on improving the quality of complex systems by addressing every failure point is beyond the scope of this article. Instead, this article focuses on two points: design errors and software coding errors. These errors will manifest in the software design and coding phases of the V diagram.

Examples of design errors include:

- Dead logic (for software combinatorial logic involving AND, OR, and NOT)
- Unreachable states or modes in state machines
- Deadlock conditions
- Nondeterministic behavior
- Overflow or divide-by-zero conditions in arithmetic operations

There are many different types of coding errors. This article covers those that are classified as run-time errors, that is, errors that only express themselves under particular conditions when the software is running. These errors are particularly troublesome because the code may appear to function normally under general test conditions, but may later cause unexpected system failures under other conditions. Some causes of run-time errors include:

- Uninitialized data. When variables are not initialized, they may be set to an unknown value.
- Out of bounds array access. This occurs when data is written or read beyond the boundary of allocated memory.
- Null pointer dereference. This occurs when attempting to reference memory with a pointer that is NULL.
- Incorrect computation. This is caused by an arithmetic error due to an overflow, underflow, or divide-by-zero operation, or when taking a square root of a negative number.
- Concurrent access to shared data. This occurs when two or more different threads try to access the same memory location.
- Dead code. Although dead code (code that will never execute) may not directly cause a run-time failure, it is important to understand why the code will not execute.

3. Traditional Methods of Verifying and Testing Software

Typical software verification processes include manual reviews and dynamic testing. Code review involves line-by-line manual inspection of the source code with the goal of finding errors in the code. The process comprises a team that will perform the review (moderator, designer, coder, and tester), the preparation process (including the creation of a checklist), and the inspection activity itself. Based on the outcome, the development team may need to address errors found and others in the organization will follow up to ensure that issues and concerns raised during inspection are resolved. With this process, detecting subtle run-time errors can be difficult. For example, an overflow due to complex mathematical operations that involve programmatic control can easily be missed. Additionally, the code review process can be inconsistent; since it is highly dependent on human interpretation, results can vary based on the team and context of the review process.

Complementing code reviews, dynamic testing is used to verify the execution flow of software, that is, to verify decision paths, inputs, and outputs. This process involves creation of test cases and test vectors and the execution of the software using these tests. Dynamic testing is well suited to the goal of finding design errors, in which the test cases often match functional requirements. Test teams then compare the results to the expected behavior of the software. Because of the complexity of today's software and tight project deadline requirements, dynamic testing is often not exhaustive. Although many test cases can be generated automatically to supplement those created manually, it is not feasible to expect dynamic testing to exhaustively verify every aspect of embedded software. This kind of testing can show the presence of errors, but not their absence.

In theory, performing code review and executing the right set of test cases can catch every defect no matter the type. In practice, however, the challenge is the amount of time spent reviewing code and applying enough of the right tests to find all the errors in today's complex systems. Even for the simplest operations, such as adding two 32-bit integer inputs, one would have to spend hundreds of years to complete exhaustive testing, which is not realistic [3]. Viewed from this perspective, code review and dynamic testing are bug detection techniques more than proving techniques because they cannot in practice exhaustively show that design errors and code defects have been eliminated.

4. Employing Formal Methods for Verification

To address the shortcomings of code reviews and dynamic testing, which are not exhaustive and can miss design or coding errors, engineers are turning to tools that implement formal methods to prove the absence of certain design and run-time errors, and ultimately to gain greater confidence. Formal methods refers to the application of theoretical computer science fundamentals to solve difficult problems in software and hardware specification and verification. Applying formal methods to models and code gives engineers insight about their design or code and confidence that they are robust when exhaustive testing is not practical.

To better understand formal methods, consider the following example. Without the aid of a calculator, compute the result of the following multiplication problem within three seconds:

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

Although computing the answer to the problem by hand will likely take you longer than three seconds, you can quickly apply the rules of multiplication to determine that the result will be a negative number. Determining the sign of this computation is an application of a specific branch of formal methods known as abstract interpretation. The technique enables you to know precisely some properties of the final result, such as the sign, without having to fully multiply the integers. You also know from applying the rules of multiplication that the result will never be a positive number or zero for this computation.

