

DOT/FAA/TC-14/41

Federal Aviation Administration
William J. Hughes Technical Center
Aviation Research Division
Atlantic City International Airport
New Jersey 08405

Advanced Verification Methods for Safety-Critical Airborne Electronic Hardware

January 2015

Final Report

This document is available to the U.S. public through the National Technical Information Services (NTIS), Springfield, Virginia 22161.

This document is also available from the Federal Aviation Administration William J. Hughes Technical Center at actlibrary.tc.faa.gov.



U.S. Department of Transportation
Federal Aviation Administration

NOTICE

This document is disseminated under the sponsorship of the U.S. Department of Transportation in the interest of information exchange. The U.S. Government assumes no liability for the contents or use thereof. The U.S. Government does not endorse products or manufacturers. Trade or manufacturers' names appear herein solely because they are considered essential to the objective of this report. The findings and conclusions in this report are those of the author(s) and do not necessarily represent the views of the funding agency. This document does not constitute FAA policy. Consult the FAA sponsoring organization listed on the Technical Documentation page as to its use.

This report is available at the Federal Aviation Administration William J. Hughes Technical Center's Full-Text Technical Reports page: actlibrary.tc.faa.gov in Adobe Acrobat portable document format (PDF).

1. Report No. *DOT/FAA/TC-14/41	2. Government Accession No.	3. Recipient's Catalog No.	
4. Title and Subtitle Advanced Verification Methods for Safety-Critical Airborne Electronic Hardware		5. Report Date January 2015	
		6. Performing Organization Code	
7. Author(s) Brian Butka		8. Performing Organization Report No.	
9. Performing Organization Name and Address ¹ Embry Riddle Aeronautical University 600 S. Clyde Morris Blvd. Daytona Beach, FL 32114		10. Work Unit No. (TRAIS)	
		11. Contract or Grant No. DTFACT-11-C-00007	
12. Sponsoring Agency Name and Address U.S. Department of Transportation Federal Aviation Administration 950 L'Enfant Plaza FAA National Headquarters 950 L'Enfant Plaza North, S.W. Washington, DC 20024		13. Type of Report and Period Covered Final Report Aug. 2011 – Sept. 2013	
		14. Sponsoring Agency Code AIR-110	
15. Supplementary Notes The Federal Aviation Administration William J. Hughes Technical Center Aviation Research Division COR was Srini Mandalapu, and the Alternate COR was Charles Kilgore.			
16. Abstract The purpose of this research is to provide safety input to the Federal Aviation Administration for developing policy and guidance for the verification coverage analysis of complex airborne electronic hardware, such as Field Programmable Gate Arrays, programmable logic devices, and application-specific integrated circuits. The recommended verification process is as follows: All designs are reviewed; the design strategy is analyzed and discussed, and the ability of the design to meet all of the requirements is documented. In addition to the design review process, constrained random verification is used to generate large numbers of random test cases that can detect problems with the requirements and remove human biases from the verification effort. Assertions are used throughout the design hierarchy to detect if any violations of the requirements or of the designer's intent occur at any time during the verification process. The requirements-based verification required for DO-254 needs to be extended to a more robust functional verification that considers the combinations of inputs and system state. Robustness tests should explore negative compliance issues to enable the hardware to gracefully recover from unexpected conditions. The verification process recommended in this report includes three coverage metrics: code coverage, assertions, and functional. Using these coverage metrics, coverage targets are proposed for DAL A, B, and C hardware. Formal methods should be applied in parallel to the simulation-based verification. If assertions are used, formal verification of the assertions should be performed. Formal techniques, such as sequential equivalence checking and model checking, should be applied on hardware that is suitable for these analysis techniques.			
17. Key Words Programmable logic, Tools, Coverage metrics, Airborne electronic hardware		18. Distribution Statement This document is available to the U.S. public through the National Technical Information Service (NTIS), Springfield, Virginia 22161. This document is also available from the Federal Aviation Administration William J. Hughes technical Center at actlibrary.tc.faa.gov	
19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 70	22. Price

TABLE OF CONTENTS

	Page
EXECUTIVE SUMMARY	viii
1. INTRODUCTION	1
1.1 Objectives	1
1.2 Research Method	2
1.3 Audience	2
1.4 Document Structure	3
2. IDENTIFY CURRENT INDUSTRY PRACTICES FOR VERIFICATION COVERAGE ANALYSIS OF CEH	4
2.1 Verification Process Overview	4
2.2 Coverage Metrics	7
2.3 Functional Coverage	9
2.4 Assertions	10
2.5 The CRV	11
2.6 Formal Verification	13
2.6.1 Sequential Equivalence Checking	13
2.6.2 Model Checking	14
2.6.3 When Are Formal Methods Effective?	14
2.6.4 Formal Verification in Practice	16
2.7 Verification Process Details	17
2.8 Robustness Testing	19
2.9 Verification Methodologies	20
2.10 Hardware-Based Verification	21
3. INDUSTRY SURVEYS	22
3.1 Design and Verification Language	22
3.2 Verification Methodology	23
3.3 Assertions	23
3.4 Formal Methods	23
3.5 Safety-Critical Verification	23
4. IDENTIFY KNOWN AND EMERGING OBSTACLES, PROBLEMS, OR ISSUES	24
4.1 Issues With Coverage Metrics	25
4.2 Issues With Functional Coverage	25
4.3 Issues With Formal Verification	26
4.4 Issues With Assertions	26

4.5	Issues with COTS IP	27
5.	IDENTIFY POTENTIAL APPROACHES AND CRITERIA TO DEMONSTRATE SUFFICIENCY OF VERIFICATION COVERAGE ANALYSIS OF CEH LEVELS A, B, AND C	27
5.1	Hardware Verification Plan	28
5.2	Simulation-Based Verification Plan	28
	5.2.1 Creating the Functional Coverage Specification	28
	5.2.2 Writing and Running the Test Bench	29
	5.2.3 Coverage Analysis	29
5.3	Formal Verification Plan	30
6.	RECOMMENDATIONS	30
7.	CONCLUSIONS	31
8.	REFERENCES	31
APPENDICES		
	A—Poll of Verification Tools and Coverage Metrics	
	B—Test Cases	
	C—Test Case Very High-Speed Integrated Circuit Hardware Description Language Source Code	

LIST OF FIGURES

Figure		Page
1	The AEH Stakeholders	3
2	Coverage Closure vs. the Number of Tests Generated	13
3	A Typical Hardware Verification Flow	17
4	The Cadence Recommended Verification Process	18
5	How Hardware Verification Is Used in Industry	21

LIST OF TABLES

Table		Page
1	The Top-Two Areas Needing Advancement in the Design and Verification Process	24
2	The Top-Three Challenges in Managing IP	24

LIST OF ACRONYMS

AEH	Airborne electronic hardware
Cadence	Cadence Design Systems, Inc.
CEH	Complex electronic hardware
CRC	Cyclic redundancy check
CRV	Constrained random verification
COTS	Commercial off-the-Shelf
DAL	Design assurance level
DC	Decision coverage
FAA	Federal Aviation Administration
FPGA	Field Programmable Gate Array
FSM	Finite state machine
HDL	Hardware description language
IP	Intellectual property
MCDC	Modified condition design coverage
OVM	Open Verification Methodology
RBV	Requirements-based verification
RTCA	RTCA, Inc., (formerly Radio Technical Commission for Aeronautics)
RTL	Register transfer language
UVM	Universal Verification Methodology
VMM	Verification Methodology Manual
VHDL	VHSIC HDL
VHSIC	Very high-speed integrated circuit

EXECUTIVE SUMMARY

The purpose of this research is to provide safety input to the Federal Aviation Administration for developing policy and guidance for the verification coverage analysis of complex airborne electronic hardware (AEH), such as Field Programmable Gate Arrays, programmable logic devices, and application specific integrated circuits.

RTCA, Inc. (RTCA)/DO-254 [1] defines the verification process used to assure that the designed hardware meets the requirements and identifies additional verification processes that should be performed for design assurance level (DAL) A and B systems. While RTCA/DO-254 identifies potential verification methods for level A and B systems, it does not define any criteria to determine when the verification process is sufficient or complete. This research investigates advanced verification coverage methods suitable for safety-critical AEH, identifies applicable coverage metrics, and proposes verification methods and coverage targets for DAL A-, B-, and C-level hardware. In addition, the need for the qualification of verification tools and the use of commercial off-the-shelf (COTS) intellectual property (IP) are investigated for potential safety issues. It should be noted that, although a wide variety of COTS solutions exist for aviation systems, this research is focused on AEH, which is certified using DO-254.

RTCA/DO-254 requires that all of the functional requirements are verified through a process called requirements-based verification (RBV). The effectiveness of RBV is limited by the quality and precision of the requirements. Automated tools, such as constrained random verification (CRV), can be used to help identify vague or weak requirements early in the design and verification process. Although the verification process can identify weaknesses in the hardware requirements, the verification process must be independent of the requirements specification process. When using RBV, how much testing and analysis is required to claim that a functional requirement has been verified? This research suggests that defining a verification plan that exercises the device for all possible input signals and all possible device configurations is required, and that multiple tests should be run for each combination.

To fully verify hardware, human biases in interpreting the requirements must be eliminated. The CRV process randomly generates conditions that are allowed by the requirements, and often generates unusual combinations that can cause the hardware to fail the requirements. The constraints used by the CRV process can be adjusted or tuned to fill in the gaps in the coverage metrics. The verification process needs to incorporate both human-written directed tests and random tests.

Assertions are used to document the correct operation of signals and also to document the designer's intent for how the register transfer language code should be used. Assertions at the interfaces of modules assure that all of the module's inputs and outputs meet all of the requirements and the designer's assumptions. Knowing the designer's assumptions helps identify problems when the hardware is integrated into the overall system. Assertions are especially useful with COTS IP, assuring that the IP meets the requirements without requiring detailed knowledge of what is contained within the IP. The verification of COTS IP is similar to the verification process used in model-based design.

Formal methods should be used along with the simulation-based verification. This report also recommends that assertions be used in all hardware designs and that formal verification of the assertions be performed. Formal techniques, such as sequential equivalence checking and model checking, should be applied on hardware that is suitable for these analysis techniques. The results of these formal techniques should be independently verified with simulation.

Three coverage metrics are used to assess the completion of the verification process: functional coverage, code coverage, and assertion coverage. Verification is considered to be complete when all three of these coverage metrics achieve their targets. This research proposes that the verification processes used for DAL A, B, and C hardware are quite similar. The difference between level C and level B is that the coverage targets for level C will be lower. Level A takes level B and adds additional robustness tests including negative compliance tests.

1. INTRODUCTION.

Modern aviation systems, both airborne (e.g., avionics, engine control) and ground (e.g. radar, air traffic control consoles), exemplify safety and mission-critical dependable systems. These systems continue to become more complex, and they often operate in uncertain environments. In addition to being correct, the hardware needs to be robust, handling any unusual conditions allowed by the requirements as well as gracefully recovering if conditions outside of the requirements occur.

This report, produced under a contract sponsored by the Federal Aviation Administration (FAA), describes research focusing on the verification process and verification tools used for airborne electronic hardware (AEH) devices, such as programmable logic devices and application-specific integrated circuits . The scope of this research has been limited to the verification process and focuses on both simulation-based and formal verification processes and coverage metrics. This research does not address such subjects as requirements tracing, bug tracking, and version management.

1.1 OBJECTIVES.

The main objective of this study is to provide the FAA with input on what verification process should be used and what criteria should determine completeness of the verification process for design assurance level (DAL) A, B, and C hardware. The following questions will be addressed:

1. What approaches are being used to demonstrate sufficiency of verification coverage of complex electronic hardware (CEH)? That is, how can it be shown that the embedded logic on the chip has been fully exercised and tested?
2. What are appropriate verification criteria applicable to CEH levels A, B, and C?
3. What are the safety issues with current and emerging industry practices and approaches to verification coverage analysis to CEH? How can these safety issues be mitigated?
4. What verification coverage approaches and criteria applicable to CEH will provide a level of confidence similar to DO-178B requirements-based test coverage and structural coverage analysis of software?
5. What are the obstacles that the industry is currently experiencing in its efforts to demonstrate verification coverage of CEH (e.g., lack of mature tools, complexity of the CEH, embedded commercial off-the-shelf (COTS) intellectual properties (IPs) within application specific CEH devices, etc.)? What is the industry doing to overcome these obstacles?

1.2 RESEARCH METHOD.

This research contains four major components:

1. Conduct a literature search and an industry survey, as appropriate, and document reference resources based on the results.
2. Identify current industry practices for verification coverage analysis of CEH.
3. Identify known and emerging obstacles, problems, or issues that the industry faces when attempting to perform verification coverage analysis of CEH, as well as the industry's recommendations for addressing these obstacles.
4. Identify potential approaches and criteria to demonstrate sufficiency of verification coverage analysis of CEH levels A, B, and C, and provide a model checking similar to DO-178B requirements-based test coverage and structural coverage analysis of software.

1.3 AUDIENCE.

The report is primarily intended for use by certification authorities in the development of policy and guidance. The designated engineering representatives and aircraft certification office engineers directly involved in the certification process are also part of the target audience. The research outcome will likely also be of interest to program and procurement managers; project leaders; system, hardware, and software engineers; and to all others directly involved in DO-254-compliant AEH projects. Figure 1 identifies the stakeholders involved in the presented investigation. It must be noted that several industry representatives shared their valuable comments and opinions with the research team through e-mails, phone interviews, and personal contacts; their names cannot be listed for reasons of confidentiality.

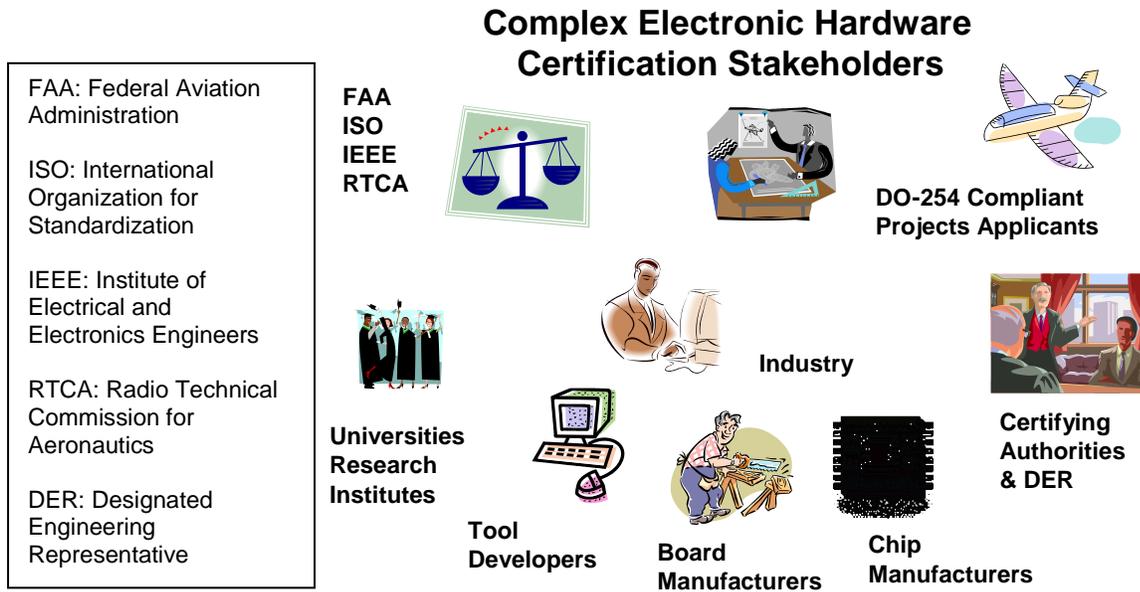


Figure 1. The AEH Stakeholders

1.4 DOCUMENT STRUCTURE.

This report consists of eight main sections:

1. Section 1 provides introductory material, including the purpose, scope, objective, and audience.
2. Section 2 describes verification practices used for CEH. Major verification tools and techniques are identified, and multiple coverage metrics are discussed.
3. Section 3 describes the results of the literature search and industry survey.
4. Section 4 examines safety issues and problems in the verification process.
5. Section 5 proposes verification processes and coverage metrics for DAL A, B, and C hardware.
6. Section 6 presents the recommendations and identifies the areas to be addressed in the remaining work.
7. Section 7 presents the conclusions of the research.
8. Section 8 provides the references.

There are three appendices accompanying the body of the report:

1. Appendix A contains the results of the industry survey.

2. Appendix B contains the results of the ambiguous requirements and COTS IP test cases.
3. Appendix C contains the very high-speed integrated circuit (VHSIC) hardware description language (HDL)—referred to as VHDL—code used for the test case implementations.

2. IDENTIFY CURRENT INDUSTRY PRACTICES FOR VERIFICATION COVERAGE ANALYSIS OF CEH.

RTCA, Inc. (RTCA)/DO-254 details the verification processes to be used in the certification of CEH in the aviation industry. The safety record of the RTCA/DO-254 is very good. However, the document was written in 2000, and AEH designs have increased in complexity since that time. This increase in hardware complexity dramatically increases the risk of an error slipping through the verification process. The RTCA/DO-254 process is known for describing what needs to be done, but providing limited guidance on how it should be done. This section focuses on how the verification process is performed within the industry, beginning with a survey of the relevant verification processes and an examination of how they are applied. This material is based on the results of the literature search and industry poll, which is summarized in section 3.

