



PACIFIC NW
28TH ANNUAL
SOFTWARE
QUALITY
CONFERENCE

OCTOBER 18TH – 19TH, 2010



ACHIEVING
QUALITY
IN A COMPLEX
ENVIRONMENT

*Conference Paper Excerpt
from the*
CONFERENCE
PROCEEDINGS

Permission to copy, without fee, all or part of this material, except copyrighted material as noted, is granted provided that the copies are not made or distributed for commercial use.

Improvement Processes that Create High Quality Embedded Software

Jay Abraham
jabraham@mathworks.com

Marc Lalo
mlalo@mathworks.fr

Scott Runstrom
srunstro@mathworks.com

Abstract

The development of embedded software encompasses a wide range of best practices and development methodologies. For quality-critical projects intended for highly reliable applications, delivery of high quality software is an absolute requirement. In these situations, development and test teams must complete code reviews, perform unit and regression test, and testing on the target system. But is it enough? What if a critical defect escapes to software deployment and then to production? Formal methods based mathematical techniques may alleviate some of the doubt. Application of formal methods based code verification may provide precision in guiding software engineering teams to know which parts of code will not fail and isolate those aspects of code that will fail or most likely to fail. This paper will discuss the practical application of these techniques for the verification of software. As part of the application of this improvement process, the paper will explore how these techniques enable the creation of high quality embedded software.

Biography

Jay Abraham is currently Product Manager at the MathWorks. Jay began his career as a microprocessor designer at IBM followed by engineering and design positions at hardware, software tools, and embedded operating systems companies such as Magma Design Automation and Wind River Systems. Jay has a MS in Computer Engineering from Syracuse University and a BS in Electrical Engineering from Boston University.

Marc Lalo is currently Product Manager at the MathWorks with responsibility for Polyspace code verification products. He has been working in the embedded software verification and safety standards field for more than 10 years. Marc is a graduate of Ecole nationale supérieure des Télécommunications.

Scott Runstrom is currently a Principal Technical Writer at the MathWorks. Scott has worked in the technical communications profession for 20 years. His previous experience includes Applied Biosystems, Waters Corporation, and Stratus Computer Inc. Scott has an M.S. in Professional Communications from Clark University, and a B.S. in Technical Writing (with a background in mechanical engineering) from Worcester Polytechnic Institute.

Copyright Jay Abraham, Marc Lalo, Scott Runstrom 2010

1. Introduction to software quality and verification processes

The goal of most software development teams is to maximize both quality and productivity. However, when developing software, there are always three related variables: cost, quality, and time. Generally, the criticality of the application determines the balance between these three variables. With classical testing processes, development teams generally try to achieve their quality model by testing all modules in an application until each meets the required quality level. Unfortunately, this process often ends before quality objectives are met, because the available time or budget has been exhausted. To achieve maximum quality and productivity, however, one cannot simply verify and test code at the end of the development process. It must be integrated into the verification process, in a way that respects time and cost restrictions.

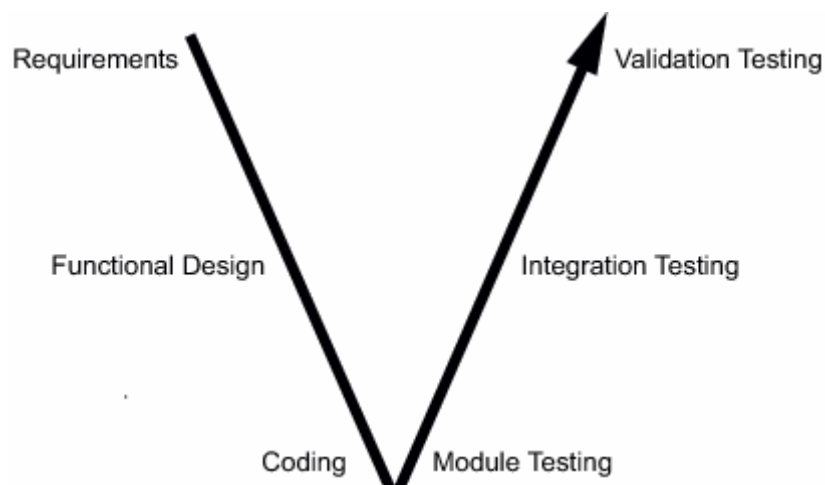


Figure 1 – Software Verification and Validation

The verification and validation process (often referred to as V&V) of complex embedded systems, consists of determining that the software requirements are implemented correctly and completely and are traceable to the system requirements. The main objective of the V&V process is to completely analyze and test the software during the development and test phase to assure that the software performs its intended functions correctly, to make sure that it performs no unintended operations, and provide information about its quality and reliability. Software V&V also determines how well the software meets its technical requirements and its safety, security, and reliability objectives relative to the system. The V&V tasks analyze, review, demonstrate or test all software development outputs (1). This process is often described with a V diagram as shown in Figure 1.

