Model-Based Design for Large High-Integrity Systems: A Discussion on Verification and Validation

Mike Anthony [*], Matt Behr [†], Matt Jardin [‡], and Richard Ruff [§]
MathWorks
www.mathworks.com

The increasing prevalence of safety standards, including DO-178B, DO-278, and DO-254, for unmanned systems is forcing organizations to re-evaluate strategies for system verification and validation. One result of these re-evaluations is that more and more organizations are adopting a Model-Based Design approach for system design. However, although simulation is well-accepted for requirements validation, models are often not fully leveraged throughout verification and validation processes. This paper discusses how models can be used throughout requirements validation, system design, implementation, and testing. While the discussion references the safety standards mentioned above, the principles can be applied to any high-integrity project employing Model-Based Design. Specifically, techniques for establishing traceability, ensuring conformance to design standards, and verifying the output of each design stage are highlighted. The discussion is centered on fundamental concepts that can be applied to both embedded software and hardware. It builds upon two previous discussions on Model Architecture and Data Management.

INTRODUCTION

"Ninety percent of all innovation in cars today is driven by software," said Ingolf Krueger, an associate professor of computer science and engineering at the University of California in San Diego.[1] As modern systems become more dependent on software to deliver new functionality, the scope and complexity of that software continues to grow. This is true not just in the automotive industry, but also in the aerospace industry, particularly in the realm of unmanned systems. Furthermore, aerospace-related projects are also using more software in safety-critical and mission-critical applications. This creates a unique set of challenges, as the software standards for such applications on aircraft are particularly rigorous.

To help meet the challenges inherent in developing large, complex embedded systems for commercial and defense related aerospace projects, many organizations have adopted Model-Based Design in DO-

[*] Mike.Anthony@mathworks.com. Application Engineering, MathWorks, 3 Apple Hill Dr, Natick, MA.

[†] Matt.Behr@mathworks.com. Aerospace Industry Marketing, MathWorks, 3 Apple Hill Dr, Natick, MA.

[‡] Matt.Jardin@mathwork.com. Consulting Services, MathWorks, 3 Apple Hill Dr, Natick, MA.

[§] Richard.Ruff@mathworks.com. Application Engineering, MathWorks, 3 Apple Hill Dr, Natick, MA.

178B, DO-254, and other certification processes. Whereas coding standards such MISRA C or JSF++ provide guidance on coding, DO-178B and DO-254 provide guidance on the development process itself. Creating and deploying the embedded system is one step in a much larger process that must also include methods for verification and validation (V&V).

NOTES ON WORKFLOWS AND THE AFFECT OF CERTIFICATION

Most modern avionics systems comprise software and hardware components. One of the strengths of Model-Based Design is that it can be applied on both software and hardware projects. The Society of Automotive Engineers (SAE) developed the ARP 4754 standard (see Figure 1) in 1996 to provide guidance on how system level requirements are decomposed into software (governed by DO-178B) and hardware (governed by DO-254).

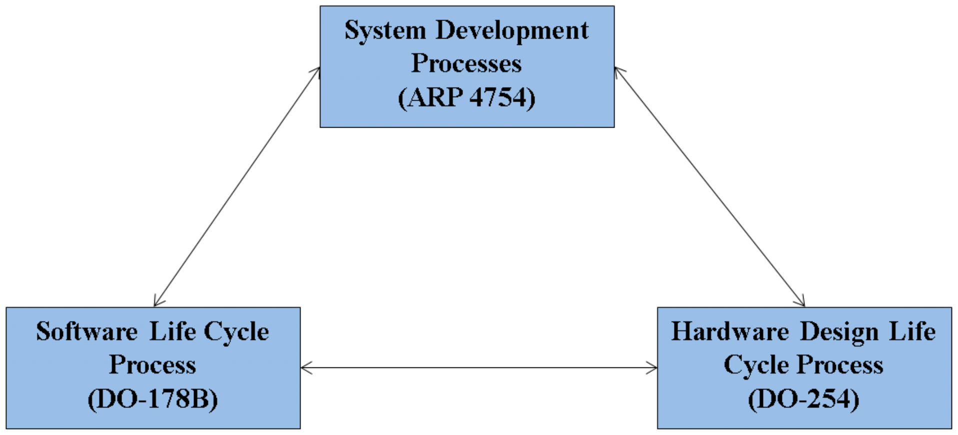

System Development Processes (ARP 4754)
Software Life Cycle Process (DO-178B)
Hardware Design Life Cycle Process (DO-254)

Figure 1: Documents governing aerospace development processes

In November 1981, the commercial segment of the aerospace industry established the DO-178 standard, which governs the development of software. The standard was revised to DO-178A in March 1985 to describe in more detail the process required for software development and verification. DO-178A also introduced the concept of software certification levels corresponding to different levels of criticality. The current versions of software certification standards are DO-178B and DO-278 for airborne and ground-based software respectively. More recently, these standards have been adapted to the development of complex electronics like Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs), as these are also starting to be used more frequently for aerospace applications. The governing document for complex electronics is DO-254.

Details on using Model-Based Design on a project requiring DO-178B can be found in *Model-Based Design for DO-178B*[2] and *Model-Based Design for DO-178B with Qualifiable Tools.*[3] Details on using Model-Based Design on a project requiring DO-254 certification can be found in *Enabling Model-Based Design for DO-254 Compliance with MathWorks and Mentor Graphics Tools*.[4] This paper will concentrate on the process and will not repeat the specific details outlined in the sources above. Refer to the above sources for specifics on tool recommendations, artifacts that can be automatically generated, and the DO objectives that these tools and artifacts satisfy.