Although the goals of this research focus on the use of coverage metrics to determine when the verification process is complete, the entire verification process must be examined as a whole. Coverage metrics should only be used to assess the completion of the verification process. If coverage metrics become the driver of the verification process, engineers may write tests that improve the coverage metrics, but do not improve the verification of the hardware. Unfortunately, 100% coverage on any coverage metric cannot assure that the design is correct.

In 2006, Gluska [2] reported that only 4% of all of the bugs on a major microprocessor design were found using code coverage metrics. Gluska noted that there were also bugs that were caught for indirect reasons, such as improvements to the verification process that were required to use the code coverage metrics. Including both direct and indirect components, the total number of bugs found by code coverage metrics was 8%. It must be recognized that code coverage metrics are most useful for identifying obscure design errors. The majority of design errors will be found using other methods, such as requirements-based and functional verification.

2.1 VERIFICATION PROCESS OVERVIEW.

In addition to the author's industry poll summarized in section 3, information from much larger surveys of the verification process has been published by Synopsys [3], Aldec [4], and the National Microelectronics Institute of the United Kingdom [5]. Including the verification process descriptions from published sources [2, 6, and 7] leads to the following broad conclusions:

- Semiconductor companies building diverse products use a similar verification flow.
 - There are company-to-company variations in the rigor of any given process.
 - Formal methods are used by roughly half of the companies.

- Aviation-related companies follow the verification process specified by RTCA/DO-254 with some additional verification work performed, but usually not claimed, for certification credit.
- The verification flow used by the semiconductor industry contains elements that are not required in the RTCA/DO-254 verification flow.

In a highly abstracted view, a typical RTCA/DO-254 verification flow for DAL A hardware consists of verifying that the designed hardware meets all of the requirements. This is referred to as requirements-based verification (RBV). In addition to RBV, such analyses as elemental analysis, formal analysis, and safety-specific analysis are performed. Because automated tools exist to assist with elemental analysis, it is the most common method of additional verification. It is usually performed as a structural coverage analysis. In many respects, this process is a parallel to RTCA/DO-178's [8] verification requirement for software.

Focusing on just the verification process, the RTCA/DO-254 verification flow for a DAL A design can be summarized as follows:

- RBV—Use directed test vectors to verify all requirements.
- Elemental analysis—Several metrics may be used, but statement coverage best matches the requirements of elemental analysis.
- Formal analysis is suggested for hardware with concurrency or fault tolerance.
- Robustness testing is added to assure the device functions correctly in all legal conditions.

The semiconductor industry verification flow for any hardware (rarely safety critical) can be summarized as:

- RBV—Use directed test vectors to verify all of the requirements. The requirements are verified for all input types and device configurations.
- Assertions are used as a debug tool and also as a coverage metric.
- Multiple coverage metrics are used, including:
 - Statement coverage
 - Branch coverage
 - Expression coverage
 - Path coverage
 - Finite state machine (FSM) coverage
 - Toggle coverage
 - Assertion coverage
 - Functional coverage

- Constrained random verification (CRV) is used to identify corner cases and problems in the manufacturer’s requirements. Corner cases are conditions that push one or more input variables to their minimum or maximum values to explore a “corner” in the multi-dimensional test space.
- Formal methods typically target interfaces; control and data hardware; and other hardware that is difficult to verify with simulation.
- Robustness testing is used to assure the device functions correctly in all legal conditions.
- Additional robustness testing is performed to assure that the device handles illegal conditions gracefully. Note that how a system handles illegal conditions is often not specified.
- A public verification methodology is used. The methodology provides standard verification tools and constructs as well as standard verification code for known interfaces. Common methodologies are:
 - Verification Methodology Manual (VMM)
 - Open Verification Methodology (OVM)
 - Universal Verification Methodology (UVM)

These methodologies will be elaborated on and discussed later in this section.

From the above summaries, we can see that a non-safety critical hardware device in the semiconductor industry is verified using a process that contains tools and techniques beyond what is required to certify DAL A hardware within the aviation industry. This does not mean that aviation hardware is not verified as well as semiconductor parts. Aviation-related companies can and do use verification processes similar to the semiconductor industry; however, these processes may not be used for verification credit in the certification process. For example, an inhouse-developed verification tool may prove valuable in practice, but it may not be able to meet the necessary tool qualification requirements for certification credit’s use.

The semiconductor industry’s verification process is driven by the fact that a mask set is needed to produce a part. With mask sets currently priced at over 6 million dollars, design errors can cost millions of dollars. There is a huge financial incentive to get the design correct on the first attempt. The high cost of mask sets prompts semiconductor companies to use a comprehensive verification process. The semiconductor industry seeks to assure design correctness to avoid the financial costs of an error. The aviation industry strives to assure design correctness to provide the safest hardware possible. Although the motivations are different, the goal of assuring a correct design is common for the two industries.

The number of companies in the semiconductor industry dwarfs the number of companies in the aviation electronics industry. In addition, while design cycles can approach decades in the aviation industry, the semiconductor industry’s product design, verification, and test cycles are short—often less than 18 months from the start of a design to shipping the product. This means

that in any year, the semiconductor industry produces many times the number of designs produced by the aviation industry. If there are weaknesses in the verification tools or the verification process, the semiconductor industry will encounter them and find solutions first. The aviation industry can use verification trends from the semiconductor industry as a guide in determining the most effective verification processes.

When a semiconductor company considers using a new design or verification tool, it is typically evaluated using known problems from previous designs to see if the new tool solves the old problems. Because there is a large worldwide user base, most tools have a substantial service history that can be used to assess the correctness of the tool. Unless a design requires absolute state-of-the-art tools, most companies run a revision or more behind the latest tool releases to minimize the probability of a hidden tool bug. In the semiconductor verification process, the verification tools check the output of the design tools on a daily basis. This frequent cross-checking of the design and verification tools provides confidence in the correctness of both tools.

2.2 COVERAGE METRICS.

The key to improving any process is the ability to measure it. For years, verification engineers have used code coverage metrics to measure the completeness of the verification effort. We will begin the discussion of coverage metrics with the structural coverage metrics identified in RTCA/DO-178, and then map them to their hardware equivalents.

RTCA/DO-178B [8] identifies three primary structural coverage metrics:

1. Statement coverage: Every statement in the program has been invoked or used at least once.
2. Decision coverage (DC): Every entry and exit point in the program has been invoked at least once. In addition, each decision in the program has been taken on all possible outcomes (true/false) at least once.
3. Modified condition decision coverage (MCDC): Every entry and exit point in the program has been invoked at least once. Every condition in a decision in the program has been taken on all possible outcomes at least once. In addition, to avoid conditions masking one another, each condition is varied individually while holding all other conditions fixed.

Hardware verification contains a set of metrics for measuring coverage. Metrics that measure coverage of the register transfer language (RTL) code are referred to as code coverage metrics. The statement coverage metric used for hardware is equivalent to statement coverage in RTCA/DO-178. The branch coverage metric used for hardware is equivalent to the DC in RTCA/DO-178. The expression coverage metric used in hardware is equivalent to MCDC in RTCA/DO-178.

Beyond the three metrics described above, additional metrics are often used to assess hardware verification completeness [9]. Path coverage is an abstract RTL metric requiring that every leg of the coverage flow graph be traversed at least once. There are also coverage metrics unique to

a hardware implementation. Toggle coverage requires every signal in the device to toggle (transition from a 0 to 1 and from a 1 to 0) at least once during the test bench. The FSM coverage requires every reachable state of every FSM to be reached and every transition between states to have been executed at least once. The need for these additional metrics is addressed when safety issues are discussed.

The RTCA/DO-254 document requires verification of all of the device requirements. Hardware verification often includes a high-level metric called functional coverage, which is a measure of how many of the device functions have been verified. Note that device functions can be tested with differing levels of rigor. For example, a statement coverage point of view of functional verification might declare a function tested if it is verified once. An MCDC point of view might require that all possible combinations of inputs and outputs have been tested. The same hardware test bench might achieve 100% functional coverage from a statement coverage point of view while achieving less than 10% coverage from an MCDC point of view. In the semiconductor industry, functional coverage embraces a rigorous DC or MCDC interpretation of the requirements.

Functional coverage and code coverage are complementary metrics, with neither being sufficient to assure design correctness. Functional coverage involves testing the device performance at the transaction and function level, whereas code coverage involves the detailed implementation of the RTL. If a design neglects to implement a function, it is possible to have 100% code coverage of the RTL, but still have design functions that have not been exercised. Conversely, it is possible to have 100% functional coverage, but have poor code coverage. This is because there are many ways that each function can be invoked, and only a few of the possibilities may have been exercised. In this process, the verification effort is complete when the coverage goal has been achieved for both code coverage and functional coverage.

Not every hardware design can use the same code coverage metrics. It makes no sense to apply FSM coverage to a design with no FSMs. Conversely, in a predominantly synchronous design containing many FSMs, coverage-based metrics, such as branch and expression coverage, offer little insight into how well the test bench has verified the hardware. There are wide variations in design implementations, and it is impossible to create an a-priori solution that will work for all designs.

Formal verification techniques can be applied along with the simulation-based verification methods discussed previously. Formal verification is often used to prove that certain properties (or requirements) are true for the hardware. Integrating formal techniques into the verification flow is difficult because assessing how well a block has been analyzed by a formal tool is cumbersome. Independent verification of the formal analysis results using simulations is typically performed.

In summary, statement coverage is the most commonly used coverage metric. It is simple to measure and can easily identify unreachable code, but it is a weak indicator of verification completeness. Complete expression coverage is a good indicator of logical coverage completeness for many designs, but complex designs can generate enormous quantities of data that must be analyzed. Depending on the hardware architecture, these designs may be suitable

for verification via formal methods. Toggle coverage is heavily focused on hardware implementation and should be applied late in the design process when the hardware is believed to be correct. Any signals that did not toggle indicate either an incomplete test bench or unused hardware. Designs containing FSMs need to use the FSM coverage metric to assess the completeness of coverage. Finally, functional tests based on the requirements are already required by RTCA/DO-254. The functional coverage metric allows the user to assess the completeness of the test bench with respect to the requirements.

2.3 FUNCTIONAL COVERAGE.

Although functional coverage has been discussed as a coverage metric, defining the functional coverage points is a key element in writing the verification plan. The verification plan should be written to address which device functions need to be tested and also under which conditions the functions are to be tested. The functional coverage metric is then used to measure how many of the identified device functions have been tested [2, 10, and 11]. When defining the functional coverage points, all of the functional requirements must be addressed, as well as higher-level completeness concerns, including the following questions:

- Were all possible input stimuli variations injected?
- Were all possible output conditions achieved?
- Did all possible internal state transitions take place?
- Did all the interesting events occur?

The last question is often the most important, but is the hardest to quantify. The function and implementation of the device determines which events are interesting; they are often concurrent conditions with multiple input conditions occurring simultaneously. Other interesting events often concern timing variations and signals crossing clock domains. Another interesting event involves looking at conditions that caused failures in previous designs. Although technically not a functional test, negative compliance tests are often added to assure design correctness and design robustness. Negative compliance tests question how the hardware operates when conditions outside of the requirements are applied.

For example, a packet-based interface functional verification would address such questions as:

- Have all the packet lengths been used?
- Have packets with good and bad cyclic redundancy check (CRC) results been used?
- Were the buffers tested in the full and empty states?
- Can the buffer-full bug that we saw 2 years ago occur in this design?
- Did the buffer status signals occur on the correct clock edges?

In a system with packets between 1 and 10 bytes in length, the packets can be sent to two addresses, with each packet containing CRC data. Basic functional coverage would require packets with lengths from 1 to 10 bytes to be generated and sent to one of two addresses with good or bad CRC data. By using cross parameters, functional coverage can require that packets of all sizes containing both good and bad CRC data have been sent to both addresses. It should

be noted that this type of hardware is well suited to formal analysis and that formal proofs of the hardware correctness should also be performed.

Functional coverage is a superset of requirements-based testing because it considers not only that a requirement has been tested, but also the conditions under which the requirement has been tested [12, 13]. The device state is also an important parameter. For example, the state of the hardware when an interrupt occurs is critical to determining how the interrupt is handled.

In summary, functional coverage is used to create a verification plan containing all possible input combinations and device configurations. When the hardware is suitable, formal methods should be used to prove the correctness of the hardware. Formal methods prove the correctness of hardware with respect to the rigorous mathematical properties and constraints that have been applied. However, the translation of human written requirements to the precise mathematical properties and constraints was not verified. Therefore, the outputs of formal methods must be independently assessed and sanity checked. All other hardware needs to be fully verified by simulation. For simulation-based verification, a combination of randomly generated tests and directed tests are performed. The completeness of the functional testing is assessed as the percentage of functional tests that are complete in the functional coverage plan.

2.4 ASSERTIONS.

Assertions are comments that are put into the RTL code and are continuously checked during simulation. Most assertions are derived from the requirements, such as, “The ready signal must go high three clock cycles after the reset signal transitions from low to high.” Without assertions, an error is detected only if it affects a monitored output signal. When assertions are placed throughout the hierarchy of a system, errors can be observed, even if they do not affect an output or other monitored signal.

Most assertions are generated from the requirements as the RTL code is being written. The designers can also capture low-level indirect requirements and assumptions by writing appropriate assertions. These assertions document the designer’s intent regarding how the elements, interfaces, and control logic are intended to function. A simple example would be an assertion that the maximum value of an internal counter is 10. Because this counter is not directly visible to the outside world, errors in the counter may occur without impacting a monitored signal. With assertions, if the counter exceeds ten in any simulation, the assertion will fire, making this error visible. If any assertion fires, either the hardware requirements or the designer’s intentions will have been violated. Further analysis may reveal that the underlying cause was a hidden error, or it may be that there is no issue and the assertion needs to be adjusted. In either case, the assertion is the key to identifying a potential error.

Assertions written by the designers tend to occur at low-level blocks in the system. Top-level assertions are often included in standard interface IP or are written by the verification engineers. Higher-level assertions are typically not focused on the RTL code within a block, but with the interface and control of the block. Examples of high-level assertions include those assuring that packets are correctly acknowledged and that signals indicating when data is valid occur according to the specification.

Writing assertions at the interfaces between RTL blocks is critical [14]. Both the sending and receiving blocks need to follow the same interface protocol. When using standard verification methodologies, common interface protocols have existing verification IP that can be used for this purpose. This IP not only contains assertions that assure correctness of the interface protocol implementation, but also contain scoreboards that allow the user to assess the completeness of the test suite.

The ability of assertions to increase the observability of the design can also dramatically reduce debug time. In cases when formal analysis or hardware testing has identified a bug, the defect in the hardware is observable only when the error reaches an output. The root cause of the error could have occurred long before the error propagates to an output. This means that debugging the errors detected in the hardware testing is often quite difficult and can lead to solutions that address the observed symptom but fail to address the underlying root cause. When assertions are used, a simulation of the test bench will generate a list of violated assertions. The list provides a map of how the error was generated and how it propagated through the design. This allows for the identification of all impacted modules, and it dramatically speeds up the debug process. Reducing the time spent debugging increases the time that can be spent searching for new bugs.

Although the original purpose of assertions was to document the requirements and the designer's assumptions, they can also be used as a coverage metric [15]. Because assertions are written throughout all levels of the hierarchy, they provide information about how well the hardware has been tested at all levels. To use assertions as a coverage metric, we need to assure that there is an adequate density of assertions throughout the design. Tools that measure the logical distance of hardware from the nearest assertion are available, and they can highlight areas with limited coverage from assertions.

The verification process can assess not only if an assertion has fired, but also whether an assertion has ever been evaluated. Because assertions reflect both the requirements and the designer's intent, an unevaluated assertion indicates a gap in the test bench coverage. An example might be an assertion indicating correct operation of a dual-port memory under the condition of both ports simultaneously attempting to write to the same address. If this assertion has not been evaluated, then the test bench has not attempted to write to the same address on both ports at the same time. It may be possible that this system implementation can never write to both ports simultaneously; if so, this should be noted in the documentation. However, if this condition can occur, a test case for this condition should be generated. Modern tools are automating both the process of writing assertions [16] and the use of assertions as a coverage metric. Assertions allow hard-to-reach and hard-to-verify logic or critical functionality to be identified as coverage goals.

2.5 THE CRV.

Coverage metrics provide a way to count interesting events that occur in the verification process, regardless of the metric. This list of events that occurred is compared to the list of events that were specified in the verification plan, and then a measure of verification completeness is produced. Although it is possible to hand code directed tests and achieve 100% code coverage, it is known that humans impose biases into their interpretations of the requirements. Ambiguous requirements may not be seen as such until after the design is complete. This means that legal

but unanticipated combinations of the inputs and the system state can produce errors that slip through the test bench. To remove human biases from the verification process, constrained random stimulus generation is used. The CRV tools assume that any condition that is not specifically disallowed by the requirements is valid, and then randomly generates test cases using these rules. If a requirement is vague, the constrained random process can generate test conditions that highlight this problem.

When using a device that receives data in packets over a serial communication bus, the size of the packets depends on which device is sending the packet, as well as the type of data contained within the packet. In addition, the packets can arrive at arbitrary times. With a system of this complexity, even the designer with full knowledge of the hardware implementation may be unable to determine the worst-case scenarios. For example, the worst case for the hardware could be several small packets followed by a large packet, or it could be a large packet immediately following a packet with a transmission error. The device operation depends not only on the input packet, but on when the input packet arrived and the history of previous packets. In the absence of the ability to identify all of the possible problem conditions, it is best to test as much of the input space as possible. Automated CRV was developed to address this issue.

