High-Integrity Production Code Generation

Originally Published at 2003 AIAA GN&C Conference
As Paper 2003-5488, 2003

Abstract

Automatic code generation is an automated process that translates design diagrams into source code. This code is often used for simulation, rapid prototyping, hardware-in-the-loop testing and many other purposes. More recently, the code is being deployed on embedded systems and is sometimes termed “production code.”

Production code generation helps software and electronics engineers by providing a common framework for adding software details to the original control algorithms model. Details such as data typing, function partitioning, and external code integration can be provided at such a low-level that the detailed model in many ways serves as the “source code.”

Object Oriented Programming (OOP) has been proclaimed as the fourth generation of software evolution, and production code generation has recently emerged as a strong candidate to become the fifth. As with previous software evolutions, a number of best practices need to migrate up to the next level. These practices include requirements tracing, configuration management, and verification and validation. This paper presents a complete software framework focused on automating the production of high-integrity embedded software for flight systems.

Discussion

Production code generation is part of a model-based design approach, which uses control algorithm models for functional specification and real-time plant models for embedded controller validation. It is used between the specification and validation activities and is focused on the task of building the actual control system that will be manufactured for the flight vehicle.

This type of flight code is certified as part of airborne systems by government agencies, including the FAA, JAA, DOD, MOD and others. These systems undergo a safety assessment to identify the various failure conditions and evaluate how the system responds to those failures. The response might be catastrophic, have no effect, or fall somewhere in between.

The software integrity level is then derived based on its contribution to the failure condition. If the system response to a software-induced failure is catastrophic, or severely impedes safe flight, the software must be developed as high-integrity software. In recent years, high-integrity software has become more pervasive and increasingly complex as new aircraft systems increase their reliance on autonomous embedded control strategies, such as fly-by-wire and integrated avionics.

Many standards exist throughout the world to assist developers in creating this type of software. Each standard has a governing body, which certifies that the system was developed accurately based on the evidence or artifacts that the engineers were required to produce. These standards and processes are followed regardless of whether or not the flight code was generated automatically or by hand.

The software engineering process can be separated into the following categories:

1. Development and Integration
2. Verification and Validation
3. Software Engineering /Framework Activities

Development and Integration is often where the most time is spent for software that is not developed for high-integrity systems. It starts with creating system requirements and ends with system integration and acceptance testing. Code generation is one piece of this effort and it spans the gap from detailed software design to software module integration.

Verification and Validation (V&V) is a main focus for high-integrity systems and encompasses the systematic application of reviews, analysis, and testing during each development and integration phase. The basic philosophy is to ensure that the right product is being built, and that it is being built in the right way. For high-integrity systems, module testing and code coverage often involves a major effort. Of great importance and equal in terms of effort, is the need to ensure that the requirements are valid and that the design satisfies the requirements.

Software Framework Activities include the basic information management tasks. Topics under this umbrella include configuration management and

Copyright © 2003 by the American Institute of Aeronautics and Astronautics, Inc. Reprinted with permission

version control, documentation, and requirements traceability. These tasks are critical for high-integrity software development. They provide certification agencies with the artifacts needed to earn their approval and trust for each development step.

Production code generation technology is of little use for high-integrity systems if it cannot fit within the various steps and activities just described.

This paper walks through each step and describes how model-based design and production code generation technologies fit within and enhance this type of software engineering environment. To illustrate each point, and to demonstrate that the capability is viable and available, example products supporting model-based design and code generation are shown [1].

Development

Model-based design places great emphasis on process iterations, early testing, and reuse throughout the development process. It is also practical in that it does not just create nice pictures, but also produces embedded code and automates target integration.

In model-based design the model consists of block diagrams and state machines, each with specific timing and real-time execution characteristics. For complete system models, the components include sensors, actuators, a plant, and of course, the control system.

In industry applications, these models and their simulation results have served at different levels of abstraction as the formal:

- System requirements;
- System architecture;
- Software requirements;
- Software architecture;
- Software detailed design; and,
- Source code.

