

Vector Software

WHITEPAPER

Understanding Verification and Validation of software under IEC 61508-3:2010

Abstract

This paper is intended to serve as a reference for developers of systems that will contain software that must be certified or follow a process conformant to IEC 61508. This paper will focus on the verification and validation of safety-related software (IEC 61508:3:2010, Part 3). It will describe a selection of techniques and measures that are relevant to the software safety validation based on the Safety Integrity Level (SIL). Some of these techniques can be applied to reduce the cost and effort required to certify software to IEC 61508-3:2010.

Introduction

Competition in Industrial Automation Systems is intense. Successful companies must constantly innovate by introducing new products and features, many of which contain significant amounts of software. These products have been transformed from primarily being mechanical, into integrated devices with embedded software in all major systems including: the industrial process-control automation industry, automotive, heavy machinery, and mining.

Controlling the costs of industrial automation embedded systems is extremely important for industry suppliers since there is a much higher volume of software than in other safety-critical industries like avionics and railway.

Software testing has traditionally been very expensive, but the cost of finding software bugs now versus the direct costs and damaged product branding associated with recalls or system downtimes makes thorough testing a necessity in the industrial automation industry.

Industrial Automation Systems, also known as electrical, electronic or programmable systems that perform safety related functions are often developed to the IEC 61508 standard. This standard developed by the International Electrotechnical Commission (IEC) is well-established. IEC 61508 is the standard governing functional safety of programmable electronic systems. The standard is divided into seven parts, in this paper we will focus on the software section, "Part 3: Software requirements (required for compliance)".

This document is intended to serve as a reference for understanding verification and validation of software under IEC 61508-3:2010. It is not intended to be an exhaustive review of the standard, but rather to provide a high-level view of the different testing categories and requirements and how they can be satisfied.

For this document, the following legend is used:

- R** Recommended activity
- HR** Highly recommended activity

Software Verification

By analyzing 7.4.7, 7.4.8, 7.5, 7.7, 7.8 and 7.9 and relating this to the Tables in Annex A, we can summarize the following activities related to verification and validation of software.

Technique / Measurement	SIL			
	1	2	3	4
A. Probabilistic testing	-	R	R	R
B. Dynamic analysis and testing	R	HR	HR	HR
C. Data recording and analysis	HR	HR	HR	HR
D. Functional and black box testing	HR	HR	HR	HR
E. Performance testing	R	R	HR	HR
F. Interface testing	R	R	HR	HR
G. Test management and automation tools	R	HR	HR	HR
H. Traceability between software design specification and test specifications	R	R	HR	HR
I. Offline numerical analysis	R	R	HR	HR
J. Impact analysis	HR	HR	HR	HR
K. Tool and translators certified or increased confidence from use	R/HR	HR	HR	HR

Table 1 – Technique / Measurements extracted from IEC-61508-3:2010, Part 3, Annex A

A. Probabilistic Testing

This exists to derive measures like MTBF (Mean Time Between Failures), availability, or probability of safe execution. The probability of error can be reduced by repeating the test with several independently chosen values for commonly used tests.

B. Dynamic Analysis and Testing

This section covers a variety of tests, many of which can be achieved through a mix of testing on the entire software build (hereafter referred to as “system testing”) and unit/module testing. Most of these activities are described in IEC-61508-3:2010, Part 3, Annex B Table B.2 shown below.

Technique / Measurement	SIL			
	1	2	3	4
1. Test case execution from boundary value analysis	R	HR	HR	HR
2. Test case execution from error guessing	R	R	R	R
3. Test case execution from error seeding	-	R	R	R
4. Test case execution from model-based test case generation	R	R	HR	HR
5. Performance modeling	R	R	R	HR
6. Equivalence classes and input partition testing	R	R	R	HR
7a. Structural coverage (entry points) 100%	HR	HR	HR	HR
7b. Structural coverage (statements) 100%	R	HR	HR	HR
7c. Structural coverage (branches) 100%	R	R	HR	HR
7d. Structural coverage (MC/DC) 100%	R	R	R	HR

Table 2 – Dynamic analysis and testing - IEC-61508-3:2010, Part 3, Annex B Table B.2

Although these activities can technically be done at the system level, the IEC 61508:3 standard specifies that during the Dynamic analysis and testing phase they must be performed at a unit or module level, and must be based on the specifications of the software and/or the specification and the code. It is possible to provide a high level of automation to all of these techniques, as explained in the following sections.

