2010-01-0938

Large-Scale Modeling for Embedded Applications

Kerry Grand, Vinod Reddy, Gen Sasaki, and Eric Dillaber
The MathWorks

ABSTRACT

As the demand for high-integrity and mission-critical embedded software intensifies, many organizations have adopted Model-Based Design to overcome the challenges associated with design complexity, reliability, quality, and time-to-market for embedded-systems development. The breadth and scope of projects applying Model-Based Design continues to increase rapidly, resulting in models that are exceptionally large and complex. Consequently, project teams have increased in size, thereby increasing the need for communication and collaboration. Model-Based Design facilitates parallel development in large-scale modeling projects by enabling multiple project teams to independently design models, integrate them with others, generate production code, and verify different model components within a larger collaborative infrastructure.

To facilitate and increase the efficiency of large scale modeling throughout the entire development life-cycle, a thorough understanding of the various steps and techniques involved in successfully applying Model-Based Design is required. These include a logical architecture to divide the model into components, clear definition of the interfaces prior to component design, maintenance of those interfaces, the production code generation approach, and the development infrastructure. The resulting design should use the same model components for design, verification, automatic document generation, and production code generation. This paper recommends best practices for creating an infrastructure and deploying large-scale models for embedded applications using Model-Based Design. The intended audience is individuals who plan to deploy a design with greater than 100,000 Simulink blocks and have experience using MATLAB, Simulink, and Real-Time Workshop Embedded Coder.

INTRODUCTION

Over the last few years, many companies have been deploying highly complex and large-scale embedded systems using Model-Based Design [1, 2]. Due to the ability for engineers to communicate using models, create frequent and faster iterations of the design, and generate production code, Model-Based Design for large embedded systems creates both opportunities and challenges. The definition of a large model is however subjective. A model is considered large if it is too big for one person to know all its details, if it exceeds 100,000 blocks, or if it contains over 100 inputs and outputs. A model is certainly large when it requires significant overhead to coordinate the modeling effort across multiple disciplines or teams.

Project teams working with large models targeting an embedded application often experience a common set of challenges and can benefit from industry-proven approaches. These challenges include architecture, design, and

implementation. In this paper, we discuss each one of these areas in detail and recommend approaches to overcome these challenges.

MODEL ARCHITECTURE

Model architecture is one of the key development tasks when designing a large-scale, highly complex, and highly interoperable system. It primarily defines system qualities, such as manageability, performance, modifiability, and testability. With Model-Based Design centered on a system model [3], a common a set of questions arise for model architecture that include:

- How do you partition the model into manageable components to aid in parallel development and reuse?
- How do you define and manage a consistent set of interfaces?
- How do you control execution of the model?
- How does model architecture affect testing?

In this section, we address these questions about componentization, interface management, execution control, and the impact of model architecture on testing.

COMPONENTIZATION

A component is a piece of your design, a unit level item or a subassembly, that you can work on without needing the higher level parts of the model. You can perform component-level design, simulation, testing, code generation, and verification and validation. If the component is a model, you can run it as is in Simulink; if the component is a library-based atomic subsystem, you will need a harness model and will need to ensure that the harness model propagates the same interface into the subsystem. Smaller items like library blocks usually perform as utility functions and are not really a component of the design (even though you design and test them much like a component, they are more generic and reusability may come at the expense of some kinds of efficiency). Additionally, dividing a model into components provides the ability to control smaller portions of the design using version control and configuration management systems [4].

Two architecture constructs are considered in this paper: model reference architecture in Accelerator mode and library models containing atomic subsystems. While model reference architecture and libraries can be used in other ways, these constructs generally provide the most benefits for large-scale models. The choice of which construct to use should be based on the tradeoff in benefits that they offer.

Recommendation 1.

Partition the top level model components using model reference architecture in Accelerator mode.

