Model Style Guidelines for Embedded Code Generation

T Erkkinen

▶ To cite this version:

T Erkkinen. Model Style Guidelines for Embedded Code Generation. Conference ERTS'06, Jan 2006, Toulouse, France. hal-02270442

HAL Id: hal-02270442
https://hal.science/hal-02270442

Submitted on 25 Aug 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Model Style Guidelines for Embedded Code Generation

T. Erkkinen

The MathWorks, Inc., Novi, MI 48375
tom.erkkinen@mathworks.com

Abstract: Embedded systems are increasingly being developed using models. These models may have started with the system engineer or algorithm developer as an executable specification or algorithm description. However, these models also now serve as the entry point for software engineering, thanks to automatic embedded code generation. As a result, software engineers want to take advantage of these same models, adding constraints on system behavior; describing characteristics that are needed for implementation, such as fixed-point details; or linking components in the design to relevant parts of requirements and specification documents.

This paper describes model style guidelines for automatically generating fixed-point and floating-point code for embedded systems. The guidelines are based on best practices and techniques derived from actual industry examples in aerospace and automotive companies worldwide.

Keywords: Model-Based Design, Production Code Generation, Simulink

1. Introduction

The model structure and code generation configuration options significantly impact the efficiency, clarity, and traceability of the automatically generated code. While model style and verification may not be a top issue for early algorithm development or rapid prototyping, it is extremely important for embedded system development, especially flight software applications. Furthermore, the composition of the model also eases the process integration with the “test-in-the-loop” (e.g., SiL, PiL, HiL) phases that are needed, while improving component reusability and Intellectual Property management.

Increasingly, companies that use model-based design approaches -- particularly in aerospace and automotive -- are collaborating to define model style guidelines, which they can individually tailor and apply as a key aspect of their software development processes.

This provides system engineers and algorithm developers greater freedom to innovate early in the process, while resulting in models that can be augmented, refined, and constrained with implementation details for efficient embedded code generation later in the development process.

2. Discussion

A system model has components for modeling both the algorithm and the environment where the algorithm executes. The algorithm may be a control law or a signal processing algorithm. The environment may include actuators, sensors, and the plant. Or, the environment may represent a communication transport layer with varying latencies and noise.

The algorithm model will eventually be deployed as embedded software on a microcontroller, a digital signal processor, or a multi-core device. The plant model may eventually be deployed on a test system for HiL testing and system validation.

A typical model is comprised of block diagrams, state machines, and embedded language scripts. Examples of system models are shown in Figures 1 and 2. The examples use MathWorks products.

Note that the system models are comprised of individual components. The components may be subsystems pulled from libraries, which have dependencies based on the system model they are placed in. Or the components may be separate models that

are atomic and isolated from the properties of the system model they are placed in.

