

Verification and Validation Spanning Models to Code

Jay Abraham¹
MathWorks, Natick, MA, 01760

Stringent performance requirements and shorter development cycles are driving the use of modeling and simulation. Model-Based Design with an “executable specification” is at the core of this development process. The model is transformed to code using code generation tools for use on embedded processors. Model-Based Design permits verification and validation tasks to be performed earlier in the process when it is easier and cheaper to fix errors. These verification and validation tasks in context of Model-Based Design span models to code.

I. Introduction

AVIONICS systems require rigorous verification and validation (V&V) processes and often utilize the V-model to achieve verification objectives. As shown in Figure 1, the left and right branches of the V correspond to distinctly different activities. The left branch starts with system-level requirements of the design, which are partitioned into subsystems and components. These elements of the design are then specified and implemented at a detailed level, e.g., with Simulink models. V&V with simulation confirms that the correct design is being developed and that the design is being developed correctly. The bottom of the V-model is the coding phase, where the design is transformed to code. The right branch represents the realization and testing of the subsystems and components and their final integration. V&V tasks for the right branch are performed on the code and on the target hardware.

This paper explores comprehensive Model-Based Design V&V processes that can be applied from requirements to delivery of the verified code. V&V activities that span models to code will be discussed. The discussion will be in the context of various aerospace examples to demonstrate specific V&V activities. Simulink and Polyspace will be used for reference purposes. Where applicable, reference to standards such as DO-178C / ED-12C will be provided.

II. Left Branch of the V-Model

Let us begin examining the V&V process by starting with the left branch of the V-model. In order to provide context, we will use an aircraft thrust reverser controller example. The design process for our example thrust reverser controller begins by specifying requirements in a textual format. The next step is to implement the design in a Simulink model. Commercial avionics must be certified to standards such as DO-178C / ED-12C. These standards stipulate traceability from requirements to design to code. The example will show how we can trace from design to requirements.

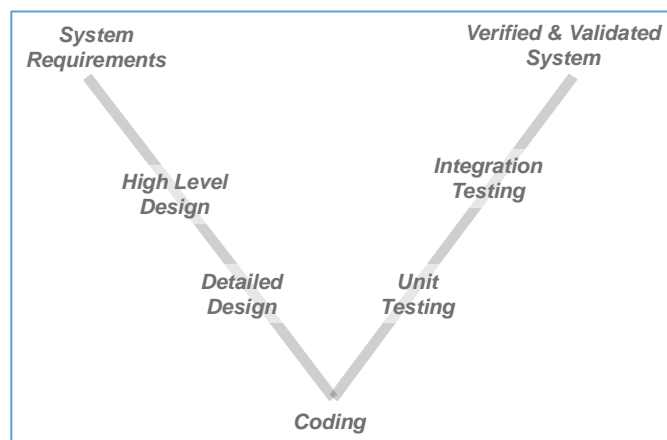


Figure 1. V-Model.

¹ Manager, Product Marketing, 3 Apple Hill Drive, Natick, MA 01760 [13].

A. Requirements Traceability

Consider the model shown in Figure 2. This model designed with a Stateflow chart implements deployment logic for an aircraft engine thrust reverser. One of the requirements for the thrust reverser stipulates that it must never deploy during aircraft flight. The deployment logic uses redundant sensor measurements of air speed, wheel speed, and weight on landing gear wheels to confirm that the aircraft is on the ground. Using requirements linking capability of Simulink Verification and Validation, critical aspects of the design in the model are linked to requirements. The requirements management interface (RMI) allows us to create navigation links between the requirements modeled in Simulink, the associated Simulink objects, and related test cases. For example, as shown in Figure 2, right-clicking on the Stateflow chart traces to a Microsoft Word document that describes a specific requirement.