Model reference lets a model component exist as its own independent model file and as part of a larger model. Model references are created using Model blocks which can be run in Accelerator mode. In Accelerator mode code is generated from the model and compiled into an optimized format for simulation with respect to memory and CPU cycles. As a result, models that are too slow for simulation can be accelerated if enough of their subsystems are replaced with Model blocks. Using model reference does degrade performance slightly when updating a model (update diagram) because each reference model is checked for changes to enable incremental builds. Using one instance or just a few reusable instances of a particular reference model, model references of 500 to 5000 Simulink blocks as a lower threshold is recommended. If there are many instances of a reference

model, model references can be even smaller. Alternatively, with small components that are reused in many places within the same model file, consider instead using a library model with a reusable atomic subsystem. Model reference lets users apply version control to a reused component independently of the models that use it. One final note is that when a user updates a Simulink model, all of its blocks are loaded into memory. Because model reference architecture in Accelerator mode acts as a single Simulink block, the memory requirements are much lower when compared to updating an atomic subsystem of identical block content.

Recommendation 2.

Use atomic subsystem library models for components with fewer than 500 blocks.

As with model reference architecture, library models also let a model component exist as part of a larger model and it can be stored as a separate model file. However, libraries cannot be used independently. They require a parent model and do not reduce memory consumption or CPU cycles in normal simulation mode. Typically, libraries are intended for low level utility functions, which are used multiple times in a design. The key difference between model reference architecture and libraries is that libraries can be used with different data types, sample time, and dimensions, in different contexts, without changing the design. Making a library component an atomic subsystem provides a method of grouping blocks into a single execution unit within a model. Therefore, the use of atomic libraries is recommended when libraries are used for model components. The generated code can optionally be placed in a separate function and source file. Libraries do allow users to apply version control to a component independently of the models that use it. However, because libraries are context-dependent and need a parent model to generate code, code generated for the library model may differ in each instance. A benefit of this context-dependent approach is that libraries can adapt to various interface specifications. However, for large-scale model components this property is not generally desired because the interfaces are usually managed and locked down to a specific data type and dimension.

Note that we do not recommend partitioning a million-block system model into 200 Model Reference blocks that are each 5000 blocks in size. We recommend using model reference Accelerator blocks at the top level component partitions and blending model reference Accelerator and atomic subsystem libraries at lower levels.

INTERFACE MANAGEMENT

As in a traditional software development process, model interfaces should be managed by using a centralized repository to capture interface information, such as data type, range, description, initial value, dimension, and sample time. These interfaces exist at the model boundary, with internal states, parameters, and signal buses. The mechanisms used to manage interfaces are Simulink data objects for signals, parameters, and buses.

Recommendation 3.

Design for portability and reusability by using Simulink data objects.

Signal and parameter data objects are instantiations of Simulink or module packaging tool (MPT) classes and exist within the base workspace. Using these data objects to define a signal, state, or block parameter associates the information within the object to the model component. For example, a signal in Simulink and an MPT signal object with the same name share properties. As a result, the interface data can be managed separately from the model, enabling a centralized data repository external to the model, such as a data dictionary, and improving model reuse. Signal and parameter data objects can also specify storage classes, which control code generated

for data objects. Using the Custom Storage Class Designer tool, you can create specialized software interfaces such as unique data access methods and packaging. Additional properties can be added to existing Simulink data objects or new data objects can be created using the Simulink Data Class Designer.

We recommend deriving a custom data class package from the MPT class and inheriting all of the MPT custom storage classes. This workflow enables teams to create new custom storage classes or add properties without changing the MATLAB/Simulink installed products. The custom data class package can be managed independently from the MATLAB/Simulink environment and used simply by adding the package folder location to the MATLAB path.

Recommendation 4.

Make buses virtual except for model reference component boundaries.

Another form of interface management exists through the use of buses, which group signals or other buses to help route and off load the developer from managing large numbers of signals or interfaces. Buses are created using the Simulink Bus class. Unlike a Simulink or MPT signal, a Simulink Bus class is analogous to a type definition in C. The Simulink Bus class specifies only a structure type and does not instantiate memory in the generated software. Additional properties cannot be added to the bus package and the storage class definition must come from a Simulink or MPT signal object. Buses also can be virtual, meaning that they do not affect the generated code except for cases where they cross a model reference component boundary. With a model reference, virtual buses are converted to a non-virtual bus; when code is generated using Real-Time Workshop Embedded Coder, the interface is represented as a C structure. It is recommended that buses be kept virtual to enable further code generation optimizations on signals except with model reference [5] where non-virtual buses are required. Atomic subsystems that are converted over to model reference will have to convert the buses to non-virtual at the boundary.