It might surprise some readers to learn that the model may serve as the source code, but this makes sense when source code is regarded as simply the lowest level at which the developer manually describes the software. In previous times, this would have been at the machine level. Today, C, C++, or Ada code is often treated as the source, but within this context, the model is clearly the source.

Requirements Specification

The use of executable specifications and automated model checkers also allow for a much more effective V&V effort prior to a final build. The benefit of early V&V is clear. The concept is taught in introductory programming classes: catch as many bugs as you can soon as you can. This saves costly and lengthy iterations to fix them.

Figures 1&2 show a variety of models using control system block diagrams for feedback control, state machines for discrete events and conditional logic, and additional blocks for digital signal processing (DSP) filters. All of these models can be simulated so that the intended system behavior can be fully understood. Additional discussion on these topics will not be provided here since aerospace GN&C engineers who read this paper are likely to be very familiar with these concepts and diagrams.

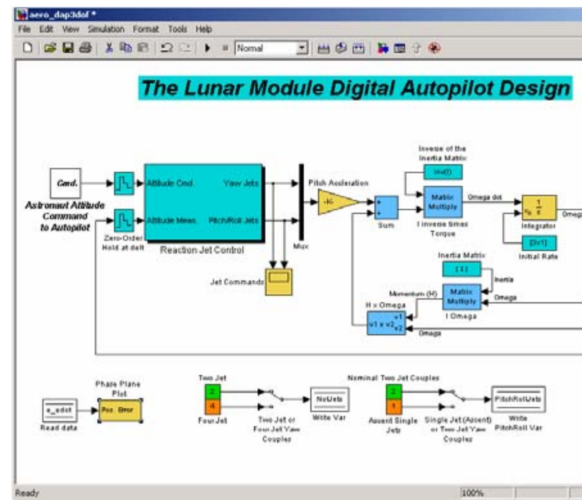

Figure 1: Lunar Module Autopilot Model

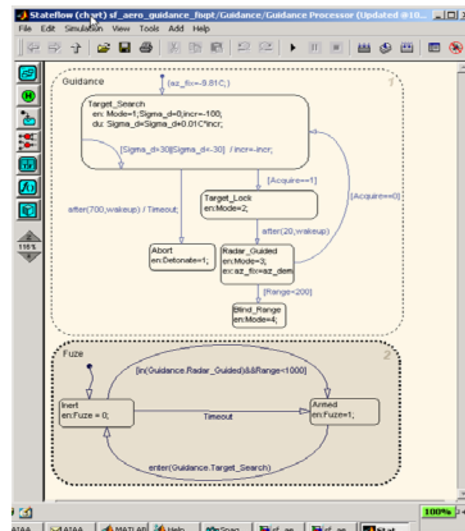

Figure 2: Fixed-Point Guidance Algorithm

Detailed Software Design

Detailed software design begins once the software requirements have been specified using the models, as shown previously. In many circumstances, this is where the systems and controls groups end their work and the embedded software and integration teams begin. Unfortunately, this is the also the point where communication often weakens and lengthy manual conversion processes begin.

Production code generation helps bridge this gap by providing mechanisms for performing detailed software design directly on the same model. Once the software has been fully designed, automatic code generation produces the final code.

Examples of a few of the detailed software design activities include:

- Data type definition;
- Storage class specification;
- Fixed point scaling;
- Real-time tasking;
- Built-in-test and diagnostics;
- Initialization and shutdown routines;
- Function and file partitioning;
- Legacy code inclusion; and
- Device driver integration.

With model-based design, the algorithm model is in effect constrained by the software design. For example, the algorithm may be developed in double precision for the ideal behavior and then fixed-point constraints are added to get the real behavior on the microcontroller. Figure 3 shows a model and parameter form for the Sum block. A checkbox “show additional parameters” is where design information or constraints are entered.