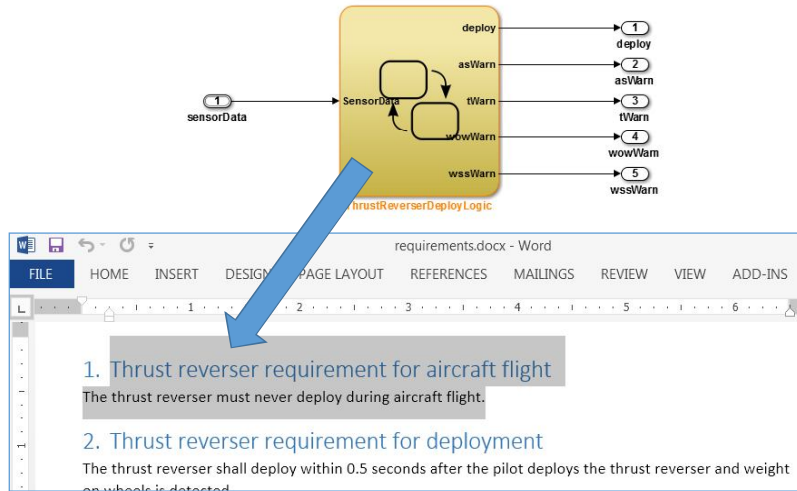


Figure 2. Linking Requirements to a Model.

B. Self-Validating Models

Once the requirements have been associated, they can be used to derive the expected behavior of the model. The expected behavior can be used to create self-validating models with the use of assertion blocks in Simulink. We can confirm that signals do not exceed specified limits or that a particular logical condition does not arise during simulation testing. For example, for the thrust reverser logic, we can specify conditions such as the *deploy* signal and the *weight on wheels* warning signal must never be active at the same time. As shown in Figure 3, we have developed a test harness that encapsulates the thrust reverser model—component under test (center), and we are driving it with tests (left). The logical constructs (right) are checking that the *deploy* and the *weight on wheels* warning signals are never

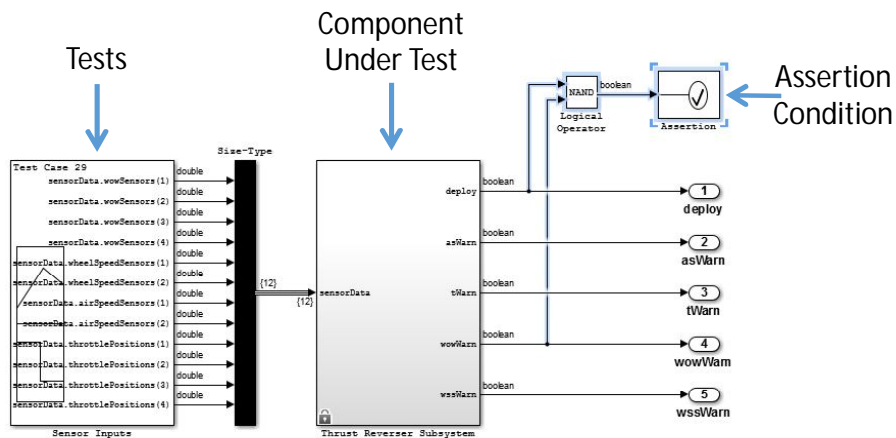


Figure 3. Self-Validating Model.

active at the same time. An assertion is triggered if this condition is violated. By executing these test cases, we can check to be sure that this condition never arises.

C. Checking Models

We then perform requirements consistency checks and confirm compliance of the model to rules and guidelines with Simulink Verification and Validation. These types of checks can identify requirements with missing documents

and inconsistent descriptions as well as confirm compliance to DO-178C standard checks. Basic checks range from support for updating the model to be compatible with the current product release version, to identifying unconnected lines and ports, to checking the root model interfaces. These checks may have to be repeated as the model is updated to address bugs and other issues from subsequent verification tasks.

Sections 6.3.1.f and 6.3.2.f of the DO-331 Model-Based Design supplement of the DO-178C standard states that “High-level requirements trace to system requirements” and “Low-level requirements trace to high-level requirements.” Using Simulink Verification Validation, we are able to determine whether Simulink blocks and Stateflow objects link to a document containing engineering requirements for traceability. If any of the blocks lack traceability information, these blocks are flagged for analysis and follow-up.