The CRV's lack of human biases allows it to excel at finding unusual conditions that escape human analysis. For example, most components have a reset signal, and testing the functionality of that signal is required. The author has seen hardware that functioned correctly when a reset was applied, but malfunctioned if a reset was immediately followed by another reset. There was no reason in the normal operation of the hardware that multiple sequential resets could not occur. Although neither the designers nor the verification team identified this condition as being a problem, constrained random tests generated the condition that identified this failure.

Because constrained random test case generation is automated, large numbers of test cases can easily be generated. This allows the device functions and RTL code to be exercised repeatedly in varying conditions. The randomness that is CRV's strength is also its Achilles heel. For most coverage metrics, CRV rapidly achieves 80% coverage and then asymptotically approaches 100%, as shown in figure 2. The new, randomly generated test cases begin to overlap previously tested conditions. The slow convergence of random conditions to complete coverage is well known and referred to as the "coupon collector problem" [17]. In a scenario for which there are 1,000 possible test conditions, initially every test case generated is unique and the coverage increases rapidly. However, each new test case increases the probability that the next test case generated will be the same as one already covered. For the example of 1,000 possible test conditions, on average, more than 7,000 cases will need to be generated to achieve 100% coverage.

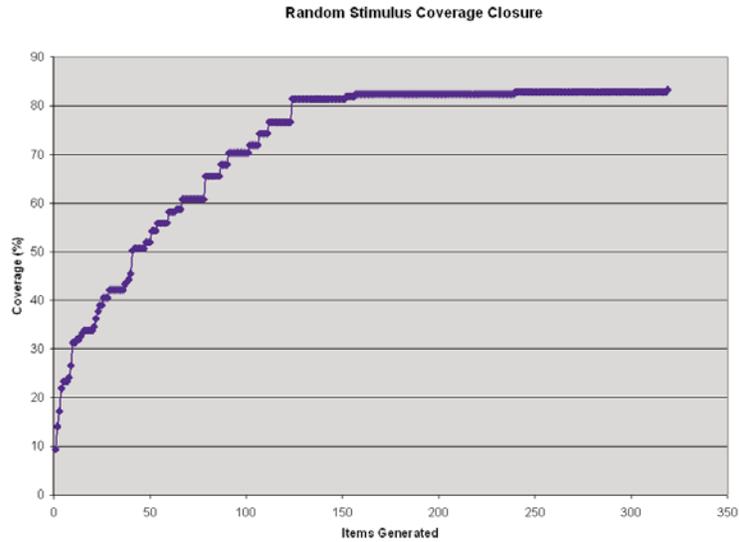


Figure 2. Coverage Closure vs. the Number of Tests Generated

Adding coverage-based metrics to the test case generation process improves constrained random testing. The coverage metrics are then used to steer the parameter constraints to target untested conditions. Modern tools, such as Mentor Graphics® inFact™, can automate the parameter adjustment process to significantly speed up the coverage closure process.

2.6 FORMAL VERIFICATION.

Formal verification confirms the correctness of a design with respect to a specified behavior by checking whether the labeled state-transition graph that models the design satisfies the specified behavior. Formal methods offer the ability to prove that hardware is correct for all possible combinations of the inputs and system state. A key feature of formal verification is that if a design is found to violate a requirement, an example demonstrating how the violation can occur is produced. Examples of areas in which formal verification has proven its value include proving the correctness of design interfaces, clock domain crossing circuits, and FSM implementations.

Formal verification proves that a design property holds for every point of the search space. There are two major formal verification approaches, both of which are complementary: sequential equivalence checking and model checking. The sequential equivalence approach checks that two models are equivalent, and the model checking approach proves that certain properties are true for the model.

2.6.1 Sequential Equivalence Checking.

Sequential equivalence checking is used to formally compare two models. These models can be quite different in structure. For example, one model could be a reference model for the design and the other the RTL implementation of the model. The reference model could be written using unsynthesizable behavioral RTL constructs or could even be written in a different language, such as C.

Sequential equivalence checking can prove that the two models are equivalent with respect to defined properties, such as the behavior of the output signals. The analysis needs to be constrained to prevent degenerate cases from interfering with the proof. A simple restriction could be a constraint that the clock signal will always toggle and that the enable signal must change.

In hardware design, there are often multiple ways to implement the same logical function. These implementations offer tradeoffs that vary with respect to performance, power dissipation, and area. As the design nears completion, it is common for the hardware implementation to shift to meet timing or power requirements. If the previous design was known to be logically correct, it is much faster to prove that the new implementation is equivalent to the old implementation by using sequential equivalence checking than it is to prove that the new design is logically correct.

Common models that are compared include:

- A behavioral model vs. the RTL implementation
- The RTL implementation vs. the pre-routing netlist
- The pre-routing netlist vs. the post-routing netlist
- The post-routing netlist vs. a new revision of the post-routing netlist

Equivalence-checking methods can prove that two models are equivalent, but the methods provide no guidance toward whether the model is correct. Model-checking techniques are used to prove the correctness of a model.

2.6.2 Model Checking.

Formal verification tools can be used to verify the correct operation of many control and datapath blocks by proving that the designs meet a set of formally defined properties. These tools are known as model checkers and perform an exhaustive state-space search to prove that the properties are true under all conditions. They explore the state space for all possible corner cases and provide examples that demonstrate how a violation can occur. The output of a model checker is one of three possible results:

1. The properties have been proven to hold.
2. There is a known failure and a counterexample is given.
3. The system could neither prove nor disprove the properties.

The third result is problematic because it provides no information.

2.6.3 When Are Formal Methods Effective?

Formal methods are best suited to more abstract, high-level models. As the complexity of the model increases, the formal methods lose their ability to exhaustively search the state space. The primary limitation of these models is the amount of memory available to the processor and the amount of CPU time that can be devoted to the analysis.

DO-254 Appendix B 3.3.3 states, “Formal methods may be applied to the whole design or they may be targeted to specific components” [1]. The document goes on to suggest, “Protocols dealing with complex concurrent communication and hardware implementing fault-tolerant functions may be effectively analyzed with formal methods.”

In the current state of the art, it is rare that formal methods are successfully applied to the whole design. It may be possible to apply formal methods to a high-level model of the design, but low-level hardware is usually analyzed in smaller blocks.

Concurrent hardware is characterized by hardware for which multiple input streams of data arrive at arbitrary times and can collide with each other. An example of this type of hardware would be an Ethernet router. The input data arrives at arbitrary times and is transferred to the correct output without modification. This type of hardware is difficult to analyze in simulation because of the arbitrary data timings, but is well suited to formal verification.

As opposed to concurrent hardware, sequential hardware manipulates the input data through a sequence of hardware operations. Because nearly every register in the device is involved in processing the data, the state space grows exponentially large. Sequential hardware, such as a digital signal processing circuit, is usually best analyzed via simulation and proves difficult to verify using formal methods.

Fault-tolerant hardware requires a hardware failure to occur to test if fault tolerance is operating correctly. This makes it difficult/impossible to validate fault-tolerant hardware in the final system, and it is also difficult to verify in simulation. Proving the correctness of the hardware using formal methods is an excellent approach to this problem.

Control circuits (e.g., memory controllers and interrupt controllers) that do not directly modify data, but control the operation of the hardware, are well suited to formal verification.

Data-transport circuits that move data without performing mathematical operations on the data are also suited to formal verification. Data-transformation circuits that perform mathematical operations on the data are less suited to formal verification.

Foster’s DVCon paper in 2006 [18] provides guidance on the types of hardware that are suitable for analysis with formal methods. Examples of hardware that are suitable for formal verification include:

- Input/output interfaces, especially standard interfaces
- Arbiters
- Bus bridges
- Power management units
- DMA controllers
- Host bus interface units
- Scheduler controllers
- Clock gating
- Interrupt controllers

- Memory controller

Examples of hardware that is not well suited for formal verification [18] include:

- Floating point units
- Graphics processors
- Convolution unit in a signal processor
- MPEG decoders

2.6.4 Formal Verification in Practice.

Formal methods can be applied at varying levels of intensity depending on the experience of the verification team. The method requiring the least expertise in formal methods is proving that the assertions are never violated [19]. Assuming the design is instrumented with well-written assertions, most verification tools with formal capabilities automatically attempt to prove all of the assertions. Because most assertions have a very local scope, they are well suited to formal analysis. Proving assertions may require the application of constraints for the proofs to complete.

Another common use of formal verification is the formal analysis of the interfaces. This focuses on the interface requirements and not on the data passing through the interface. Interface verification is usually done at a high level of abstraction and is well suited to formal analysis. For many common commercial interfaces, the verification tool vendor can provide the properties and constraints needed to perform this formal analysis.

The most common use of formal verification tools is the use of model-checking tools to prove properties in support of finding bugs. Rather than attempting to prove that hardware is correct for all properties at all times, this technique is used to prove limited properties under restricted conditions. This may include limiting the depth of the state-space search or using simulation to reach a particular state, and then exploring the state space using formal methods. Any bugs that are identified are real, and this technique offers additional assurance, but not proof, that the hardware is correct.

The properties to be proven using formal methods are chosen based on a ranking given in the verification plan. Properties that would have a high rank for formal verification may include such factors as:

- A previous project had a bug in this area.
- The property is difficult to cover using simulation.
- The hardware is conducive to proving this type of property (i.e., control hardware).

It should be noted that properly constraining the model to prove properties using model checking can require knowledge that can only be provided by the designer. This may be impossible if COTS IP is used. The exchange of information needed to constrain the model can require a communication path between the design and verification teams that must be structured to limit the possible loss of independence between the two efforts.

It is possible to perform a full proof of the correctness of the hardware in applications for which the hardware is suitable for formal verification and sufficient expertise with formal verification exists. This ultimate goal of formal verification has proven difficult to achieve in practice.

2.7 VERIFICATION PROCESS DETAILS.

The verification process begins by defining the verification plan. The planning process begins by determining the goals of the verification process. These goals will include verification of all of the requirements as well as coverage targets for multiple metrics. Because it is not possible to fully verify a complex design, these goals are prioritized in terms of their impact on system safety; whether the relevant hardware is proven or a new design; the complexity of the hardware; and many more criteria.

A typical verification process is shown in figure 3. The verification methods are divided between those based on simulations of test benches and those that are formal. Simulation-based verification includes directed tests coming directly from the requirements, random tests to cover corner cases, and assertions to assure that the design meets the designer's intent. Formal verification methods include sequential equivalence checking to prove model equivalence and model checking to prove properties.

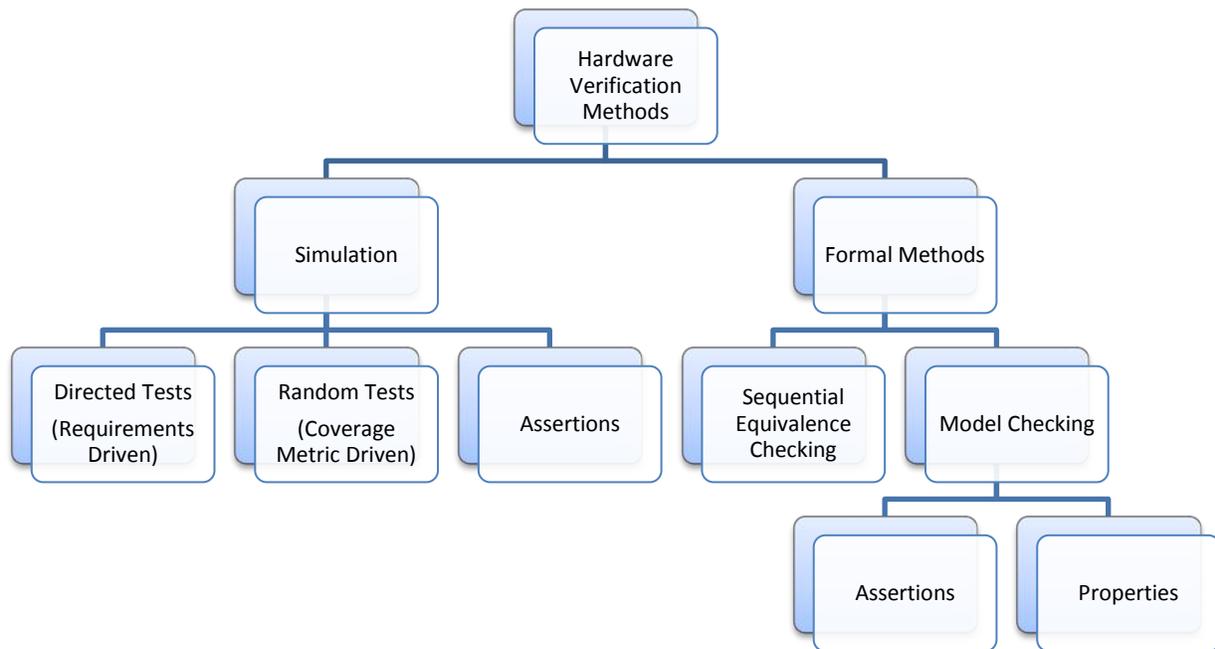


Figure 3. A Typical Hardware Verification Flow

The verification goals identified in the verification plan are analyzed to determine which verification method will best achieve this goal. The distinction between verification by simulation and verification by formal methods is significant. The tools and expertise required to perform formal analysis are quite distinct from those used for simulation, so these tasks will be

assigned to separate teams. The verification plan is not fixed. If a goal assigned to formal verification proves unsuited to formal techniques, it can be covered with simulation. Conversely, if the use of simulation makes the achievement of coverage metrics difficult, a goal may be added to the formal verification plan.

Assertions are used in both the simulation-based and the formal-method-based verification flows. The assertions may come from high-level requirements and be written by the verification team, or they can be written by the designers to document low-level design assumptions.

Common industry practice for the simulation-based verification process and verification coverage analysis is shown in figure 4. The figure comes from a recent Cadence Design Systems, Inc. (Cadence) webinar on state-of-the-art verification processes [20]. Although not a major supplier of tools to the aviation industry, Cadence is the world’s largest design and verification tool supplier.

The arrows at the top of figure 4 show that the test bench development begins at the same time as the RTL design. The verification test bench development moves up through the hierarchy in parallel with the RTL development.

There are several major events on the timeline. “Feature/protocol finished” means the RTL designers believe the block is complete. “Bug rate leveled” indicates the rate of bug discovery is asymptotically approaching 0.

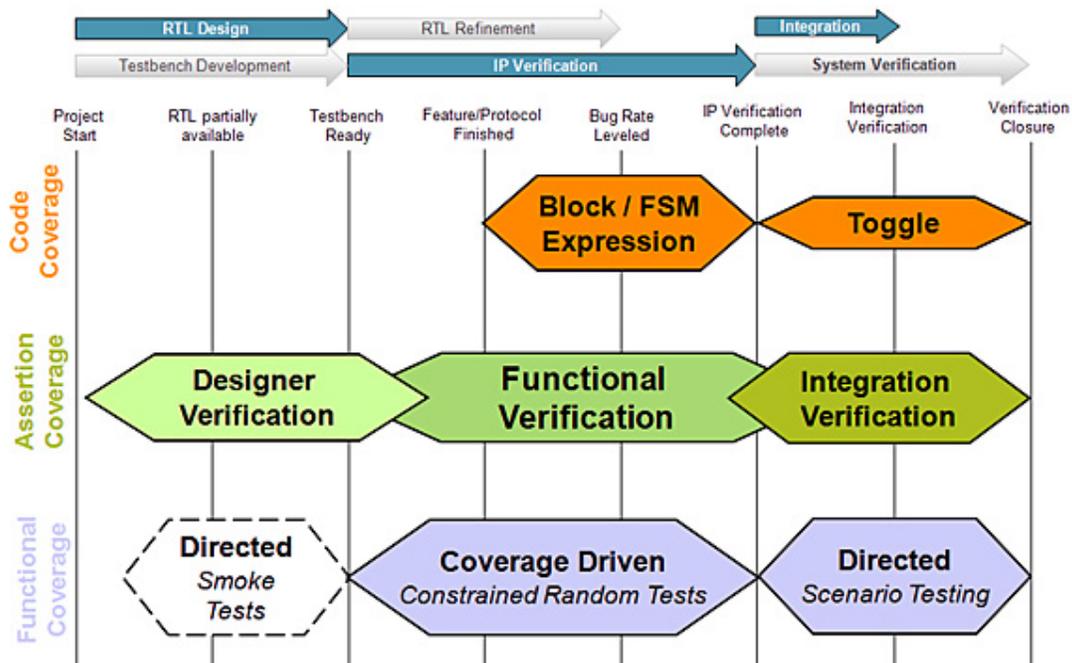


Figure 4. The Cadence Recommended Verification Process

Notice that unlike RTCA/DO-254, for which there are RBV and code coverage requirements, there are three distinct coverage regimes identified: code coverage, assertion coverage, and

functional coverage. Before the verification team has written the test bench, verification is still occurring for the tests that the designer has written for specific purposes. Because the designer's test benches lack independence, they are not counted toward verification completion and are shown within a dotted line labeled "smoke tests." Once the test bench is written, functional verification and constrained random tests begin. Ambiguous or poorly written requirements are often identified as part of the verification process.

Once the RTL designer believes a feature is complete, code coverage metrics are run. In this case, block coverage (which is another name for statement coverage), FSM, and expression coverage are used. As the major features are completed and meet the coverage goals, final integration begins. At that time, the design is mature enough to apply toggle coverage. Any holes in the toggle coverage or the functional coverage are addressed with directed tests.

