

# Early verification and validation using model-based design

WITH A MODEL-CENTRIC APPROACH, EMBEDDED-SYSTEM DESIGNERS CAN VERIFY AND VALIDATE THEIR PRODUCTS AS A PARALLEL ACTIVITY THROUGHOUT THE DEVELOPMENT PROCESS.

When The MathWorks introduced Matlab technical-computing software more than 20 years ago, many of the first users were control-system designers. Anyone who had laboriously inverted matrices by hand to crank through to a controller design fell in love with Matlab at first sight.

Matlab simplified linear-control design, but, in the real world, systems are seldom linear. So even after you designed the controller, testing and tuning it meant building a hardware prototype of the system and coding the algorithm. In other cases, there was no prototype, so testing couldn't take place until late in the development process.

To help verify algorithms before committing them to hardware, engineers turned to numerical techniques to simulate the behavior of the system that the algorithm would control, or the "plant." Control engineers learned to hack C or Fortran programs together to try to model the system, borrowed numerical integration routines that they thought might work for their type of system, duplicated their control algorithm in the system-model program, and simulated the whole thing. This entire simulation-development process was time-consuming and challenging—if you could make it work at all.

The MathWorks in 1990 released Simulink, a software environment for modeling and simulating dynamical systems. Using Simulink in control design offered two big advantages. First, it provided an intuitive, block-diagram environment for modeling both the algorithms and the plant, as well as nonlinear, real-world effects that might affect system behavior. Second, it included a simulation engine that its creators based on state-of-the-art numerical-integration methods. These core features greatly simplified control engineers' task of verifying controllers through simulation. But control engineers still had to eventually code algorithms to test them on hardware prototypes or actual systems.

This process got much easier about five years later with the introduction of automatic code generation from Simulink models. Control engineers no longer had to worry about errors from translating the algorithm model into code for debugging and testing the code running in the prototype system.

The next step in the evolution of control engineering was a major challenge: production-code generation. Rapid prototyping code generally contains a lot of debugging routines, data-

collection code, host-target communications code, and other additions useful for interactive testing. Generally, this code was not streamlined enough for deployment in the deliverable system. Code-generation tools evolved to produce code efficient enough to deploy in the production embedded system. Today, many industries consider automatic code generation from control models for production deployment a best practice.

## MODEL-BASED DESIGN

The rapid growth in processor speed and memory that enabled the development of modeling, simulation, and code-generation tools on the desktop also enabled embedded-software developers to increase the functions and complexity of embedded controllers. This step in turn drove the need to move beyond traditional code-development techniques using text editors and debuggers to center design on models. This model-centric development approach is known as model-based design (Figure 1).

With model-based design, teams use models to develop their

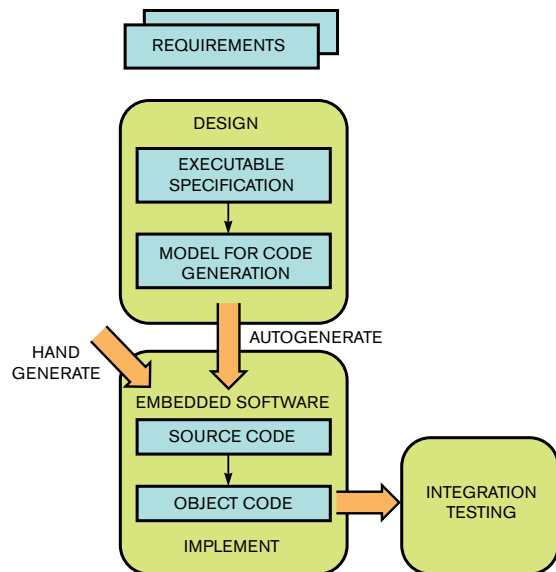


Figure 1 Model-based design streamlines embedded control design with modeling, simulation, and automatic code generation.

designs from the written requirements. With a simulation engine, these models become an “executable specification.” The ability to execute the design is a huge benefit to a team trying to develop and review a specification. Once you have reviewed the high-level model, you embellish the model with design details in preparation for translating it into code. Automatic code generation from the detailed design models greatly streamlines the implementation process and removes the chance of introducing translation errors going from the design to the code.

Embedded control systems have traditionally followed the V diagram as a development process (Figure 2). This process leaves all verification and test on the right side of the V, after design and implementation are complete. For a traditional, C-code-based embedded-control-development process, integration testing often precedes other forms of increasingly high-level testing, such as hardware-in-the-loop testing and final system test with the actual system under control. Although this development sequence has helped organize complex system design, it does have some drawbacks: The sequence does not consider verification and test until the end, when it is most expensive and time-consuming to fix any errors you find; you must implement all components to test a system; and it fails to account for iteration in a development process.