D. Functional Testing

The next step is to perform simulation-based tests of our thrust reverser algorithm in Simulink. Closed-loop testing of the controller with a plant model helps engineers investigate all aspects of the system, including algorithms, components, the plant model, and the environment. Tests are created in textual form (e.g., in a spreadsheet or text file) or with graphical waveforms. These testing scenarios create test vectors for simulation with inputs, plant parameters, environmental factors, and other elements. Expected outputs are also required to confirm whether the test passed or failed. Test cases should also have a description that explains its purpose with traceability to requirements for certification to DO-178C / ED-12C.

Formal methods-based functional verification can then be performed to confirm that the design provably meets certain functional requirements. The DO-333 formal methods supplement to DO-178C discusses use of formal methods to complement testing. Functional requirements for discrete systems are typically explicit statements about expected behaviors that a system exhibits and behaviors that it must never exhibit. To formally verify that the design behaves according to these requirements, the requirements statements first need to be translated from a human language into the language understood by a formal analysis engine. Each requirement has one or more verification properties associated with it. These verification properties are used to report whether or not the design meets the functional requirements. Using Simulink Design Verifier, the design is formally verified to conform to the specified property. If a verification property is not met, Simulink Design Verifier will generate a counter example test case to show the conditions under which the property (requirement) will not be met. This test case can then be used to diagnose the error and fix the model.

For our thrust reverser example, if two weight on wheels sensors are false, then deploy cannot be true. This property is described in Simulink as shown in Figure 4. If this property is not correct for the thrust reverser design, Simulink Design Verifier will generate a counter example test case showing an example of a condition that will cause this property to fail.

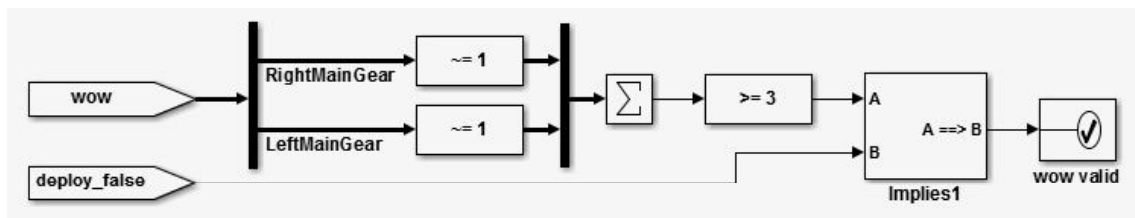


Figure 4. Verification Property. *If two weight on wheels sensors are false, deploy cannot be true.*

E. Verification for Robustness

As our thrust reverser example is being hardened and readied for subsequent verification tasks, it is important to confirm that it is robust. Robustness provides assurance that the design performs as implemented and does not have any unexpected behavior. For example, if the algorithm were to suffer from an overflow or divide-by-zero condition, that could result in unexpected behavior. Because testing may not be exhaustive, formal-methods-based-design error detection with Simulink Design Verifier can be applied to prove that the algorithm will not suffer from these types of defects. Furthermore, if our design is state-based or logic-based, it is important to confirm that there are no dead branches or dead logic. Avionics standards such as DO-178C / ED-12C specifically prohibit existence of dead or unreachable code.

In relation to dead code, DO-178C / ED-12C require coverage analysis. This type of analysis checks that cumulative tests that are executed will activate all aspects of the design. Simulink Verification and Validation produces model coverage reports to identify untested elements of the design. It displays coverage information on the model,

letting us traverse the model for missing coverage and navigate to the associated requirements. We can then determine whether we need to modify the requirements, test cases, or design in order to meet coverage goals. Using Simulink Design Verifier, we can also automatically generate test cases to improve coverage analysis. Various types of coverage analysis can be performed: decision coverage, condition coverage, modified condition/decision coverage (MC/DC), lookup table coverage, boundary value analysis, and signal range coverage.

