AIAA Modeling and Simulation Technologies Conference and Exhibit
10-13 Aug 2009, Hyatt Regency McCormick Place, Chicago Illinois AIAA 2009-6233

Model-Based Design for DO-178B with Qualified Tools

Tom Erkkinen[1]
The MathWorks Inc., Novi, MI
Bill Potter[2]
The MathWorks Inc., Phoenix, AZ

Model-Based Design with automatic code generation is an important and established technology for developing aerospace embedded control systems. Early verification, validation, and test of models and generated code using software tools with accompanying workflows are increasingly used. In 2009, The MathWorks released tool qualification kits for verification tools based on the commercial aviation software standard DO-178B. The use of Model-Based Design for DO-178B applications using qualified verification tools is described herein.

I. Introduction

Model-Based Design allows engineers to design embedded systems and simulate them on their desktop environment for analysis and design. Model-Based Design provides a variety of code generation capabilities that teams use to generate source code for many purposes including simulation, rapid prototyping and hardware-in-the-loop testing. The use of Model-Based Design for flight code design and embedded deployment is also well established [1-4].

Flight software needs to undergo rigorous and well-documented verification activities for flight certification, such as with the commercial airborne software certification standard DO-178B [5]. With DO-178B, the tool that performs the development or verification task needs to be qualified or its output needs to be verified. The procedure for qualifying a tool according to DO-178B is based on the tool's role. If the tool is used for development activities, a rigorous qualification procedure applies; for verification tools, a substantial but less rigorous procedure is used.

This paper will show a framework for using commercial off the shelf (COTS) Model-Based Design technology to develop embedded flight software. It will present a workflow that includes textual requirements, detailed design models, automatic code generation, and a variety of automated verification steps. It will compare this to traditional development processes that use paper designs and hand coding. This paper will also examine tool qualification artifacts and benefits that can be achieved with a Model-Based Design workflow that includes qualified tools. An example implementation using commercially available Model-Based Design tools from The MathWorks [6] will guide the discussion.

II. Model-Based Design Overview

A. Modeling and Simulation

A system model includes the algorithm and environment where the algorithm executes. An example algorithm may be a control law or a signal processing filter design. For control systems, the environment includes actuators, sensors, and the plant. For signal processing systems, the environment may represent a communication transport layer with varying latencies and noise. The algorithm model will eventually be deployed as generated code on an embedded processor such as a microcontroller (MCU) or digital signal processor (DSP). The plant model can be deployed on a real-time test platform for hardware-in-the-loop (HIL) testing of the embedded flight system. It takes time and effort to develop models. However, reusing them by generating code for implementation and test is a good way to leverage a Model-Based Design investment.

[1] Manager, Embedded Code Generation and Certification
[2] Embedded Code Generation and Certification

A typical system model is developed using Simulink® block diagrams, Stateflow® state machines and truth tables, and Embedded MATLAB™ code. Embedded MATLAB is a subset of the MATLAB® language that supports both simulation and C source code generation. Initially released as a small subset in 2004, Embedded MATLAB has grown to include most of operators and functions typically used for embedded deployment and real-time simulation. Code can be generated for Embedded MATLAB files (.m files) directly from the MATLAB command line, from Embedded MATLAB blocks created in Simulink, or from Embedded MATLAB functions in Stateflow. A related feature is that Simulink diagrams can now be directly embedded inside Stateflow charts. Together, these features provide engineers with a highly flexible, multiple domain modeling environment for expressing system, software, and hardware designs.

In order for the model to be clearly understood, it needs to execute or simulate. Simulation first requires that the model compile successfully, per the model's diagnostic settings. Syntax and semantic checks are performed during the model compilation stage to check that the model is well specified and complete, for example no missing connections between blocks, giving models an immediate advantage over paper or document-based specifications. Subsequent run time analyses occur during the normal course of simulation such as array out-of-bounds and overflow checks, adding to the integrity of the *executable spec*.

The simulation results can be shown in many ways using scopes, gauges, and animation. Examples of system models with corresponding simulation results are shown in Figures 1 and 2.