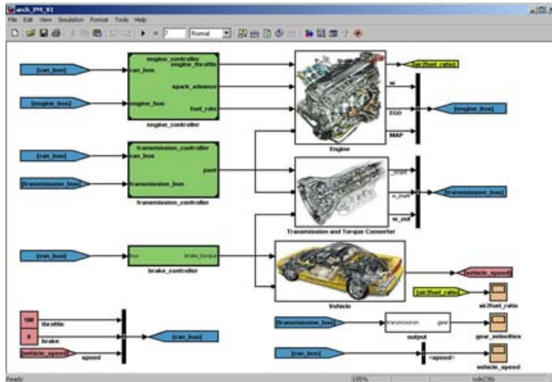

Figure 1: Vehicle and ECU System Model

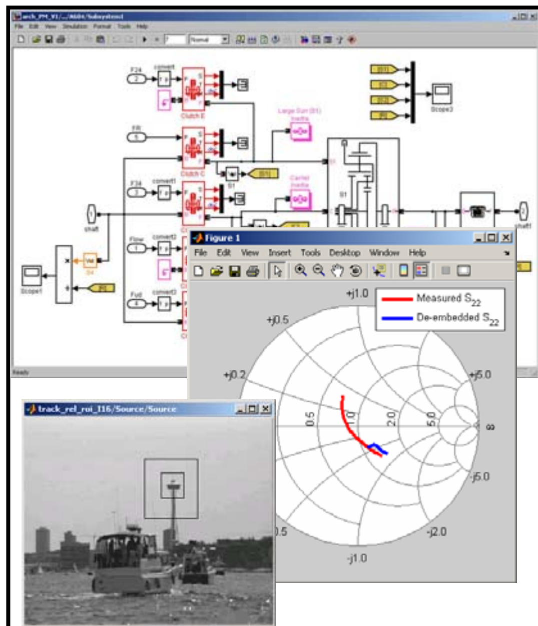

Figure 2: Radar Application System Model

2.1 Simulation

In order for the model to be clearly understood, it needs to simulate. Successful simulation requires that the model compile and execute based on the values established for the model's diagnostic settings, such as array out-of-bounds error checking.

The simulation results can be shown in many ways such as scopes, gauges, and virtual animation. See Figure 3 for an example of an aircraft system simulation interfaced to a flight simulator. The results can also be produced as massive amounts of data that are reused as test suites during detailed design and implementation.

Figure 3: Aircraft System Model Animation

The simulation model can also be used as an executable specification. The fidelity of the model dictates whether or not the model specification is high-level or low-level.

There are other aspects in addition to model fidelity to be considered during the model development process. Some aspects span multiple groups or even different companies, such as OEMs and suppliers. For example, Toyota Motor Corporation and DENSO Corporation discussed important aspects that guide their modelling and production code generation environments [1].

Toyota needs automatic code generation tools that are *system development-oriented:*

- *Ease of simulation*
- *Easy operation*
- *Seamless linkage with other system development tools and data*

DENSO needs code generation tools that are *software development-oriented:*

- *Conformance to coding rules*
- *Well defined software structure*
- *Seamless linkage with other software development tools and data*

One of the keys to a successful model style is to develop models that allow for fast, easy, and seamless transitions between system engineering and software design.

2.2 Detailed Design

Technology improvements starting in the late 1990s have allowed high-level models to be refined and elaborated to great levels of detail. Software design details such as fixed-point data type, scope and storage class, function and file partitioning, and explicit identifier control are now possible.

Back and forth transition between model fidelities improves system and software communication and also fosters a development approach that encourages iterations for optimizing code, addressing derived requirements, and improving design integrity.

Data type design is a good example of this back and forth flow. Early during high-level specification, numerical accuracy involving data types is not a major consideration. The models typically use double precision data for assessing algorithm performance.

Once the performance is deemed satisfactory, however, detailed data type design should occur. Single precision or fixed-point types are then derived for each signal and parameter in the model based on embedded target resources or other factors. Once the data types are assigned, it is important to compare the detailed types with the original high-level specification.

Data type override is one way to accomplish this comparison. With data type override, a developer is able to make the simulation and code generation tool ignore the detailed data types and act upon the model as if all data elements were specified as a specific floating point type (e.g., double precision real). This greatly facilitates comparison and verification of the detailed design with its higher-level representation.

One could also use a data dictionary driven approach. For example, a floating point data dictionary could be loaded and used for rapid prototyping simulation, while a fixed-point data dictionary could be loaded for embedded, mass-production target environments. To do this, the model needs to be developed in a style that supports inheriting data types from an external or workspace data dictionary. The external data dictionary could be provided, for example, using a corporate database tool, Excel, or XML file.

An example of a data dictionary and a data dictionary element for a lookup table is shown in Figure 4.