For our thrust reverser example, using Simulink Design Verifier we are able to automatically generate tests to check for coverage analysis. When we run these tests, we are able to measure cumulative coverage results. As shown in Figure 5, we are able to achieve 100% condition coverage, decision coverage, and modification condition decision coverage.

Upon completion of all of the above V&V tasks, the thrust reverser model in an executable specification

form is ready for production code generation. Using Embedded Coder, we can generate readable, compact, and fast C or C++ code that can be compiled for additional tests on a host computer or the target embedded processor. We have now reached the bottom of the V-model.




















		D1	C1	MCDC
1. thrustrvs	46	100% 	100% 	93% 
2. Thrust Reverser Subsystem	45	100% 	100% 	100% 
3. ThrustReverserDeployLogic	45	100% 	100% 	100% 
4. SF: ThrustReverserDeployLogic	44	100% 	100% 	100% 
5. SF: airspeedOK	2	100% 	NA	NA
6. SF: isRolling	4	100% 	100% 	100% 
7. SF: onGround	17	100% 	100% 	100% 

Figure 5. Coverage Summary Report.

III. Right Branch of the V-Model

The right branch of the V-model focuses on verification of the code and execution of that code on an embedded target. Although code generation automates the task of generating code from models, the model may include legacy code or other handwritten code such as S-functions in Simulink models. Furthermore, generated code may have to be integrated with other legacy code, driver code, RTOS services, and so forth. The end result is mixed code consisting of handwritten and generated code. Interface issues and bugs in the mixed code may result in functional and robustness problems. In order to confirm the integrity of this mixed code, it is necessary to perform checks on the source code.

For example, consider the addition of two variables in a programming language such as C. If the target hardware does not have a floating-point unit, the addition operation must be performed with integers and care must be taken to avoid an overflow.

```
int32_t a, b, c; /* declare signed 32bit integer variables */
... /* some other code here to initialize variables */
c = a + b; /* add variables */
```

The addition operation $c = a + b$ is risky because of the potential of an overflow. The summation could produce results that are greater than $2^{31}-1$. This result will not fit in the 32bit variable c . This condition will result in incorrect computation. This type of error is known as a run-time error. It is important to pay attention to run-time errors as these errors may exist in code, but unless specific tests are executed under particular conditions, the run-time error may not be detected. The errors may cause unexpected system failures.

One may expect that these types of issues should have been detected at the simulation model level. Issues such as overflow and divide-by-zero can be detected at the modeling level with Simulink Design Verifier, but they can only be detected if the model implementation is identical to the code implementation. For example, if the simulation is accomplished in double-precision floating-point but the C code is implemented in fixed-point (integer), then an overflow or divide-by-zero condition may not be detected at the model level.

Examples of code level run-time errors are as follows:

- Non-initialized data—if variables are not set to an initial value, the value is unknown
- Out of bounds array access—occurs when writing or reading beyond the boundary of the array
- Null pointer dereference—occurs when attempting to access a pointer that is NULL
- Wrong results caused by an overflow, underflow, divide-by-zero, negative square root, and so forth
- Unprotected reading or writing to shared variables by multiple threads
- Dead code that will never execute (may lead to a run-time failure)—DO-178C does not allow dead code

A software verification technique known as static analysis can be used to identify and prove absence of certain run-time errors. This technique can be applied to the verification of avionics software at the code level.

A. Static Analysis of Source Code

Static analysis is a verification activity in which source code is analyzed for quality. This technique allows software developers to find and diagnose errors. Results produced by static-code analysis provide a means by which robustness of the code can be measured and improved. In contrast to other verification techniques, static analysis can be performed without executing the program or running tests.

Static analysis ranges from checking source code for compliance to standards, to bug finding techniques. Bug finding can utilize simple techniques and sophisticated methods. The advantage of simple techniques is that they are very fast; however, they may produce false negatives and false positives. False negatives are situations in which the static-code analyzer misses a real error. False positives are incorrectly identified errors. Both false negatives and false positives are problematic. The former may produce a false sense of security, whereas the latter may delay production of the software or create unnecessary rework that impacts the performance or memory footprint of the code.