Establishing a more consistent development process allows an organization to be more consistent in hardware and software design activities. This paper will first outline tools and workflows used at the model level for both software and hardware. It will then discuss software- and hardware-specific considerations. The steps in the workflows described below are outlined in a linear fashion. In practice, it is highly unlikely that each of these steps would be performed only once. It is almost certain that requirements will change, or that a lack of clarity in requirements will be identified, causing the entire process to be repeated. This is especially true for low-level requirements that are derived from high-level requirements. Even within the workflow, there are many cases where parts of the process will be repeated.

Wherever possible, opportunities for automation will be highlighted. The rigor of certification requires that the design be verified at each stage in the development process. Reusing designs and tests is therefore an essential aspect of an efficient process. Opportunities for reuse are emphasized in the workflows outlined below. Certification also requires that many design artifacts be generated throughout the process. These artifacts are needed to document that the appropriate process was followed, but writing them manually for each step is time consuming. Automating artifact generation when possible can help achieve further efficiency.

The demands imposed by certification (exhaustive design iterations, precise recreation of design and tests, and artifact generation) all lead to increased costs on certified projects. Although these activities are required for both traditional and Model-Based Design approaches, the costs associated with them depend on the approach used. Presuming basic (SEI CMM and CMMI Level 2 or 3) software principles are employed from the outset, increased costs is estimated to be 25 to 40 percent.[5] These activities are also among the least enjoyable for engineers. Particular attention will be given to automation of design steps, design and test case reuse, and automatic artifact generation in the discussion below.

The following workflow assumes the use of Model-Based Design. This workflow is intended to include all of the steps necessary to help achieve compliance to standards such as DO-178B or DO-254. It can be adapted to help address the verification and compliance needs of particular organizations. The discussion of the workflow is broken into three sections. The first section discusses elements of the workflow performed at a model level. These elements are common to both software and hardware development. The second discusses software development and verification processes. The third discusses hardware development and verification processes.

TERMINOLOGY

Though *verification* and *validation* are commonly used terms, their definitions are surprisingly varied in engineering. For the purposes of this discussion, the following definitions will be used. Verification answers the question, “Is the design right?” In other words, verification is the activity of showing that the design was done correctly and behaves as expected. Validation answers the question, “Is this the right design?” In other words, validation is the activity of showing that the expected behavior of the design solves the correct problem.

There are several other common terms that are related to V&V. These include *traceability*, *conformance to standards* (hereafter referred to as *conformance*), *testing*, and *proving*. For clarity these terms are defined as follows:

Traceability. Traceability is a documentation activity. At each phase of software or hardware development, when a new expression of the algorithm is created (for example, when creating a model from textual requirements, or when creating code from a model), it must be documented that the new expression does everything that its predecessor did, and nothing that its predecessor did not do. For example, when the design is created based on the textual requirements, it must be shown that all of the textual requirements are addressed by the design, and that there is nothing in the design that does not explicitly address one or more requirements. Likewise, when the code is created based on the design, it must be shown that all of the features of the design are implemented in the code, and that nothing is implemented in the code that does not explicitly implement one or more of the design features.

C*onformance:* Conformance is the activity of checking an expression of the algorithm against a set of rules. This is a common task in traditional development, particularly in coding. For software, there are many forms of coding standards ranging from industry-wide standards such as MISRA C or JSF++ to company- or program-specific coding standards. For hardware, there are similar standards, including European Space Agency (ESA) VHDL modeling guidelines[6] and vendor-specific standards. In traditional approaches, conformance is checked via a static analysis tool, a manual code review, or both. The goal of this activity is to reduce errors. Conformance standards are not restricted to code or hardware design. For example, textual requirements documents are typically written in a standard format within a given project to promote readability and consistency. This is essentially a conformance activity in the requirements domain. Likewise, the design can be expressed in many different ways. The least dynamic of these would be some form of Algorithm Design Document (ADD). As with requirements, these types of documents are typically written from a predefined template, again to promote readability and consistency. In a Model-Based Design workflow, conformance to standards in the modeling environment becomes especially important because conformance facilitates reuse. Establishing and adhering to a coding standard makes it easier for the next person or project to reuse existing code. The same can be said of a model that adheres to a modeling standard.

Testing: Because *testing* is a very broad term with many interpretations, there is often confusion about what it means. Types of testing include functional testing, structural testing, white-box testing, black-box testing, robustness testing, acceptance testing, unstructured testing, and coverage testing. The workflow described in this paper focuses on functional testing, which is requirements-based. In functional testing, there exists at least one test case for each requirement. Test cases comprise a set of inputs and expected behavior. Functional testing is the activity of executing these test cases against an expression of the algorithm (for example, the design or code) and comparing the results produced by the algorithm with the expected behavior as defined by the requirements. Functional testing is not limited to test cases based on textual requirements. In Model-Based Design, the model is often considered to be an expression of low-level requirements. Test cases generated from the model can be considered low-level requirements-based tests. Functional testing helps to ensure the *design is right*, and thus is one way to perform verification.