We do not recommend partitioning every component interface into a bus to ease signal routing. Getting the correct number of buses and separate inputs/outputs is an interface design problem. A large bus can be undesirable if it:

- unnecessarily hides the interface requirements
- increases component complexity, because the bus has to be packed and unpacked
- negatively impacts code generation optimization and embedded target throughput, because some modeling constructs can force a bus copy

Hundreds of separate signals can increase the amount of function arguments in the generated software. As a result, the model becomes difficult to understand because of the large number of interfaces, and desired operations on contiguous groupings of data are not possible. Typically, buses make sense where high cohesion exists with model components.

Recommendation 5.

Make your model interfaces explicit by minimizing or eliminating global data stores, global events, and global Goto/From blocks.

Large-scale models containing components often employ one of two strategies to share data: global data or propagating signals throughout the model. Global data sharing within Simulink is typically accomplished through the use of global data stores and global Goto/From block tags. In Stateflow, global events can be triggered by the Stateflow global events feature. The main drawback of using global data is that the source and destination become implicit, which makes the design difficult to debug because the signal flow is not presented in the model. Alternatively, connecting and sharing the data via signals makes the interfaces explicit. We recommend making the model interfaces explicit through the use of signal routing, keeping in mind the best practices for data stores [6]. Explicit signal routing has the added benefit of reducing the effort required to convert a component over to model reference since global Goto/From blocks and Stateflow global events can not cross the model reference boundary.

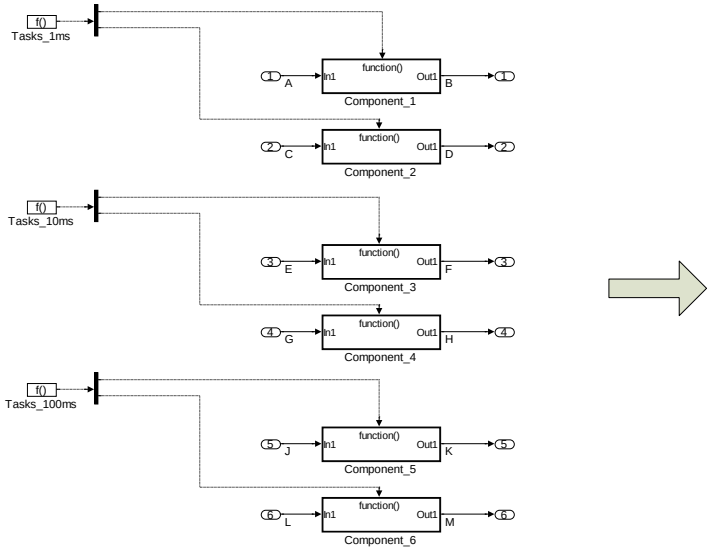
EXECUTION CONTROL

Recommendation 6.

Use function-call scheduling if the goal is to match the model to existing software architecture. Otherwise, let Simulink determine the best order based on data dependency analysis.

Simulink provides a variety of mechanisms to easily specify multiple execution rates, tasks, and data protection. Synchronous components are commonly modeled using Simulink's intrinsic scheduling capabilities. Asynchronous components are commonly modeled using function-call scheduling. When the goal is to map a component to existing software architecture, function-call scheduling is commonly used. However, blocks can be grouped into an atomic subsystem or model block and explicitly triggered by a function-call or a Stateflow chart to model predefined or an existing embedded scheduler. This approach lends itself to matching the model architecture with the existing embedded software. Figure 1 shows a function-call subsystem approach to simulating the embedded scheduler. In this example three execution rates (1 ms, 10 ms, and 100 ms) are generated from Function-Call Generator blocks. Given the resulting call tree for each execution rate, the 1-ms scheduler calls Component_1 and Component_2, the 10-ms scheduler calls Component_3 and Component_4, and the 100-ms scheduler calls Component_5 and Component_6. Asynchronous rates can also be modeled.