1. Test Case Execution from Boundary Value Analysis

These types of tests aim at removing potential software errors at parameter limits or boundaries based on the parameter's type. The best way to achieve this goal is to unit test the software so as to input these limit values directly to functions. There are some "special cases" that should be duly tested. These include:

- Setting parameters to zero if they happen to divide another variable within the function
- Testing blank ASCII characters
- Testing an empty stack or list element
- Testing a null matrix
- Testing a zero table entry
- Testing the maximum and the minimum values of the type, and potentially the functional limits
- Testing values outside the boundaries

Some test automation tools can automate the construction of these test cases even further by providing a way to automatically generate test cases that test all input values to their minimum, maximum and median values. These sorts of test cases are referred to as MIN-MID-MAX test cases.

The minimum and maximum values are determined by testing the range of every type present in the program on the target board or simulator. Thus, using the test automation tool on either the board or on a simulator will guarantee that the range of boundary values tested through automatically generated MIN-MID-MAX tests is valid in your system.

These tools can further test special values as specified above. They can even test other values not directly mentioned, such as Not-A-Number (NaN), positive and negative infinity on floating-point variables, etc.

2. Test Case Execution from Error Guessing

These test cases are based on testing experience and intuition combined with knowledge and curiosity about the system under test. This may lead to the creation of additional test cases to try to cause errors within the software.

This activity can be performed at system level, but there is also value in performing this at the unit level (as required by IEC-61508-3:2010, Part 3, Annex A, Table A.5), thus ensuring that the components of the software are as error-proof as possible. In fact, the experience gained by testing at implementation time (as introduced earlier in this document) may lead to the creation of additional test cases that can be recycled during regression testing phases.

3. Test Case Execution from Error Seeding

The goal of this test strategy is to intentionally provoke errors to see if test cases will identify their presence. If some errors are not detected, additional test cases must be provided.

Error Seeding can easily be done at unit level. A test case in a test automation tool can also be set to (1) input error conditions and, in many cases, (2) confirm that these error conditions were detected. For instance, pointers may be left intentionally null, or exceptions may be intentionally raised to see how the system would cope with such situations.

4. Test case execution from model-based test case generation

This test strategy allows early exposure of ambiguities between the original model and the developed/generated source code. As it extends from the test cases defined during Model Based Testing (MBT), it allows for automatic generation of test cases when working with the source code.

Most test automation tools are able to import test case data in a comma separated values (csv) format. If model based test cases can be exported into a csv format, or some other easily parse-able format, they can quickly be imported into the unit test automation tool for source code verification.

5. Performance Modeling

These activities aim at calculating performance, such as processor time, communications bandwidth, storage devices utilization, and so on.

Test automation tools can be used to inject test vectors for these types of activities, however the calculation of processor time, bandwidth, throughput, etc., must be done with the help of other tools, such as the debugger. An interesting feature to keep in mind when selecting a test automation tool is the capability for it to be executed under the control of a debugger.

6. Equivalence Classes and Input Partition Testing

This strategy aims at testing the software adequately by determining partitions on the input domain necessary to exercise the software. These test cases aim at testing the program sufficiently. They can be based on either software specifications or the internal structure of the program (or both).

These tests can obviously be conducted at system testing level, especially if testing high-level requirements. However, engineers may also want to test low-level requirements and/or base some of their partition test cases on the internal structure of the program (as required by IEC-61508-3:2010, Part 3, Annex A, Table A.5).

These last activities become practical when using a test automation tool. A unit test tool can test ranges of values and lists of values, either in a combination or non-combination mode. Execution of these complex test cases is done with the click of a button.

Unit test automation tools also typically have a feature called a partition test case generator to automatically create additional test cases on a provided domain.