Proving: Simply stated, *proving* is the activity of using formal analysis to help ensure, from a mathematical perspective, a certain behavior for all possible scenarios. For example, consider a design based on a set of requirements. For a specific requirement, a functional test is included as part of the verification activities for the design. Assume for this case that the functional test included an input stimulus to the design, and via simulation it was shown that the output of the design as a function of that stimulus met the requirements-based expectation. In this situation, all that has been shown is that for that one specific stimulus, the design behaved as expected. There is, however, still an unanswered question: Will the design behave as expected for all possible input stimuli? There are a few possible methods to address this question. The first is a brute-force approach. This entails simulating every possible stimulus driving the design and comparing the output of the design against the expected behavior, a process which quickly becomes computationally unfeasible for even a small number of inputs. Another approach uses stochastic methods, usually in the form of Monte Carlo simulations. This approach relies on running enough simulations to statistically cover the entire range of stimuli. However, because this approach does not run every possible combination, there is a risk of missing crucial edge cases that cause unexpected behavior. This risk can be mitigated by applying worst-case test cases. Since worst-case analysis is incomplete, it is frequently supplemented with boundary-value testing, in which test cases are executed with values below, equal to, and above thresholds related to a given requirement or data type. Even with techniques such as boundary-value testing, significant effort is required to develop all of these test cases. And once developed it can be difficult to assess whether they are truly exhaustive.

Formal analysis is an alternative to exhaustive testing used to help ensure, from an algorithmic or mathematical perspective, a certain behavior for all possible stimuli. Formal analysis entails capturing the expected behavior as a mathematical expression. A formal mathematical proof is then used to show that the algorithm in question will always exhibit the expected behavior. In other words, formal analysis proves that it is mathematically impossible for the algorithm to provide an output other than the expected behavior. The math required to accomplish this can be challenging. As a result, formal analysis is not suitable for all classes of problems. Some nonlinear algorithms, for example, are difficult to prove using formal methods because the mathematics become complex enough that finding a closed-form solution to the problem is impossible. Other classes of problems are well suited to formal analysis. For example, logic-intensive algorithms and linear math can be proved relatively easily with this approach. Formal analysis can be an excellent complement to functional and stochastic testing.

ARCHITECTURAL AND CONFIGURATION MANAGEMENT CONSIDERATIONS

Developing a componentized design with a modular architecture helps facilitate verification and validation. This paper does not cover methods for defining architectures in the requirements, design, code, or hardware environments, because the definition of these architectures is discussed in detail in *Large-Scale Modeling for Large High Integrity Systems: A Discussion on Model Architecture*[7] and *Model-Based Design for Large High Integrity Systems: Data Management*[8].

Regardless of what tools and processes are used on a large project, the basic concept of verifying small pieces and integrating verified small pieces together in a hierarchy is very common. Indeed, for projects of any appreciable size, this is not only advisable, but essential. It is practically impossible to achieve rigorous traceability, conformance, and verification when these activities are left until the end of a large project and attempted only at the system level. This does not mean that system-level testing is unnecessary. Instead, the system-level efforts should only be attempted after completing a well-organized set of unit-level or component-level tests, which correspond to a modular architecture. This notion of componentization holds for both Model-Based Design and traditional development approaches.

This paper takes a component-level view of a development, verification, and validation process, in which the process defined for a single component is essentially the same as the process for the overall system. The primary difference is that the system comprises components that have already completed the development, verification, and validation process.

A rigorous process will necessarily generate many artifacts showing completion of each step. These artifacts must be managed along with the project files for each version that is verified. Configuration management provides a means of maintaining the results of development and verification efforts for each component. In fact, the DO safety standards require that a configuration management plan be put in place for certified projects. A detailed discussion of configuration management can be found in *Configuration Management in Model-Based Design*.[9]

DEVELOPMENT, VERIFICATION, AND VALIDATION OF THE MODEL

Figure 2 illustrates the steps performed at the model level in both hardware and software development. These include validation of requirements, model development, tracing of the model to requirements, conformance of the model to standards, and verification of the model.

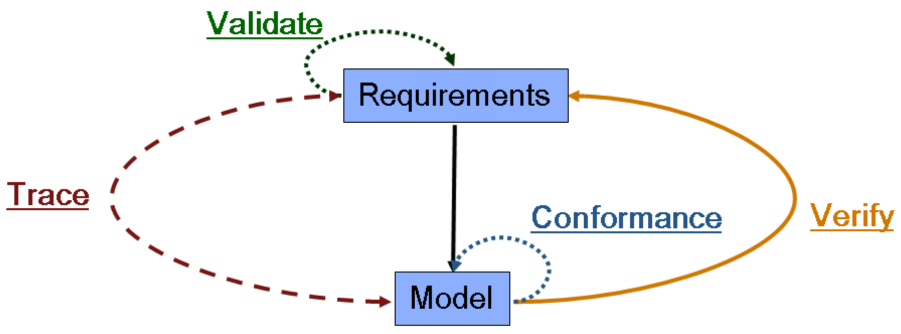


Figure 2: Model verification and validation

Validating Requirements

The first step in the process is validation of the requirements. Recall from the terminology defined above that validation answers the question, “Is this the right design?” Fundamentally, this is a check on whether the textual requirements are complete and accurate. For a component, the textual requirements are usually derived from higher-level requirements. In this case, the validation step is to check that the component-level requirements accurately capture the intent of the higher-level requirements.

Regardless of the tools being used, validation is almost exclusively accomplished via a manual review of the textual requirements. Questions about intent and the “right” design are almost always subjective in nature, and thus difficult to answer solely with a tool. There are techniques related to requirements management that assist in traceability of derived requirements to higher-level requirements. These techniques can assist with requirements validation, but they are not typically sufficient.

Developing the Model

The second step in the process is the development of the model. In a rigorous process, this is the exercise of developing the algorithm to meet the requirements. This implies that the algorithm must be developed to address each and every textual requirement. If the validation of the requirements was done correctly, then building the model that fulfills those requirements will result in the “right design.”