A benefit of function-call subsystems is that the algebraic loops normally encountered can be automatically resolved through the specified execution order, which eliminates the need for adding a unit delay to feedback loops. However, care must be taken to avoid data dependency violations within the final architected model [7]. Data dependency violations occur when Simulink signal data is not valid prior to execution of a function-call subsystem or model block. While function-call architecture provides a direct mapping between model and code, it is often unnecessary as Simulink can determine the execution order and enable different rates to communicate through Rate Transition blocks.




```
/* Model step function for TID0 */
void call_tree_example_step0(void)     /* Sample time: [0.001s, 0.0s] */
{
  /* S-Function (fcncallgen): '<Root>/Tasks_1ms' incorporates:
   *  SubSystem: '<Root>/Component_1'
   *  SubSystem: '<Root>/Component_2'
   */
  call_tree_examp_Component_1();
  call_tree_examp_Component_2();
}

/* Model step function for TID1 */
void call_tree_example_step1(void)     /* Sample time: [0.01s, 0.0s] */
{
  /* S-Function (fcncallgen): '<Root>/Tasks_10ms' incorporates:
   *  SubSystem: '<Root>/Component_3'
   *  SubSystem: '<Root>/Component_4'
   */
  call_tree_examp_Component_3();
  call_tree_examp_Component_4();
}

/* Model step function for TID2 */
void call_tree_example_step2(void)     /* Sample time: [0.1s, 0.0s] */
{
  /* S-Function (fcncallgen): '<Root>/Tasks_100ms' incorporates:
   *  SubSystem: '<Root>/Component_5'
   *  SubSystem: '<Root>/Component_6'
   */
  call_tree_examp_Component_5();
  call_tree_examp_Component_6();
}
```

Figure 1. Subsystem execution control using Function-Call Generator blocks.

Recommendation 7.

Avoid the use of block priority to control the Simulink execution order.

Simulink enables fine grain control of the execution order through block priority, which should be avoided for two reasons. First, when a block is copied, its block priority specification is copied too, which can lead to unintended order of execution. Second, the block priority has to follow an execution order allowed by Simulink. Otherwise, the priority will be ignored with a warning or an error as determined by the model configuration settings. This can be time consuming to debug.

Inherited and specified sample times pose another concern. To increase component reusability, a preferred mechanism is to allow Simulink to propagate sample times from the source block to downstream blocks through inheritance. This mechanism can be accomplished by specifying an inherited -1 sample time. Be mindful of filters and other blocks with dynamics (discretized blocks) because they may need to execute at a rate other than their source data to assure stability of the filter.

Simulink block sorting assumes that nothing executes simultaneously on multiple processing units and that all rates are interrupting other processing, so no conditions can exist when two or more pieces are active at the same time. As a result, simulations do not presently utilize Parallel Computing Toolbox.

ARCHITECTURE IMPACT ON TESTING

Recommendation 8.

Architect the design such that engineers can independently test their areas of responsibility in the model domain.

Team organization should be considered in architecting a large-scale model. When a large model is managed by multiple engineers, engineers are often responsible for discrete sections of the model. If testing responsibility is also partitioned, the engineers should test only those parts of the model based on their responsibility, as shown in Figure 2.

