Proceedings of the 17th World Congress
The International Federation of Automatic Control
Seoul, Korea, July 6-11, 2008

# Verification and Validation Integrated within Processes Using Model-Based Design

**Brett Murphy, Chris Hayhurst, Jon Friedman, Coorous Mohtadi, Richard Anderson and Pieter Mosterman**

***The MathWorks, Inc.***

Abstract: Verification and Validation have always been a key part of the process for producing embedded control systems. With the advent of Model-Based Design as an alternative method for generating embedded software, the need for verification and validation remains and, up to the present, conventional approaches for doing verification and validation have largely been followed. However, conventional and new techniques fully integrated into Model-Based Design have the potential for greater returns, and will be presented in this paper.

Keywords: Model-Based Design, Verification, Validation, Coverage, Static Analysis

## 1. INTRODUCTION

Traditional control system software development involves paper specifications, design and hand coding followed by verification activities such as code inspections and unit/integration test. Many of these activities lack tool automation and involve manual interaction. Thus they are error prone and time consuming**.** Lack of tool chain integration provides another opportunity for errors to be injected into the software that are often detected late and at high costs to the development process.

In addition, two trends exacerbate the traditional development problems:

- the increasing amount of software in embedded control systems
- the need to meet safety-critical software development standards like DO-178B and IEC61508.

To help address these development challenges, controls engineers are increasingly adopting Model-Based Design for creating executable specifications and automatically generating control code. This approach is now widespread in the automotive industry and adoption is increasing rapidly in aerospace and industrial automation.

Experience now shows that a key enabler of maturity and effectiveness in this approach is the use of integrated Verification and Validation (V&V) techniques. Without a repeatable and tool based approach to V&V, errors found in the implementation tend not to be corrected at the source of the problem but further down the 'V'. The barriers to refining or changing the design after the initial cycle are therefore too high to fully realize the potential benefits of Model-Based Design. Tool based V&V, fully integrated in Model-Based Design enables the engineer to have confidence to make changes in the executable specification model and re-test quickly at every stage of the 'V' this retaining the integrity of the whole process.

This paper will:

- Give a brief overview of Model-Based Design for embedded control design
- Describe the current possibilities of V&V techniques that are integrated into Model-Based Design
- Explain the various levels of adoption of Model-Based Design that such integration makes possible and the potential returns

## 2. MODEL-BASED DESIGN FOR EMBEDDED CONTROL

Control software development using Model-Based Design often begins with system requirements. The requirements are allocated to hardware and software and a refinement phase occurs. Eventually a detailed software design model is produced and a V&V analysis/model test phase are performed to ensure the model satisfies the known and derived requirements. The test results are also examined to ensure that the design satisfies the requisite structural model coverage – see Aldrich, B. (2002).

Requirements traceability between model design components and the requirements specification is usually involved. Designs specified using state machines and block diagrams can have automated links that support bi-directional traceability to higher-level requirements in popular documentation and requirements management tools. Industry standards such as DO-178B RTCA (1992) and IEC-61508 require these high degrees of traceability.

Automated documentation tools help complete the design phase by making it easy to prepare and execute formal and informal requirements and design reviews. These software development reviews are now executed much more quickly and easily than before by having rigorously tested models with high degrees of traceability easily accessible.

After satisfying the design phase, the "golden reference model" is placed into configuration management and code is automatically generated.

The code is then assessed for performance, size, and resource consumption. Iterations on the model will often occur to optimize the generated code for a particular hardware platform. Model guidelines used during design can make it easier to get more efficient code during the first iteration. Software integration gradually takes place during the iterations. Integration involves application software and real-time operating system software; external or legacy code integration for some software components such as look-up tables, calibration data, scaling choices and low level device drivers for software to hardware interaction.

## 3. INTEGRATED VERIFICATION AND VALIDATION

As implementation and integration winds down, V&V of the generated software takes place. However the sooner V&V efforts occur, the sooner errors are detected and resolved.

With recent technology developed from lessons learned in aerospace and automotive development programs, it is now possible to perform enhanced V&V of the models and the generated code. The techniques and methods described here are based on approaches from Erkinnen (2006) but are updated to include new tools and technology. This section describes the main approaches and their use in Model-Based Design.