Specific classes of software defects that can be introduced in the coding phase of the V&V process are coding errors, run-time errors (software errors considered as latent faults) and design errors. 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. Software with these types of defects cannot be considered robust. 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.
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.
3. Null pointer dereference – A Null pointer dereference occurs when attempting to reference memory with a pointer that is NULL. Any dereference of that pointer leads to a 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.
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.
6. Illegal type conversions – Illegal type conversions may result in corruption of data.
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. Note that DO-178B prohibits the existence of dead code.
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.

The next sections examine how various techniques can be used to minimize or eliminate these errors in software and how these techniques can be incorporated into a V&V process to create high quality embedded software.

2. Static analysis

Static analysis or verification is a broad term, and is applicable to any tool which derives dynamic properties of a program without actually executing it. However, most static analysis or verification tools only verify the complexity of the software, in a search for constructs which may be potentially dangerous. As described in (2), these tools can certainly find some errors in code, however they may miss errors which strongly depends on dataflow (like run-time errors or design errors). Because these tools do not exhaustively analyze all behaviors these tools are considered unsound (miss errors) and therefore produce false negatives. Decreasing the probability of false negatives will increase the probability of false positives. i.e. falsely identifying a defect that is not real. 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 negatives and thus miss errors in the process.

3. 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 such as unit, integration, and system (3). Dynamic testing involves the creation of test-cases and test-vectors and execution of the software against these tests. This particularly suits the goal of finding design errors, which test cases often matching functional requirements. 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.

4. Abstract interpretation

Abstract interpretation is the application of mathematical formal methods based techniques to abstract the semantics of a software program. It can be an effective means of performing certain types of software verification. Abstract interpretation as a proof-based verification method can be illustrated by multiplying three large integers in the following problem:

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

Although computing the answer to the problem by hand can be time-consuming, we can quickly apply the rules of multiplication to determine that the sign of the computation will be negative. Determining the sign

of this computation is an application of abstract interpretation. The technique enables us to know precisely some properties of the final result, such as the sign, without having to multiply the integers fully. We also know from applying the rules of multiplication that the resulting sign will never be positive for this computation.

In a similar manner, abstract interpretation can be applied to the semantics of software to prove certain properties of the software. Abstract interpretation bridges the gap between conventional static analysis techniques and dynamic testing by verifying certain dynamic properties of source code without executing the program itself. Abstract interpretation investigates all possible behaviors of a program – that is, all possible combination of values – in a single pass to determine how the program may exhibit certain classes of run-time failures. The results from abstract interpretation are considered sound because we can mathematically prove that the technique will predict the correct outcome as it relates to the operation under consideration.

5. Code verification

Code verification or code verifiers based on abstract interpretation can be used as a static analysis tool to detect and mathematically prove the absence of certain run-time errors in source code, such as overflow, divide by zero, and out-of-bounds array access as part of the V&V process. The verification is performed without requiring program execution, code instrumentation, or test cases. Cousot describes the application and success of applying abstract interpretation to static program analysis (4). Deutsch describes the application of this technique to a commercially available solution (5). Descriptions of the application of abstract interpretation for software and hardware can be found in both academia and industry. Some of these include:

1. Stack overflow checking of embedded software (6)
2. JULIA for abstract interpretation verification of Java code (7)
3. DAEDALUS for validating critical software (8)
4. Hardware verification of the timing of transistor gates (9)

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

```

static void Pointer_Arithmetic (void)
{
    int array[100];
    int i, *p = array;
    for(i = 0; i < 100; i++, p++)
        *p = 0;

    if(get_bus_status() > 0) {
        if (get_oil_pressure() > 0)
            *p = 5;
        else
            i++;
    }

    i = get_bus_status();
    if (i >= 0) { *(p-i) = 10; }

    if ((0 < i) && (i <= 100)) {
        p = p - i;
        *p = 5;
    }
}

```

Figure 2 – Polyspace color scheme

To describe the use of abstract interpretation based code verification in this paper, we used Polyspace® (10). Polyspace is a code verifier that detects and proves the absence of certain run-time errors such as

overflows, divide by zero, out of bound array access, etc. The input to Polyspace is C, C++, or Ada source code. Polyspace first examines the source code to determine where potential run-time errors could occur. Then it generates a report that uses color-coding to indicate the status of each element in the code, as shown in Figure 2. Polyspace results that are all tagged in green indicate that the code is free of certain run-time errors. In cases where run-time errors have been detected and color coded in red, gray, or orange, software developers and testers can use information generated from the code verification process to fix identified run-time errors.