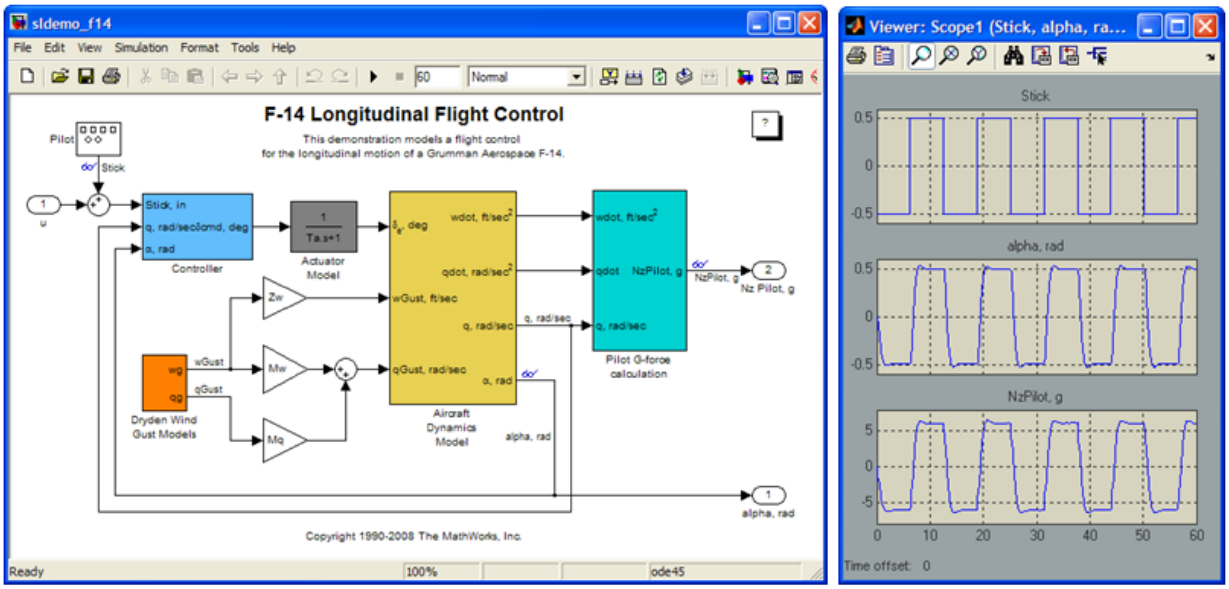

Figure 1: Aircraft Flight Control Model and Simulation Output Plot

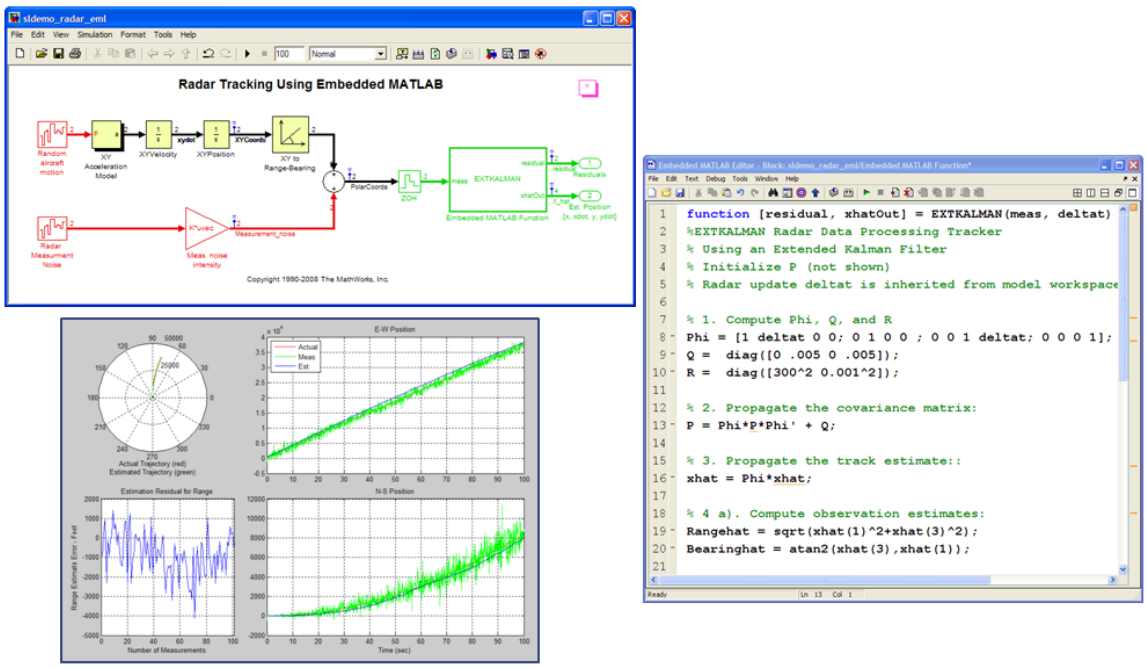

Figure 2: Radar Application System Model (top), Embedded MATLAB Function (right), and Output (bottom)

Simulation is accomplished in two ways. One is to use an in-memory representation of the model and execute the simulation in an interpretive manner, normal mode. Normal mode simulation provides users with more control of the execution environment and greater user interaction, but it can be slow for large models. The other way to simulate is to generate code from the model and execute it during simulation. This provides less user interaction but it executes faster and is known as accelerated mode, although there is some overhead for the first simulation run to generate code. A new rapid acceleration mode is also available. It leverages multiple processing cores to offer faster execution, but less user interaction, than accelerated mode.

The simulation model serves as an executable specification. The contents of the model define if it is a system model, a software model, or a verification model. The fidelity of the model dictates whether the model is a high-level or low-level specification. There are many organizational and development aspects to be considered for Model-Based Design. Some span multiple groups or different companies. For example, aircraft manufacturers need modeling tools that are system oriented and aid validation. Subsystem suppliers may need tools that are more software-oriented, aid verification, and produce software according to certification standards such as DO-178B. Successful use of Model-Based Design considers these aspects and allows for fast, easy, and seamless transitions between systems and software engineering. Best practices for Model-Based Design adoption are available [7].

B. Architecture and Design

A model contains a hierarchy of components, starting with the top level model. Common components include subsystems, subsystems in libraries, and referenced models. The components can be virtual or non-virtual. Virtual components are graphical conveniences that do not impact simulation semantics or behavior. Non-virtual components impact simulation behavior because they are treated as atomic units and thus affect sorting and execution order. A related impact also occurs during code generation where C functions can be created from atomic subsystems yet virtual subsystems always produce in-lined code.

Subsystems are part of the model they are placed in and cannot be changed independent of the parent model. From a version control standpoint, this means that a small change to a subsystem results in a new version of the entire model and its corresponding model file.

Subsystems in libraries do exist in separate library models, however, they are not fully atomic or independent of the model they are placed in. For example, a subsystem placed in a parent model will inherit attributes of the parent

including sample time, data type, and signal width. In software engineering literature, this characteristic is commonly referred to as polymorphism and offers developers a great deal of flexibility during component creation and assimilation. However, for flight software integration, it is also helpful to have components that are fully atomic and whose behavior does not change when combined with other components. This is where model reference helps.

Model reference allows parent models to reference other models, in separate model files, through use of model blocks. A model block placed in parent model references a child model in way that preserves the child model's interface. This means that one can develop, code, and verify a model once and then reuse it as a component in another model or multiple models without changing its interface or interface control description (ICD).

Model reference components can be simulated in normal or accelerated mode. Figure 3 shows an example model architected using model blocks. An architecture graph created by Simulink's model dependency viewer is also shown. Note that the example has model blocks that contain small triangles in each corner to distinguish them from other blocks or subsystems. The triangles can be black (filled in) or white (not filled in). White indicates that normal mode is used, as shown for the Algorithm model block. Black means accelerated mode is used, as is used for the other model blocks. A menu selection in the model block's parameter form makes it easy to switch simulation modes. A third simulation mode is not shown but was recently added, processor-in-the-loop (PIL) mode. PIL mode makes it easy to verify the generated code based on the model behavior and is described in the following section.