The ability to quickly build and simulate the model is an advantage in the iterative validation process, because the process of model building and simulation often yields further insight into the requirements and design. Whenever possible the model should be traced to the requirements *as it is being developed*. If the model is developed to completion without regard to the requirements, it is likely that the model will fail to satisfy one or more requirements or that it will contain extra or unintended functionality.

Tracing the Model to the Requirements

Tracing the model to the requirements helps ensure that the model does indeed address all of the requirements it should, and that it includes no extraneous functionality. This two-way traceability (or top-to-bottom and bottom-to-top traceability) does not preclude a requirement from being addressed by many sections of the model. Nor does it prohibit a section of the model from satisfying more than one requirement. The key is that the relationship between the model and the requirements must be documented and maintained throughout the design process.

Conforming to Modeling Standards

Conformance to standards is a widely accepted practice in the world of software engineering. Coding standards are used to ensure readability, ease integration, and improve understanding. These same benefits are realized by applying modeling standards. In fact, when automatically generating code from the model, the best way to ensure that the generated code meets a coding standard is by enforcing a set of modeling rules and code generation options in the modeling environment. The modeling standard must not only enforce the coding standard on the generated code, but also enforce a common simulation environment including solver options, data management options, and so on to facilitate integration of models in a large project. Measuring conformance is typically done through manual review and static analysis, although other methods exist. Again, tools that perform static rules checking are common in software engineering. The same capabilities should be applied in the modeling environment. In Simulink®, a commonly used modeling and simulation tool, the Model Advisor feature can be used to check and enforce modeling standards.

Verifying the Model against Requirements

Some of the more common verification techniques include visual inspection, simulation, and formal analysis. Although each of these can be valid in certain cases, visual inspection is, in general, the most difficult to perform and document. Visual inspection is usually completed via a review by several people, and it is not always easy to create and track data artifacts that adequately show completion of the review. Like any manual process, visual inspection is also inherently time consuming and prone to errors.

In a Model-Based Design process, verification is typically performed by simulating the model. The formality with which these simulations are run can vary widely depending on the project. In the most rigorous case, each textual requirement maps to at least one test case. Each test case contains inputs and expected outputs. To execute a test case, the inputs are fed to the model and the model is simulated. The outputs of the model are then compared to the expected outputs. If the output of the model matches the expected output, then the model can be said to meet the requirement on which that test case is based. When the tests case is also traced to a requirement, verification shows that the model behaved as expected based on the textual requirement.

While simulating the model driven by these requirements-based tests, a good practice is to measure model coverage. There are different levels of model coverage, condition coverage, condition/decision coverage, and modified condition decision coverage (MCDC). Choosing the appropriate level of coverage is a function of the project and/or level of certification required. The model coverage achieved with the requirements-based tests is a measure of the requirements coverage. A gap in requirements coverage is apparent if each textual requirement is associated with a test case, all of these test cases have been simulated, and the achieved model coverage is less than 100 percent. This gap can be resolved in two possible ways. The first is that the uncovered functionality in the model in unintended functionality, and should be removed. The second is that the uncovered functionality is necessary for the model to function properly to meet the other requirements-based test cases, meaning that the requirement for this functionality is missing or is incomplete. In this case a new requirement should be written to clarify the need for the functionality in question. This refinement of the requirements at the model level is a key enabler of a more efficient and cost-effective process, as lack of clarity and missing requirements are identified much earlier than the code verification phase, where these gaps would be identified in a traditional process.

Showing that the model behaves as expected for a single specific set of inputs is far different than showing that the model will behave as expected for every possible set of inputs; a task that is better handled via formal analysis. One way to show that the model will meet the requirement for every possible input scenario is to create test cases that represent every possible permutation of input values, execute these test cases against the model, and compare the model outputs to the expected outputs. Even for simple models, developing and running these test cases quickly becomes unworkable. An alternative method is *proving* through the application of formal analysis. In this case, if the expected behavior from the requirement can be expressed in mathematical terms, then it is possible to perform a mathematical proof to show that the model will always exhibit the desired behavior. Because formal analysis is a mathematical analysis rather than a dynamic analysis (simulation), it can be applied only to the types of problems for which closed-form solutions to the mathematical proofs exist. As previously noted, algorithms that contain nonlinear math are difficult or impossible to prove; however, algorithms that are mathematically linear or logic-intensive are excellent candidates for a formal analysis approach.

SOFTWARE DEVELOPMENT WORKFLOW EXAMPLE

Figure 3 expands on the modeling workflow illustrated in Figure 2 to show a more complete software development workflow.