Coverage metric completeness determines when the verification process is complete. Ideally, a part would require 100% of the code coverage, assertion coverage, and functional coverage metrics. Other coverage metrics are monitored, but may or may not be used to determine verification completeness. Because time to market is critical in the semiconductor industry, parts may be shipped with less than 100% coverage on all metrics. Typical verification completeness criteria would be 95%-97% code and assertion coverage, and as much functional coverage as the schedule will allow.

Verification engineers and managers were asked in a poll and in personal interviews about how their verification flow would differ between a nonsafety-critical part and a safety-critical part. The overwhelming response was that the two verification processes would be identical, differing only in the coverage targets and the rigor applied to the robustness testing. A nonsafety-critical part could ship with less than 100% coverage. This was viewed as a business decision for which the risk of a design error requiring a new mask set is balanced against the cost of being late to the market. A safety-critical part would target 100% coverage on all metrics used. It should be noted that 100% coverage in any of the metrics can be impossible to achieve in certain implementations. In those cases, an analysis explaining why 100% coverage cannot be achieved is required.

2.8 ROBUSTNESS TESTING.

Robustness testing has two major thrusts. The first is to assure that the product operates correctly under all legal operating conditions. This includes voltage and temperature variations as well as variations in other operational parameters, such as clock speed and throughput. The second thrust is often referred to as negative compliance verification. This thrust concerns itself with how the system operates when subjected to conditions that are outside of the requirements. Most protocol documents are very precise on what constitutes valid inputs and valid configurations. However, the documents often do not address how the system should respond to conditions outside of the system requirements. For example, serial protocols—such as Arinc-429, RS-232, and I2C—define special signals known as start and stop bits to delimit the beginning and end of a data transmission. The requirements state that the system will begin a transmission with a start bit followed by the data, and then the transmission will be terminated with a stop bit. But how should a system handle the case if start bits and data are received without a stop bit? The protocol requirements do not address how the system is using the data

and cannot determine the correct way to handle this case. It may make sense to ignore all of the received data bits, accept the data bits when a start bit arrives, set an error flag, stop receiving data, or employ another method. A large number of cases deal with input conditions that are not covered by the requirements, such as missing start bits, missing stop bits, too few data bits, too many data bits, too many stop bits, stop bits in the middle of the data, etc. Given the large test space and unknown interactions with the hardware, negative compliance testing is usually accomplished using CRV to randomly generate noncompliant inputs and system states. Including negative compliance testing in the verification process greatly expands the verification test space and increases the verification time. Formal methods can prove useful in assessing negative compliance by assuring that certain failure conditions cannot occur.

Negative compliance robustness tests are performed to assure that the system handles out-of-requirements conditions gracefully and recovers, so when valid conditions return, the system returns to correct operation. Consider the case of a data transmission with a missing stop bit. When this condition occurs, a straightforward implementation of the protocol would cause the system to wait indefinitely for the stop bit. Although waiting forever is highly undesirable, this is a valid implementation of the protocol. If reception of this data transmission was handled with an interrupt, the missing stop bit could prevent the interrupt from clearing. If the only way to receive a stop bit or a new start bit was required for the interrupt to clear, the entire system would be locked up, waiting indefinitely for the missing stop bit that could never arrive. Although this condition is technically allowed by the requirements, it should not be allowed to occur in practice.

2.9 VERIFICATION METHODOLOGIES.

The aviation and military industry predominantly uses VHDL as its design and verification language. The verification process in the semiconductor industry uses a combination of SystemVerilog with the native design languages of either Verilog or VHDL. SystemVerilog is a popular Verilog extension that offers object-oriented programming tools to speed development and improve verification IP reuse. Several verification methodology standards are currently in use, as follows:

- VMM was developed by Synopsys and was the first successful and widely implemented set of best practices for creation of reusable verification environments in SystemVerilog. The VMM has the object-oriented capabilities of SystemVerilog and allows constrained random and functional coverage verification.
- UVM is an open source SystemVerilog-focused library of reusable verification components that include assertions. The UVM's purpose is to combine the VMM and OVM features. The goals are test bench reuse and the development of reusable verification IP.
- OVM is a tool-agnostic library of objects and procedures covering the fundamental processes in verification, such as stimulus generation, data collection, and control of the verification process. The OVM focuses on higher level, transaction-level verification. The OVM is attempting to develop object-oriented capabilities for the VHDL language.

Although UVM is popular at the moment, all three of the above methodologies are in current use in industry. The need for standardized verification methodologies is clear.

2.10 HARDWARE-BASED VERIFICATION.

Hardware-based verification is typically performed by building a hardware system to apply test vectors to the component under test and to store and analyze the outputs of the device. A common hardware verification test is to apply the hardware’s RTL test suite using full-speed clocks and data signals. From a RTCA/DO-254 point of view, this hardware test provides a critical independent assessment of the design tool’s output and, therefore, can be used to avoid the need to qualify the design tool. This test can detect timing problems within the component, problems because of power quality, and signal-integrity issues related to the component. Major tool manufacturers endorse the use of hardware-based verification. For example, “Aldec provides DO-254/CTS (Compliance Tool Set) which allows testing of designs in a wide range of test combinations and compares outputs generated by RTL simulator with the target Field Programmable Gate Array (FPGA) device outputs” [21].

Synopsys conducted a survey of 1912 users on how they used hardware-based verification. The results are shown in figure 5.

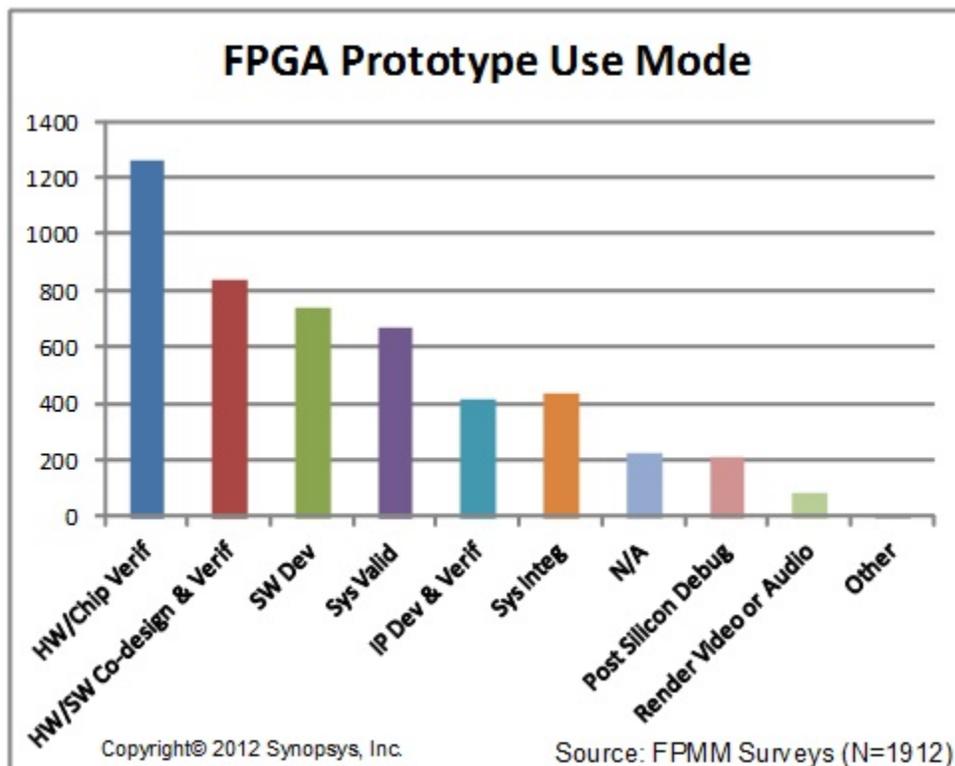


Figure 5. How Hardware Verification Is Used in Industry

The data are interesting in that the hardware verification system is being used for both verification and validation. When performing a test on the component in an environment similar to its final application, it is not clear whether it is quiet validation or quiet verification. The most popular answer was that hardware-based verification was used to locate bugs in the HDL code that escaped the normal verification process.

There is always a tradeoff between simulation speed and accuracy. The analysis of high-level transaction-based conditions, such as maximum throughput, is usually done by creating a software model of the hardware. However, system-level models are rarely cycle accurate and signals that need cycle-level accuracy, such as first in, first out full and empty signals, are always a problem. Hardware-based verification offers a system-level model that is both cycle-level accurate and high speed. Hardware-based verification allows timing problems to be identified as well as longer and more thorough simulations to be run in the same time period. In addition, the system can be subjected to robustness tests by verifying that the system not only meets, but exceeds, the requirements. This could be achieved, for example, by applying input signals at a rate in excess of that specified by the requirements. Negative compliance testing can also be performed to ensure that the system handles out-of-requirement conditions gracefully.

Not everything is improved with at-speed hardware-based verification. If a bug is found while running the same test bench that was used for verification, the ability to find the cause of the bug is limited because the hardware can only be observed at the outputs. Simulating the test bench that produced the bug can take days. The hardware-based verification speeds up bug detection, but it does not accelerate the debugging process. There are debugging tools that allow the user to observe internal signals within the FPGA, but those tools often degrade the system timing.

3. INDUSTRY SURVEYS.

Twelve verification engineers and managers representing five semiconductor companies and one aviation-related company participated in a detailed poll for this report. The small number of participants allowed the author to interact with each of the respondents and clarify ambiguous responses. The results of the author's poll are representative of conditions in the semiconductor industry.

Aldec Corporation published selected results from a customer survey that had over 2,400 respondents. Aldec design and verification tools are not widely used in the semiconductor industry and are more common in military and aviation applications. The Aldec survey should be considered representative of military and aerospace users.

A summary of the surveys follows. The results of both the author's poll and the Aldec survey are presented in detail in appendix A of this report.

3.1 DESIGN AND VERIFICATION LANGUAGE.

There are two major RTL languages used for the design and verification of hardware: VHDL and Verilog. The VHDL is a strongly typed language derived from Ada and is widely used in military applications. Verilog is a language modeled on C and is widely used in the semiconductor industry. Verilog has been extended to include object-oriented structures in a

language called SystemVerilog. SystemVerilog has built-in constructs to aid in the verification process, such as scoreboards to assess verification coverage and constrained random testing modules. There are some efforts to develop a similar extension to the VHDL language, but these efforts have had limited success. Although at least one company develops the design of their hardware using VHDL and verifies it using a combination of VHDL and SystemVerilog, the design flow is much cleaner if all of the tools are based on the same language.

The Aldec survey notes that, for its military- and aerospace-centric user base, “SystemVerilog is growing in popularity,” and that, with respect to training courses, “it is only recently that we’ve seen SystemVerilog overtake VHDL” [21]. This indicates that a shift from VHDL to Verilog is occurring in the military and aerospace industries.

The Aldec survey shows that 31% of the respondents will use Verilog and 32% of the respondents will use SystemVerilog for new designs. This probably indicates that all of the SystemVerilog users are using Verilog as their design language. The author’s poll indicated a much higher SystemVerilog usage of 66% in the semiconductor industry.

3.2 VERIFICATION METHODOLOGY.

The Aldec survey revealed that the majority of the respondents used no particular verification methodology or inhouse-developed verification methodology. Standardized verification methodologies, such as VMM and OVM/UVM, were used by 24% of the respondents, compared to 60% of the respondents in the author’s poll. This difference in the adoption of standardized verification methodologies is probably because semiconductor companies are free to embrace new verification technologies, but the certification processes of the military and aviation industries slow the adoption of new methodologies.

3.3 ASSERTIONS.

The author’s poll revealed that assertions were used by more than 80% of the respondents. The assertions were written at all levels of the design hierarchy, with all respondents using low-level assertions, and most respondents using higher-level assertions. Although automatic tools help identify assertions and assess assertion density, assertions were overwhelmingly written by hand. Assertions are a key element of the verification process.

3.4 FORMAL METHODS.

The author’s poll investigated the use of formal methods in the verification process. Half of the respondents used formal methods and a majority of the respondents considered formal methods essential to the verification process. Those respondents that did consider formal methods essential were quite adamant that formal methods must be utilized.

3.5 SAFETY-CRITICAL VERIFICATION.

The poll asked the respondents to consider a safety-critical component and a nonsafety-critical component, then to identify how the verification process would differ for these components. The responses were unanimous that the verification processes would be the same. The difference

would be that safety-critical components would have higher coverage targets, and that more effort would be expended on the formal verification effort.

4. IDENTIFY KNOWN AND EMERGING OBSTACLES, PROBLEMS, OR ISSUES.

A Deepchip survey [22] in 2011 asked what two areas of the system on a chip and integrated circuit design process needed the most advancement over the next 2 years. As table 1 shows, verification tools are viewed as the greatest weakness. The second highest response was IP collaboration and IP reuse tools. The high ranking of the verification and reuse of IP indicates concern about the widespread use of COTS and inhouse-developed IP throughout the industry.

Table 1. The Top-Two Areas Needing Advancement in the Design and Verification Process

EDA verification tools	63%
IP collaboration tools (selection-integration-reuse)	50%
EDA design tools	42%
Embedded software tools	26%
Other	2%

The same survey asked respondents their top three challenges for managing semiconductor IP. The responses are shown in table 2.

Table 2. The Top-Three Challenges in Managing IP

Verifying IP	62%
Integrating IP in design	53%
Making internal IP reusable	50%
Managing IP updates/bug fixes	48%
Finding/Selecting optimal IP	39%
Tracking IP usage	21%
Other	2%

Verification of the IP topped the list of challenges. Whether the IP is produced inhouse or is COTS, verifying IP without a detailed understanding of the implementation is a major concern. Problems integrating IP into the designs was second, and making internal IP reusable ranked a close third. This indicates that even when companies have developed the IP source code in their own design flow, IP reuse is still a major issue. The reasons for this vary. The IP reuse problems because of coding styles and documenting designer intent are among the most common.

These surveys indicate that verification of significant blocks of IP is one of the industry's greatest challenges. Finding and selecting optimal IP and tracking IP usage were also among the top challenges, showing that COTS IP is widely used in modern products.

4.1 ISSUES WITH COVERAGE METRICS.

For DAL A hardware, RTCA/DO-254 requires the verification process to test all of the requirements—RBV—and perform an analysis to assure that all design elements have been tested. This analysis is often accomplished by assuring that every line of code was executed. If there are lines of code that were not executed, then either the code is unnecessary and should be removed, or there is a missing requirement.

There are safety issues with this methodology. By definition, complex airborne hardware cannot be exhaustively tested. All that code coverage can assure is that each line of code has been executed at least once. For example, a line of code calculates the aircraft altitude from the barometric pressure sensors. Consider a situation for which an incorrect altitude calculation occurs when the system is only using altitude data from the satellite navigation system. Even though the line executes and an error occurs, the error has not affected the system outputs and, therefore, remains hidden. The code coverage metric now indicates that this section of code has been executed and, therefore, does not need further verification. This could allow the error to escape detection.

Consider the VHDL code below.

```
if (a = '1' and b = '1') then  
    c <= '0'; Statement 1  
else  
    c <= '1'; Statement 2  
end if;
```

This code checks to see if the variables *a* and *b* are both equal to 1. If both variables in a test case are equal to 1, then statement 1 will be executed and *c* will equal 0. If a test case with *a* equal to 1 and *b* equal to 0 is applied, then statement 2 will execute and *c* will equal 1. These two test cases achieve 100% statement coverage of the code, but leave two untested combinations of the inputs. In this simple case, using expression coverage would solve the problem by requiring test cases for all of the input combinations, but coverage metrics do not have knowledge of when signals are valid or being observed. For example, when a system is reset, all of the outputs are set to known values. However, the input circuits can react to the inputs applied during a reset and the coverage metrics, such as expression coverage, will indicate that features have been tested when their output could not be observed to see if an error occurred. To avoid the possibility of an error escaping the verification process, assertions can be used to check the correctness of signals, even when they do not affect observable pins.

4.2 ISSUES WITH FUNCTIONAL COVERAGE.

Defining a functional test plan requires elaborating possible input combinations and system states. Although there are tools that can help elaborate the test plan, determining what constitutes a function of the device requires human input. As mentioned previously, interesting events need to be identified. Although there are tools to assess the functional coverage metrics, there are currently no tools to assess the completeness of the functional test plan.

4.3 ISSUES WITH FORMAL VERIFICATION.

A formal verification tool can produce the following outputs:

- The property has been proven to hold.
- The property has been shown to not hold and a counterexample is provided.
- The property could not be proven to hold or was proven not to hold. The result is inconclusive.

Regardless of the output, there are always concerns about the quality of the result.

If the tool proves a property to hold, the hardware could still be incorrect for several reasons, including:

- The property did not accurately describe the requirement.
- The analysis was unrealistically over-constrained.
- The analysis found a solution using degenerate/illegal states because of a lack of constraints.

If the tool proves the property not to hold and gives a counterexample, the result may be of limited value for several reasons, including:

- The property did not accurately describe the requirement.
- The counter example is so complex that it provides no insight into the root cause of the failure.
- The analysis found a counter example using degenerate/illegal states because of a lack of constraints.

If the tool gives an inconclusive result, there is little feedback to indicate how close or far a solution may be. Would more simulation time help? Are more constraints needed? Is this hardware better verified using simulation?

Unlike simulation tools that have standard coverage metrics, there are limited tools to assess what code was actually used in the formal proof. There are currently no indications of whether there is unused hardware in the system. If the unused hardware does not interfere with the correct operation of the hardware needed to prove the properties, there is no way of identifying it. Currently, there is research into developing coverage metrics for formal tools [23].