Sophisticated static analysis combines formal methods with bug finding techniques. An example of a formal method technique is abstract interpretation, a mathematical rigorous approach to prove correctness of code. Tools that use this methodology find errors in the source code and prove the absence of certain critical errors.

B. Software Metrics and Monitoring Quality

Code metrics such as cyclomatic complexity metrics help quantify the complexity of handwritten code when they contain complex decision logic and paths. This metric quantifies the number of linearly independent paths or decision logic.

Using static-code analysis tools like Polyspace Bug Finder, we can generate project-level, file-level, and function-level metrics to evaluate the complexity of code. Using an online dashboard to monitor cyclomatic and other code complexity metrics, we can drill down to the function level to obtain more detailed information about the software. We can also apply thresholds to check if certain software quality objectives have been met. As shown in Figure 6, reports from Polyspace Bug Finder identify project-level, file-level, and function-level metrics. This report can be submitted as part of the code analysis report for DO-178C / ED-12C certification.

C. Code Rule Checking

Languages such as C and C++ provide freedom and flexibility in software program development. This flexibility can result in the construction of inherently unsafe programs. For example, it is possible to write code that has multiple levels of pointer deference as shown in the C code fragment below.

```
*****ptr = 12;
```

Although this code will compile and execute, it is likely that the complex nature of the multi-dereference will result in an error elsewhere in the program. For example, the program author may subsequently omit a level of dereference and this could result in a run-time error. Therefore, significant care must be taken to keep track of all of the levels of pointer dereference and unless extensive analysis has been performed, it may not be possible to confirm that the code is safe. DO-178C / ED-12C stipulate that code standards must be used for development of avionics software. Code standards restrict the use of the programming language to a safer subset. Examples of code standards are Motor Industry Software Reliability Association (MISRA) and Joint Strike Fighter (JSF++). MISRA includes standards for handwritten code and automatically generated code. For example, the MISRA Rule 6.17 specifies guidelines for use of pointers and arrays. The rule stipulates that “*pointer arithmetic shall only be applied to pointers that address an array or array element.*” Polyspace Bug Finder can be used to analyze source code for code-rule standards and produce reports of code-rule compliance.

Table 3.1. Project Metrics

Metric	Value
Project Name	Autopilot
Number of Direct Recursions	0
Number of Files	11
Number of Headers	42
Number of Recursions	0
Number of Protected Shared Variables	0
Number of Unprotected Shared Variables	0

Table 3.2. File Metrics

Metric	Values (Min .. Max)
Comment Density	0 .. 100
Estimated Function Coupling	0 .. 8
Lines	18 .. 207
Lines Without Comment	5 .. 91

Table 3.3. Function Metrics

Metric	Values (Min .. Max)
Cyclomatic Complexity	1 .. 11

Figure 6. Project, File, and Function Metrics.

D. Range Analysis and Results

In order to understand the operational limits of software, it is necessary to know the ranges of outputs and intermediate variables. Static analysis, with the use of formal methods techniques, provides an excellent technique for calculating the ranges of global variables and function outputs. For example, Polyspace Code Prover uses a technique known as abstract interpretation to track control and data flow through the software. It displays range information associated with variables and operators. We can use range information that is calculated to determine that the software does not violate specified range limits.

As illustrated in Figure 7, for the code relating to our thrust reverser example, the display from Polyspace Code Prover identifies that the division operation consists of a range between -1701 to 3276 for the left operand and 9 for the right operand. The resulting range from performing the division is -189 to 364.

```

50
51 static s32 new_speed(s32 in, s8 ex_speed, u8 c_speed)
52 {
53     return (in * 9 + ((s32)ex_speed + (s32)c_speed) / 2 );
54 }
55
56 static char re
57 {

```

operator / on type int 32
left: [-1701 .. 3276]
right: 9
result: [-189 .. 364]

Figure 7. Results from Range Analysis.