7a. Structural Coverage – Entry Points

Entry point coverage attempts to ensure that each function/procedure is called at least once. It is not concerned with how much code inside the function/procedure has actually been executed.

```
if(i < 10 && j == 0 || k > 12)
```

It should be noted that a single test case evaluating to false at this line will not cover additional lines that may be contained within that 'IF' statement. Likewise, if this 'IF' statement has an 'ELSE' statement attached to it, a test case evaluating to true will not cover the contents of that 'ELSE' statement. However, either the true or false test case will cover the 'IF' statement per se.

Some unit test automation tools with automatic test case generation capabilities can help engineers devise test cases seeking to maximize statement coverage. For instance, automatically generated test cases based on the basis paths (the independent paths in the code) will often provide a high degree of statement coverage.

7c. Structural Coverage – Branch Coverage

Branch coverage concentrates on the state of decision points, which can be either true or false. For instance, the following line of code would require two test cases to be covered – one where the 'IF' statement returns true, and another where it returns false.

```
if(i < 10 && j == 0 || k > 12)
```

To comply with IEC-61508:3, a test automation tool should be able to produce both Statement and Branch levels of coverage during a single test execution.

7d. Structural Coverage – MC/DC (Modified Condition/Decision Coverage)

MC/DC requires the greatest number of tests to accomplish the coverage requirement. This level of coverage demonstrates that all sub-conditions that are part of a conditional statement can independently affect the outcome of the conditional statement itself. For instance, in the case of the following statement:

```
if(i < 10 && j == 0 || k > 12)
```

One should demonstrate that by changing the value of 'i' while keeping the value of other sub-conditions stable, the end value will change.

This task can be very arduous even for the most experienced engineer. However, this testing can be done efficiently by using a truth table. This can be automatically generated from the code and it should indicate clearly which test case pairs are required to achieve MC/DC coverage, and then flags which test cases and test case pairs have been provided as shown in Figure 1.