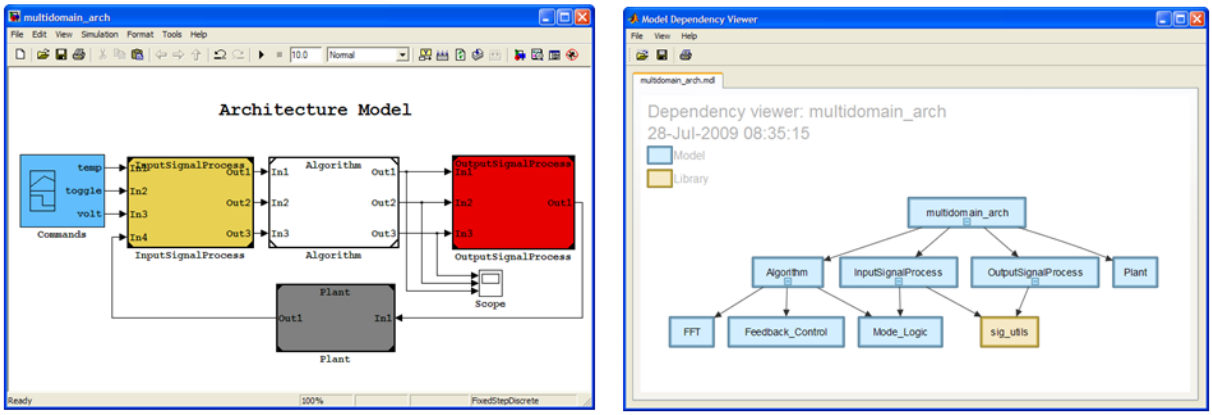

Figure 3: Example Model Architecture Model (left) and Corresponding Dependency Graph (right)

C. Embedded Code Generation and Verification

For companies generating code, it is important to understand the impact on the code from block usage, block parameters, and code generation settings. As with any language or tool, engineering teams should agree on the how to use the code generator based on its target environment, software integrity level, and other criteria. The target environment settings are crucial to establish since they are not defined by ANSI/ISO-C, for example, integer word sizes. By selecting the appropriate target word sizes, developers may realize significant code efficiency improvement and help ensure that their source code conforms to the processor architecture. Conversely a wrong choice may produce errors.

The Real-Time Workshop® Embedded Coder™ product should be used to generate the flight software because it offers the greatest control of code output options relating to items such as efficiency, traceability, and verifiability. It is also commonly used to integrate generated code with independent system simulation environments because it provides high degrees of code integration options, including encapsulated C++. A code generation advisor for Real-Time Workshop Embedded Coder, shown in Figure 4, helps developers quickly establish settings based on criteria such as efficiency, traceability, or safety-precaution.

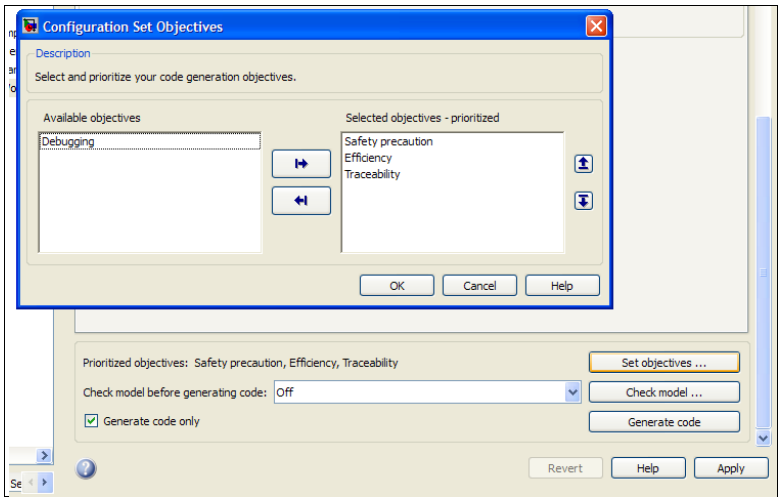

Figure 4: Code Generation Advisor for Establishing and Checking Code Objectives

C code is the traditional language used for flight code generation. But recent advances now support the generation of C++, Verilog, and VHDL. This makes it easy to target a variety of processors and hardware depending on the program's need as shown in Figure 5.

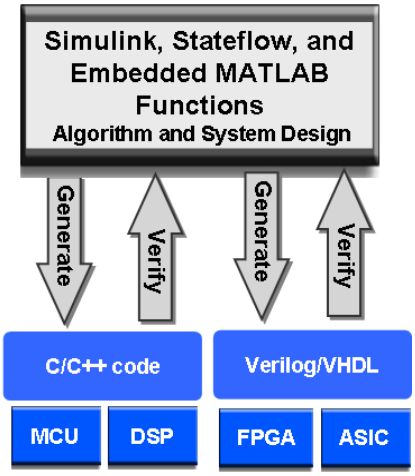

Figure 5: Generating and Verifying Code for Multiple Hardware Devices

Figure 5 also indicates the need to verify the object code executing on the target against the model. Automated code verification using software-in-the-loop (SIL) and processor-in-the-loop (PIL) testing is available using Model-Based Design with Simulink. SIL testing occurs by compiling and executing the source code on the host and comparing it to the model results. PIL works in a similar way but compiles and executes the code on the target hardware or Instruction Set Simulator. SIL is often used as a quick, incremental verification step towards successful PIL, which is the ultimate goal. Structural code coverage analysis should also be examined during the test process based on the DO-178B coverage criteria required such as Modified Condition/Decision Coverage (MC/DC).

With Model-Based Design, engineers can develop an automated SIL and PIL test environment that compares numerical results using tolerances agreed upon by the project engineering team. One option is to use the new model block PIL simulation mode, mentioned in the previous Architecture and Design section. This mode requires a communication mechanism between the host and target (such as Serial or TCP/IP) and application of the PIL API. Once established, engineers can set the model block simulation mode to PIL and quickly invoke their PIL test.

Code generation verification APIs automate SIL and PIL using scripts for batch testing with numerical differencing and plotting. SystemTest™ provides a similar capability but does so using a graphical interface and automated reporting.