Code verification reduces development time by automating the verification process and helping to efficiently review verification results. It can be used at any point in the development process, but using it during early coding phases it enables development and test teams to find errors when it is less costly to fix them. This process takes significantly less time than using manual methods or using tools that require modification of code or run test cases.

Knowing where potential run-time errors could exist or that certain run-time errors are proven not exist helps with productivity. Less time is spent debugging because the exact location of an error in the source code is known. After errors are fixed, verification can be performed on the updated code. Code verification helps developers and testers use their time effectively. Because they know which parts of code are run-time error-free and they can focus on the code that has definite run-time errors or might have run-time errors.

6. Overview of a code verification process

Code verification cannot magically produce quality code at the end of the development process. Code verification is a tool that helps measure the quality of code, identify issues, and ultimately achieve set quality goals. To successfully implement such a solution within a development process the implementers of a high quality software development process must define quality objectives, define a process to match quality objectives, apply the process to assess the quality of code, and then continuously improve this process.

Before one can verify if code meets a prescribed set quality goals, these goals must first be defined. The first step in implementing a code verification process is to define quality objectives. Since the ultimate goal is to create zero defect software, this goal may not be realized in the short term. Therefore intermediate steps and milestones should be established.

The type of code verification that is to be performed is also important. One can perform robustness or contextual verification. Robustness verification helps to show that the software works under all or most conditions or environments and contextual verification shows that the software works under normal conditions or environments. Some verification processes may incorporate both robustness and contextual verification. For example, developers may perform robustness verification on individual files early in the development cycle, while writing the code. Later, the team may perform contextual verification on larger software components. Additionally, quality objectives may involve the use of coding rules or standards and defining software quality levels.

6.1 Robustness verification

Robustness verification proves that the software works under all conditions, including "abnormal" conditions for which it was not designed. This can be thought of as "worst case" verification. Code verifiers such as Polyspace, by default perform robustness verification. In this use-case the code verifier:

- Assumes function inputs are full range
- Initializes global variables to full range
- Automatically stubs missing functions

While this approach ensures that the software works under all conditions, it can lead to unproven results (i.e. *orange checks* in Polyspace vernacular). The software developer or tester then inspects code fragments identified as unproven in accordance with their software quality objectives.

6.2 Contextual verification

Contextual verification shows that the software works under predefined working conditions. This limits the scope of the verification to specific variable ranges, and verifies the code within these ranges. When performing contextual verification with code verifiers such as Polyspace, it will reduce the number of unproven results. Various techniques are available to help with this task.

- Using data range specifications to specify the ranges of function input and global variables, thereby limiting the verification to these cases.
- Create a detailed main program to model the call sequence, instead of using automatically created simple main generators.
- Provide manual stubs that emulate the behavior of missing functions, instead of using the default automatic stubs.

6.3 Choosing coding rules

Coding rules are one of the most efficient means to improve both the quality of your code, and the quality of your verification results. Popular coding rules to choose from include MISRA-C, MISRA-C++, and JSF++ (11). In addition to improved quality, in general the software will be easier to read, maintain and port. Some certification standards also stipulate the use of coding standards and rules (such as DO-178B in the aviation industry). A side benefit is that when enforcing coding standards, improved code verification results may also be observed. In general, use of coding standards and rules does not mean that all rules specified in a certain standard must be enforced.

6.4 Defining software quality levels

Defining and identifying software quality levels is an excellent method by which to create intermediate milestones to achieve a goal of getting to zero defect software. Table 1 identifies an example set of quality levels that can be used as part of a code verification process. The table identifies four quality levels from level QL-1 (lowest) to level QL-4 (highest). Each quality level consists of a set of software quality criteria that represent a certain quality threshold. Quality levels such as these are beginning to be used in the automotive industry. For example, by original equipment manufacturers to stipulate specific Software Quality Objectives (SQA) to their suppliers (12).

Criteria	Software Quality Levels			
	QL1	QL2	QL3	QL4
Document static information	X	X	X	X
Enforce coding rules with direct impact on code verification results	X	X	X	X
Review all systematic run-time errors (e.g. red checks)	X	X	X	X
Review all dead code (e.g. gray checks)	X	X	X	X
Review first criteria level of unproven code (e.g. orange checks)		X	X	X
Review second criteria level of unproven code (e.g. orange checks)			X	X

Criteria	Software Quality Levels			
	QL1	QL2	QL3	QL4
Enforce coding rules with indirect impact on code verification results			X	X
Perform dataflow analysis			X	X
Review third criteria level of unproven code (e.g. orange checks)				X

Table 1 – Example definitions of software quality levels

Static information – Includes information about the application architecture, the structure of each module, and all files. This information must be documented to ensure that the application is fully verified.