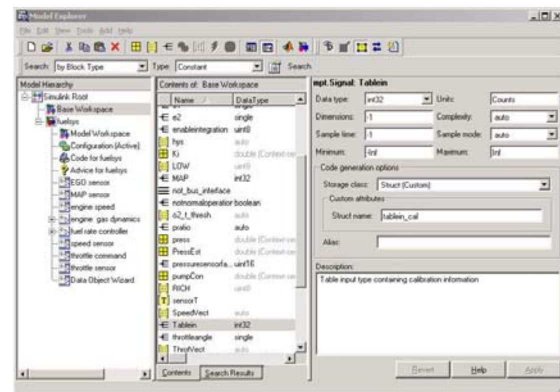

Figure 4: Data Dictionary Showing Lookup Table using Structure Data Types

Visteon Corporation has described their data dictionary driven approach for ECU software using automatic production code generation in an SAE paper [2]. This paper also includes code metrics comparing hand-written production code with automatically generated code as shown below.

Table 1: Code Size comparison between a fixed-point hand code and auto code.

		Code Size
Hand Code		928
Auto Code	No overflow/underflow check	904
	Check OF/UF everywhere	1562
	Check only where necessary	934

*Based on Tasking Compiler for ST10

The above table shows that the overflow/underflow checking increases code size significantly, so it is not practical to do this check everywhere. Fixed-point modeling is the best way to determine where the check is necessary. It is clear that the customized Embedded Coder® is able to generate efficient fixed-point code.

Table 2 shows ROM and RAM comparisons between hand code and auto code for a floating-point component in some typical powertrain software.

Table 2 ROM and RAM comparison between a floating-point hand code and auto code.

	Hand Code	Auto Code
ROM	6408	6192
RAM	132	112

Figure 5: Visteon Production Code Metrics

General Motors Powertrain also described their use of data dictionary driven development and code metrics at a recent conference [3]. They also presented hand vs. automatic code metrics as shown below in Figure 6.

	Hand Code	Auto-code	Percent Change
Calibration ROM	9464	9464	0%
Code ROM	2952	2900	-1.76%
RAM	240	238	-0.83%

Figure 6: General Motor Powertrain production code metrics

2.2 Code Generation

For companies trying to maximize code efficiency, it is important to have a good understanding of the appropriate blocks, appropriate block parameters settings, and the appropriate code generation settings.

As with any language, teams should agree on the language aspects to use or not use for a particular project. The aspects may be based on the target environment, the software integrity level, or other criteria. Model tool documentation such as the block data types table shown in Figure 7 guides developers on which constructs to use.

Web Browser - Simulink Block Data Type Support

Sublibrary	Block	Double	Single	Boolean	Base Integer	Fixed-Point	Code Generation Support
Math Operations	Abs	X	X		X	X	X
	Algebraic Constraint	X					
	Assignment	X	X	X	X	X	X (N2)
	Complex to Magnitude-Angle	X	X				X
	Complex to Real-Imag	X	X	X	X	X	X
	Dot Product	X	X	X	X	X	X
	Gain	X	X		X	X	X
	Magnitude-Angle to Complex	X	X				X
	Math Function (Exponential)	X	X				X
	Math Function (log)	X	X				X
	Math Function (10^u)	X	X				X
	Math Function (log10)	X	X				X
	Math Function (magnitude^2)	X	X		X	X	X
	Math Function						

Figure 7: Block Data Type Table for Production Code Generation

In addition to the blocks, the block settings heavily influence the format of the generated code. One of the key settings in a fixed-point environment is the check for numerical overflow involving fixed-point calculations. The block parameter form for a simple gain block in Figure 8 shows this setting.

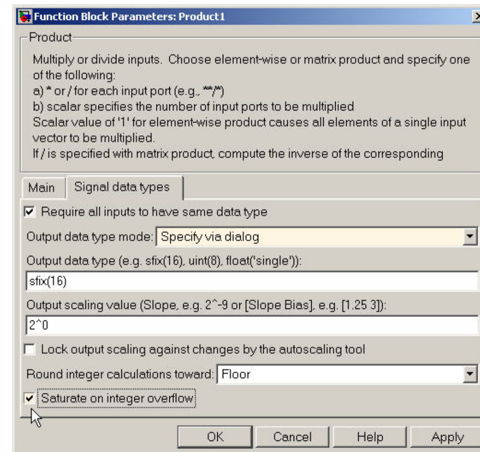

Figure 8: Saturation on integer overflow block setting enabled

The Visteon metrics shown in Figure 5 noted that this saturation setting determined whether or not the automatically generated code was smaller, larger, or equal to hand code. If the setting was used "where

necessary" the automatic code was approximately equal to hand code.