D. Model Verification and Validation

Using Model-Based Design, verification and validation activities occur throughout development. A number of new technologies have been introduced that assist with early model verification such as requirements traceability, model checking, model coverage, formal methods, and test case generation.

The Model Advisor with Simulink Verification and Validation™ checks models for areas that may impede the model's use in flight software environments. Some checks focus on simulation aspects, others on code efficiency, and a new series of checks address certification standards including DO-178B. For example, one of the checks examines model input and output ports to see if they are well defined and do not inherit characteristics such as data type and sample rate. Another check examines that user has established their target environment settings. There is also an API and graphical editor for adding custom, project-specific, checks.

Another important verification step is to develop and execute model tests. For safety-related systems, the tests should be based on the requirements, usually in textual form. Bi-directional links between the model and the high level requirements in documents, data bases, or requirements management tools is supported with the requirement management interface in Simulink Verification and Validation. Requirements can also appear in the generated code as comments with automated linking to and from the model.

With Simulink Design Verifier™ product, test cases can be automatically generated from the model based on desired model coverage criteria (e.g, MC/DC). Figure 6 shows the tests generated from a model. Note that the test shows an incomplete coverage for a particular path. It was not exercised due to poor design logic which needs to be resolved before generating code. Finally, Simulink Design Verifier provides special blocks that help engineers perform formal proofs to assess the design logic and enhance its robustness.

Status

Table 2.1. Objectives Proven Unsatisfiable

#:	Type	Model Item	Description
118	Decision	[in(MultiFail)]	Transition " [in(MultiFail)]": Transition trigger expression F

Table 2.2. Objectives Satisfied

#:	Type	Model Item	Description
2	Decision	Speed_Sensor_Mode	State "Speed_Sensor_Mode": Substate executed State "speed_norm"
4	Decision	Throttle_Sensor_Mode	State "Throttle_Sensor_Mode": Substate executed State "throt_norm"
6	Decision	Pressure_Sensor_Mode	State "Pressure_Sensor_Mode": Substate executed State "press_norm"
9	Condition	[speed==0 & press < zero_thresh]/ Sens_Failure_Counter.INC	Transition "[speed==0 & press...": Condition 1 T
10	Condition	[speed==0 & press < zero_thresh]/ Sens_Failure_Counter.INC	Transition "[speed==0 & press...": Condition 1 F
11	Condition	[speed==0 & press < zero_thresh]/ Sens_Failure_Counter.INC	Transition "[speed==0 & press...": Condition 2 T
13	Decision	[speed==0 & press < zero_thresh]/ Sens_Failure_Counter.INC	Transition "[speed==0 & press...": Transition trigger expression F
14	Decision	[speed==0 & press < zero_thresh]/ Sens_Failure_Counter.INC	Transition "[speed==0 & press...": Transition trigger expression T
15	Mcdc	[speed==0 & press < zero_thresh]/ Sens_Failure_Counter.INC	Transition "[speed==0 & press...", MCDC Transition trigger expression with Condition 1 T
16	Mcdc	[speed==0 & press < zero_thresh]/ Sens_Failure_Counter.INC	Transition "[speed==0 & press...", MCDC Transition trigger expression with Condition 1 F
17	Mcdc	[speed==0 & press < zero_thresh]/ Sens_Failure_Counter.INC	Transition "[speed==0 & press...", MCDC Transition trigger expression with Condition 2 T

Figure 6: Model Test Objectives Report using Design Verification Tools

Once the model has satisfied its requirement-based test and model coverage requirements, flight code can be automatically generated, as described earlier. Verification of the code with the model is performed using software-in-the-loop (SIL) and processor-in-the-loop (PIL) as previously described. Hardware-in-the-loop (HIL) testing then occurs after software/hardware integration as a final system validation effort in the lab. For hardware-in-the-loop testing, code is generated for the plant model. It runs on a highly deterministic, real-time computer. Sophisticated signal conditioning and power electronics are needed to properly stimulate the hardware inputs (sensors) and receive the outputs (actuator commands). Fault injection hardware may also be included.

Finally, in addition to these simulation based test approaches, it is important to analyze and verify your software using formal analysis, for example to show absence of certain run-time errors or to perform MISRA-C® and JSF++ code checking. PolySpace® code verification products enable this and support C, C++, and Ada source code.

III. DO-178B and Related Standards Overview

A. DO-178

DO-178 is a commercial aviation standard used to certify software in airborne systems. It is also heavily used in military, space and other applications, due in part, to the lack of another broadly accepted aerospace standard. DO-178A was released in March 1985, DO-178B was released in December 1992, and DO-178C is under development. DO-178B has clearly defined objectives for key software lifecycle process activities including software requirements, software design, coding, integration, verification, and configuration management. For each objective, outputs are specified that need to be created, verified, and ultimately, used for certification. The objectives and outputs are summarized in Tables A-1 to A-10 and are based on the software integrity level. Level A is the highest integrity and is used when failure would cause or contribute to a catastrophic failure.

Model-Based Design was in its infancy when DO-178B was introduced. A quick review of the Tables and examination of the document shows little notion of the use of models. This is one reason why development of version C is underway. The 11th Joint meeting for DO-178C occurred in June 2009 and was attended by more than 100 participants representing the major aircraft manufacturers and suppliers [8]. Aviation administration officials also participated including members from FAA, EASA, and even China's CAA. When released, DO-178C will supplant DO-178B as the world's primary aerospace software development standard.

MathWorks participates in DO-178C, and in particular, the Model-Based Design subcommittee [9]. A key deliverable in support of this effort is a proposed update of objective and output Tables based on Model-Based Design activities previously described. The workflow described in Section IV leverages this information to compare traditional approaches to DO-178 with those using Model-Based Design.

B. Related Standards

Simulink is unique in its ability to support multiple modeling and deployment domains. As such, there are many aerospace safety standards used with Simulink but not covered by DO-178B including systems engineering, code deployed in ground-based DSPs, or HDL-based hardware. Engineering teams that base their DO-178B programs on Simulink are able to leverage that investment for reuse in other standards and applications as summarized below.