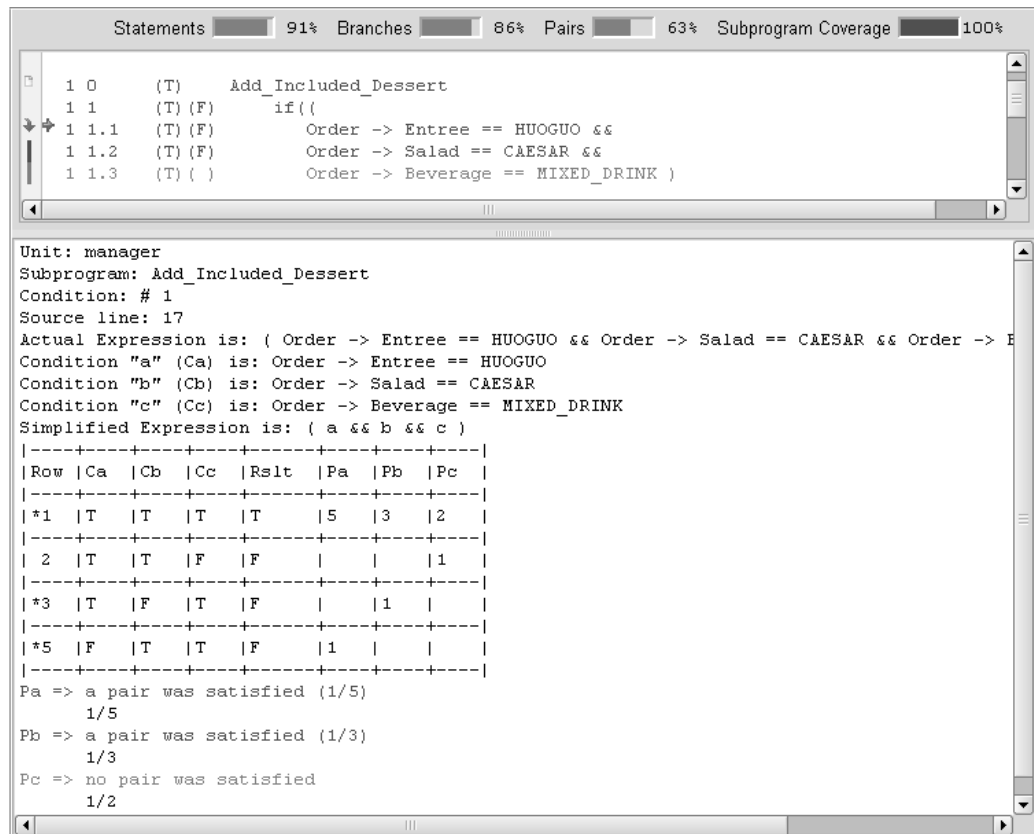


Figure 1 – MC/DC Truth Table

Additional Considerations

All of the testing described can be done on host, simulator, or directly on a target board. For more accurate results it is recommended these tests be performed at least on an appropriate simulator. They should be performed on the board whenever possible so as to (1) guarantee that the results correspond to the specificities of the environment, and (2) accelerate the testing linked to software/hardware integration testing activities detailed in Table A.6.

C. Data Recording and Analysis Testing

When test cases are created and executed, it should be possible to make notes of any specifics in relation to the test case, why they exist, and any interpretations of their results. Test Automation tools typically permit appending additional information to test case scenarios to capture this information.

D. Functional and Black Box Testing

Testing during software design and implementation is highly recommended for all SIL levels. Different types of suitable testing strategies are described in IEC-61508-3:2010, Part 3, Annex B, Table B.3 (shown below in Table 3).

Technique / Measurement	SIL			
	1	2	3	4
1. Test Case Execution from Cause Consequence Diagrams	-	-	R	R
2. Test case execution from model-based test case generation	R	R	HR	HR
3. Prototyping/Animation	-	-	R	R
4. Equivalence Classes and Input Partition Testing, including boundary value analysis	R	HR	HR	HR
5. Process Simulation	R	R	R	R

Table 3 – Functional and black-box testing – in IEC-61508-3:2010, Part 3, Annex B, Table B.3

1. Test case execution from Cause Consequence Diagrams

Cause Consequence Diagrams refers to modeling, in a diagrammatic form, the sequence of events that can develop in a system as a consequence of combinations of basic events.

2. Test case execution from model-based test case generation

As was discussed above, it is often desirable to be able to test the actual source code in addition to the model based simulation. Code based test tools have the ability to import model based test cases to alleviate the need to redefine the test data. Executing these tests will result in code coverage to prove the completeness of the test cases generated.

3. Prototyping/Animation

Prototyping and Animation check the feasibility of implementing the system against the given constraints.

4. Equivalence Classes and input partition testing, including boundary value analysis

Boundary Value Analysis and Equivalence Classes/Input Partition Testing have been described earlier in this document (during Verification and Testing). If these tests were directly performed on the target board, these should not have to be modified at this point in time. If they were done on a host or a simulator, the test cases associated with these activities can be easily reused within a unit test automation tool. At this point in time, additional test cases performed at system testing level may also be advisable.

5. Process Simulation

Process Simulation tests the function of a software system without actually putting the complete system in action.

E. Performance Testing

According to in IEC-61508-3:2010, Part 3, Annex B, Table B.6, Avalanche/Stress Testing, Response Timing and Memory Constraints, and Performance Requirements are “recommended” and “highly recommended” depending on the SIL. These can be partially automated using a test automation tool. For instance, the time a function takes to execute can be calculated at the unit test level. Another example of such tests may involve running the same test case several times, perhaps alongside system-level threads.

F. Interface Testing

During software design and implementation, it is recommended to test the interface of the code (at the function call level). There are several levels of detail or completeness of testing that are feasible. However the most important levels would require:

- Testing all function parameters using “extreme” values at once or one at a time (while using “normal” values for other parameters)
- Test all values for all parameters, including in combination through specific test conditions

Extreme values can be specified according to functional range or type, and illegal values can also be tested. Combinational testing should allow for testing all possible permutations of inputs.