### 3.1 Requirements Tracing

Industry standards, such as DO-178B, see RTCA (1992) and CMMI, see CMMI web page, require bidirectional traceability of requirements, perhaps originating in other requirements tools, throughout development. It is also important for the code to be traceable to the model, so that it may be reviewed and verified. Using a requirements management interface, textual requirements can be traced to the model and then the C code and vice versa. Together, these capabilities provide a complete traceability path from the code to the requirement (FIG. 2). Even more important, linking the requirements to the whole design flow including test vectors ensures the requirements are consistent and unambiuous and the design minimal – see Ghidella, J (2005). Production code generators place high emphasis on code clarity, and can include the generation of HTML links into the generated C code that when clicked highlight the source block(s) used to create the code and the requirements that motivated the use of the block so creating end to end tracebility.

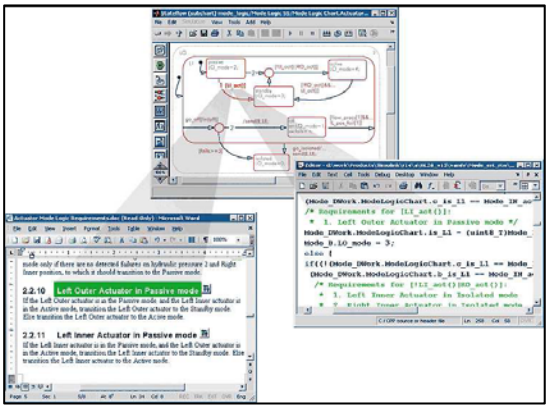

FIG. 1: Requirements Tracing

### 3.2 Structural Coverage Testing

Because of the cost and scarcity of physical prototypes, it is very useful to test the model before deploying it on the target embedded controller for build and integration. Source code-based testing has existed for many years, but new tools and techniques now allow for model testing and structural coverage, see Aldrich (2002) and Edwards (2004). The usage scenario is that a developer fully stresses the control algorithm to verify its design integrity using simulation and coverage analysis. Examples of poor design integrity include numerical overflow and/or unreachable logic. Stress testing of the model using minimum and maximum numerical values helps ensure that overflow conditions will not occur. This type of testing is easy with simulation, but unreachable logic is not so easily detected because detection requires structural coverage.

Unreachable logic differs from deactivated logic in that the latter is known to the developer and is deactivated for a reason. Unreachable logic means that something was missed during specification, implementation, or test creation.

Model assertions are then used to determine if the tests passed or failed. If a signal exceeds its boundary during a simulation or test, an assertion is raised that can either stop the execution or be recorded for post analysis.

Model coverage analysis assesses the cumulative results of single or multiple test suites to determine which modeling elements were executed and which were not. Tests can also be designed to set pass/fail criteria based on achieving certain coverage metrics. Although not reaching a coverage target can have many causes, a common one is missing test cases. Also, lack of coverage can also result from unreachable logic, discussed above. Coverage analyses are well established in imperative programming languages such as C, and Ada, but these types of analysis were not made available at the model level until recently. Modified Condition/ Decision Coverage (MC/DC) is considered by the Federal Aviation Administration (FAA) to be the most stringent coverage level and is necessary in the development of safety-critical systems. This coverage analysis, among others, is now available as an integrated part of software tools using Model-Based Design and is conducted by performing simulation runs. When done within the modeling and simulation stage this enables the automatic logging and reporting of coverage metrics for the model (FIG. 1).

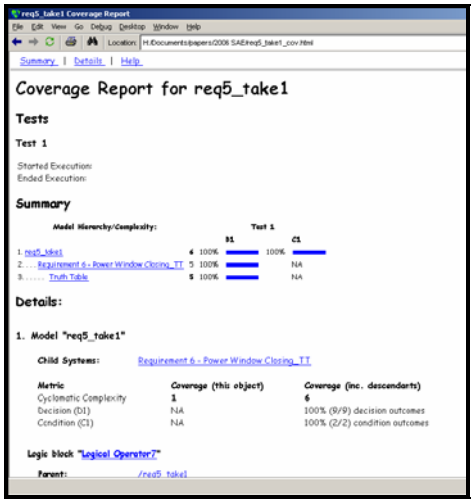


FIG. 2: Model Coverage Analysis

An important aspect of performing this model testing and structural coverage is the definition of appropriate test patterns that exercise all aspects of the model and, equivalently, of the code that is generated from the model. Many of these test patterns are identified interactively during the model development and from requirements.