- ARP-4754 "Certification Considerations for Highly Integrated or Complex Aircraft Systems," addresses Systems Engineering including:
  - Systems Requirements
  - Requirements Validation
  - Systems Design
  - System Verification
- ARP-4761 "Guidelines for Conducting Safety Assessment Process on Civil Airborne Systems and Equipment," addresses Systems Safety Assessment including:
  - Fault Tree Diagrams
  - Common Cause Analysis
  - Zonal Safety Analysis
  - Functional Hazard Assessment
- DO-278 "Guidelines for Communication, Navigation, Surveillance, and Air Traffic Management (CNS/ATN) Systems Software Integrity Assurance," addresses Software Engineering for ground and space based systems including:
  - Software lifecycle
  - Objectives to be satisfied
  - Evidence required
  - Covers software used on microprocessors and digital signal processors
- DO-254 "Design Assurance for Airborne Electronic Hardware," addresses Hardware Engineering certification including:
  - Hardware lifecycle
  - Objectives to be satisfied
  - Evidence required
  - Covers FPGAs and ASICs

C. DO-178B Tool Qualification

Section 12.2 of DO-178B describes Tool Qualification. As stated, *Qualification of a tool is needed when processes of this document are eliminated, reduced, or automated by use of a software tool without its output being verified as specified in Section 6.* DO-178B provides two types of qualification criteria, one for software development tools such as code generators and compilers that can introduce errors, and another for software verification tools such as code checkers and static analyzers that cannot introduce errors but may fail to detect them. A tool cannot be qualified on its own by the tool vendor. It can only be qualified in the context of the project or flight application being certified.

The criteria for qualifying development tools is extremely rigorous, whereas the criteria for verification tools is less, but still significant. As a result, there are many verification tools but few development tools with commercially available qualification kits or support. In fact, there are no DO-178B commercially qualified C compilers as one aircraft manufacturer noted in a presentation to the FAA Tools Forum, along with some rationale [10].

The top reasons why commercial compilers are not FAA-qualified:

- *Complexity*
- *Unclear benefits*
- *Market forces*
- *Language trends*

By unclear benefits, it is important to consider that there are many tools in the process of transforming an algorithm into flight code including the code generator, cross compiler, assembler, linker, and flasher. If a linker is qualified, it is important to understand what, if any, benefit it has on the requirements-based object code verification process. Similarly, if a coder is qualified but the compiler and linker are not, one should verify that the object code does not contain errors inserted by the compiler or linker.

An aircraft supplier also presented to the FAA Tools Forum [11] and noted that *Qualified development tools are typically either very simple tools or end up way behind the technology curve.* This simplicity is due to a practical necessity of tool qualification, which is to reduce the application space and thus reduce the qualification effort. The supplier stated that their internal studies have shown that qualifying a development tool was 20 times more expensive than qualifying a verification tool. Further, they reported that development tool qualifications do not offer a return on investment until after several programs have used the tool, whereas *Use of qualified verification tools results in savings on first program where it is introduced*.

Thus, a common approach by DO-178B practitioners is to use unqualified but state-of-the-art development tools with automated and qualified verification tools. This lets developers express their design in the most natural and appropriate way, affords a high degree of verification rigor, and offers an investment return even on the first project.

Verification of object code to the model is vital for high integrity systems, even if a fully qualified toolchain of coder, compiler, and etcetera existed. One reason is simply the added confidence; some refer to this as a belt-and-suspenders approach to safety. Another is based on software numerical issues resulting from floating point approximations on different platforms. See the classic paper of *What Every Computer Scientist Should Know About Floating Point Arithmetic* [12]. In short, floating point is an approximation of real numbers. For example, squaring 0.1 in single precision floating point does not yield 0.01; there are extra digits to far left of the decimal point. These approximations manifest as subtle but critical differences in execution results of math-intensive algorithms.

This may result in an algorithm yielding different results when it executes as code compiled on the host versus when it executes on target. Or in another example, the same code on the same processor with the same compiler but using updated math libraries may yield different answers. Today's algorithms involve coordinate transformations, Fast Fourier Transforms, and trigonometric functions for dynamic inversion and modern flight control laws. Processing resources usually demand use of single precision data types. So engineers should expect differences, but they should test for them using established tolerances, and strive to develop algorithms that are robust to these affects. One should be wary of approaches that advocate eliminating testing. Do not eliminate testing; automate it using SIL, PIL, and other techniques.

IV. DO-178B Development Approaches

A. Traditional Hand Development Approach

Traditional approaches used to develop and verify software for DO-178B in the 1990s were mostly designing and coding by hand but with some automated tool support for compiling and verifying the software. The main development process artifacts described in DO-178B include High Level Requirements, Low Level Requirements, Source Code, and Object Code. Tools such as text editors and IDEs or code compilers were used for development. Verification activities include reviews, conformance to standards, testing, and coverage analysis, with some tool automation for code based activities. As stated in the previous section, qualified code verification tools were often used but qualified code compilers were not. Figure 7 illustrates the development and verification activities traditionally done and indicates the tools used with their qualification status.