4.4 ISSUES WITH ASSERTIONS.

There are tools that assess the density of assertions and can identify areas for which additional assertions are needed. These tools merely count assertions and cannot distinguish between well-

written and poorly written assertions. There are also tools that automatically generate assertions based on the RTL code, but these assertions can never completely document indirect requirements or the designer's intent [24].

4.5 ISSUES WITH COTS IP.

Both COTS IP and inhouse-developed IP present challenges to the verification process because the implementation is either unknown or too complex to easily analyze. Because the IP is a black box, the primary verification method is to write a functional coverage plan for the IP. Assertions can then be written to describe the interface for all inputs and outputs. A formal analysis of the interface can be performed if the hardware is suited to formal analysis. Human written and random tests are then performed until the functional and assertion coverage goals are met.

5. IDENTIFY POTENTIAL APPROACHES AND CRITERIA TO DEMONSTRATE SUFFICIENCY OF VERIFICATION COVERAGE ANALYSIS OF CEH LEVELS A, B, AND C.

Given how well the coverage metrics from RTCA/DO-178 map to hardware metrics, it would seem that implementing similar coverage metrics for hardware should produce a confidence level similar to that achieved in RTCA/DO-178 designs. However, industry practice has shown that the combination of functional coverage and code coverage is insufficient to fully verify a hardware device.

It has been demonstrated that RBV/functional coverage, code coverage, and formal methods are complementary verification techniques, and that using one without the other risks allowing errors to escape. It is proposed that DAL C hardware use the existing DO-254 process of verifying all requirements, and use elemental analysis or statement coverage to determine verification completeness. Formal methods cannot be universally applied, so one can only encourage, but not require, their use. The verification plan needs to balance how well formal methods can be used to verify the hardware with the expertise within the verification team. In some applications, formal methods can offer a substantial reduction in the time required to verify the product, and it is expected that verification teams will introduce formal methods to reduce the cost of verification:

- DAL B—In addition to the verification performed for DAL C, DAL B hardware is required to document all module interfaces with assertions and to use CRV. Verification completeness will be determined by meeting the coverage goals for RBV/functional coverage, assertion coverage, and statement and branch coverage.
- DAL A—In addition to the verification performed for DAL B, DAL A hardware will use toggle coverage and implement robustness tests and negative compliance tests as part of the functional tests. Verification completeness will be determined by meeting the coverage goals for RBV/functional coverage; assertion coverage; and statement, branch, and expression coverage.

Because a wide spectrum of hardware designs are covered by RTCA/ DO-254 and that there are times when 100% code coverage cannot be achieved, there is reluctance to recommend coverage level targets. It must be emphasized that the following numbers are targets and that there may be designs for which meeting the targets will be either impossible or require inordinate effort. The targets should not be viewed as inflexible numbers or the testing will focus on meeting the coverage metrics instead of assuring the correctness of the design. A reasonable coverage metric for a DAL B design would be 97% branch coverage, 95% assertion coverage, and 95% functional coverage. For a DAL A design, a reasonable target would be 100% expression coverage, 97% assertion coverage, 97 % functional coverage, and 95% toggle coverage.

Formal methods should be applied to hardware of DAL B or A. The DAL B hardware will be using assertions, which should be verified with formal tools. The DAL A hardware should be using sequential-equivalence checking and model-checking tools on all hardware suitable to formal methods.

The following verification flow should be used for hardware of all DAL.

5.1 HARDWARE VERIFICATION PLAN.

The verification plan will verify all requirements. The plan will develop coverage goals and then assess which goals are best achieved via simulation and formal methods. This plan will have two subplans: the simulation-based verification plan and the formal verification plan. If formal plans are used, the type of formal verification and the goals of the verification will be explicitly traced to the requirements. Because formal methods prove that the hardware meets the specified properties, it is critical that the properties be carefully traced to the requirements.

The hardware verification plan will assess how well each of the verification goals has been achieved on a scoreboard. The results of formal verifications will be checked with simulations to provide independent verification of the formal tool results. If the simulations and the formal methods disagree, one or both of the analyses are incorrect. A careful investigation of any discrepancies must be performed.

5.2 SIMULATION-BASED VERIFICATION PLAN.

Those verification goals assigned to simulation in the verification plan should include all possible hardware configurations and all variations of the inputs.

DAL B hardware will also include constrained random testing and assertion-based coverage. The verification plan will identify sequences of interest for all input signals and identify significant corner cases, concurrency issues, and error conditions.

DAL A hardware includes all of DAL B criteria, plus toggle coverage and robustness tests with negative compliance conditions.

5.2.1 Creating the Functional Coverage Specification.

Creating the functional coverage specification requires the following steps:

1. Define what functions should be covered.
2. Decide on the interesting inputs as well as internal states.
3. Identify the legal values, illegal values, and boundary values for all inputs.
4. Examine the interfaces and internal state machines to identify the important state machines and key transitions.
5. Identify key relations between the input data and system state.
6. Write assertions at all levels of the hierarchy for DAL B and A. The assertions should cover all inputs, outputs, and important internal signals.
7. Use known verification IP for standard interfaces.

5.2.2 Writing and Running the Test Bench.

Writing and running the test bench requires the following steps:

1. Write the test bench using parameters to enable constrained random testing.
2. Begin verification runs to debug the verification suite and identify vague and/or incomplete requirements as soon as there is RTL code to verify.
3. Address any problems with requirements as they are identified.

5.2.3 Coverage Analysis.

As the RTL code nears completion, follow these steps to begin coverage analysis:

1. Start looking for unreachable code at the module level and move up in the hierarchy.
2. Identify holes in the verification test bench and focus additional tests in those areas. The goal is 100% code coverage. If this is not possible, document why this is so.
3. Assure that all of the states in all state machines have been reached for DAL B.
4. Achieve 95% coverage for assertion and functional coverage.
5. Run toggle coverage looking for gaps in the test bench coverage for DAL A.
6. Assure that all of the transitions between state machine states have been traversed.
7. Ensure that all assertions have been evaluated at least once (100% assertion coverage) and that all device functions have been verified (100% functional coverage).

8. Analyze why and document the reason if 100% coverage cannot be achieved. An example of 100% coverage not being achieved might be circuits that operate only when a radiation-induced failure occurs. These circuits may not be fully exercised in simulation and, therefore, may prevent 100% coverage. These circuits would need to be analyzed independently to assure design correctness.

5.3 FORMAL VERIFICATION PLAN.

Formal verification methods are suitable for all DALs; however, the methods used can vary. The DAL C hardware would be encouraged, but not required, to use formal methods. The DAL B hardware would be expected to use formal methods to prove the assertions and to formally verify all interfaces. The DAL A hardware that is suitable to the application of formal methods would be expected to use equivalence-checking/model-checking techniques to improve the assurance that the design is correct.

The integration of formal methods into the coverage analysis is problematic; formal analysis can assess whether the desired functional properties are correct, but the formal results need to be independently verified with simulations to assure that the properties proven to hold are the correct properties. When a formal proof fails to be proven correct, an example is given to illustrate the failure. These examples are routinely analyzed and debugged using simulation tools.

6. RECOMMENDATIONS.

The existing RTCA/DO-254 verification process works well. One of the goals of this research was to determine coverage criteria that would yield assurance levels comparable to RTCA/DO-178. Published research and semiconductor industry practices indicate that the process of RBV combined with code coverage metrics is insufficient to adequately verify complex hardware. Several improvements to the existing process are suggested. A CRV can be used to generate large numbers of random test cases to detect problems with the requirements and remove human biases from the verification effort. Use of assertions throughout the hierarchy to detect any violations of the requirements or the designer's intent occur at any time during all simulations. It is believed that incorporating these additions into the RTCA/DO-254 verification processes will best improve the design assurance quality.

The RBV required for RTCA/DO-254 needs to be extended to a more robust functional verification that considers all combinations of inputs and system state. Robustness tests should also explore negative compliance issues to enable the hardware to recover from unexpected conditions, such as loss of communications, signals stuck at levels, and transmission errors. The hardware is not expected to operate correctly when subjected to unexpected conditions, but once the unexpected conditions have been removed, the hardware should resume normal operation as quickly as possible. Unexpected conditions should not cause the hardware to lock up or enter unknown states or conditions.

According to Foster [18], formal methods can provide substantial benefits to the verification process, but formal methods are best applied to hardware architectures that are concurrent data-transfer, control-functions, or fault-tolerant devices. Hardware that does significant processing

to the data is not well suited to formal methods. Because a particular design may or may not be suitable for formal methods, it is difficult to provide regulatory guidance on when and how formal methods should be applied.

The verification process includes three coverage metrics: code coverage, assertions, and functional. Using these metrics, coverage targets are proposed for DAL A, B, and C hardware.

7. CONCLUSIONS.

It is difficult to prevent errors from reaching a complex hardware final product. Even design and verification tools that are known to be correct can introduce errors if the tools are applied incorrectly or the requirements are incomplete or incorrect. The goal of the verification process is to verify the hardware using multiple orthogonal techniques to minimize the possibilities of an error escaping the verification process. Coverage metrics by themselves cannot assure design correctness. The overlapping verification processes of constrained random testing, code coverage, assertion coverage, functional coverage, and formal methods represent the best known techniques for assuring design correctness.

8. REFERENCES.

1. RTCA, Inc., RTCA/DO-254, “Design Assurance Guidance for Airborne Electronic Hardware,” Washington, DC, 2000.
2. Gluska, A., “Practical Methods in Coverage-Oriented Verification of the Merom Microprocessor,” *43rd ACM/IEEE*, San Francisco, California, July 24–28, 2006, pp. 332–337.
3. Synopsys, Inc., “Verification or Validation? What Do You Think?,” Available at: <http://blogs.synopsys.com/breakingthethreelaws/2012/02/%E2%80%9Cverification-or-validation-what-do-you-think%E2%80%9D/> (accessed July 2, 2014).
4. Aldec Inc., “Verified: The Need for Continued VHDL Support,” May 2012, Available at <http://www.aldec.com/en/company/news/2012-05-09/114> (accessed May 2012).
5. National Microelectronics Institute, “Verification Roadmapping,” Available at: <http://www.nmi.org.uk/assets/files/networks/verification/NMI%20VerificationRoadmapReport%20Final.pdf> (accessed February 2010).
6. Sherer, A., “A New Approach to FPGA Verification,” *Electronics Weekly*, October 12–18, 2011, p. 18.
7. Keithen, P., Landoll, D., Marriott, P., and Logan, B., “The Use of Advanced Verification Methods to Address DO-254 Design Assurance,” *IEEE Aerospace Conference*, Big Sky, Montana, March 2008, pp. 1–11.
8. RTCA, Inc., DO-178B, (ED-12B), “Software Considerations in Airborne Systems and Equipment Certification,” Washington, DC, 2001.

9. Rameni, K., "Coverage Metrics for Hierarchical Validation of Complex Behavioral Hardware Designs," Ph.D. thesis: University of California, Irvine, 2007.
10. Piziali, A., *Functional Verification Coverage Measurement and Analysis*, Kluwer Academic Publishers, Chapter 4, 2004.
11. Rajan, A., Staats, M., Whalen, M., and Heimdahl, M., "Coverage Metrics for Requirements-Based Testing," *NASA Formal Methods Symposium*, Washington, D.C., April 2010.
12. Tasiran, S., and Keutzer, K., "Coverage Metrics for Functional Validation of Hardware Designs," *Design & Test of Computers, IEEE*, vol. 18, no. 4, July/August 2001, pp. 36–45.
13. Jerinic, V., Langer, J., Heinkel, U., and Miller, D., "New Methods and Coverage Metrics for Functional Verification," *Proceedings of Design, Automation and Test in Europe*, vol. 1, March 2006, pp. 1–6.
14. Mentor Graphics, "Assertion-Based Verification for ARM-Based SoC Design," Available at: <http://www.mentor.com/products/fv/resources/overview/assertion-based-verification-for-arm-based-soc-design-1afc1e3f-7550-477e-96d8-36e379dfa3a0> (accessed 7/2/14).
15. Landoll, D., and Beland, S., "Using Assertions to Satisfy DO-254 Elemental Analysis," *Digital Avionics Systems Conference (DASC) 2011 IEEE/AIAA 30th*, Seattle, Washington, October 16–20, 2011, pp. 7C4-1-7C4-19.
16. Lingyi, L., Sheridan, O., Tuohy, W., and Vasudevan, S., "A Technique for Test Coverage Closure Using GoldMine," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 5, May 2012, pp. 790–803.
17. Blom, G., Holst, L., and Sandell, D., "7.5 Coupon Collecting I, 7.6 Coupon Collecting II, and 15.4 Coupon collecting III," *Problems and Snapshots From the World of Probability*, vol. 191, pp. 85–87, 2004.
18. Foster, H., Loh, L., Rabii, B., and Singhal, V., "Guidelines for Creating a Formal Verification Test Plan," *Proceedings of DVCon*, San Jose, CA, 2006.
19. P. Yeung, "Applying Assertion-Based Formal Verification to Verification Hot Spots," available at: <http://www.mentor.com/products/fv/resources/overview/applying-assertion-based-formal-verification-to-verification-hot-spots-78f2ea3b-f6a0-431e-8901-4c7956f381d5> (accessed July 2, 2014).

20. Cadence Design Systems, Inc. “Archived Webinar: Which Verification Coverage Metrics to Use When,” available at:
http://community.cadence.com/cadence_blogs_8/b/ii/archive/2011/11/14/archived-webinar-what-verification-coverage-metrics-to-use-when (accessed 2014).
21. Aldec Inc., “Superior Approach to DO-254 Hardware Verification,” Available at:
<http://www.aldec.com/en/downloads/private/110> (accessed 2012).
22. DeepChip, “ESNUG 491 Item 9,” May 2011.
Available at: <http://www.deepchip.com/items/0491-09.html> (accessed July 2, 2014).
23. Chockler, H., Kupferman, O., and Vardi, M., “Coverage Metrics for Formal Verification,” *International Journal on Software Tools for Technology Transfer*, vol. 8, no. 4-5, 2006, pp. 373-386.
24. Litvinova, E., Hahanova, A., Gorobets, A., and Priymak, A., “Verification System for SoC HDL-code,” *11th International Conference on Modern Problems of Radio Engineering Telecommunications and Computer Science (TCSET)*, Lviv, Ukraine, p. 348.

APPENDIX A—POLL OF VERIFICATION TOOLS AND COVERAGE METRICS

Most semiconductor companies view their verification processes as proprietary information. Through an anonymous poll and personal interviews, the author was able to poll 12 verification engineers and managers representing five semiconductor companies and one aviation-related company. The author acknowledged that the poll results have a bias towards the semiconductor industry.

Figure A-1 shows question 1 of the author’s poll.

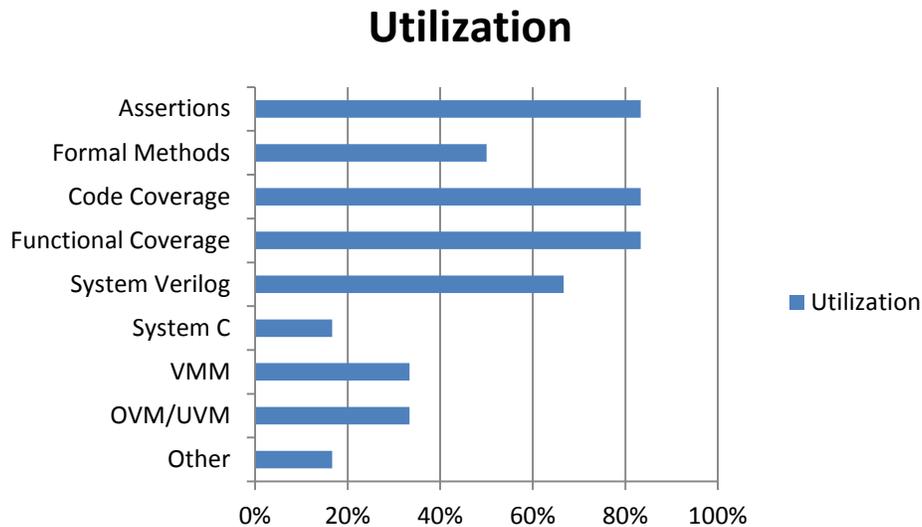


Figure A-1. Author’s Poll Question #1: Which of the Following Tools/Techniques Are Used in Your Verification Flow? (check all that apply.)

Analysis of the responses indicates that assertions are widely used. The verification coverage metrics of code coverage and functional coverage are used by nearly all of the respondents. This is consistent with a verification process using three coverage metrics: code coverage, assertion coverage, and functional coverage. The high-level verification language SystemVerilog is used by roughly 66% of the respondents. Verification methodologies such as the Verification Methodology Manual (VMM) and Open Verification Methodology/Universal Verification Methodology (OVM/UVM) are used by roughly 66% of the respondents.

A similar survey with more than 2400 respondents was conducted by Aldec Corporation. Aldec design and verification tools are not widely used in the semiconductor industry and are more common in military and aviation applications; therefore, the Aldec survey is biased towards military and aerospace users. Aldec questioned what design verification languages the respondents would use for their next design (see figure A-2). A project for which multiple languages were used would indicate all of them, meaning that the totals exceeded the 2400 respondents.