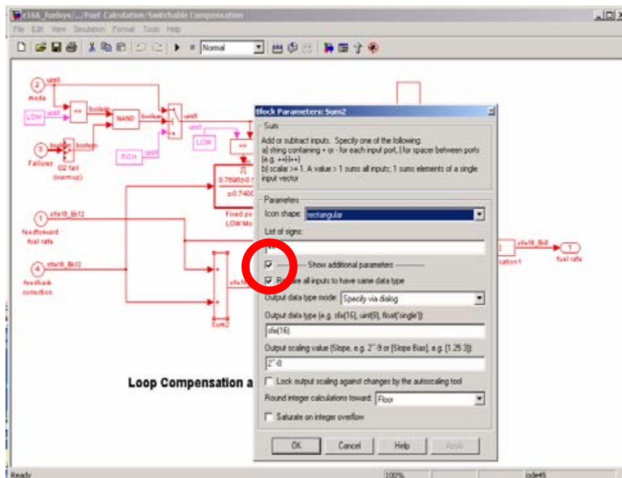

Figure 3: Detailed design for fault tolerant fuel system controller

The floating-point version of the Sum Block used during algorithm specification is converted into a detailed software design simply by changing fields in the block’s parameter form. Note that it is also possible to use a data dictionary or workspace to set the various signal and parameter attributes.

The fixed-point information can also be ignored, allowing for simulation and code generation in floating-point. This makes it easy to go back and forth between requirements and design, which is a problem with some development methods, especially during maintenance.

High-integrity software needs clearly defined data types and data qualifiers. Figure 4 shows how storage classes can be defined and illustrates a few of the detailed data and data type modifiers available. User-specified data types and attributes could also be added using the advanced custom storage classes feature.

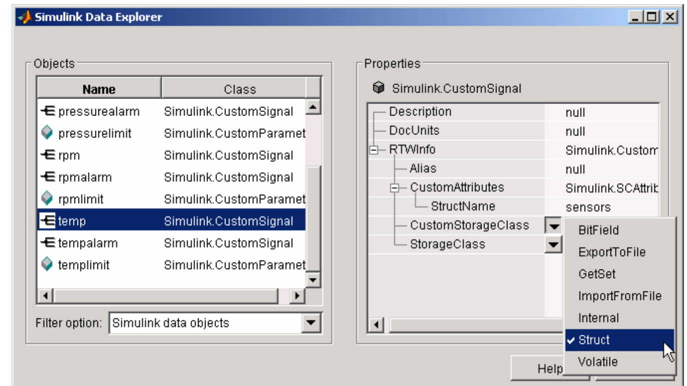

Figure 4: Custom Storage Classes

High-integrity software also requires explicit use of data and restricts implicit promotion and conversion of data types during math or other operations. For simulation and code generation, it is possible to set warning and errors flags that detect a variety of mistyped operations. This ensures that proper typing is used when the model is the source. This is a great benefit over using the C language as the source, since C is inherently a weakly-typed language.

Production Code Generation

With the detailed model available, simulation on the host or code execution on the target lets developers verify that the implementation satisfies (or closely approximates) the requirements model. It is then time to generate the actual production embedded code.

As with a C or Ada compiler, this process is straightforward. Various optimization settings and user

configuration options exist. The key is to make sure that the code is efficient, accurate, deterministic, and integrates well with other code or tools. It is also important for the code to be traceable to the diagram, so that it may be reviewed and verified.

Here are the source code requirements as stated in the FAA Guidelines document RTCA/DO-178B [2]:

"The objective of the software coding process is:

a. Source code is developed that is traceable, verifiable, consistent, and correctly implements low-level requirements."

Tracing and Reviews

Being able to trace from high-level requirements through final implementation is a key part of high-integrity software development. Figure 3 previously demonstrated how to trace from a requirements model to a detailed design model, and back again. The following diagram now demonstrates how to trace the automatically generated C code back to the model.

The example shown in Figure 5 uses an HTML report generation capability with hyperlinks back to the model. As shown, when the developer selects the Sum descriptor in the code, the Sum Block in the Rate Monotonic model is highlighted.