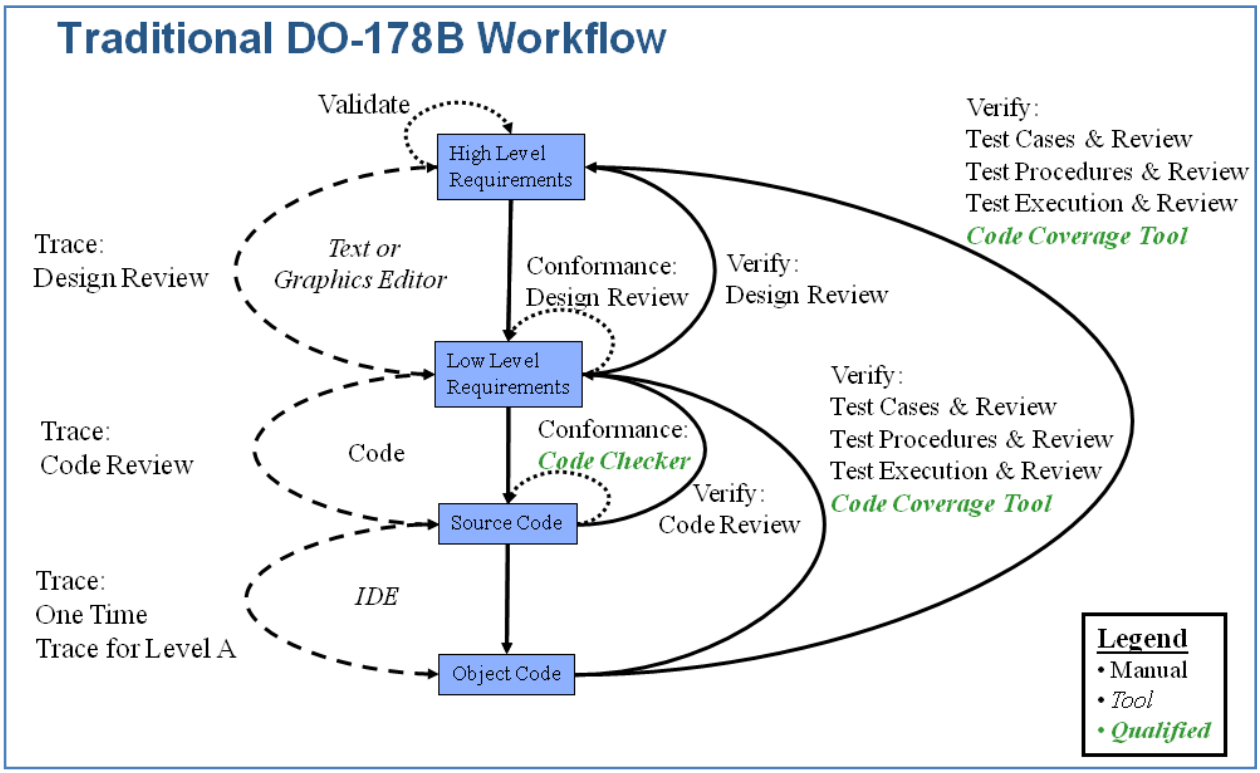

Figure 7: Traditional Hand Development Approach to DO-178B Applications

B. Traditional Model-Based Design Approach

By the millennium, Model-Based Design and code generation were starting to be used more heavily for DO-178B applications. In 2004, one company reported that in just the past year they had generated and certified over a million lines of code to DO-178B, including software certified to Level A. Further they found that automatic code generation resulted in six-sigma software quality, vastly exceeding hand code quality [13]. As is shown in Figure 8, modeling allowed for earlier and more rigorous verification of the previously paper-based design. However, the benefits of these verification tools could not be fully realized because they were not qualified. This changed in March 2009, when MathWorks released DO Qualification Kit (for DO-178B).

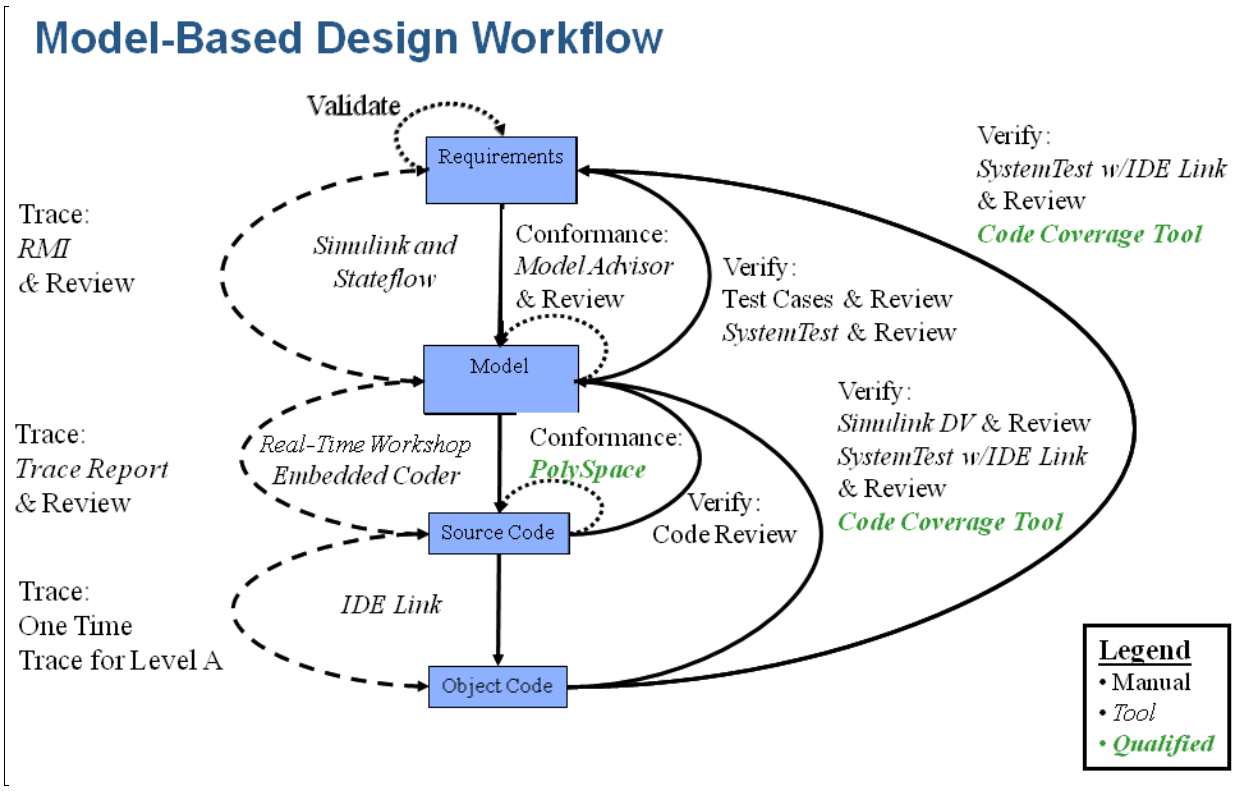

Figure 8: Traditional Model-Based Design Approach to DO-178B Applications

C. **Today's Model-Based Design Approach**

The MathWorks recently released the DO Qualification Kit (for DO-178B), which allows selected verification tools to be qualified for a DO-178B project. The version 1.0 of this product includes tool qualification data for the following tools and can be currently be used with releases R2008b or R2009a.