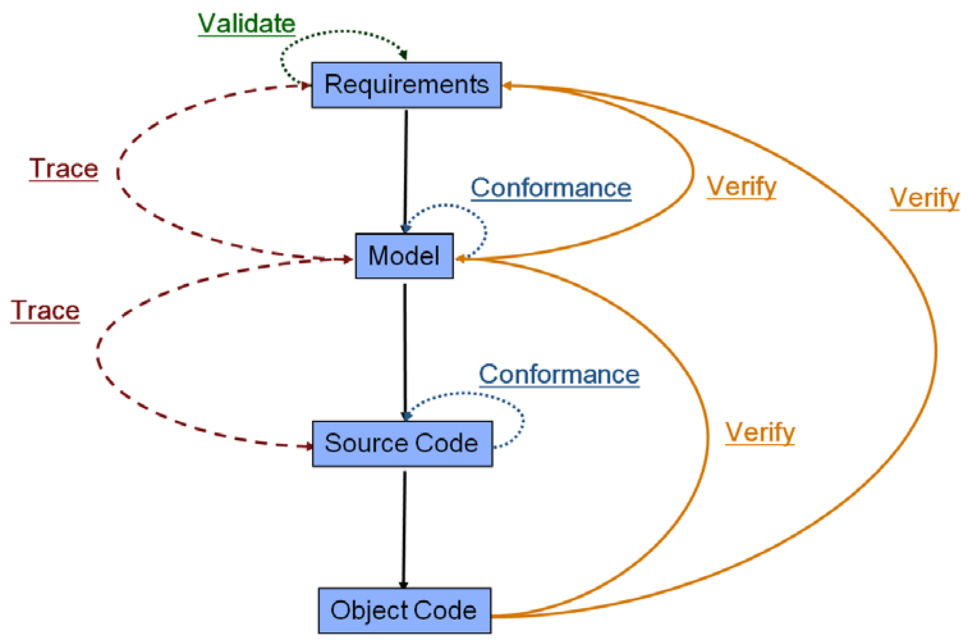


Figure 3: Software development workflow

Automatic Code Generation

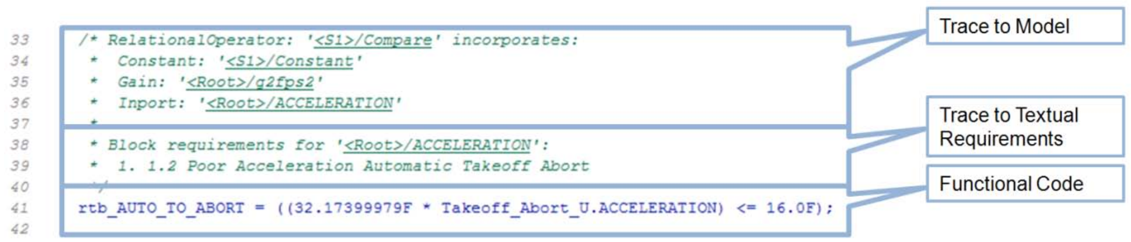
After fully verifying the model against the requirements, the next step is translating the model into the form used for final implementation in an embedded system. For high-integrity software projects, this is almost always C code, though in increasingly rare cases Ada is also used. On some occasions C++ is also used, and there is a subset of C++ that is strongly recommended in these cases[10]. Essentially, it omits any part of the C++ language for which the compiler must determine what code to execute in a given situation.

In a Model-Based Design process, embedded code can be automatically generated from the model. Code generation can help reduce translation errors and accelerates the propagation of changes from the model to the code. For further reading on the advantages of automatic code generation, refer to *Model-Based Design for DO-178B with Qualifiable Tools*[3] and *Checking Code and Models in Production Environments*[11].

Tracing the Code to the Model and Requirements

In a Model-Based Design process, the code is traced to elements of the model, which are in turn traced to the requirements. Code generation helps ensure the traceability of the code to the model. It also provides documentation that can help show that everything in the model is represented in the code, and that nothing extra is in the code that was not in the model. Thus, this traceability maps each line of code to the corresponding piece of the model, and maps each piece of the model to lines of code.

Traceability of the code to the model is not sufficient. It must also be shown that the code implements an algorithm that satisfies all of the requirements for that software component. Ideally, each requirement should be traced to one or more lines of code, and each line of code should be traced to one or more requirements. If the model was already traced to the requirements, some automatic code generation tools can be configured to include the textual requirements for all pieces that were linked to a requirement (see Figure 4). In addition, a *traceability report* detailing the mapping between the model blocks and the resulting code can be generated automatically. This artifact can be used for credit in the certification process. A Model-Based Design process using automated tools as described here can provide traceability of the code all the way to the requirements.



```
/* RelationalOperator: '<S1>/Compare' incorporates:
 *  Constant: '<S1>/Constant'
 *  Gain: '<Root>/g2fps2'
 *  Inport: '<Root>/ACCELERATION'
 *
 * Block requirements for '<Root>/ACCELERATION':
 *  1. 1.2 Poor Acceleration Automatic Takeoff Abort
 */
rtb_AUTO_TO_ABORT = ((32.17399979F * Takeoff_Abort_U.ACCELERATION) <= 16.0F);
```

Figure 4: Generated c-code from Real-Time Workshop Embedded Coder

Conforming to Coding Standards

Having traced the code to the model and to the requirements, the next step is to show that the code conforms to the coding standard specified for the design. Coding standards are common in software development and provide a way to ensure that code written by different programmers has the same structure, organization, style, and level of readability.

Coding standards generally evolve over time, from project to project. The standards are a collection of best practices and lessons learned, and they are derived from issues that have been encountered in the past. The value of coding standards is based on the idea that code written to conform to a well-defined standard is less likely to have errors.

The activity of code conformance checking is an examination of the code against certain rules. This can be done using a manual review process, a static analysis tool, or both. Static analysis is done without compiling and executing the code. One of the more popular C coding standards for the aerospace and automotive industries is MISRA C, developed by the Motor Industry Software Reliability Association. For a detailed discussion on using Model-Based Design and conforming to MISRA-C, see *Checking Code and Models in Production Environments*[10].

Verifying the Absence of Run-Time Errors

Although code that conforms to the standard is less likely to contain errors, there are some situations in which “less likely” is not good enough. This is especially true for DO-178 projects where an error can lead to a catastrophic event, resulting in loss of life. In these situations, organizations must apply more rigor to design and testing to ensure the code is safe. Specifically, this involves ensuring that the code is free of any defects, including run-time errors, which would lead to unintended behavior. Quite often, run-time errors occur during off-nominal conditions, and thus may go undetected during normal testing. Run-time errors can be caused by out-of-bounds array indexes, out-of-bounds pointers, dividing by zero, data type overflows, and uninitialized data. These types of software errors are usually easy to fix, but can be very difficult to detect via manual review, static rules checking, and standard functional tests.