G. Test management and automation tools

The challenge here is that during implementation, engineers are not likely to have access to the entire source code – by definition, it is still under development. This supposes a unit or perhaps module-based testing approach. In order to be able to test the software as it gets implemented, software stubs will need to be provided. These are pieces of code that take the place of regular code (for instance, one that has yet to be implemented) so linker closure can be obtained on a specific file, class or even module. Without linker closure, the code cannot be successfully tested.

Software Unit Test Automation tools are well-suited to this task. Engineers can use these tools to quickly generate a complete test environment – stubs included – without providing any test code or scripts. All that is needed is the code to be tested and the path to the “includes”. With this minimal amount of information, these tools can automatically generate all of the test harness code (test driver, stubs...) required for unit or module testing. What would normally take hours of test code development is done in a matter of minutes. An example is shown in Figure 2.

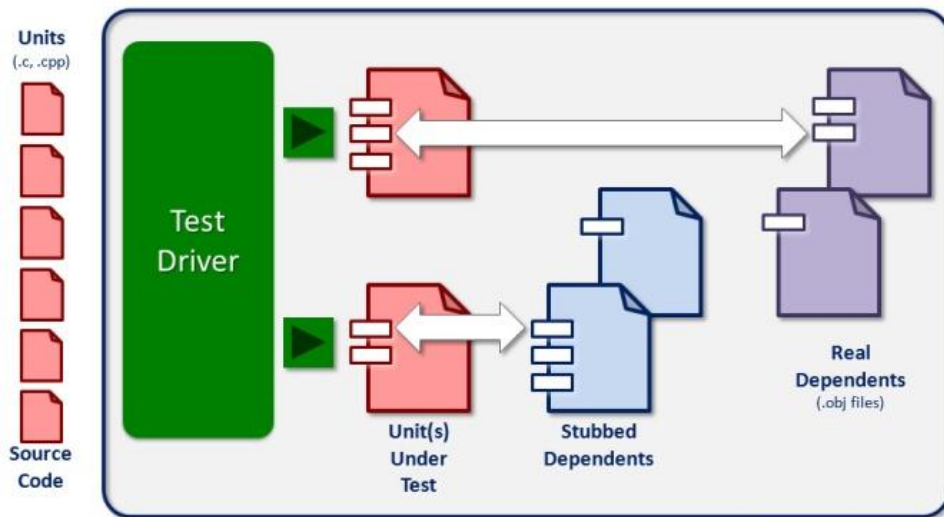


Figure 2 – Software Test Tool Automation

Unit test automation tools allow test execution on host, simulator, or directly on the product platform (also referred to as embedded target board). This is done typically through an interface called the Runtime Support Package (RSP). All of the facilities to execute tests and capture all test artifacts are controlled by the RSP. An example of this workflow is shown in Figure 3.

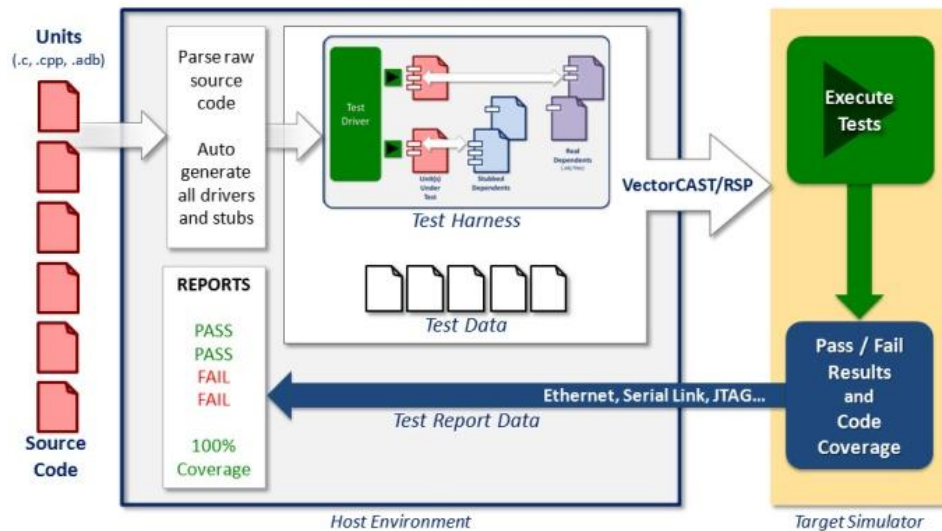


Figure 3 – Test Automation Tool working with Runtime Support Package

Even if the environment changes (for example, a new target processor, or with a different code version), test cases can be easily exported from and imported into test environments using a unit test automation tool. Typically in these tools the test case data is kept separate from the test harness in a text file, and can be re-imported in test harnesses focusing on specific units – all at a click of a button. This not only saves time but also provides assistance during test case definition activities.