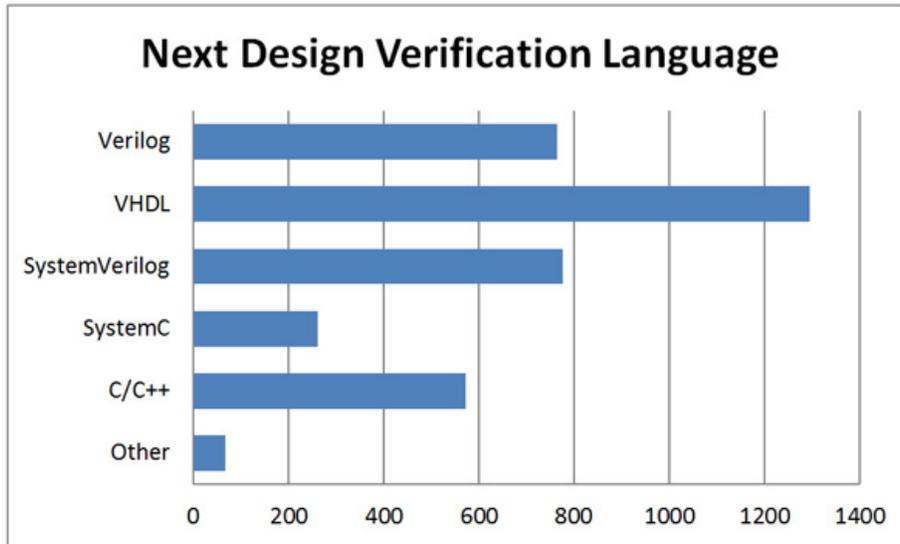


Figure A-2. Results of an Aldec Survey Question Asking What Verification Language the Respondents Will Use on Their Next Design

The survey shows that 51% of the respondents would use very high-speed integrated circuit (VHSIC) hardware description language (HDL)—referred to as VHDL—as their verification language. However the chief technology officer of Aldec noted that “SystemVerilog is growing in popularity” and that, with respect to training courses, “it is only recently that we’ve seen SystemVerilog overtake VHDL.” The Aldec survey shows that Verilog® will be used by 31% of the respondents and SystemVerilog will be used by 32% of the respondents. Because the VHDL, Verilog, and SystemVerilog percentages exceed 100%, some users are likely using SystemVerilog and VHDL at the same time. The author’s survey indicated a SystemVerilog usage of 66%, but the author’s survey did not break out Verilog and SystemVerilog individually. Because SystemVerilog is a superset of Verilog, the results appear to be consistent between the two surveys and indicate a trend of using SystemVerilog for all industries.

Aldec also surveyed users on the verification methodology they used on their most recent project (see figure A-3).

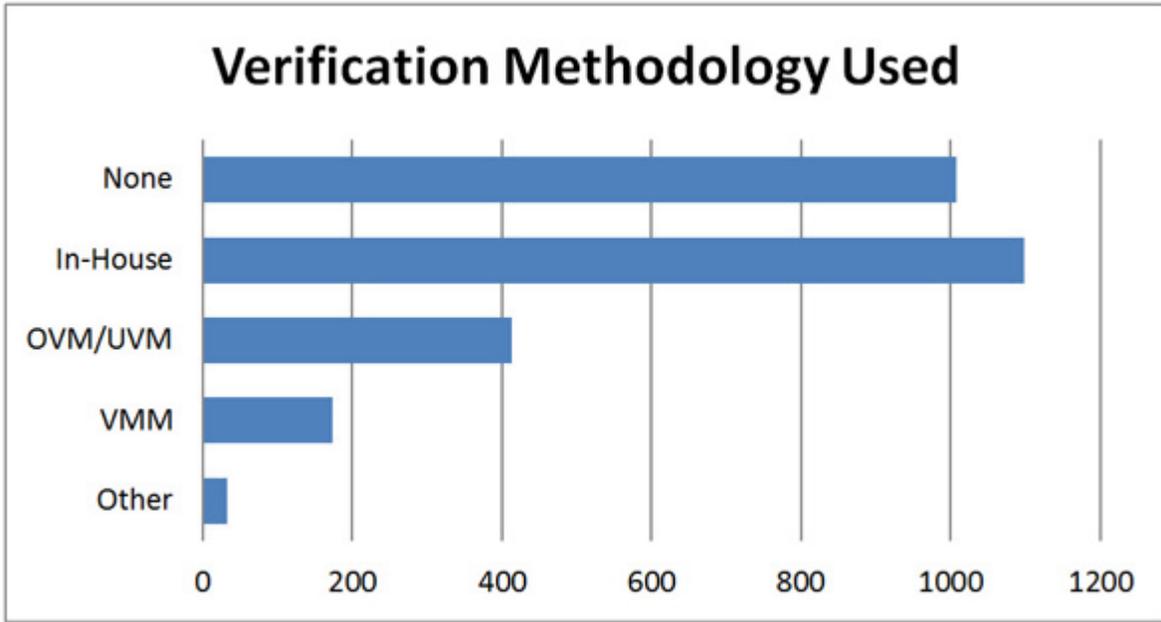


Figure A-3. Results of an Aldec Survey Asking Respondents What Verification Methodologies Were Being Used

The Aldec survey revealed that 41% of respondents used no particular verification methodology and 45% used an inhouse-developed verification methodology. Published verification methodologies, such as VMM and OVM/UVM, were used by 24% of the respondents compared to 60% of the respondents in the author’s poll. This real difference between the industries is probably because semiconductor companies are free to embrace new verification technologies, but the certification processes of the military and aviation industries slow the adoption of new methodologies.

The author’s poll next investigated the use of assertions. From question #1 we know that the majority of respondents are using assertions. The value of the assertion process hinges on quality assertions being present at all levels of the design hierarchy. There are automated tools to identify where assertions are needed. Question #2 (see figure A-4) asked whether these automated tools were being used. Only 17% of respondents have adopted automatic tools. In practice, the assertions generated by the automated tools are usually used to supplement manually generated assertions.

How do you identify where to insert assertions?

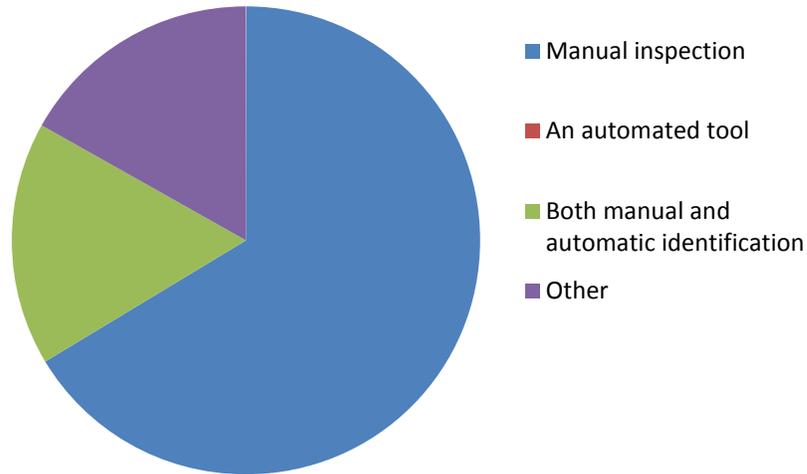


Figure A-4. Authors Poll Question #2: How Do You Identify Where to Insert Assertions?

Assertions should be applied throughout the design hierarchy. It was expected that the register transfer language (RTL) designers would implement low-level assertions and the verification team would write high-level assertions. Question #3 asked who inserted the assertions into the RTL (see figure A-5). The data from question #3 show that, in general, both the design and verification teams write assertions, but that with some respondents, only the RTL designers write assertions.

Who puts the assertions into the RTL code?

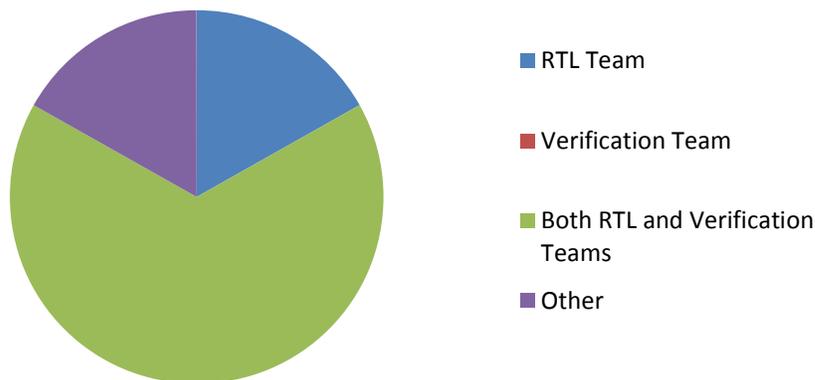


Figure A-5. Author's Poll Question #3: Who Puts Assertions Into the RTL Code?

Question #4 (see figure A-6) addressed the issue: At what hierarchical level were the assertions used? An answer of 100% at all levels of hierarchy was expected and the data confirmed this.

Some respondents did not use top-level assertions. A follow-up investigation revealed that these respondents produce modules that are integrated by another group and do not work at the top level.

Where are assertions used?

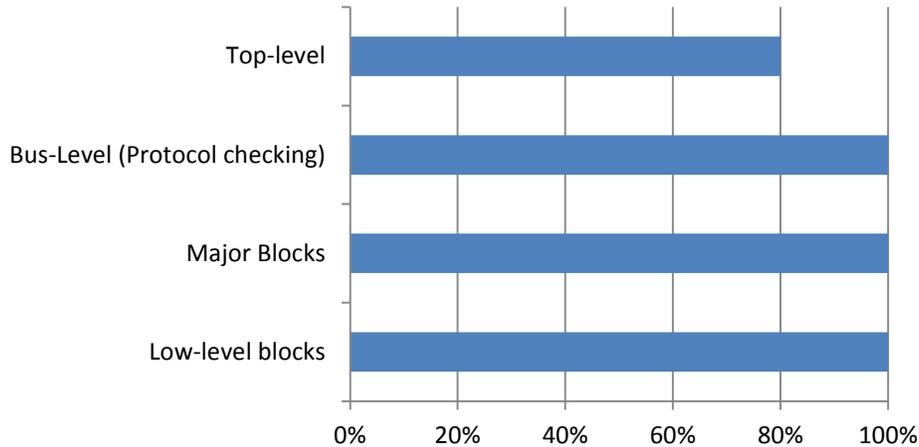


Figure A-6. Author's Poll Question #4: Where Are Assertions Used? (check all that apply)

Question #5 (see figure A-7) asked which coverage metrics were used to assess verification completeness.

Which coverage metrics do you use to assess verification completeness?

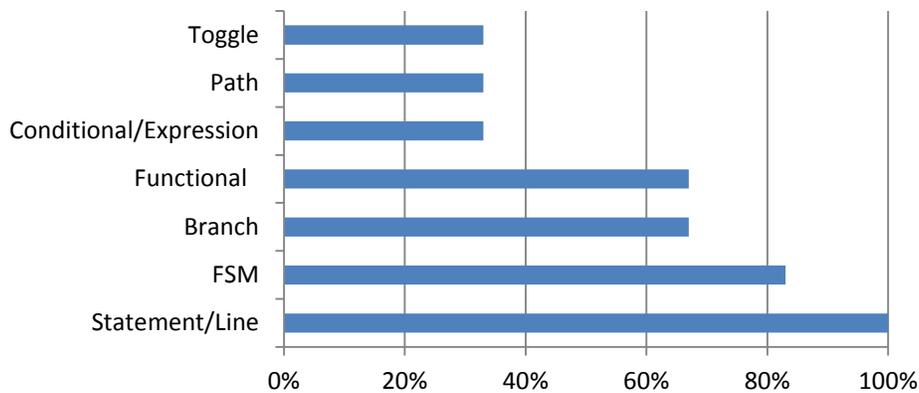


Figure A-7. Author's Poll Question #5: Which Code Coverage Metrics Do You Use? (check all that apply.)

Statement coverage was used by all respondents, as expected. Only 33% of the respondents utilized expression coverage. This is probably a reflection of the high cost of expression coverage. This question did not break out which metrics were used to determine when verification was complete. Some respondents monitor nearly all of the coverage metrics.

Question #6 (see figure A-8) asked how often code coverage checks were run.

How often are code coverage checks run?

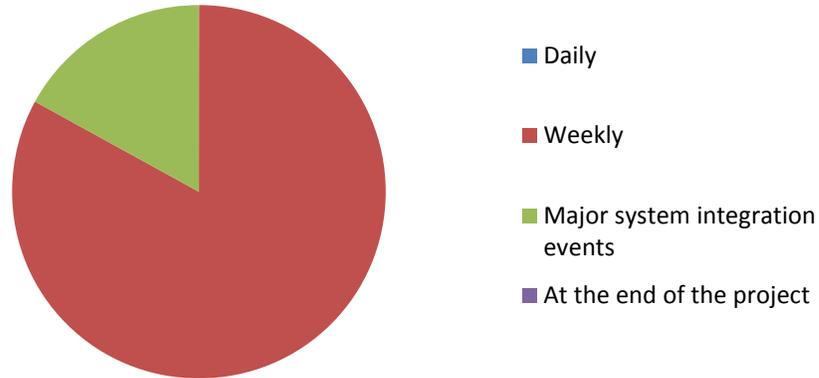


Figure A-8. Author’s Poll Question #6: How Often Are Code Coverage Checks Run?

It was expected that the code coverage checks would be run at major system integration events. The fact that code coverage checks were run weekly indicates that the coverage percentage is part of determining when verification is complete and weekly runs are being used to monitor verification progress.

Question #7 (see figure A-9) asked if the respondent used constrained random verification (CRV).

Do you use constrained random verification?

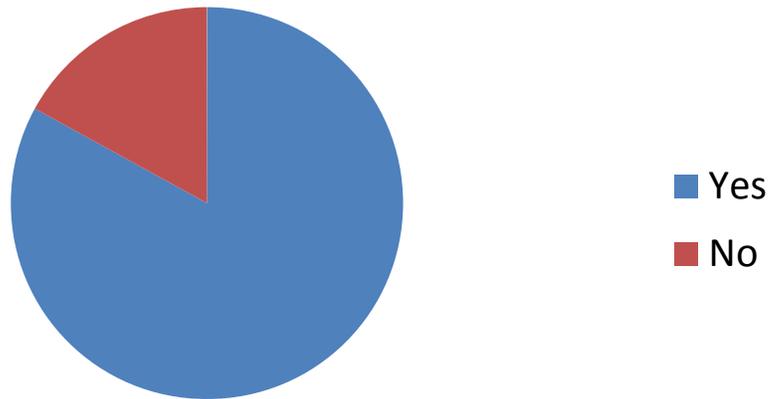


Figure A-9. Author's Poll Question #7: Do You Use CRV?

It is clear that CRV is widely used. Question #8 (see figure A-10) asked: “How do you know when the CRV process is complete?” The responses can be summarized as follows: CRV is run

until the code coverage goals are met. If there is still time remaining in the project because of tapeout or other reasons, continue running CRV until the project is complete.

Question #9 (figure A-10) asked whether formal methods were part of the verification process.

Do you use formal methods in your verification process?

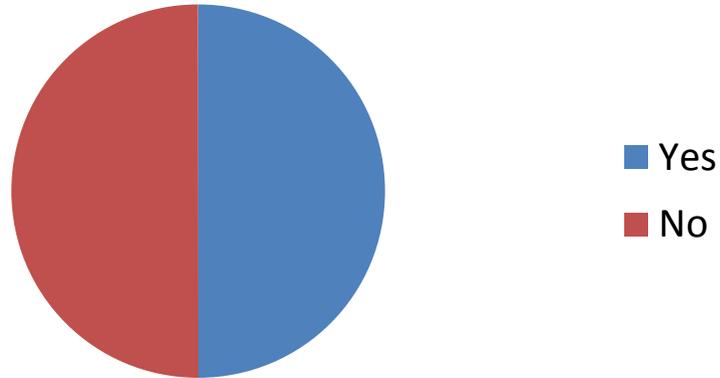


Figure A-10. Author's Poll Question #9: Do You Use Formal Methods in Your Verification Process?

The results indicate that formal methods have been adopted by half of the respondents. Figure A-11 shows question #10 of the author's poll.

How essential are formal methods to your verification process?

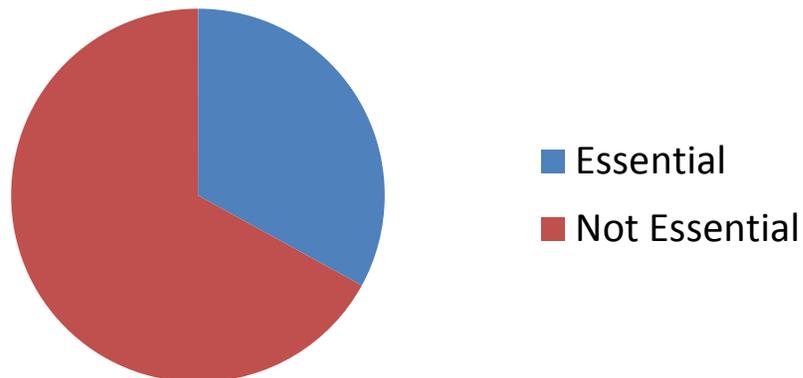


Figure A-11. Author's Poll Question # 10: How Critical/Essential Are Formal Methods in Your Verification Process?

The responses to questions #9 and #10 reveal that verification flows where formal methods are essential components of the verification process, but they are not widely used. This is a bit of a surprise because the tool vendors announce the value of formal tools in the verification process. Formal methods are being used to supplement other verification processes.

The final question of the poll was an open-ended question asking the respondents to consider a nonsafety-critical part and a safety-critical part, and to describe how the verification process would differ between the two parts. In summary, the responses show that there would be no difference in the verification process between the two parts. Both parts would be assessed for completeness using appropriate code coverage metrics, assertion coverage, and functional coverage, but for the safety-critical part, there would be more effort to reach 100% coverage on all of the metrics. There would also be a greater effort to prove the correctness of the device using formal proofs. In addition, the safety-critical part would also be subjected to a more rigorous robustness test regime.