There are several techniques available to help analyze code for run-time errors. As with the verification of the model against requirements, the approaches can be broadly grouped into three categories: brute force, Monte Carlo, and formal analysis. With a brute-force dynamic approach, code is executed with every possible permutation of input values. For example, a function with two double precision inputs would require executing the function (1.797693134862316e+308)^2 times to cover every possible permutation of two doubles. It is clear that for anything but trivial functions a brute-force approach is not feasible. A Monte Carlo approach could be used to show the code is statistically safe, but it does not prove that the code is safe for all possible inputs.

The most rigorous approach uses *abstract interpretation*, a formal analysis technique used to prove the absence of run-time errors. Described simply, this technique solves the problem by mathematically showing the ranges of all variables as they propagate through the code, and analyzing if any possible value within this range could cause a run-time error at each operation.

Compiling the Code

Execution of code is only possible if it is compiled, and there are many options for how and where the code can be compiled. It could be compiled locally on a desktop computer (for Software-in-the-Loop or SIL testing), or it could be compiled using an Integrated Development Environment (IDE) and executed in an Instruction Set Simulator (ISS). The code could also be compiled using an IDE and placed on a target processor (for Processor-in-the-Loop or PIL testing). Each of these approaches results in different object code. Although compilers are an important part of software development and the verification process, they will not be covered here in detail. It is important to recognize that the same source code can be compiled using different compilers and different options, resulting in noticeable differences in the compiled code. Because of this, some software development processes carefully define where testing must be performed. For example, DO-178B requires that tests to be counted for certification credit must be executed against the code on the target processor. This means that the user must execute the code on the same processor as the embedded system; a *similar* target processor will not do. Often a surrogate processor is used for initial testing until the actual embedded processor is available, but in the end, the code must be tested on the actual embedded processor for certification.

One of the advantages to Model-Based Design is that it readily supports SIL and PIL testing. SIL verification shows that the compiled code produces the same results as the original model. Even if SIL verification shows that the code is equivalent to the model, PIL testing on the actual embedded processor is still needed, as there may be numerical differences between SIL and PIL results. This can happen when the target processor implements word sizes differently from the machine being used for SIL. Consider an algorithm that calculates position relative to the center of the earth. Running the code in SIL testing on a PC may give significantly different answers than running the code on a single precision SHARC processor due to rounding errors. If a user assumes these numerical errors do not exist or are negligible, then PIL verification may be seen as unnecessary. This is not a conservative assumption, however, and that is why DO-178B requires testing on the target processor. PIL testing eliminates this potentially dangerous assumption by verifying behavior on the actual processor. Because this is the more rigorous approach, the following sections assume the target code is executed via PIL testing rather than SIL testing.

Verifying the Code against the Model

The next step in the process is verifying the code behaves the same as the model and produces the same results. Determining what test cases should be executed is vital. Frequently the test cases are based on the textual requirements, but this is not always sufficient. To show that the compiled generated code is functionally equivalent to the model in every possible case, the test cases used for this step must provide 100 percent model coverage. Since the model is an expression of low-level requirements, these test cases are verifying the code against the low-level requirements, which is still considered functional testing.

Though closely related, showing 100 percent coverage of the model is not the same as showing 100 percent coverage of the code. There are different forms of code coverage, including statement coverage, branch coverage, and modified condition decision coverage (MCDC). The concept of coverage also applies to the model, though model coverage has a slightly different meaning. For details on these differences, refer to the MathWorks product documentation for Simulink Verification and Validation.[12] Some projects require 100 percent code coverage in testing, which may necessitate additional effort beyond the functional testing.

After choosing the most appropriate level of coverage for the project and creating the test vectors to achieve that level of coverage in the model environment, the task is to execute test cases against the model and the code to ensure equivalent functionality. If the test vectors used provide 100 percent model coverage, and the model and the code produce the same outputs for these test cases, then it has been shown that the code and the model are functionally equivalent in every possible case.

As noted earlier, the most rigorous verification requires execution of the code on the target processor. With Model-Based Design, this is possible via PIL testing, given the appropriate tool chain and target hardware. To realize the maximum benefit from PIL testing, this tool chain must be both efficient and easy to use.

Verifying the Code against Requirements

The final step in the process is very similar to the previous step. The only difference between verifying the code against the model and verifying it against the requirements is the origin of the test cases being executed. Assuming that the previous step verified that the executable code on the target processor is functionally equivalent to the model, the final step is to rerun the high level requirements-based test cases on the executable code on the target processor. A key benefit of PIL testing in a Model-Based Design process is test case reuse. Recall that the requirements-based tests were executed against the model during the verification of the model against the requirements. Reusing the same instantiation of the test cases directly without having to rewrite them is a significant efficiency benefit. To directly reuse test cases, the tool chain being used for Model-Based Design must support PIL testing.

HARDWARE DEVELOPMENT WORKFLOW EXAMPLE

When using Model-Based Design, the process for developing complex electronics such as FPGAs and ASICs is very similar to the process for developing software. In fact, one of the significant advantages of Model-Based Design is that the modeling portion can be abstracted from the target implementation. As a result an algorithm that meets the requirements can be developed before the target implementation is even known. This is illustrated in the hardware development workflow example shown in Figure 5.