- Simulink Verification and Validation
  - DO-178B Model Checks
- SystemTest
  - Limit Test Element
- PolySpace Code Verifiers
  - PolySpace code verification products for C/C++

As shown in Figure 9, a highly automated verification workflow with qualified tools is now available. The modeling tools provide engineers with high degrees of flexibility for expressing designs. The code generation tools produce efficient results and provide many code optimization options. In one example, a company developed an algorithm for multi core processing using Simulink, and later, performed a benchmark and found that the generated code was 30% faster than the hand code [14].

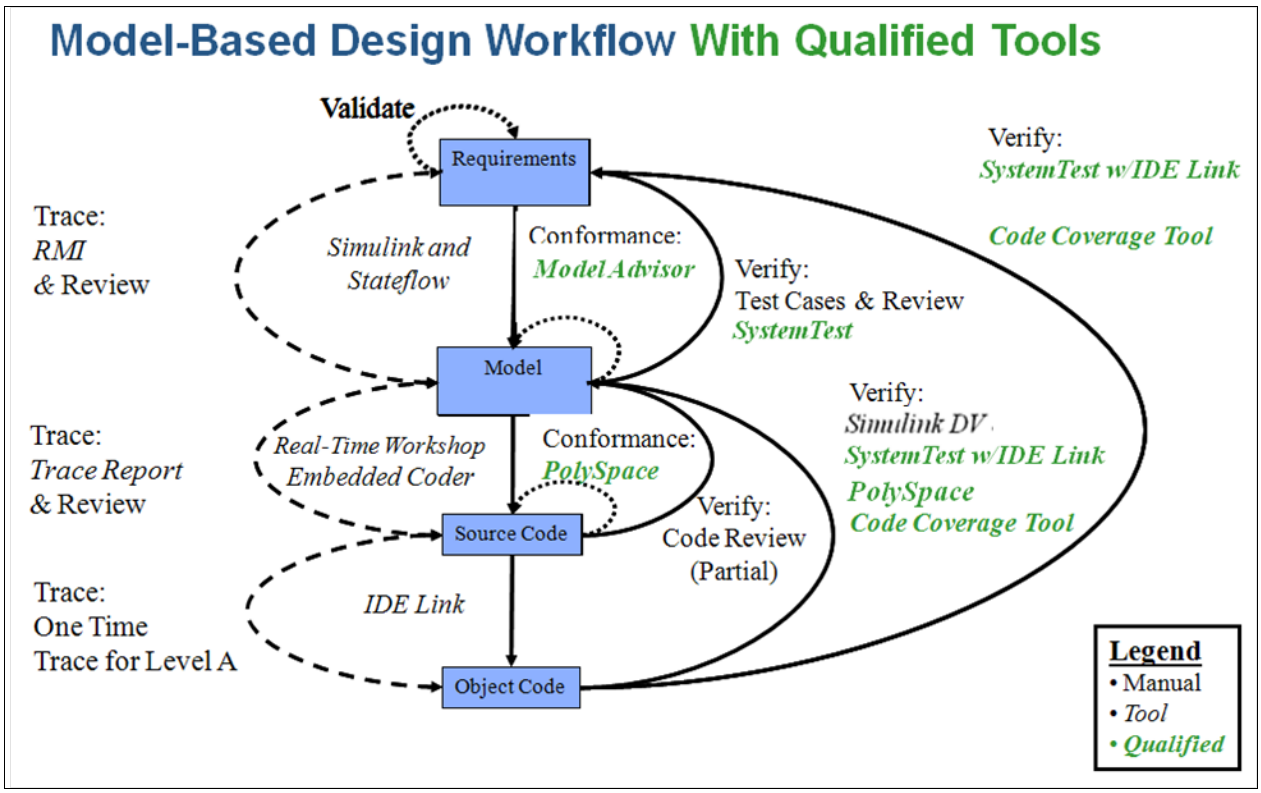

Figure 9: Today's Model-Based Design Approach to DO-178B Applications using Qualified Tools

Recommendations for DO-178B Projects

The following outline describes a recommended workflow for using The MathWorks products in DO-178B with qualified verification tools.

1. Requirements Process

For requirements validation, use traditional peer reviews of the requirements. No changes are proposed for this process. For requirements linking, use the Requirements Management Interface to establish links between the model and high level requirements in textual or third-party tool form.

2. Modeling Process

For the low level requirements, use Simulink and Stateflow for modeling. Use one model since it provides a single "truth" in the design. Note that not every domain is appropriate for the use of Simulink or Stateflow, and this must be considered for each project. It is also understood that for any given project, some of the software will need to be developed in traditional ways. Avoid conversion from Simulink to third party design tools as this introduces a separate step, and thus a new error introduction point, and adds no real value to the process. This also results in an additional artifact that must be reviewed.

To achieve the full benefits of Model-Based Design, early verification of the design should occur at the model level rather than being a downstream activity. Perform simulation tests based on higher level requirements. There are some suggested improvements in the model verification process that can achieve savings:

- Use Model Advisor and qualify it for DO-178B projects
- Use the Requirements Management Interface for requirements linking and reporting
- Use SystemTest to automate simulations and reporting, and qualify it for DO-178B projects
- Use model coverage during simulations to assess the effectiveness of the high level test cases

One of the typical activities in a DO-178B process is a design review. The purpose of this review is to show that the low level requirements comply with the high level requirements and conform to standards. The methods used to

achieve these activities are reviews and analyses. The Model Advisor performs an automated review of the model that covers most of the conformance to standards assessment, but manual inspection of items not easily automated should also be done. Model compliance to high level requirements can be assessed using simulation, which is a form of analysis, rather then performing traditional design reviews. Simulations are much more effective than design reviews in finding real issues in the design.

Simulation results should be documented, assessed and archived for a project. SystemTest provides a mechanism for running the simulations, automatically generating test reports and assessing the results with a qualified pass/fail checker. It automates simulation and reduces the simulation results review activity to a review of the pass/fail results reported by the tool. In the case where failures are detected, it also provides the ability for engineers to investigate the failures using data that is automatically stored during test execution. It is also possible to use this tool for regression testing when changes are made to the models.