Writing out a set of tests that achieve 100% MC/DC is challenging and often takes a design engineer longer than the original model took to design. Finding the right sequence of inputs to trigger a complex piece of logic may be close to impossible. At best it is an art which is learned from years of experience and is a skill not easily transferable from engineer to engineer. New tools and techniques based on formal methods or other analysis technologies are now available to automatically generate tests from models. Automatic test generation can be useful to a design group handing off a model to a software group as a software specification. The designers can use test generation technology applied to the models to create tests that they can use as acceptance criteria once the software is delivered by the software group. The software group can generate tests from a more detailed version of the model to ensure the code they develop matches the behavior of the model before they deliver the software. Both sides are better assured that the code is correct before it is integrated into a controller.

### 3.3 Taking Advantage of Style Checking and Patterns in Models

The models that are used to design and implement embedded systems undergo a series of transformations, some small, others more significant. Initially, the models may serve the system engineer or algorithm developer as an executable specification or algorithm description. Later, the models may serve as a description of how the algorithm will be partitioned – See Schlosser (2006). The evolving models now serve as the entry point for software engineering, thanks to automatic embedded code generation. As the models move closer to the implementation phases, software engineers want to annotate and transform the models, adding constraints on system behavior or describing characteristics that are required for implementation, such as fixed-point details, see Erkinnen (2004). These transformations of the model also provide an opportunity to constrain the design in a manner that makes the implementation easier to test and verify.

Large organizations that use Model-Based Design have worked together to define common model style guidelines and best practices. In addition, these industry style guidelines are often tailored by individual organizations (and sometimes specific development teams working on a specific type of application) to ensure that the design is safe, complete, unambiguous, and appropriate for embedded code generation. These modeling style guidelines are analogous to coding style guidelines.

Depending on the preferred workflow, whether a design represents a new feature or a modification to an existing feature, and what the development team prefers, there are two basic approaches: A priori and midstream/a posteriori. The constraints are applied a priori by only giving the designer a palette of authorized blocks that can be used. This might be more relevant to safety-critical systems or designs that are small modification to an existing implementation. The resulting constrained subset of designs does not need to be checked in every case, because it adheres to these restrictions by

construction. Alternatively, the constraints can be applied midstream or a posteriori by permitting the designer to use a larger palette of blocks, then checking the model to ensure that it follows a given set of rules or guidelines. Ideally, this ability to check the model is done in the model creation environment, so that the developer can quickly flag issues and edit the model in an efficient and iterative manner. By applying a set of custom rules to models, exceptions will be flagged, and the user can go immediately to the modeling environment to address the exceptions. Such an advisor tool, being closely related to an organization's model style guides, helps keep models from different authors similar in style and structure, and at the same time easily usable for production code generation.

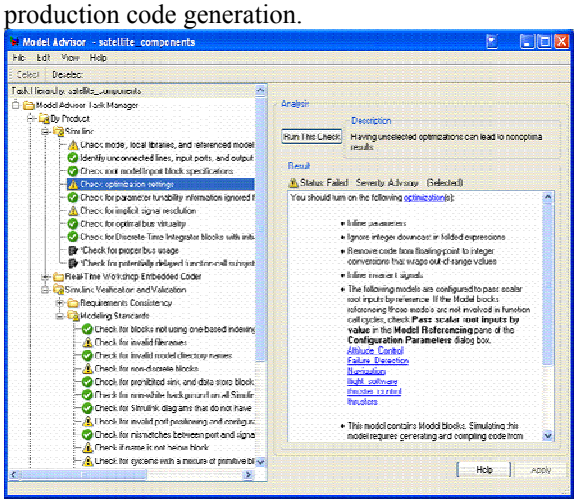


FIG. 3: Model Style Checking

### 3.4 Approaches to Code Compliance Checking Based on Models and Code

The Motor Industry Software Reliability Association (MISRA) published the "Guidelines for the use of the C language in critical systems" in Edwards (2004). MISRA-C has been adopted as a coding standard by a growing number of organizations. A significant amount of checking for MISRA-C compliance can be done by looking at what the automatic code generation tool systematically generates, rather than looking at each instance of the generated code based on the assumption that each code segment will have unforeseen variation. However, MISRA-C is still important, for at least two scenarios that might occur in Model-Based Design:

- Importing or interfacing to legacy code that violates a rule or rules.
- Making conscious or accidental changes to simulation and code generation targets and settings, resulting in violations.