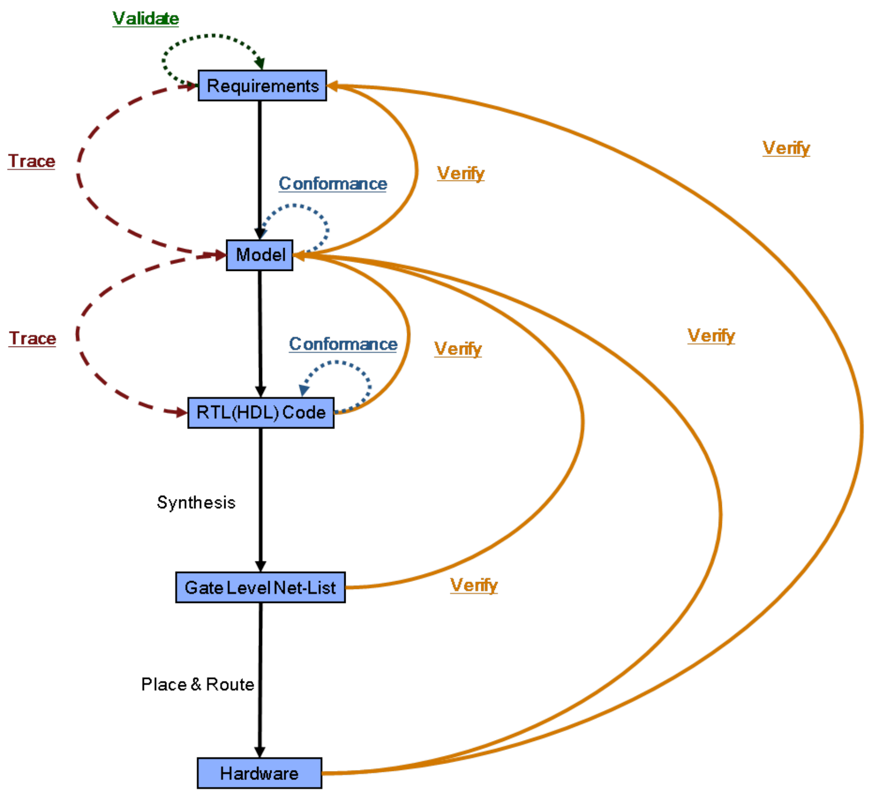


Figure 5: Hardware development workflow

This workflow mirrors the software workflow from requirements through verification of the model against the requirements. In fact, the process is the same through traceability of the code to the model. This portion of the workflow is the same regardless of whether the target is software or hardware. An overview of the Model-Based Design hardware development workflow follows. The discussion is less detailed than in the section on software development because in many steps (for example, traceability) the concepts and the workflow are essentially the same.

Additional Modeling Considerations

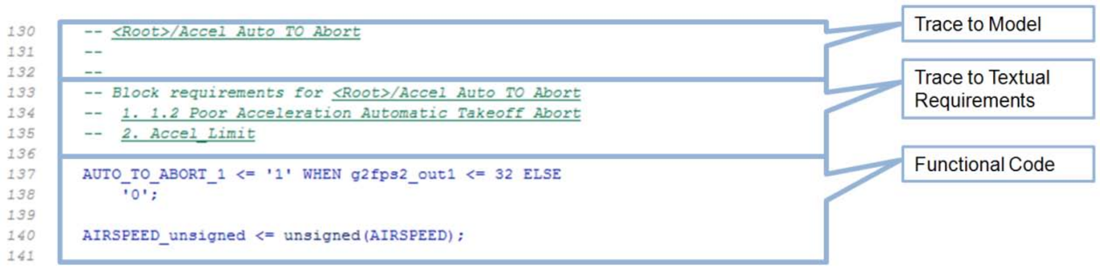
For hardware development, models must be elaborated to include fixed-point design considerations. Hardware development may also require elaboration to specify implementation details such as pipelining and parallel versus serial implementations. This is similar to the elaboration of a model with multiple sampling rates for software development, though the modeling constructs are different. As models evolve, the test cases can be reused to verify that designs still meet their requirements.

Automatic HDL Generation

In the case of hardware design, the design will be implemented in hardware description language (HDL) or Verilog code. Just as for software, this implementation can be automatically generated from the model in a Model-Based Design environment. The same benefits are also realized, specifically the reduced likelihood of translation errors and the ability to more quickly propagate changes from the model to the code.

Tracing the HDL to the Model and Requirements

For the same reasons enumerated in the software workflow, traceability of the HDL to the model and requirements is required. Using Model-Based Design, the capabilities for maintaining and reporting on traceability from requirements to model to HDL code are the same as for software (see figure 5).



```
-- <Root>/Accel Auto TO Abort
--
--
-- Block requirements for <Root>/Accel Auto TO Abort
--  1. 1.2 Poor Acceleration Automatic Takeoff Abort
--  2. Accel_Limit

AUTO_TO_ABORT_1 <= '1' WHEN g2fps2_out1 <= 32 ELSE
    '0';

AIRSPEED_unsigned <= unsigned(AIRSPEED);
```

Figure 6: Generated HDL code from Simulink HDL Coder**Conforming to HDL Standards**

Similar to software, organizations have developed coding standards for VHDL. One example is the set of VHDL standards developed by the European Space Agency[6]. Ultimately, HDL checking is performed in an HDL authoring tool. However, as with software, model structure and configuration options can affect generated code. Proper enforcement of standards at the model level will help in generating HDL code that also conforms to standards.

Verifying the HDL against the Model

HDL differs from c-code in that it can be simulated before being synthesized and deployed to hardware. As a result, there is an additional verification step required. In order to verify generated HDL, it can be cosimulated with Simulink®. Cosimulation allows test cases to be reused and results to be compared with those from the original design. This is roughly equivalent to SIL testing in the software development workflow.