H. Traceability between software design specifications and test specifications

According IEC 61508:3, the objective of Traceability is to ensure that all requirements can be shown to have been properly met and that no untraceable material has been introduced. This includes – but is not limited to – traceability between software requirements and test cases.

Normally a test automation tool integrates requirements through a module known as a “Requirements Gateway”, with third-party tools whose function is to maintain software requirements, such as IBM® Rational® DOORS®. Using this integration, users can download and then link specific software requirements to test cases, and then upload that information back to the requirements management system with the status of the test case (PASS/FAIL). This makes the drafting of the traceability matrix much easier.

I. Offline numerical analysis

The goal of these metrics is to predict the attributes of programs from properties of the software itself rather than from its development or test history. Typically test automation tools include cyclomatic complexity as part of their standard analysis. Additional metrics can typically be obtained using a static analysis tool.

J. Impact Analysis

Impact analysis aims at identifying the effect that a change or an enhancement to a software system will have to other modules in that software system as well as to other systems. This entails re-verifying the changed module, all the affected modules or the complete system, which itself depends on impact analysis.

Unit Test Automation tools typically have a complete regression testing capability. This means that the test cases that were designed for earlier versions of the code can be re-run seamlessly on the new code. If a test case becomes antiquated (because of a change in the number of parameters or their type for example), then that test case is simply ignored or flagged for review.

Thanks to this characteristic of test automation tools, it becomes possible to re-run the whole set of test cases on a given model against updated code in a matter of minutes. This streamlines the amount of maintenance that needs to be done. These tests can also be re-executed overnight, as the regression testing process can be fully automated.

K. Tool and translators certified or increased confidence from use

Tools and translators that are used can gain credit for use either through a formal certification of the tool, or through prior use. Where the tool is formally certified, it should come with the following documentation:

- A certificate from the authority stating its certification (see Figure 4 for a sample).
- Tool Operational Requirements (TOR)
 - This document contains verifiable requirements for the tool or translator under verification along with information about the project's operational environment that may have an impact on the results produced by the tool (e.g. microprocessor architecture)
- Tool Qualification Data (TQD)
 - This document contains the test data and results from running the qualification test suite required to verify all requirements in the TOR using the project's operation environment
- Compliance analysis to IEC 61508 standard
 - This is typically captured in a workflow manual describing how to use the tool within set boundaries to ensure the results produced with it are acceptable for proving conformance to a standard



Figure 4 – Test Automation Tool working with Runtime Support Package

Where the tool is relying “from increased confidence from use”, evidence should be provided of other similar safety related systems where the tool has been used.

Conclusion

In this paper we have discussed all of the Clauses in the IEC 61508:3 standard that relate to the validation and verification of the safety-related software. Software unit test solutions provide automation and flexibility which radically decrease the time spent on complying with the software verification and validation requirements specified in standards such as IEC 61508:3. We have also discussed techniques through the usage of these tools that can be used to improve the efficiency in working through these clauses.

References

International Electrotechnical Commission. 2010. *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems – Part3: Software Requirements*. (IEC 61508-3:2010).

About Vector Software

Vector Software, Inc., is the leading independent provider of automated software testing tools for developers of safety critical embedded applications. Vector Software's VectorCAST line of products, automate and manage the complex tasks associated with unit, integration, and system level testing. VectorCAST products support the C, C++, and Ada programming languages.

Vector Software, Inc.

1351 South County Trail, Suite 310
East Greenwich, RI 02818
USA
P: +1 401.398.7185
F: +1 401.398.7186
E: info@vectorcast.com
W: vectorcast.com