A vital verification technique to enforce such compliance is to define and run industry and organization level rule checks on the model. Using this approach, the checks are performed after model creation and then prior to code generation to ensure that code passes the checks. Another option is to include a MISRA-C code checker or static analysis into the code generation and build process. For these and other reasons, it makes sense to add a code-based checker to your software development process including those using Model-Based Design.

### 3.5 Property Proving

Testing and checking of the model and code are important, particularly to ensure that the system has the desired functionality. For many embedded systems, particularly those with safety implications, it is important to go further and prove certain properties of both the model and the generated code;

#### 3.5.1 Model checking to prove functional properties

While many functional requirements can be verified in a design model using testing in simulation, some cannot. For example, there is no way to ensure the following requirement using testing alone: "Reverse thrust operation shall not engage while aircraft is in flight," even with tests that generate 100% structural coverage. Property proving on the model can verify these types of requirements. The designer uses some scripting or modeling mechanism for capturing the property to be proven. For functional requirements, the designer is essentially modeling the requirements to be proven.

Beyond requirements, property proving is valuable as a model "debug" technique that provides deeper and more extensive insight into a design than simulation alone. With property proving on the model, designers can explore edge cases, answer complex logical questions about the design and discover unexercised or "dead" logic.

#### 3.5.2 Code checking to prove non-functional properties

For algorithm developers and system engineers, model testing and structural coverage are powerful techniques and can be done early in the design process. However, there is a particular category of errors that can be difficult to detect via simulation that can cause significant problems during software development and testing: runtime errors.

These are problematic for several reasons MathWorks (2007):

- Runtime errors are latent faults that often surface under very specific combinations of data values, thus making them very costly to find by dynamic testing. Moreover, to truly find "all" runtime errors, one would have to exhaustively test for all combinations of values, which is impossible.
- Runtime errors cannot be readily associated with the code using dynamic testing. In fact, they are generally caught through their consequences on functional behaviors, including sending unexpected commands to actuators, as well as unexplained and hard-to-reproduce software failures. Moreover, once a functional failure

presents itself, a lengthy debugging is then necessary to trace the problem to its source.

- Likewise, the detection of runtime errors during tests implies exponentially increasing debugging efforts. Finding and debugging an overflow from an engine controller crash during system test is arduous, time-consuming, and indicative of the kind of effort required to solve the problem through dynamic testing.
- Runtime errors often negatively impact functional tests. If such an error is found during test, it has to be fixed and regression tests performed so as to make sure the error did not mask other problems and its solution did not create a bug elsewhere. In other words, a passed test scenario may become a failed test later on if a runtime error is detected and fixed.
- Runtime errors represent between 30–40% of errors found during maintenance, see Sullivan (1991). Static analysis is one approach with which software engineers may be familiar. Vendors of code-based checking tools acknowledge, see MathWorks (2007) that traditional static analysis promised substantial savings in terms of testing time and costs, but was unable to fully deliver on these promises for two reasons:
    - The number of "false positive" messages is generally too high—which can waste precious engineering time;
    - Since traditional static analyzers are partial in nature, code still needs to be manually inspected or tested for robustness as it includes numerous hidden "false negative".

In order to solve those issues, static code verification based on abstract interpretation techniques has been applied to detect runtime errors. These techniques can handle dynamic properties of programs by solely relying on source code (without code compilation) and can exhaustively diagnose the sections of code that may or may not lead to failures.