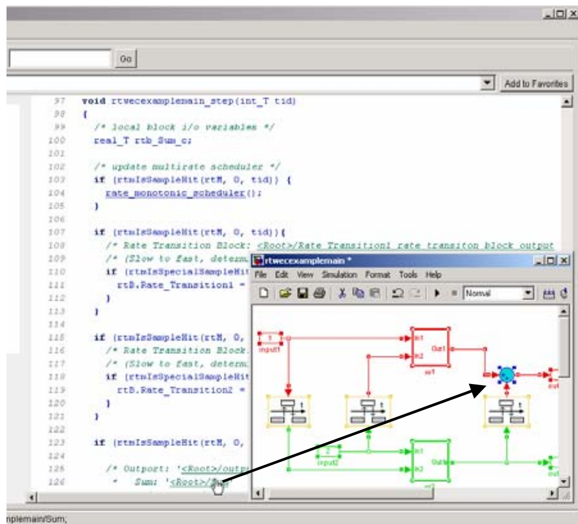

Figure 5: Code review for Rate Monotonic Model

Embedded Target Integration

The figure above uses Rate Transition Blocks, but direct links also exist to commercial RTOSs, including Wind River products and OSEK variants. Additionally, integration with device drivers and legacy code can be accomplished in many ways depending on the exact needs of the development environment.

Some development organizations like to put drivers in the model so that the model can fully specify the entire software system. Other groups prefer to keep the application software (AS) separate from the operating system software (OS) to improve modularity and portability.

Verification and Validation

DO-178B provides detailed guidelines on an acceptable verification strategy for high-integrity software. The focus is on requirements-based testing and code coverage. The following diagram shown in Figure 6 was recreated from an equivalent diagram in DO-178B.

The key point to note here is that the testing is fully based on the requirements. After running the requirements-based tests, the structural coverage is derived, and additional test cases are added as needed to achieve the coverage requirements. For the highest level of integrity (what DO-178B refers to as "Level A"), a coverage known as "Modified Condition/Decision Coverage" is required. Obtaining 100% MC/DC coverage is often one of the biggest challenges for developing high-integrity software.

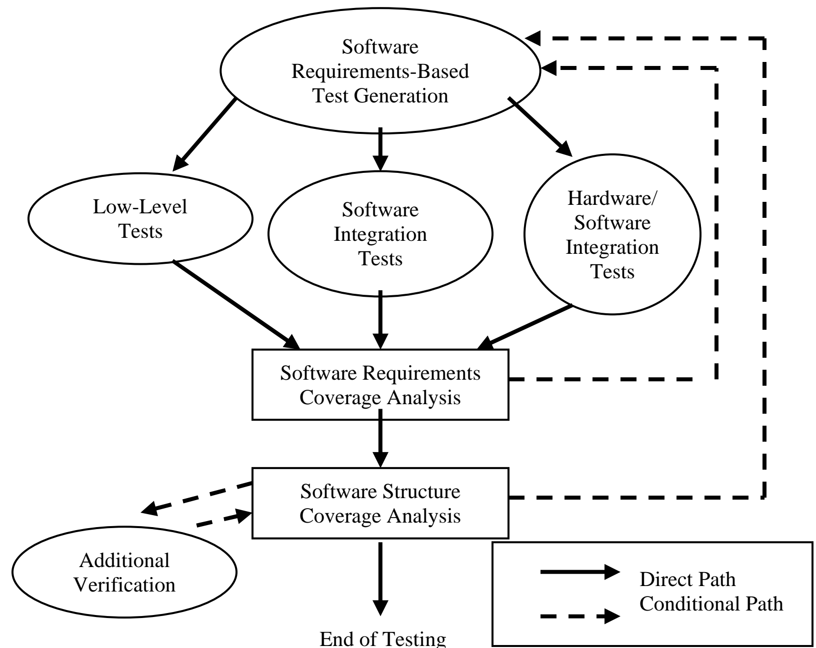

Figure 6: DO-178B Verification Software Process

One way to increase the likelihood of achieving high levels of structural coverage is to do extensive testing of the requirements. This is a strong feature of model-based design and its automated testing and coverage capabilities, some of which were discussed at the 2002 AIAA GN&C conference [3].