Model-based design enables the use of verification as a parallel activity that occurs throughout the development process (Figure 3). Doing test and verification along every step of the development process means finding errors at their point of introduction. You can reiterate, fix, and verify the design faster than in the traditional V-diagram process. How do you go about achieving early verification, therefore lessening the time you spend at the end of development testing and debugging your design? The following sections outline some best practices.

### MODEL VERIFICATION AND VALIDATION

The primary way model-based design achieves verification and validation is through testing in simulation. Although

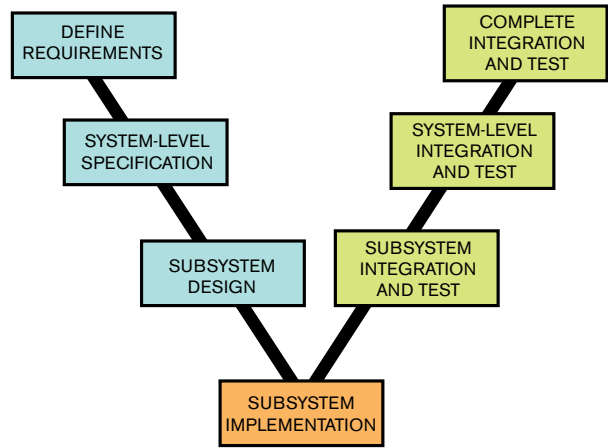


Figure 2 You can capture the traditional embedded-system process as a V diagram showing the reliance on test and verification on the right side of the V, late in the development process.

many organizations do some form of modeling, too many apply simulation in an ad-hoc manner that does not maximize the potential verification benefits. Simulation alone cannot find all errors; however, it is a huge step forward, and you can do it almost as soon as you design a model. Iterating in a modeling environment is fast and easy.

Modeling individual components is useful and may be necessary to complete a complex design; however, don't let that advantage stop you from first modeling the system or environment your component will operate in. By modeling the whole system in a single environment, you can quickly see how the functions of your component will interact with other components and how the integrated components will behave in the deployed system or environment. You can find missing requirements for your component or others. By having a system model to return to as you iterate one component, you can assess how design iterations will affect system functions.

Developing tests in parallel to design and development allows early detection of potential problems, and it significantly reduces the cost and time required for fixing them. By thinking about testing while you develop the model, you will design better for testability, thus ensuring that the design is fully testable. This principle applies far outside the embedded-system area, but embedded-system developers often overlook it. Maybe it is a case of thinking that you can do anything in software, or maybe the documented development process overlooks it. As with new agile-software-development processes, however, developing tests before or in parallel with the design models is a best practice.

Almost every testing scenario involves varying something: inputs, plant parameters, environmental factors, or other elements. Time and expense often limit how much variability you can test for; however, by testing in a simulation environment, you can simulate test cases more quickly and, if the processing power is available, in parallel. Exploring the entire parameter space in simulation can also narrow down the critical tests to run in real time.

Every organization has standards or best practices for design and implementation. Many of these standards are undocumented but exist in key people's heads. Formalizing the standards and incorporating standard checks into your model-development

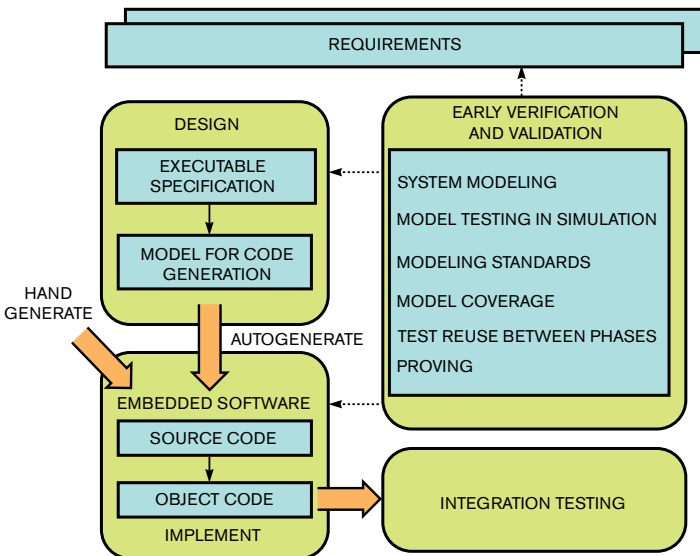


Figure 3 Model-based design enables new techniques for verification and validation to help teams find and fix errors early in requirements, designs, and code.

process is straightforward but can have a large impact by reducing early the number of “silly” errors, ensuring models are more readable for sharing between team members and are more maintainable in the future. Modeling standards can be as simple as verifying that all your inputs and outputs are connected to something as complex as meeting industry standards. The key is developing consistent checks and then driving compliance with them throughout the organization.