APPENDIX B—TEST CASES

To demonstrate how errors might creep into a design, two simple test case scenarios were developed. The first investigates how ambiguity in the requirements can result in conflicting implementations. These ambiguities can be identified by constrained random tests, the use of assertions, or the use of formal methods. The second test case investigates the use of commercial off-the-shelf (COTS) intellectual property (IP) and how using assertions to document the interface to the COTS IP can prevent similar errors. An alternative is to use formal methods to prove the correctness of the COTS IP interface.

B.1 TEST CASE #1 - REQUIREMENTS AMBIGUITY.

The requirements should capture the intent of the system designer, but any ambiguity in the requirements may be unintentionally misinterpreted by the hardware designer. This test case will explore two implementations of a system and will demonstrate that, although both implementations meet the same system requirements, they both produce different results for key output signals.

This example describes a highly simplified control system for an aircraft's braking and reverse thruster system.

System inputs:

1. `in_air_sensor`—Indicates that the aircraft is in flight. When this sensor is active (logic level 1), the reverse thrusters should not deploy.
2. `deploy_reverse_thrusters_switch`—Cockpit switch controlling deployment of the reverse thrusters. When this switch is active (logic level 1), the reverse thrusters should deploy.
3. `apply_brakes_switch`—Cockpit switch controlling activation of the aircraft's braking system.

System outputs:

1. `reverse_thruster_control_signal`—Signal output that enables/disables the aircraft's reverse thrusters.
2. `brake_control_signal`—Signal output that enables/disables the aircraft's brakes.

Output behavior:

1. The `reverse_thruster_control_signal` will be active when the `deploy_reverse_thrusters_switch` is active, unless the `in_air_sensor` is active.
2. The `brake_control_signal` will be active when the `apply_brakes_switch` is active.

The aircraft braking and reverse thruster control system was implemented in very high-speed integrated circuit (VHSIC) hardware description language (HDL)—referred to as VHDL—using two different methods, both of which meet the system requirements, yet produce different behavior on the system outputs.

B.1.1 SYSTEM IMPLEMENTATION 1.

The following VHDL code is a system implementation based on the provided system requirements.

```
process (deploy_reverse_thrusters_switch, apply_brakes_switch, in_air_sensor)
begin
    brake_control_signal <= apply_brakes_switch;
    reverse_thruster_control_signal <= (deploy_reverse_thrusters_switch and not
in_air_sensor);
end process;
```

Let's analyze the register transfer language (RTL) code by tracing the first requirement to the RTL.

The requirement: System Outputs 1: The *reverse_thruster_control_signal* will be active when *deploy_reverse_thrusters_switch* is active, unless the *in_air_sensor* is active.

is met by the VHDL statement:

```
reverse_thruster_control_signal <= (deploy_reverse_thrusters_switch and not in_air_sensor);
1.
```

The *reverse_thruster_control_signal* will be active only when the *deploy_reverse_thrusters_switch* is active and the *in_air_sensor* is inactive; otherwise, the *reverse_thruster_control_signal* will be inactive. The second requirement is next traced to the RTL.

The requirement: System Outputs 2: The *brake_control_signal* will be active when the *apply_brakes_switch* is active

is met by the VHDL statement:

```
brake_control_signal <= apply_brakes_switch
```

The *brake_control_signal* will be active when the *apply_brakes_switch* is active; otherwise, the *brake_control_signal* will be inactive.

The output of a test bench for this system is shown in figure B-1. The test bench achieves 100% code coverage and 100% functional coverage. Examination of the system output waveform for this implementation demonstrates that this system operates as intended. The

brake_control_signal is active only when the *apply_brakes_switch* is active. The *reverse_thruster_control_signal* is active only when the *deploy_reverse_thrusters_switch* is active and the *in_air_sensor* is inactive.

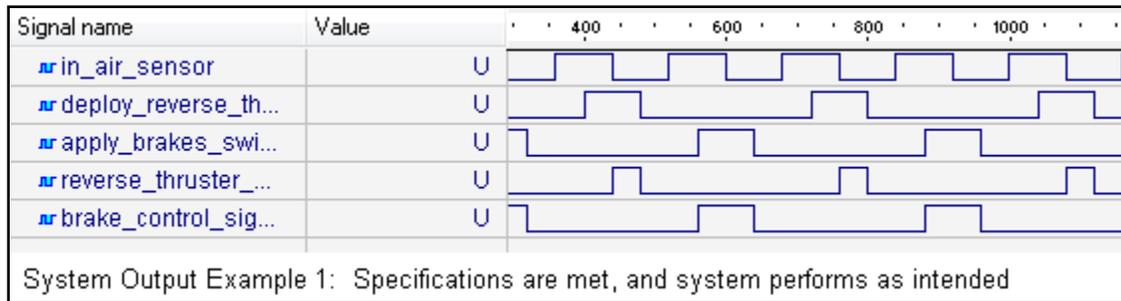


Figure B-1. Test bench Output of Implementation 1

B.1.2 SYSTEM IMPLEMENTATION 2.

The following HDL code is an alternative implementation of the aircraft braking and reverse thruster system.

```
process (deploy_reverse_thrusters_switch, apply_brakes_switch, in_air_sensor)
begin
  brake_control_signal <= '1';
  if (deploy_reverse_thrusters_switch = '1' and in_air_sensor = '1') then
    reverse_thruster_control_signal <= '0';
  else
    reverse_thruster_control_signal <= '1';
  end if;
end process;
```

Tracing the requirements to the VHDL again shows that the requirement

System Outputs 1: The *reverse_thruster_control_signal* will be active when the *deploy_reverse_thrusters_switch* is active, unless the *in_air_sensor* is active

is met by the VHDL statement:

```
if (deploy_reverse_thrusters_switch = '1' and in_air_sensor = '1') then
  reverse_thruster_control_signal <= '0';
else
  reverse_thruster_control_signal <= '1';
end if; .
```

The *reverse_thruster_control_signal* will be active unless the *deploy_reverse_thrusters_switch* and the *in_air_sensor* are active simultaneously. The second requirement is next traced to the RTL.

The requirement:

System Outputs 2: The *brake_control_signal* will be active when the *apply_brakes_switch* is active

is met by the VHDL statement:

```
brake_control_signal <= '1'; .
```

The *brake_control_signal* will always be active, so it will inherently be active when the *apply_brakes_switch* is active.

The results of executing the test bench on the system are shown in figure B-2. Once again, the test bench achieves 100% code coverage and 100% functional coverage. The *brake_control_signal* is active regardless of input behavior. This behavior meets the requirement that *brake_control_signal* will be active when the *apply_brakes_switch* is active, but this is probably not the intent of the system designer. By not including what to do when the *brake_control_signal* is not active, the system specification contains a vague requirement. Most humans will read the requirements in a manner consistent with implementation #1, but this error in the requirements needs to be identified. Writing assertions written for the interface between this block and others would quickly reveal the error in the requirements and prevent this from escaping detection until late in the design process.

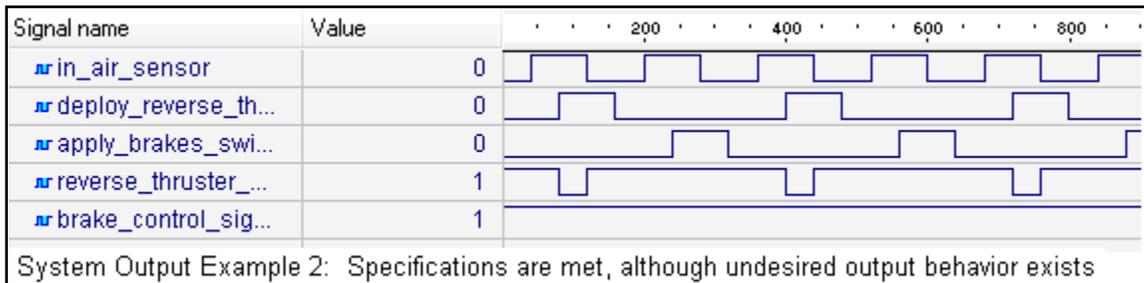


Figure B-2. Test Bench Output for Implementation 2

The identified ambiguity can be avoided through the use of fully defined requirements. Any requirement of system behavior must define the behavior for all possible input conditions. Updating the aircraft braking and reverse thruster requirements to remove ambiguity produces the following requirements:

System Outputs 1: The *reverse_thruster_control_signal* will be active when *deploy_reverse_thrusters_switch* is active, unless the *in_air_sensor* is active.

This requirement contains two inputs and one output, which means that there are four possible input combinations. This requirement does not define what to do when *deploy_reverse_thrusters_switch* is inactive. The word “unless” can often be interpreted multiple ways.

System Outputs 1 corrected: Often the use of a table, such as table B-1, is the simplest way to fully describe a requirement.

Table B-1. Logic for the *Reverse_Thruster_Control_Signal* in Terms of the *Deploy-Reverse-Thrusters-Switch* and *In_Air_Sensor* Inputs

reverse_thruster_control_signal	deploy_reverse_thrusters_switch	in_air_sensor
Inactive	Inactive	Inactive
Inactive	Inactive	Active
Active	Active	Inactive
Inactive	Active	Active

Original requirement 2: The *brake_control_signal* will be active when the *apply_brakes_switch* is active.

Requirement 2, corrected:

The *brake_control_signal* will be active when the *apply_brakes_switch* is active, otherwise the *brake_control_signal* will be inactive.

There are two ways to alleviate problems due to flawed requirements. The first is to document all interfaces from both inside and outside the module. Differences between the interface documentation can be identified through manual inspection or the use of formal method tools. A second method is to use random tests to generate test cases that highlight the problem. In this case, the hardware is control-based, so a formal proof of the correctness of the hardware could be performed. This would force the ambiguity to be resolved so the formal properties could be written.

B.2 TEST CASE #2 – THE COTS IP EXAMPLE.

This test case examines a design scenario using purchased COTS IP as part of a larger design. This example focuses on the design process and not the complexity of the design. The hardware examined in this test case is a basic traffic light and crosswalk signal. To allow us to identify the root cause of the problems, we have full access to the COTS IP source code. We begin by detailing the design and verification of the COTS IP traffic light, and then integrate this IP with a separately designed and verified crosswalk light module. The modules were verified separately based on the requirements and were found to meet all of them; however, the combined system produced errors. This test case investigates the use of assertions to identify the problem before system integration.

B.2.1 THE COTS IP – BASIC TRAFFIC LIGHT SYSTEM DESIGN

The COTS IP in this test case is the design of a simple traffic light, which controls traffic at the intersection of two streets, continuously cycling with fixed timing for both the green light duration and the yellow light duration. The system produces six traffic output signals, which are

green, yellow, and red light control signals for each road. The traffic light system is implemented by defining and designing a state machine to control the system outputs as presented in Figure B-3. The system also outputs the signal's *state* and *next_state* for use with other modules. In the VHDL source code (appendix C of this report), the output signals are *r0*, *y0*, *g0*, *r1*, *y1*, *g1*, *state* and *next_state*.

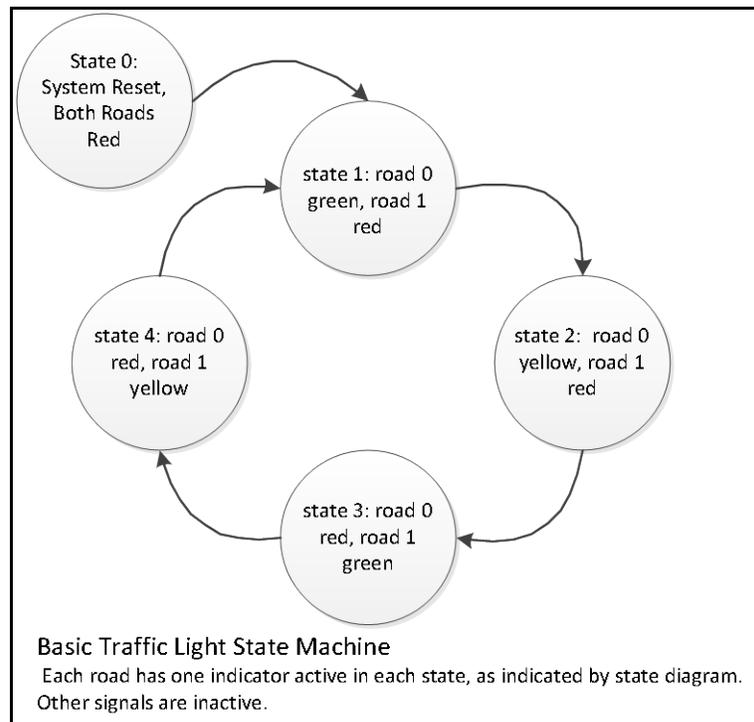


Figure B-3. The State Machine Implemented in the COTS IP

The state machine VHDL code was verified through simulation by examining the six state machine output signals using a waveform viewer. Additionally, the following assertions were used to ensure design safety criteria were never violated:

- Assertion one: Green 0 and Green 1 are never simultaneously active.
- Assertion two: If Green 1 or Yellow 1 are active, then Red 0 must always be active.
- Assertion three: If Green 0 or Yellow 0 are active, then Red 1 must always be active.

Figure B-4 shows the waveform simulation indicating that the state machine was functioning without error. The outputs cycled properly with appropriate timings. The assertions also verified that the system was not violating the design criteria.

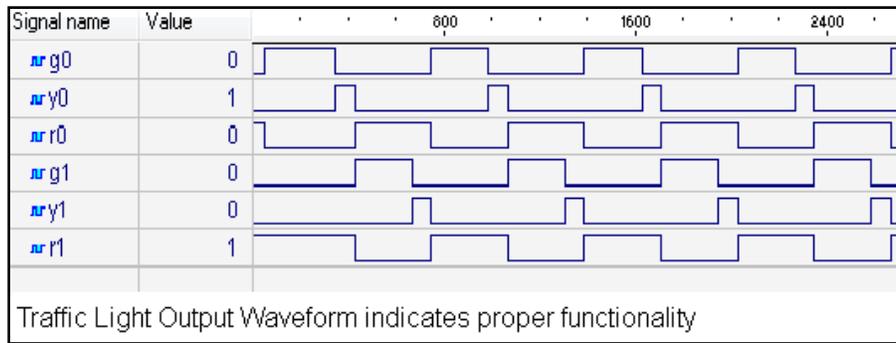


Figure B-4. Test Bench Showing That the Light Outputs of the COTS IP Meet the Requirements

Examining system output performance is a commonly accepted method for performance complex electronic hardware verification. However, it is possible that problems exist within the design, even if these problems do not immediately produce system output failure. In this traffic light example, the traffic light was accepted as error free from the output-based verification process.

B.2.1.1 Using the COTS IP– Crosswalk Signal Module Design

The product being designed is a traffic light with the addition of a crosswalk signal to control the flow of pedestrian traffic crossing road 1. The COTS traffic light IP has a good service history. The crosswalk module has an input signal of the traffic light state and three crosswalk output signals (*walk*, *hurry*, *no_walk*). The crosswalk control signal was intended to be based on the next traffic light state, for which the crosswalk outputs would have the characteristics shown in table B-2.

Table B-2. Crosswalk Signal Requirements

Control signal	Condition (traffic light state)	walk	hurry	no_walk
000	system reset	0	0	1
001	road 0 green, road 1 red	1	0	0
010	road 0 yellow, road 1 red	0	1	0
011	road 0 red, road 1 green	0	0	1
100	road 0 red, road 1 yellow	0	0	1

After designing the crosswalk module, the test bench was run. Figure B-5 shows the output waveform analysis, which indicates that the crosswalk functions as expected. Design assertions were also used to verify the crosswalk system and did not indicate any errors with the system. The assertions covered the following rules:

- Assertion 1: *walk* and *hurry* are never active at the same time
- Assertion 2: *walk* and *no_walk* are never active at the same time
- Assertion 3: *no_walk* and *hurry* are never active at the same time

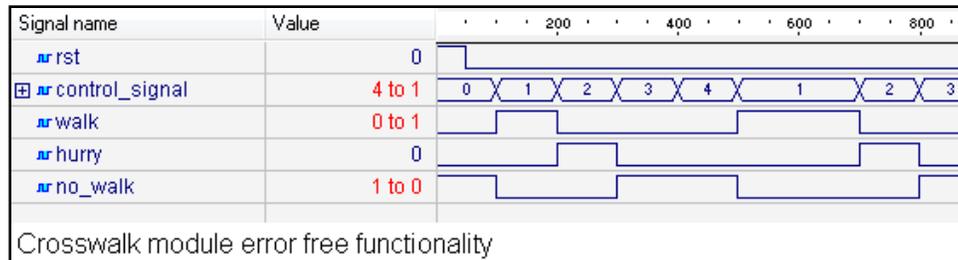


Figure B-5. Test Bench Demonstrating That the Crosswalk Module Is Correct

B.2.1.1.1 IP Integration

The working crosswalk module was added to the basic traffic light design. However, the crosswalk exhibited significant problems. The crosswalk signals were behaving erratically, creating a dangerous situation for the pedestrians crossing road 1. Figure B-6 shows a more detailed simulation of the integrated system where it can be seen that the crosswalk signals are changing at times when the traffic light signals were not changing.