E. Finding Bugs

An obvious goal of software development is to produce code that does not contain bugs. Static-code analysis tools like Polyspace Bug Finder can help with this task. Using fast analysis techniques, they can pinpoint numerical, dataflow, programming, and other bugs in C or C++ source code. Bugs are identified in the source code with detailed information to help determine the root cause and source of the bug. For example, when a defect such as an overflow occurs, as shown in Figure 8, Polyspace Bug Finder traces all line numbers in the code that lead to the error condition. Software developers can use this information to debug and fix the code.

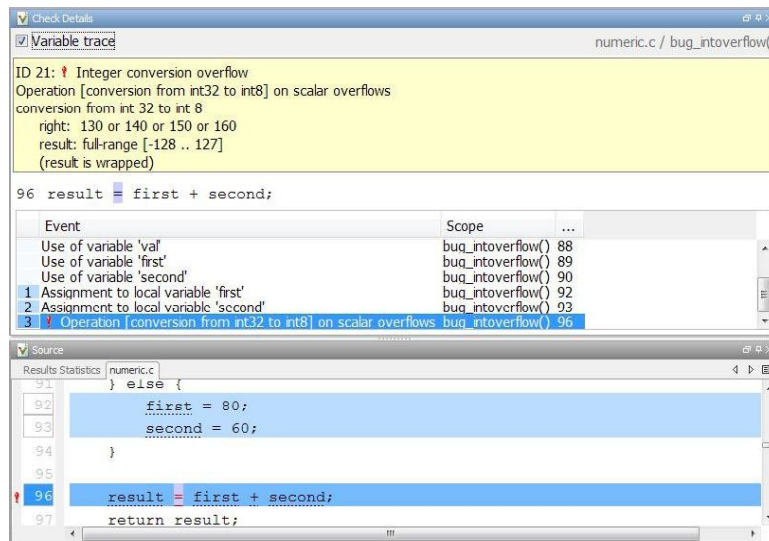


Figure 8. Overflow Bug Identification with Root Cause Information.

F. Proving Absence of Certain Run-Time Errors

When static analysis is combined with formal methods, it is possible to use mathematical proofs to check if the software is free of certain run-time errors. A formal methods technique known as abstract interpretation can be applied to static analysis. This technique uses mathematical techniques to analyze complex dynamic software programs. The source code is interpreted in a mathematical domain to understand the behavior of the program. It produces verification results without requiring program execution or test cases to find and prove run-time errors. DO-333, the formal

methods supplement to DO-178C, recognizes the use of formal methods. For example, the supplement states that requirements involving “always/never” in general are difficult to verify with a finite number of test cases, but may be verified by formal analysis.

Polyspace Code Prover is an example of a static-analysis tool that uses abstract interpretation. It is able to identify where certain run-time errors may occur and proves the absence of specific run-time errors. Using abstract interpretation, Polyspace Code Prover categorizes findings such as (a) proven to fail with a run-time error, (b) unreachable code, (c) proven to be free of certain run-time errors, or (d) unproven. Using Polyspace Code Prover, it is possible to obtain certification credit for certain aspects of DO-178C standard. For example, in Table A-5(6) of the standard, Polyspace Code Prover can be used to identify fixed-point arithmetic overflows, use of uninitialized variables, use of uninitialized constants, and data corruption due to tasks or interrupts conflict. As shown in Figure 9, Polyspace Code Prover depicts color-coded proof-based static-code analysis results directly on the source code.

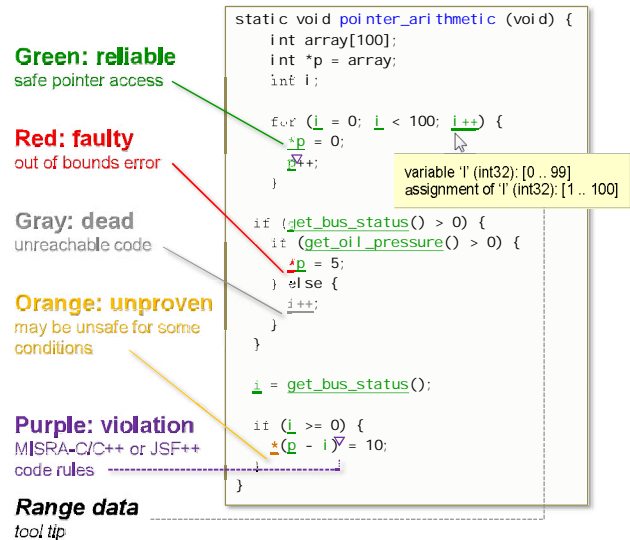


Figure 9. Color-Coded Static Analysis Results.