Coding rules – This includes the application of selected or all coding rules from standards such as MISRA-C, MISRA-C++, or JSF++. There are two classifications of the application of coding rules. The first set has direct impact on the results of code verification and a second set with indirect impact. Examples of MISRA coding rules that have a direct impact on code verification results are (list is not exhaustive):

- The *static* storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage
- Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void
- Floating-point expressions shall not be tested for equality or inequality
- The *goto* statement shall not be used
- Etc.

Good design practices generally lead to less code complexity, which can indirectly improve the results from code verification. The following set of coding rules help address design issues that have indirect consequence on code verification results (list is not exhaustive).

- Identifiers (internal and external) shall not rely on the significance of more than 31 characters
- *typedefs* that indicate size and signedness should be used in place of the basic types
- Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures
- Bitwise operations shall not be performed on signed integer types
- Etc.

Systematic run-time errors – These are definite run-time errors that have been identified by the code verifier. They represent errors that occur every time the code is executed.

Dead code – Represents unreachable code. This often highlights a design error, as it questions the presence of a statement which is not reachable under any data conditions.

Unproven code – Indicate unproven code, meaning a run-time error may occur. The three criteria called out in Table 1 indicate threshold levels that should be established by the software development team for code that is not proven to be free of run-time errors. For example, the software team could decide that for divide by zero run-time errors, for the first criterion level 60% of divide by zero errors shall be proven, for the second criterion level 75%, and for the third criterion level 95%.

Dataflow analysis – Identifies errors such as non-initialized variables and variables that are written but never read. This can include inspection of application call tree, read/write accesses to global variables, and shared variables and their associated concurrent access protection.

6.5 Achieving certification

High quality software often implies achieving certification. The verification activities have to be consistent with requirements defined by the certification standards, such as DO-178B or IEC 61508. These standards often mention verification activities such as checking coding rules, finding run-time errors and design errors as relevant steps in the process. Improving the quality up to the proof of absence of certain class of errors naturally fits into these standards. This can enable software development teams reduce alternative verification activities thanks to evidences of these proofs.

7. Deploying an improvement process

In summary, the overall improvement process that can enable software development teams to develop high quality code would consist of the use of code verification as part of the overall software verification and validation (V&V) process, use of coding rules and standards, and identifying milestones for achieving certain software quality levels. For the code verification process, you can also use robustness verification, contextual verification or a combination of the two. It is also important to use code verification early in the development cycle because it improves both quality and productivity. That is, it allows users to find and manage run-time defects soon after the code is written. This saves time because each user is familiar with their code, and can quickly determine why code can or cannot be proven to be reliable from a run-time perspective. In addition, defects are cheaper to fix at this stage, since they can be addressed before the code is integrated into a larger system.

8. Creating high quality embedded software

Critical software defects in the form of run-time errors may be introduced in the coding phase of software development. These defects may not be detected with traditional testing methods applied during unit and regression phases of software verification and validation (V&V). Abstract interpretation based code verification techniques introduce precision and guidance to the software development process. The technique provide precision in guiding software engineering teams to know which parts of code may not fail and isolate those aspects of code that may fail or are most likely to fail. Using these techniques, software developers and testers can minimize effort related to hunting for defects in wide swaths of code. The application of code verification as part of the overall V&V processes is an incremental quality improvement process. This process helps software development teams get on the path of creating zero defect software. It is a robust verification process that helps achieve high reliability in embedded devices.

References

1. **NIST**. *Reference Information for the Software Verification and Validation Process*. 1996.
2. **Bessey, Al**. *A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World*. ACM, 2010.
3. **Wagner, Stefan** . *A Literature Survey of the Software Quality Economics of Defect Detection Techniques*.. ACM, 2006.
4. **Cousot, Patrick**. *Abstract Interpretation: Theory and Practice*. Springer, 1996.
5. **Deutsch, Alain**. *Static Verification of Dynamic Properties*. SIGDA, 1996.
6. **Regehr, John, Alastair Reid and Kirk Webb**. *Eliminating Stack Overflow by Abstract Interpretation*. EMSOFT, 2003.

7. **Spoto, Fausto.** *JULIA: A Generic Static Analyzer for the Java Bytecode.* 1982.
8. **Programme, European IST.** DAEDALUS. [Online] 2002.
<http://www.di.ens.fr/~cousot/projects/DAEDALUS/>.
9. **Viladrosa, Robert.** *Abstract Interpretation Techniques for the Verficiation of Timed Systems.* Thesis, 2005.
10. **MathWorks.** Polyspace Code Verification Products. [Online] 2010.
<http://www.mathworks.com/polyspace>.
11. **MISRA.** The Motor Industry Software Reliability Association. [Online] 2010. <http://www.misra.org.uk/>.
12. **MathWorks.** Using Polyspace to implement the “Software Quality Objective for Source Code Quality” Standard. [Online] 2010. <http://www.mathworks.com/matlabcentral/fileexchange/27525>.