Now, consider the following simplified application of the formal mathematics of abstract interpretation to software programs. The semantics of a programming language can be represented by concrete and abstract domains. Certain proof properties of the software can be performed on the abstract domain. In fact, it is simpler to perform the proof on the abstract domain than on the concrete domain.

The concept of soundness is important in the context of a discussion on abstract interpretation. Soundness means 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, for example, that the software will not exhibit certain run-time errors [4].

Cousot and Cousot [5] describe the application and success of abstract interpretation to static program analysis. Deutsch describes the application of this technique to a commercial software tool [6]. The application of abstract interpretation involves computing approximate semantics of the software code with the abstraction function, which maps from the concrete domain to the abstract domain 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.

Lattices are used to represent variable values. For the sign example described earlier, the lattice shown in Figure 4 can be used to propagate abstract values in a program (starting at the bottom and working to the top for conditions such as <0, =0, and so forth). 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.

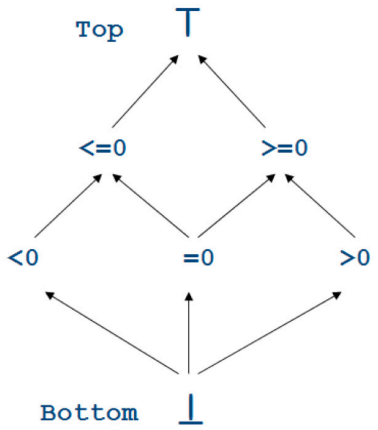

Figure 4: Lattice representation of variables.

Over approximation is applied to all possible execution paths in a program. Analysis techniques can identify variable ranges. That information is used to prove either the existence or the absence of run-time errors in source code.

To better understand the application of abstract interpretation to code verification, consider the following operation:

```
X := X / (X – Y);
```

If X is equal to Y, then a divide by zero will occur. In order to conclusively determine that a divide by zero cannot occur, the range of X and Y must be known. If the ranges overlap, then a divide-by-zero condition is possible.

In a plot of X and Y values (see Figure 5), any points that fall on the line representing X=Y would result in a run-time error. The scatter plot shows all possible values of X and Y when the program executes the line of code above (designated with +). Dynamic testing would execute this line of code using various combinations of X and Y to determine if there will be a failure. However, given the large number of tests needed to be run, this type of testing may not detect or prove the absence of the divide-by-zero run-time error.

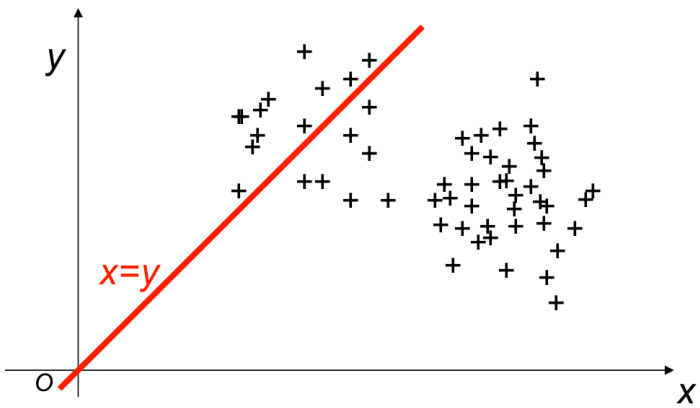

Figure 5: Plot of data for X and Y.

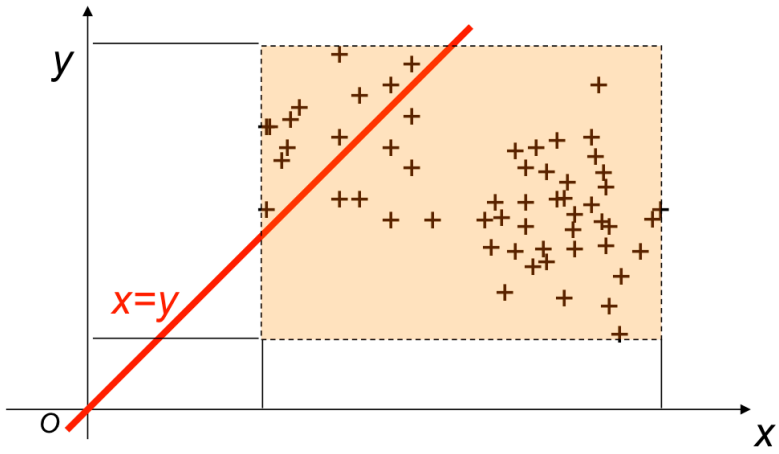

Figure 6: Creating a bounding box to identify potential errors.