G. Testing Source Code

Once the code has been verified with static analysis, it can be tested on the host computer (e.g., on a desktop computer) to check that the code has the same behavior as the executable specification (simulation model). This genre of comparison testing is known as “back-to-back” or “in-the-loop” testing. The specific task of testing the code on the host computer is software-in-the-loop (SIL) testing. The generated C or C++ code is compiled for the desktop computer and cosimulated with the original simulation to compare the outputs to identify errors, if any. This step confirms there have been no changes in behavior from one development stage to the next and verifies that a development step did not introduce errors.

The next phase is to cross-compile the code for the target processor and to check that the target-specific code has the same behavior as our executable specification (simulation model). This cosimulation task is known as processor-in-the-loop (PIL) testing. Similar to SIL, PIL provides a method to execute tests created on a development host computer directly on an embedded processor. Tests or closed loop simulations running on the host development computer communicate synchronously with the code running on the embedded target, enabling engineers to run the tests against the code on the target. PIL testing can also be performed with an instruction set simulator (ISS) that executes on the host computer. Running PIL tests on multiple target platforms is one way to assess an algorithm’s robustness to variations in processor, hardware, or compilers. PIL supports execution profiling to help determine Worst Case Execution Time and integration with code coverage analysis tools.

Real-time testing can be performed with hardware-in-the-loop (HIL) testing. This type of testing permits us to execute the control algorithm on dedicated target hardware connected to the physical system with a real-time plant or environment model. The behavior of the system in real time can be compared to expected results.

IV. Conclusion

Verification and validation (V&V) tasks can be applied in Model-Based Design spanning the domain of models to code. Considered in context of the V-model, V&V activities begin with models, continue to code where they confirm that avionics designs have traceability to requirements, meet those requirements, and are robust. These activities can begin early in the design development phase. Static analysis ranging from bug finding to proving the absence of certain run-time errors can be performed on the source code. In-the-loop testing helps confirm that the software behaves as expected on the target processor.

References

- [1] MISRA, Guidelines for the Use of the C Language in Critical Systems, MISRA, 2013.
- [2] T. Erkkinen and B. Potter, "Model-based design for DO-178B with qualified tools," *AIAA Modeling and Simulation Technologies Conference*, 2009.
- [3] R. Estrada, E. Dillaber and G. Sasaki, "Best practices for developing DO-178 compliant software using Model-Based Design," *AIAA Modeling and Simulation Technologies Conference*, 2013.
- [4] W. J. Aldrich, "Using model coverage analysis to improve the controls development process," *AIAA Modeling and Simulation Technologies Conference*, 2002.
- [5] T. Erkkinen, "Production code generation for safety-critical systems," in *SAE World Congress*, 2004.
- [6] MathWorks, "Simulink and Polyspace products," 2015. [Online]. Available: www.mathworks.com.
- [7] J. Pan, "Dependable Embedded Systems," in *Software Testing*, 1999.
- [8] J. Bowen, "Safety Critical Systems, Formal Methods and Standards," *Software Engineering Journal*, 1992.
- [9] P. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," *4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977.
- [10] NIST, "Reference Information for the Software Verification and Validation Process," 1996.
- [11] B. Clark and D. Zubrow, "How Good Is the Software: A Review of Defect Prediction Techniques," Software Engineering Institute, 2001.
- [12] T. Martyn, "Issues in Safety Assurance," SafeComp, 2003.
- [13] RTCA, "DO-178C, Software Considerations in Airborne Systems and Equipment Certification," RTCA, 2012.

© 2015 MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.