Up to this point the Model Coverage tool has not been used during simulations. The use of this tool during simulation provides an assessment of how effective the high level requirements based tests cover the functionality in the model. Note that there may be derived requirements in the model, and thus, the Model Coverage tool may not indicate full coverage during those simulations. Data from the Model Coverage Report can be used later in the process by using this data as input to the Simulink Design Verifier to assist in completing test coverage.

3. Coding Process

Use code generation from Real-Time Workshop Embedded Coder because it is the most efficient approach to obtaining code from a development standpoint. The verification of the source code, whether it is automatically generated or not, requires a code review as part of the DO-178B process. PolySpace products can be used to automate portions of the code review and credit can be taken for up to three of the code review objectives by using the DO Qualification Kit for PolySpace. Use of PolySpace and the DO Qualification Kit significantly reduce the cost of code reviews.

4. Object Code Testing Process

Testing of the object code and achieving structural coverage during those tests can be one of the more time consuming and expensive processes on a DO-178B project. Establish a goal to reuse the high level testing done at the model level to also test the code. This is a good plan to have, but it is likely that some low level testing will also be needed in order to achieve full structural coverage of the code.

One of the key tools to be used in this process is the Embedded IDE Link™ product. This tool allows Simulink to perform PIL testing for code that is compiled using popular IDEs and compilers. The use of the IDE link in conjunction with a CPU development board would allow test cases to be reused to verify the flight code in an efficient manner.

As described previously, the Model Coverage Tool can be used to determine the level of coverage achieved from the test cases used during simulation. It is likely that full coverage will not be achieved from the high level test cases. In order to achieve full coverage, the Simulink Design Verifier may be used to supplement the high level test cases by automatically generating test cases. An efficient method of doing this is to input the model coverage data from the high level testing into Simulink Design Verifier, and it will then produce the test cases necessary to make up the uncovered portions of the model. The engineer can then run the generated tests and confirm the model was covered using the Model Coverage Tool. The combination of these sets of test cases can then be executed on the code. A third-party code coverage tool can then assess the actual code coverage from these test cases. There may be some additional coverage holes found during the code testing that may need further analysis, but these should be a small percentage of the overall coverage and afford a significant net reduction in overall testing effort.

DO-178B also requires that equivalence class tests be performed on input data ranges. The model coverage tool also helps in this area because it can record signal ranges during testing and show the minimum and maximum values achieved. Additionally, the Simulink Design Verifier can be set up to automatically generate equivalence class test cases using the Test Objective Block on input signals. In this case the user must specify the test values based on the data range and desired equivalence classes. Use of Simulink Design Verifier to generate equivalence class test cases offers further reductions to the testing effort.

The optimum way to set up the system for use of Simulink Design Verifier is to use reference models in the design and to generate test cases for each reference model, individually. Reference Models are already used at many companies developing large scale modeling due to simulation performance improvements.

In lieu of some types of robustness testing, credit can be taken for PolySpace formal analysis. PolySpace has the ability to detect uninitialized variables, numeric overflows, infinite loops, divide by zeros, etc. Thus by using this qualifiable tool, test cases for detecting these types of errors are no longer needed. This further reduces the overall object code testing effort by eliminating the need for these robustness test cases.

IV. Conclusions

This paper presented an overview of Model-Based Design and noted important aspects for using models to develop and generated embedded flight software. Discussion focused on the verification of models and code using new technology from MathWorks. New verification tool qualification kits for DO-178B aid this process.

References

1. "Design Times at Honeywell Cut by 60 Percent," W. King, Honeywell Commercial Aviation Systems, Nov. 1999, http://www.mathworks.com/company/user_stories/
2. "Flight Control Law Development for F-35 Joint Strike Fighter," D. Nixon, Lockheed-Martin Aeronautics, Oct. 2004, http://www.mathworks.com/programs/techkits/pcg_tech_kits.html
3. "ESA's First-Ever Lunar Mission Satellite Orbits Moon with Automatic Code," P. Bodin, Swedish Space, Oct. 2005, http://www.mathworks.com/programs/techkits/pcg_tech_kits.html
4. "Automatic Code Generation at Northrop Grumman," R. Miller, Northrop Grumman Corporation, June 2007, http://www.mathworks.com/programs/techkits/pcg_tech_kits.html
5. "Software considerations in airborne systems and equipment certification," RTCA/DO-178B, RTCA Inc. Dec. 1992
6. The MathWorks, Inc., www.mathworks.com
7. "Best Practices for Establishing a Model-Based Design Culture," SAE Paper 2007-01-0777, P. Smith, etc., March 2007, www.mathworks.com/products/simulink/technicalliterature.html
8. "Eleventh Joint Meeting of Joint EUROCAE Working Group 71 and RTCA Special Committee 205 Software considerations in airborne systems and equipment certification," RTCA No. 161-09/SC205-025, June 2009, http://www.rtca.org/CMS_DOC/205sum11ja-0906.pdf
9. SC-205/WG-71 Plenary Website, SG4 Model-Based Development Group, ultra.pr.erau.edu/SCAS
10. "Crafting an FAA-Qualifiable Compiler," V. Santhanam, Boeing, ERAU/FAA Software Tool Forum, May 2004, http://www.erau.edu/db/campus/softwaretools_presentations.html
11. "Use of MathWorks Tool Suite to Develop DO-178B Certified Code," B. Potter, Honeywell, ERAU/FAA Software Tool Forum, May 2004, http://www.erau.edu/db/campus/softwaretools_presentations.html
12. "What Every Computer Scientist Should Know About Floating-Point Arithmetic," D. Goldberg, Computing Surveys, 1991, http://docs.sun.com/source/806-3568/ncg_goldberg.html
13. "Achieving Six Sigma Software Quality Through Automatic Code Generation," B. Potter, Honeywell, June 2005, http://www.mathworks.com/industries/aerospace/miadc05/presentations/potter.pdf
14. "Software Development with Real-Time Workshop Embedded Coder," N. Holliday, Thales Missile Electronics, April 2008, http://www.mathworks.com/programs/techkits/pcg_tech_kits.html