The code generation configuration settings also have a great influence on the generated code. One of the major groups of settings that is important involves the hardware implementation pane. This pane is where one establishes the target specific settings not defined by ANSI/ISO-C, such as integer word sizes.

By selecting the appropriate target hardware developers may realize significant code efficiency improvement. See Figure 9.

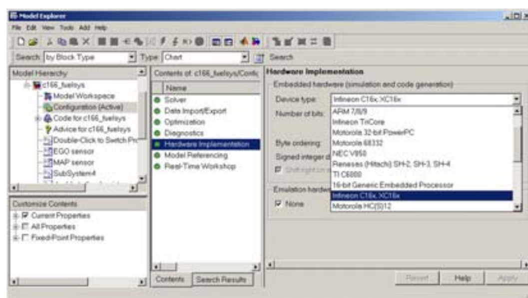

Figure 9: Production hardware settings used to deploy the automatically generated code

Taking the time to establish these and other model and code generation guidelines can lead to impressive results beyond code efficiency. Honeywell Aerospace Corporation presented that they generated over 1.6 million lines of certified flight code within the past year using a COTS tool-based environment [4]. Honeywell also noted that their software process produced greater than six-sigma code quality.

It is important to note that Honeywell's automatically generated software was developed in accordance with criteria per DO-178B Level A software integrity [5].

2.3 Verification and Validation

Using Model-Based Design, verification and validation activities occur throughout development. A number of new technologies have recently been introduced that assist with this such as model advisor, model coverage tools, and in-the-loop testing.

One of the key benefits to establishing model guidelines is that they can be automated. The model advisor analyzes the model and checks for areas that may impede its use in production software environments. Some of the checks focus on the simulation aspects, others on code generation.

For example, one of the checks makes sure that model interface ports are well defined and do not inherit important characteristics such as data type and sample rate. Another check informs users if they are missing important code optimization settings. Figure 10 shows checks based on the model, the code generation settings, and customized user-provided settings.

Even if one does not use the model analysis tool itself, developers should inspect the many guidelines that it offers. Note that the model advisor also has an API for an organization to add their model checks.

Figure 10: Interactive model coverage reporting

Another important verification step is to design and execute model tests. For safety-related systems, it is important that the test cases be based on the requirements. Bi-directional links between the model and the systems requirements in documents, data bases or requirements management tools is important and available. Requirements-based comments can also appear in the generated code.

In addition to supporting requirements-based testing and bi-directional traceability, the model should support test coverage

analysis. One way to determine test coverage is to use a model coverage tool. Figure 11 shows such a tool and highlights areas where the model was not exercised.

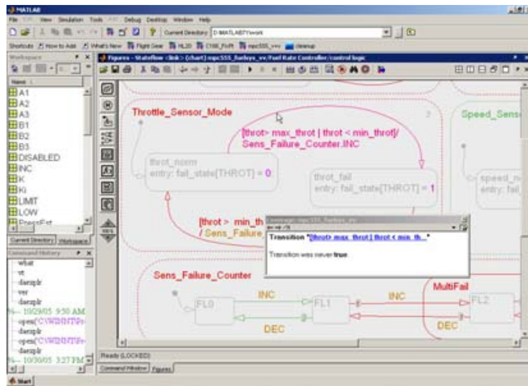

Figure 11: Interactive model coverage reporting

Once the model has satisfied its test and coverage requirements, code can be automatically generated, as described earlier. As code implementation and integration moves forward, there are several opportunities to verify the implementation using SiL or HiL.

DaimlerChyrsler Trucks noted the importance of using SiL testing [6] with automatically generated code:

SIL based function development brought a high state of maturity before the vehicle tests start. Desktop debugging instead of debugging in the vehicle allows high test efficiency. The Embedded Coder meets our demands concerning code efficiency, structure, and automatic coding.