DO-254 Levels A & B require code coverage, although less specific guidance is given compared to DO-178B. While HDL coverage is measured within the HDL simulation environment, the process of ensuring coverage is essentially the same as that for software. Developing test cases that prove requirements are met and fully exercise the design at the model level can streamline or reduce testing at the HDL level.

Verifying the Netlist against Hardware Synthesis

A major difference between software and hardware development is evident at the compile phase. Whereas software is compiled from source code into object code, HDL code is first synthesized before going through place and route for final implementation on the target hardware. The same cosimulation capabilities for the HDL code exist for cosimulating the netlist, however simulation is much slower.

Verifying the Hardware against Requirements

Place and route is the final stage in hardware implementation, and it produces an FPGA or ASIC. The output produced by this hardware must also be compared with the output from models and the expected output defined in the requirements to help ensure that requirements are still being met. This is accomplished using hardware-in-the-loop (HIL) testing. Again, test cases and analyses used at the model and HDL cosimulation stages can be reused. The software equivalent of HIL testing is PIL testing.

Additional Notes on Hardware Development

EDA tool vendors provide additional tools for hardware verification. For example, equivalency checking tools based on formal methods can be used to prove the equivalency between HDL and the netlist. There are also tools focused on analyzing implementation effects rather than algorithmic effects. An example is static timing analysis tools, which help ensure the algorithm will perform as expected on hardware despite manufacturing variations, temperature changes, voltage fluctuations, and so on. These tools can complement the workflow discussed here.

CONCLUSION

This paper introduced a rigorous Model-Based Design process for software and hardware development. Each step of the development process was described, and in some cases tools to aid in individual steps were identified.

Model-Based Design provides a common design workflow for both software and hardware development projects. This allows acceleration of the design process as model development can begin before the target is chosen. Furthermore, rework is significantly reduced if the desired target (software or hardware) is changed late in the development process. At an organizational level, increased consistency between software and hardware development processes can reduce the number of tools and processes to be managed.

The automation and reuse of test cases provide significant cost and schedule benefits compared to a traditional development process. These benefits stem from the ability to author test cases based on requirements just once, and execute these test cases in an automated way on a model, on software compiled locally or running on a target processor, or hardware designs being co-simulated or running on a target FPGA or ASIC.

The ability to automatically generate documentation and artifacts, particularly during the cyclical process of change requests and regression testing, also provides significant time and cost savings. The most rigorous processes require that each step in the workflows discussed above must be documented. In a traditional development process, this requires a significant amount of manual effort. This effort is magnified by the repetitive nature of these tasks. Model-Based Design enables many documentation steps to be automated efficiently.

While a thorough discussion is beyond the scope of this paper, it should be noted that many safety standards, including DO-178B, DO-278, and DO-254 allow for verification tools to be qualified. Qualifying a verification tool allows certain steps in the certification process to be skipped or streamlined. Tool qualification can further magnify the cost savings realized by automation. Please see *Model-Based Design for DO-178B with Qualified Tools*[3] for a detailed discussion on tool qualification for DO-178B certification.

Like many industries, the aerospace industry is seeing a continued increase in the amount and complexity of embedded software and hardware. In this environment, finding more efficient methods to develop these systems while also maintaining the highest levels of quality is essential. Model-Based Design provides an efficient process to develop, verify, validate, and document complex embedded systems.

REFERENCES

[1] Worthen, Ben. (2010, March 31). Now, Even NASA Is Involved in Toyota Crisis. *The Wall Street Journal*, p. B1.

[2] MathWorks. (2010). *Model-Based Design for DO-178B.* White Paper.

[3] *Model-Based Design for DO-178B with Qualified Tools*. **Erkkinen, Tom and Potter, Bill.** AIAA 2009-6233. Chicago, IL: 2009 AIAA Modeling and Simulation Technologies Conference and Exhibit.

[4] MathWorks. (2010). *Enabling Model-Based Design for DO-254 Compliance with MathWorks and Mentor Graphics Tools*. White Paper.

[5] Hilderman, Vance and Baghai, Tony. Avionics *Certification: A Complete Guide to DO-178 (Software), DO-254 (Hardware)*. 1st edition Avionics Communications Inc., 2007.

[6] *VHDL Modeling Guidelines,* Creasey, R. and Coirault, R. European Space Research and Technology Centre. ASIC/001, Issue 1, September 1994.

[7] *Model-Based Design for Large Safety-Critical Systems: A Discussion on Model Architecture.* **Anthony, Mike and Friedman, Jon.** San Diego, CA : s.n., 2008. AUVSI Unmanned Systems.

[8] Model-*Based Design for Large High Integrity Systems: A Discussion on Data Modeling and Management.* **Anthony, Mike and Behr,** Matt, Breckenridge, CO: 2010-9054. American Astronautical Society.

[9] *Configuration Management of the Model-Based Design Process.* **Walker, Gavin, Friedman, Jonathan, and Aberg, Rob.** Society of Automotive Engineers. SAE 2007 TRANSACTIONS. JOURNAL

[10] *Position Paper: Object-Oriented Technology (OOT) In Civil Aviation Projects: Certification Concerns.* **Certification Authorities Software Team (CAST)**. January 2000. http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-4.pdf

[11] MathWorks (2003). *Checking Code and Models in Production Environments*. **Erkkinen, Tom and Hachmeister, Damon**. White Paper.

[12] Model Coverage section of DOC - http://www.mathworks.com/access/helpdesk/help/toolbox/slvnv/ug/bsgdhe4.html