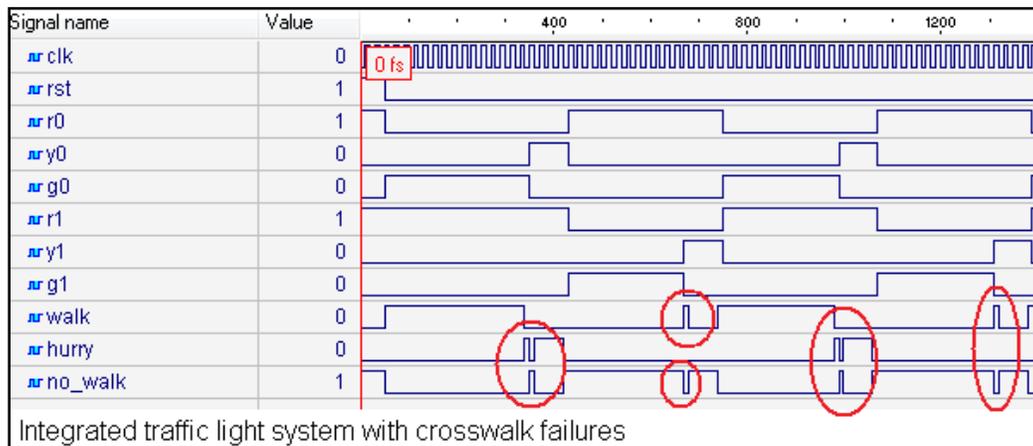


Figure B-6. Test Bench of the Crosswalk System With Integrated COTS IP Showing Glitches on the Crosswalk Output Signals

Figure B-6 shows the results of simulating the entire traffic light system in more detail. The glitches in the *walk* signal are because of glitches in the *next_state* signal coming from the COTS IP. As shown in figure B-7, further analysis demonstrates that *next_state* from the COTS IP glitches frequently. This error has always been present in the traffic light IP, but because the *next_state* signal did not create any observable error in the traffic light system, the error lay dormant until the crosswalk IP used the signal.

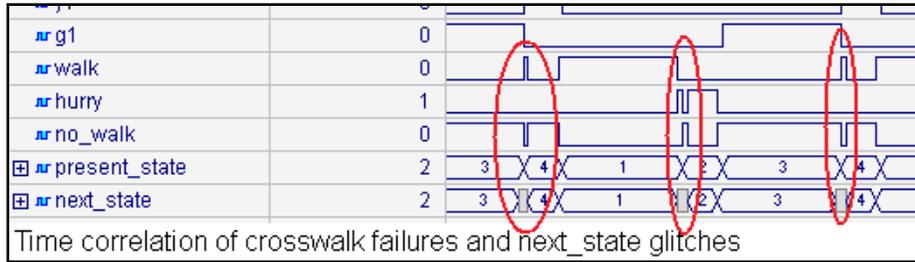


Figure B-7. Comparison of the Output Glitches to Intersignal Next_State

An error of this nature occurred because the traffic light designer assumed the glitches in the *next_state* signal were irrelevant to this design (see Figure B-8), while the crosswalk designer assumed that the *next_state* signal did not glitch. If either designer had fully documented these assumptions, this error could have been identified before system integration.

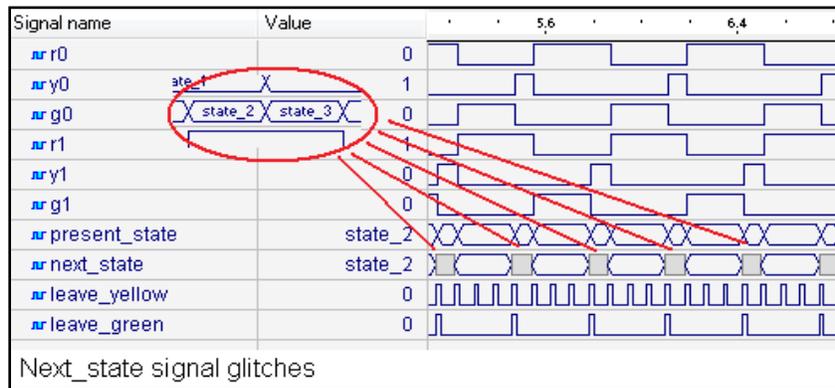


Figure B-8. Detailed Simulation of the COTS IP Showing the Cause of the Next_State Glitches

B.2.2 ASSERTIONS TO THE RESCUE.

The COTS IP vendor can avoid this problem by writing assertions to document how the interface signals should work. The customer can detect problems in the COTS IP by using assertions to document how the interface signals should operate. Assertions should not be limited to testing individual system modules; assertions should also cover the interfaces between different modules. An example of an interface assertion that would have addressed the traffic light problem in this system follows:

Assertion: The signal *next_state* should have only one transition when the signal *state* transitions.

The verification of COTS IP is always a concern because of the inability to observe the module internals. Documenting the interface to which the COTS IP using assertions ensures that if the IP violates its requirements at any time, the errors will be detected by the assertions. This provides assurance that the COTS IP is performing as required.

APPENDIX C—TEST CASE VERY HIGH-SPEED INTEGRATED CIRCUIT HARDWARE
DESCRIPTION LANGUAGE SOURCE CODE

C.1 VHDL AND TEST BENCH CODE FOR TEST CASE 1.

C.1.1 THE COMMERCIAL OFF-THE-SHELF INTELLECTUAL PROPERTY TRAFFIC
LIGHT VHDL CODE.

```
-----  
-----  
--  
-- Title      : traffic_light_state  
-- Design     : basic_traffic_light  
-----  
-----  
--  
-- Description : basic traffic light state machine  
-- 5 states: state 0:  road 0 red, road 1 red; in the event of a  
system reset  
-- state 1: road 0 green, road 1 red;  
-- state 2:  road 0 yellow, road 1 red;  
-- state 3: road 0 red, road 1 green;  
-- state 4: road 0 red, road 1 yellow;  
-- state machine cycles through states based on a long timer for  
green  
-- and a short timer for yellow.  both road get same time allocation  
per cycle  
--  
-----  
-----  
  
--{{ Section below this comment is automatically maintained  
--   and may be overwritten  
--{entity {traffic_light_state} architecture  
{traffic_light_state_arch}}  
  
library IEEE;  
use IEEE.STD_LOGIC_1164.all;  
use IEEE.std_logic_arith.all;  
use IEEE.std_logic_unsigned.all;  
  
entity traffic_light_state is  
    port(  
        clk : in STD_LOGIC;  
        rst : in STD_LOGIC;  
        r0  : out STD_LOGIC;  
        y0  : out STD_LOGIC;  
        g0  : out STD_LOGIC;  
        r1  : out STD_LOGIC;  
        y1  : out STD_LOGIC;
```

```

        g1 : out STD_LOGIC
    );
end traffic_light_state;

--}} End of automatically maintained section

architecture traffic_light_state_arch of traffic_light_state is
type possible_states is (state_0, state_1, state_2, state_3, state_4);
    --define available states
signal present_state, next_state: possible_states;

signal yellow_count: std_logic_vector(1 downto 0);
signal green_count: std_logic_vector(3 downto 0);
signal leave_yellow: std_logic;
signal leave_green: std_logic;
signal rst_ylw, rst_grn: std_logic;

begin

    clk_next_state: process (clk, rst)--clk next_state into present
state
    --this is the synchronous portion of the state machine
begin
        if ( rst = '1') then --Async reset
            present_state <= state_0;
        elsif (clk'event and clk='1') then --Rising Edge of clk
            present_state <= next_state; --Clock Next state
        end if;
    end process clk_next_state;

    green_counter: process (clk, rst)
begin
        --if (rst = '1' or rst_grn = '1') then
        if (rst = '1') then
            green_count <= "0000";
        elsif (CLK'event and CLK='1') then
            green_count <= green_count + 1;
        end if;
        if green_count = "1111" then
            leave_green <= '1';
            --green_count <= "0000";
        else
            leave_green <= '0';
            --green_count <= green_count + 1;
        end if;
    end process green_counter;

    yellow_counter: process (clk, rst)
begin
        --if (rst_ylw = '1' or rst = '1') then
        if (rst = '1') then

```

```

        yellow_count <= "00";
    elsif (CLK'event and CLK='1') then
        yellow_count <= yellow_count + 1;
    end if;
    if yellow_count = "11" then
        leave_yellow <= '1';
    else leave_yellow <= '0';
    end if;
end process yellow_counter;

state_select: process (present_state, leave_green, leave_yellow)
begin
case present_state is
    when state_0 =>
        --rst_ylw <= '1';
        --rst_grn <= '1';
        r0 <= '1';
        y0 <= '0';
        g0 <= '0';
        r1 <= '1';
        y1 <= '0';
        g1 <= '0';
        next_state <= state_1;

    when state_1 =>
        --rst_ylw <= '1';
        --rst_grn <= '0';
        r0 <= '0';
        y0 <= '0';
        g0 <= '1';
        r1 <= '1';
        y1 <= '0';
        g1 <= '0';
        if leave_green = '1' then
            next_state <= state_2;
        else
            next_state <= state_1;
        end if;

    when state_2 =>
        --rst_ylw <= '0';
        --rst_grn <= '1';
        r0 <= '0';
        y0 <= '1';
        g0 <= '0';
        r1 <= '1';
        y1 <= '0';
        g1 <= '0';
        if leave_yellow = '1' then
            next_state <= state_3;
        else
            next_state <= state_2;
        end if;
    end case;
end process state_select;

```

```

        end if;

    when state_3 =>
        --rst_ylw <= '1';
        --rst_grn <= '0';
        r0 <= '1';
        y0 <= '0';
        g0 <= '0';
        r1 <= '0';
        y1 <= '0';
        g1 <= '1';
        if leave_green = '1' then
            next_state <= state_4;
        else
            next_state <= state_3;
        end if;

    when state_4 =>
        --rst_ylw <= '0';
        --rst_grn <= '1';
        r0 <= '1';
        y0 <= '0';
        g0 <= '0';
        r1 <= '0';
        y1 <= '1';
        g1 <= '0';
        if leave_yellow = '1' then
            next_state <= state_1;
        else
            next_state <= state_4;
        end if;

    when others =>
        --rst_ylw <= '1';
        --rst_grn <= '1';
        r0 <= '1';
        y0 <= '0';
        g0 <= '0';
        r1 <= '1';
        y1 <= '0';
        g1 <= '0';
        next_state <= state_0;
    end case;
end process state_select;

end traffic_light_state_arch;

```

C.1.2 THE TRAFFIC LIGHT TEST BENCH CODE.

```
library ieee;
use ieee.STD_LOGIC_UNSIGNED.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

    -- Add your library and packages declaration here ...

entity traffic_light_state_tb is
end traffic_light_state_tb;

architecture TB_ARCHITECTURE of traffic_light_state_tb is
    -- Component declaration of the tested unit
    component traffic_light_state
    port(
        clk : in STD_LOGIC;
        rst : in STD_LOGIC;
        r0 : out STD_LOGIC;
        y0 : out STD_LOGIC;
        g0 : out STD_LOGIC;
        r1 : out STD_LOGIC;
        y1 : out STD_LOGIC;
        g1 : out STD_LOGIC );
    end component;

    -- Stimulus signals - signals mapped to the input and inout ports
of tested entity
    signal clk : STD_LOGIC := '0';
    signal rst : STD_LOGIC := '1';
    -- Observed signals - signals mapped to the output ports of
tested entity
    signal r0 : STD_LOGIC;
    signal y0 : STD_LOGIC;
    signal g0 : STD_LOGIC;
    signal r1 : STD_LOGIC;
    signal y1 : STD_LOGIC;
    signal g1 : STD_LOGIC;

    -- Add your code here ...

begin

    -- Unit Under Test port map
    UUT : traffic_light_state
        port map (
            clk => clk,
            rst => rst,
            r0 => r0,
            y0 => y0,
            g0 => g0,
```

```

        r1 => r1,
        y1 => y1,
        g1 => g1
    );

    -- Add your stimulus here ...
--assertions
--make sure the we never have two greens
--psl property two_greens is never (g1 and g0);
--psl as_one : assert two_greens;
--if g1 or y1 are on then r0 must be on;
--psl property r0_check is always ((g1 or y1) -> r0);
--psl as_two: assert r0_check;
--if g0 or y0 are on then r1 must be on;
--psl property r1_check is always ((g0 or y0) -> r1);
--psl as_three: assert r1_check;

process
begin
    wait for 10 ns;
    CLK <= not CLK;
end process;

process
begin
    wait for 50 ns;
    rst <= '0';
end process;

end TB_ARCHITECTURE;

configuration TESTBENCH_FOR_traffic_light_state of
traffic_light_state_tb is
    for TB_ARCHITECTURE
        for UUT : traffic_light_state
            use entity
work.traffic_light_state(traffic_light_state_arch);
            end for;
        end for;
    end TESTBENCH_FOR_traffic_light_state;

```

C.2 VHDL AND TEST BENCH CODE FOR TEST CASE 2.

C.2.1 THE CROSSWALK SIGNAL VHDL CODE.

```

-----
-----
--
-- Title      : crosswalk_lights
-- Design     : traffic_light_crosswalk

```

```

-- Description : crosswalk light module will be controlled by traffic
light state machine
--
-- crosswalk lights will behave as follows:
-- will add a crosswalk light signal for pedestrians trying to cross
road 1
-- when road 0 light is green, the 'walk' signal will be active;
'hurry' and 'no_walk' will be inactive
-- when road 0 light is yellow, the 'hurry' signal will be active;
'walk' and 'no_walk' will be inactive
-- when road 0 light is red, the 'no_walk' signal will be active;
'walk' and 'hurry' will be inactive

--the control signals for this module are sourced by the traffic light
state machine "state_count" as follows
-- 5 states: "000": road 0 red, road 1 red; in the event of a system
reset: no_walk
-- state 1: "001": road 0 green, road 1 red: walk
-- state 2: "010": road 0 yellow, road 1 red: hurry
-- state 3: "011": road 0 red, road 1 green: no_walk
-- state 4: "100": road 0 red, road 1 yellow: no_walk
--
-----
-----

--{{ Section below this comment is automatically maintained
-- and may be overwritten
--{entity {crosswalk_lightss} architecture {crosswalk_lights_arch}}

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity crosswalk_lights is
    port(
        rst : in STD_LOGIC;
        control_signal : in std_logic_vector (2 downto 0);
        walk : out std_logic;
        hurry : out std_logic;
        no_walk : out std_logic
    );
end crosswalk_lights;

--}} End of automatically maintained section

architecture crosswalk_lights_arch of crosswalk_lights is

begin

    crosswalk: process (rst, control_signal)
begin

```

```

if ( rst = '1') then --Async reset
    walk <= '0';
    hurry <= '0';
    no_walk <= '1';
else
    case control_signal is
        when "000" =>
            walk <= '0';
            hurry <= '0';
            no_walk <= '1';

        when "001" =>
            walk <= '1';
            hurry <= '0';
            no_walk <= '0';

        when "010" =>
            walk <= '0';
            hurry <= '1';
            no_walk <= '0';

        when "011" =>
            walk <= '0';
            hurry <= '0';
            no_walk <= '1';

        when "100" =>
            walk <= '0';
            hurry <= '0';
            no_walk <= '1';

        when others =>
            walk <= '0';
            hurry <= '0';
            no_walk <= '1';
    end case;
end if;
end process crosswalk;
end crosswalk_lights_arch;

```

C.2.2 THE CROSSWALK TEST BENCH CODE.

```

library ieee;
use ieee.STD_LOGIC_UNSIGNED.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

-- Add your library and packages declaration here ...

entity crosswalk_lights_tb is
end crosswalk_lights_tb;

```

```

architecture TB_ARCHITECTURE of crosswalk_lights_tb is
    -- Component declaration of the tested unit

    component crosswalk_lights
    port(
        rst : in STD_LOGIC;
        control_signal : in std_logic_vector(2 downto 0);
        walk : out STD_LOGIC;
        hurry : out STD_LOGIC;
        no_walk : out STD_LOGIC );
    end component;

    -- Stimulus signals - signals mapped to the input and inout ports
of tested entity
    signal rst : STD_LOGIC := '1';
    signal control_signal: std_logic_vector (2 downto 0) := "000";
    -- Observed signals - signals mapped to the output ports of
tested entity
    signal walk : STD_LOGIC;
    signal hurry : STD_LOGIC;
    signal no_walk : STD_LOGIC;

    -- Add your code here ...

begin

    -- Unit Under Test port map
    UUT : crosswalk_lights
        port map (
            rst => rst,
            control_signal => control_signal,
            walk => walk,
            hurry => hurry,
            no_walk => no_walk
        );

    -- Add your stimulus here ...
--assertions
--make sure two signals are never active at the same time
--psl property one  is never (walk and hurry);
--psl assert_one : assert one;
--psl property two  is never (walk and no_walk);
--psl assert_two : assert two;
--psl property three  is never (no_walk and hurry);
--psl assert_three : assert three;

process
begin
    wait for 50 ns;
    rst <= '0';

```

```

end process;

process
begin
    wait for 100 ns;
    control_signal <= "001";
    wait for 100 ns;
    control_signal <= "010";
    wait for 100 ns;
    control_signal <= "011";
    wait for 100 ns;
    control_signal <= "100";
    wait for 100 ns;
    control_signal <= "001";
end process;

end TB_ARCHITECTURE;

configuration TESTBENCH_FOR_crosswalk_lights of crosswalk_lights_tb is
    for TB_ARCHITECTURE
        for UUT : crosswalk_lights
            use entity
work.crosswalk_lights(crosswalk_lights_arch);
            end for;
        end for;
    end TESTBENCH_FOR_crosswalk_lights;

```