Another methodology would be to approximate the range of X and Y in the context of the run-time error condition (that is, X=Y). In Figure 6, note the bounding box created by this method. If the bounding box intersects X=Y, then there is a potential for failure. Some static analysis tools apply this technique. However, approximation of this type is too pessimistic, since it includes unrealistic values for X and Y.

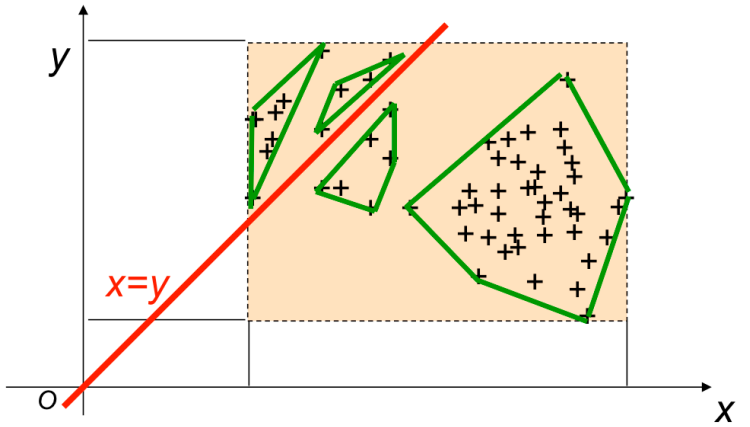

Figure 7: Abstract interpretation.

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 (for example, arithmetic operations, loops, if-then-else, and concurrency) an abstract lattice is created. A simplified representation of this concept is to consider the grouping of the data as polygons as shown in Figure 7. Since the polygons do not intersect X=Y we can conclusively say that a division by zero will not occur.

The abstract interpretation concept can be generalized as a tool set that can be used to determine variable ranges and to detect a wide range of run-time errors in software. Abstract interpretation investigates all possible behaviors of a program—that is, all possible combinations of values—in a single pass to determine how and under what conditions the program may exhibit certain classes of defects. The results from abstract interpretation are considered complete because it can be mathematically proven that the technique predicts the outcome as it relates to the operation under consideration.

The application of formal methods enables engineers to apply automation to software verification tasks. Unlike manual code reviews, automated application of formal methods is consistent. Formal methods will also provide a complete answer when it can be applied to the problem. Verification based on formal methods can be applied in the software design and coding phases of the V diagram.

5. Application of Formal Methods to Model and Code Verification

Model Verification

During the software design phase, which today typically involves creating models of the advanced control algorithms, engineers need to verify that the design they produce is robust. For example, they need to be certain that their design will not contain overflow errors or state machines with unreachable states. Because engineering teams develop and work with models in this phase, the application of formal methods for verification in this phase is termed *model verification.* The purpose is to produce a robust software design by ideally detecting all design errors or proving their absence.

However, use of models alone does not ensure a robust design. As an example, consider an algorithm that contains an addition operation. The two inputs to the addition operation are generated by other complex mathematical operations. Both inputs are 8-bit signed integers and the output is of the same type. In this scenario, it is possible that the addition operation may result in an overflow. For example, an overflow will occur if the first input has a maximum value of 2^7-1 and the other input is greater than 0. Using the traditional methods of design reviews and dynamic testing, the exact condition that results in the overflow might be missed. In contrast, using formal methods tools, engineers can determine the minimum and maximum ranges of the input to the addition. Furthermore, formal methods tools can determine that it is possible for an overflow to occur and can produce a test case or counter example to show how this overflow design error can occur.