There are many other sources of guideline information including C code guidelines [7], automotive model guidelines [8], large scale model guidelines [9], and fixed point tips [10].

Developing a complete modelling and code generation environment takes some time and effort but the rewards are worth it.

Consider what Lockheed-Martin presented regarding their use of automatic flight code generation for the Joint Strike Fighter program flight controls [11].

It is proven in CDA Phase:

- *Successful flight tests of all variants with one OFP*
- *Reduced Software Defects (Early checkout in Engineering Simulations)*
- *Overall reduction in Manhours/SLOC of ~40%*

3. Conclusion

This paper presented a few important guidelines and industry trends for using modelling and automatic code generation for production software. It was not possible to provide numerous detailed examples in the space provided but several references were provided containing numerous guidelines, examples, and tips.

This paper also discussed approaches for verification and validation of the models and generated code. Industry examples were referenced.

7. References

[1] T. Katayam, A. Ohata, Toyota; Y. Uematsu DENSO; ”Production Code Generation for Engine Control System”, International Automotive Conference, Stuttgart, 2004 http://www.mathworks.com/company/events/programs_de/iac2004/presentations/08_pp_toyota.pdf

[2] Grantley Hodge, et al, Visteon Corp, “Multi-Target Modeling for Embedded Software Development for Automotive Applications”, SAE Congress Technical Paper No. 2004-01-0269, Detroit, 2004 http://www.mathworks.com/products/rtwembedded/technicalliterature.html

[3] L. Michaels, General Motors Powertrain, "Automatic Code Generation Process", MathWorks International Automotive Conference, Dearborn MI, 2005 http://www.mathworks.com/industries/auto/iac/presentations/michaels.pdf

[4] B. Potter, Honeywell; "Achieving Six Sigma Software Quality Using Automatic Code Generation", Mathworks International Aerospace and Defense Conference, Manhattan Beach CA, 2005 http://www.mathworks.com/industries/aerospace/miadc05/presentations/potter.pdf

[5] RTCA Inc., "Software considerations in airborne systems and equipment certification", RTCA/DO-178B, Dec. 1992 http://www.rtca.org/

[6] M. Wünsche, et al. DaimlerChrysler AG; "Model based development of Cruise Control for Mercedes Benz Trucks", Mathworks International Automotive Conference, Stuttgart, 2004, http://www.mathworks.com/company/events/programs_de/iac2004/presentations/11_doc_IAC_DaimlerChysler.pdf

[7] Motor Industry Software Reliability Association (MISRA), "Guidelines for the Use of the C Language in Vehicle Based Software", ISBN 0 9524156 9 0, April 1998. http://www.misra.org.uk/

[8] MathWorks Automotive Advisory Board (MAAB), "Controller Style Guidelines for Production Intent Using MATLAB®, Simulink® and Stateflow®," April 2001, www.mathworks.com/industries/auto/maab.html.

[9] Rob Aberg and Stacey Gage; "Strategy for Successful Enterprise-Wide Modeling and Simulation Using COTS", American Institute of Aeronautic and Astronautics GN&C Conference, August 2004. http://www.mathworks.com/company/events/aiaa_04/

[10] Siva Nadarajah and Vinod Reddy: "Fixed-Point Modeling and Code Generation Tips", MATLAB Central, 2005, http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=7197&objectType=FILE

[11] D. Nixon, Lockheed Martin; "Flight Control Law Development for the F-35 Joint Strike Fighter", Mathworks International Aerospace and Defense Conference, Newton MA, 2004, http://www.mathworks.com/industries/aerospace/miadc/presentations/10_F35_Flight_Control_DevelopmentDaveNixon.pdf

8. Glossary

SiL Software-in-the-Loop
PiL Processor-in-the-Loop
HiL Hardware-in-the-Loop
ECU Electronic Control Unit
OEM Original Equipment Manufacturer
FAA Federal Aviation Agency
API Application Program Interface
OFP Operational Flight Program
SLOC Source Lines of Code