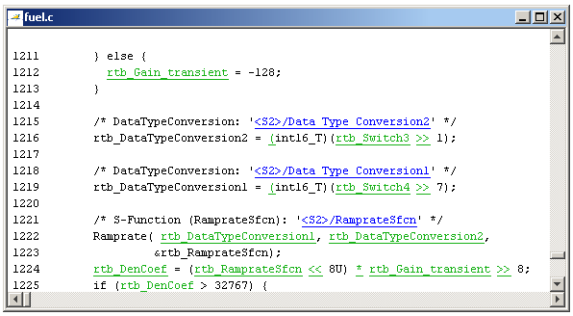


FIG. 4: Prove the absence of errors in code

That analysis is the same, regardless of whether the code was hand-written or automatically generated from a model but the integration of these techniques with modeling tools provides significant improvements to the workflow. By connecting the analyzed code and the model from which it was automatically generated, the static verification tool can present its results in both the source code and the model. Being able to navigate from the code to the model, make the change, then automatically regenerate and recheck the code, provides a powerful way to analyze, debug, and modify algorithms using both high-level and detailed perspectives. It encourages a development process in which changes are made to the model rather than directly in the code, which contributes to the reusability of the models from project to project. System and software engineers can see the results in the model or code context depending on their preferences, yet communicate with each other because the analysis results are the same. It permits each team to work in tools with which they are already familiar, which leads to more reliable results and more efficient use.

## 4. LEVELS OF ADOPTION OF MODEL-BASED DESIGN

The Japanese delegates of the MathWorks Automotive Advisory Board has suggested that there are 5 levels of adoption that organizations go through as they become more mature in their use of Model-Based Design:

| | |
|---|---|
| ***LEVEL 1*:** | Simulation in control specification development |
| ***LEVEL 2*:** | Providing Simulink/Stateflow specification to suppliers |
| ***LEVEL 3*:** | Heuristic verification and validation including Hardware in the loop simulation |
| ***LEVEL 4*:** | Code generation for target controller in production developments |
| ***LEVEL 5*:** | Systematic verification and validation including test vector generation in production developments |

It is clear that Verification and Validation techniques are significant in increasing the maturity of Model-Based Design. It is unlikely that organizations will adopt automatic code generation with all of its associated benefits without first putting in place a reasonable level of V&V and an organization will not reach the highest level without a thorough and integrated approach to V&V.

## 5. CONCLUSION

Verification and validation activities are critical to the success of any development process. Using Model-Based Design to start verifying and validating a design early and continuously throughout the design process, can lead to more successful embedded system deployments than when traditional

methods are used, which rely on verification, validation and testing at the end of the process. There are a variety of ways a development organization could apply verification and validation techniques when using Model-Based Design, but our experience with a number of process adoptions shows there are a number of clear best practices. Applying these best practices helps ensure a development process that takes full advantage of Model-Based Design to provide systematic verification and validation.

## REFERENCES

Aldrich, B. (2002) Using model coverage analysis to improve the controls development process. In AIAA Modeling and Simulation Technologies Conference and Exhibit, Montreal, Canada

CMMI (Capability Maturity Model Integrated) web page. http://www.sei.cmu.edu/cmmi , visited on 05.07.2004.

Edwards, P., S. Fisher, G. McCall, et al (2004): MISRAC: – Guidelines For The Use Of The C Lan-4Two tool vendors in this category are PolySpace Technologies and LDRA Soft-ware Technology.

Erkkinen, T. (2004): Production code generation for safety-critical systems. In 2004 SAE World Congress

Erkkinen, T. and Hote, C (2006): Automatic flight code generation with integrated static run-time error checking and code analysis. In AIAA Modeling and Simulation Technologies Conference and Exhibit, AIAA 2006-6257, Keystone, Colorado

Ghidella, J (2005): Requirements-Based Testing in Aircraft Control Design

MathWorks (2007), White Paper: “Code Verification and Run-time Error Detection Through Abstract Interpretation”

RTCA (1992) Radio Technical Commission for Aeronautics, Inc. DO 178b: Software Considerations in Airborne Systems and Equipment Certification,

Schlosser, J (2006) Architektursimulation von verteilten Steuergerätesystemen. Logos Verlag, Berlin, 2006. Technische Universität München, Fakultät für Informatik.

Sullivan, M. and Chillarege (1991): Software defects and their impact on system availability. In proc. 21th International Symposium On Fault-Tolerant Computing (FTCS-21, pp. 2–9, Montreal. IEEE Press.