Code Verification

During the coding phase, engineering teams either manually or automatically translate the design documents or models into code. The application of formal methods for verification in this phase is termed *code verification* and the purpose is to produce robust code by identifying and proving the absence of code defects such as run-time errors. This can be accomplished with formal methods coupled with *static code analysis*—the analysis of software without dynamic execution. This technique identifies the absence, presence, and possible presence of a certain class of run-time errors in the code. As a result, engineers can use this technique to prove that the code is free of detectable run-time errors.

During code development and integration, it is important to thoroughly understand the interface between various code components. For example, consider a situation in which handwritten code produced by one team generates an index value that is used for an array access in generated code produced by a second team. The first team believes that the index range can be 0 to 599. The second team believes the maximum index value is 399 and has developed the software with that understanding. Unless there is a test case that causes the index value to exceed

399, this run-time error may not be detected during the integration test. It is even possible that if this illegal array access were to occur during the execution of a test case, the error may not be detected. For example, writing to out-of-bounds memory may not cause a program to fail, unless the data at that location were to be used in some fashion.

The application of formal methods coupled with static code analysis does not require execution of the source code, so it can be used as soon as code is available. Using formal methods and static code analysis tools, engineers can validate software at the component level or as an integrated application. Because these tools propagate variable range values, they can detect or prove that the illegal array access in the example described above may or may not occur.

6. Summary and Conclusion

Today's sophisticated civilian and military aerospace applications often include a complex combination of handwritten and automatically generated code. Even after formal code reviews and dynamic testing is performed on the right-hand side of the V diagram, latent errors can still remain in a system because traditional verification and test methods are often incomplete. Formal methods enable teams to prove that aspects of their models and code are free of a specific type of error, enabling them to focus their verification efforts on the model components or code that require further attention. Applying formal methods for model and code verification instills more confidence in the engineers building modern embedded systems.

REFERENCES

1. SEI, How Good Is the Software: A Review of Defect Prediction Techniques. Brad Clark, Dave Zubrow, 2001
2. Challenges in deploying static analysis; Jain, Rao, Balan, CrossTalk – August 2011
3. Dependable Embedded Systems, Software Testing. Jiantao Pan 1999.
4. Cousot, "Abstract Interpretation", ACM Computing Surveys, 1996
5. Cousot and Cousot "Abstract Interpretation Based Formal Methods and Future Challenges", Informatics. 10 Years Back. 10 Years Ahead, 2001)
6. Deutsch, "Static Verification of Dynamic Properties, SIGAda, 2003
7. Regehr, J., Reid, A., Webb, K.: Eliminating stack overflow by abstract interpretation. In Proc. of the 3rd International Conf. on Embedded Software (EMSOFT), Philadelphia, PA, October 2003
8. <http://www.di.ens.fr/~cousot/projects/DAEDALUS>
9. Spoto A., "JULIA: A Generic Static Analyser for the Java Bytecode", 1982
10. <http://www.mathworks.com/products/polyspace>

ABOUT THE AUTHORS

Jay Abraham is the Product Marketing Manager of Polyspace products at MathWorks. Jay is part of the team leading product strategy and business development of Polyspace products, specializing in the American marketplace. Prior to joining MathWorks, Jay was a Product Marketing Manager at companies such as Wind River and Magma Design Automation, starting his career with IBM. Jay has a B.S. degree from Boston University, a Master's from Syracuse, and is a graduate of the Institute of Managerial Leadership from The Red McCombs School of Business at The University of Texas at Austin.

E-mail: Jay.Abraham@mathworks.com

Dr. Jon Friedman is the Aerospace & Defense and Automotive Industry Marketing Manager at MathWorks. Jon leads the marketing effort to foster industry adoption the MATLAB and Simulink product families and Model-Based Design. Prior to joining MathWorks, he worked at Ford Motor Company where he held positions ranging from software development research to electrical systems product development. Jon has also worked as an Independent Consultant on projects for Delphi, General Motors, Chrysler and the US Tank-Automotive and Armaments Command. Jon holds a B.S.E., M.S.E. and Ph.D. in Aerospace Engineering as well as a Master's in Business Administration, all from the University of Michigan.

E-mail: Jon.Friedman@mathworks.com