Determining when a test suite is sufficient is often more art than science because you typically base this determination on the judgment of a design or test engineer. For software components, many teams use code coverage as a more objective measure of test completeness. You can similarly use model coverage for model testing. Coverage is a measure of how much of the logic in a model—or in source code—you exercise during testing. Modified condition/decision coverage is a stringent, well-accepted measure of coverage.

A core principle of most quality standards is to document. Document your requirements. Document your process. Document your results. Without documentation, it is impossible to trace what you did; it is impossible to show someone, such as a customer, that you met his needs; and it is impossible to repeat your results. Although documentation is often tedious, many tools help automate documentation activities by generating standard reports. It is also impossible to overestimate the time savings good documentation will provide if you discover a problem later or you want to reuse a design.

## CODE VERIFICATION

These practices are a great starting point for early verification at the model level. Eventually, implementation and deployment onto production hardware become necessary. At this point, code verification becomes the focus. How does model-based design help?

Automatic code generation is one method that helps. By verifying your design at the model level and then generating code directly from it, you need verify only that the model and code are equivalent. This workflow is ideal. In the real world, it is sometimes impossible to generate all the code you need from a model. You may have middleware and device-driver code that you develop using traditional processes, or you may use legacy code for some functions. For these cases, there are some additional best practices to verify code.

You can test hardware almost everywhere; however, it is often disconnected from testing in simulation. Many factors can drive that disconnection: Groups who run tests on hardware differ from those who model the design, and the software that runs in the lab differs from the software in the design. However, when you run the same tests on your model that you run in the lab, you know exactly how the design should perform in the lab. To verify the equivalence of code to the design model, you can use the same test harness for model testing to test the software implementation of the model compiled and running on the embedded target. Running tests on a component design model simultaneously with code on an embedded target is a co-simulation step called PIL (processor-in-the-loop) testing. Tools are now available to execute the tests—which a developer created on a host computer, such as a PC—on the software synchronously with running it on an embedded pro-

[Go to www.edn.com/ms4318](http://www.edn.com/ms4318) and click on Feedback Loop to post a comment on this article.

cessor. Comparing test results of the embedded code with the original model helps you ensure that the behavior of the component remains unchanged after compilation and downloading and that the code is functionally correct.

Runtime errors in embedded code can be especially challenging to find, and, once you find them, they are difficult to debug. Examples include overflows and underflows, division by zero and other arithmetic errors, out-of-bound array access, illegally dereferenced pointers, read-only access to noninitialized data, and dangerous type conversions. Until recently, there were essentially only three options for detecting runtime errors in embedded software: code reviews, static analyzers, and trial-and-error dynamic testing. Code reviews are labor-intensive and often impractical for large, complex applications. Static analyzers identify relatively few problems and, more important, fail to diagnose most of the source code. Dynamic, or “white-box,” testing requires that you write and execute numerous test cases. When tests fail, debugging the problem can be difficult. Code-verification tools, based on formal methods, allow you to prove the absence of runtime errors and provide strong assurance of the code’s reliability. When you use them in addition to testing, as part of a development process, code-verification tools provide other techniques for identifying design and implementation errors that would be difficult to find and costly to correct in later testing phases.

System modeling and simulation tools, such as Simulink, help streamline the task of designing and verifying complex algorithms without expensive hardware. In place of hand-coded, hard-to-maintain simulations, control designers can quickly develop complex algorithms and system models and test their algorithms before committing them to hardware. Years of experience have given rise to approaches that provide automatic code generation to support real-time testing in prototyping systems and later for deployable embedded code. Today, model-based design sees use in a variety of applications, including controls, image processing, audio, communications, and signal processing.

One of the primary benefits of model-based design is the opportunity to do rigorous verification and validation in parallel with all other development steps, especially early in the development process. You can maximize these benefits using a series of best practices that adopters of model-based design have discovered through hard experience. Practices such as developing tests along with models and reusing model tests on code and hardware, among others, can significantly reduce the risk of a development project missing quality or delivery goals due to the discovery of errors late in the process. **EDN**

---

## AUTHORS’ BIOGRAPHIES

*Brett Murphy manages the technical-marketing teams for verification and validation, test and measurement, rapid prototyping, and hardware-in-the-loop products at The MathWorks. He holds bachelor’s and master’s degrees in aerospace engineering from Stanford University (Palo Alto, CA).*

*Amory Wakefield is a product manager for simulation and test applications at The MathWorks. She has bachelor’s and master’s degrees in electrical science and engineering from the Massachusetts Institute of Technology (Cambridge).*