Figure 7 shows a series of test sequences specified using a Signal Builder block. Assertion blocks are placed within the model and optionally the generated code to determine if the tests pass or fail based on expected results.

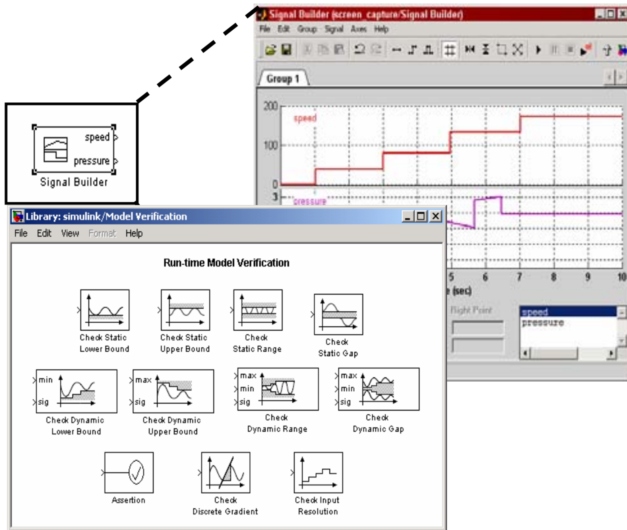

Figure 7: Test Builder and Test Assertion Blocksets

Model coverage also can be generated as part of the test procedure. The coverage reports on state reachability and gives cumulative structural coverages for multiple runs. Current coverages reported include MC/DC.

Production code generation configuration options include the use of makefiles and templates, which allow for the automatic invocation of other tools, such as instruction set simulators and code coverage tools, for further analysis and verification efforts.

Configuration Management

Configuration Management (CM) systems help ensure that changes to documents, models, software and builds are carefully controlled and managed. Developers are required to check-out software before making changes, and then check-in the changed software with revision information. These changes are then merged with other changes in a repeatable and reversible manner.

This is a fundamental requirement of any software engineering process. With high-integrity systems there is additional scrutiny and approvals involved, typically involving configuration control boards.

CM interfaces are available for model-based design environments such as Microsoft Common Source Control for Windows and other particular tools for Unix. An example CM interface is shown in Figure 8.

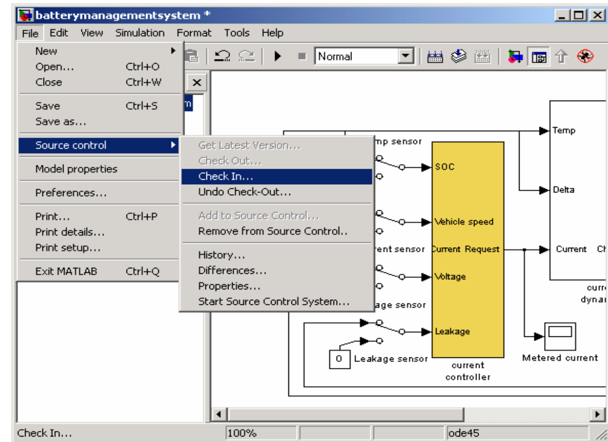

Figure 8: CM Interface using MCSC compliant tool

Requirements Management

Every project has requirements, usually in the language best understood by the customer. Many times these requirements are in a document filled with the word *shall*. Management and control of these requirements is extremely important to keep projects on schedule and within budget.

Requirements management interfaces exist for linking requirements specified in software packages [4] to the model that satisfies the requirement. (See Figure 9.)

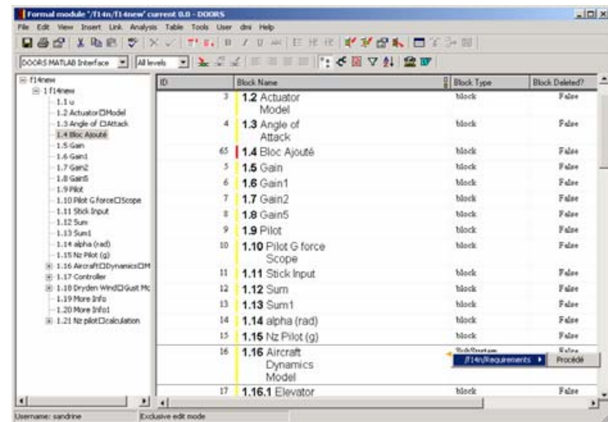

Figure 9: Requirements Traceability using commercial off the shelf requirements tools

When the requirement is selected, the model, block or diagram that satisfies the requirement is then automatically invoked and highlighted to clearly show the trace. This further expands upon the traceability discussions presented earlier in this paper.

Documentation

Documentation is needed for every project, even those based on models. It allows management, customers, and suppliers to comprehend and approve the model at key milestone events such as formal design reviews (e.g., PDRs or CDRs).

Automatic report generation is an important capability within model-based design. An example report is shown in Figure 10. The documents are based on models, thus the reports can be executed and simulated as opposed to simply being static text.

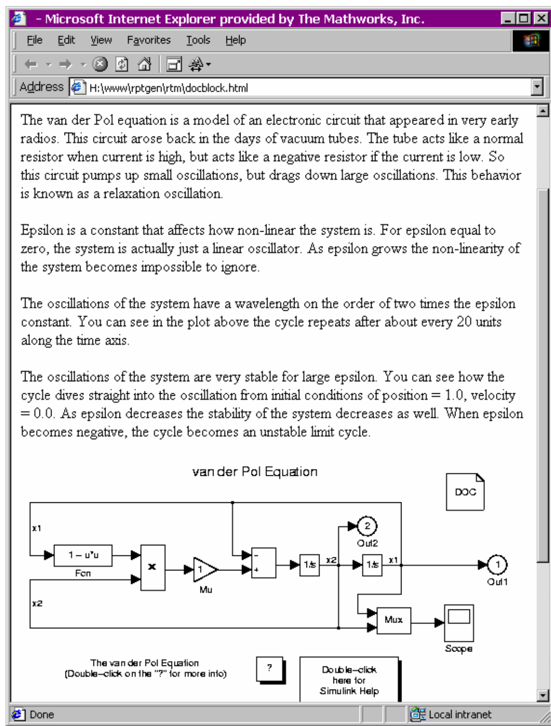

Figure 10: Automatic Report Generation

Conclusion

Generating code is a small part of production code generation. It requires supporting a full range of software engineering activities that are heavily based on the use (and reuse) of models. This paper demonstrated how production code generation and model-based design does indeed fit within software engineering frameworks for high-integrity systems. Specific methods and tools were shown to illustrate these points and to prove that these solutions are not just theoretical, but are real and available.

Each topic on its own can clearly be expanded and discussed in much greater depth. These points may be the subject of future papers. Readers are encouraged to contact the author for additional information and insight.

Finally, when developing high-integrity software, it is important to understand both the letter of the governing standard and the spirit or intent of the standard. Any standard would welcome the benefits described here, such as improved communication, reduced manual translation and the errors it may cause, and the detection of errors early and often.

References

[1] www.mathworks.com
[2] "Software considerations in airborne systems and equipment certification," RTCA/DO-178B, RTCA Inc., Dec. 1992
[3] B. Aldrich, "Using model coverage analysis to improve the controls development process," AIAA GN&C 2002
[4] www.telelogic.com

Author information

Tom Erkkinen is an Embedded Applications Marketing Manager at The MathWorks. He is heavily involved in establishing production code generation environments for embedded system developers worldwide.

Please contract Tom at:

The MathWorks, Inc.
39555 Orchard Hill Place, Suite 280
Novi, MI 48375

USA (248) 596-7943
terkkinen@mathworks.com

-- # # # --