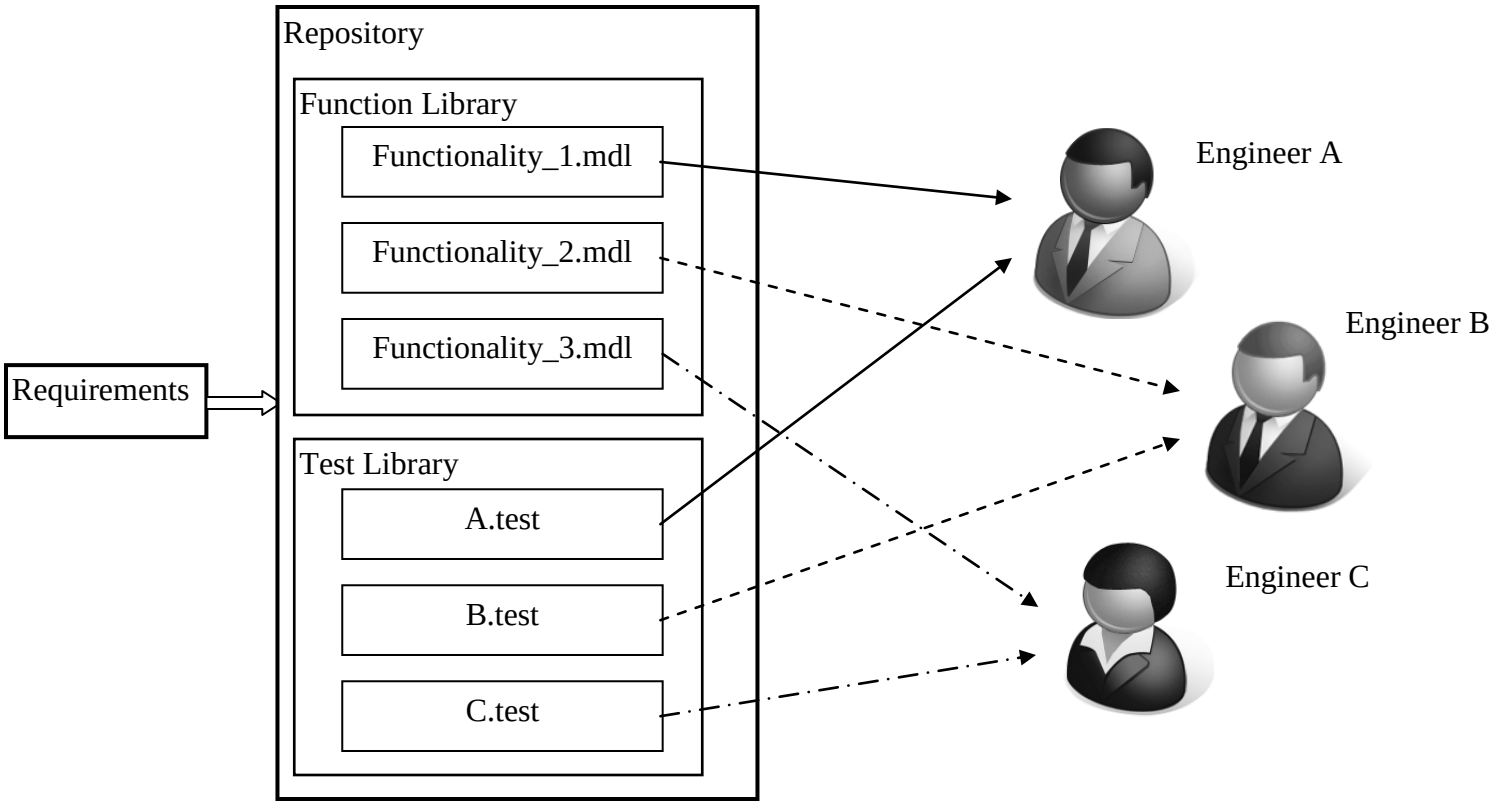


Figure 2. Modules should be partitioned to allow engineers to access them without conflict.

Figure 2 shows an architecture in which engineers are not in conflict when testing their functionality because each engineer has check-in rights to only their files. For example, Engineer A can check-out `Functionality_1.mdl` and A.test from the repository, perform the test, add any corrections to the `.mdl` file, and check the files back into the repository. Consider the case where A.test also tests functionality in `Functionality_2.mdl`. Engineer A may find problems with `Functionality_2.mdl`. If both Engineer A and Engineer B make corrections to `Functionality_2.mdl`, there will be a check-in conflict. One way to avoid this conflict is to forbid Engineer B to test while Engineer A tests, but that approach leads to lost productivity. The ideal solution is to architect the model such that there is no overlap in ownership. If overlap cannot be avoided, a version control system should be utilized to avoid conflicting or lost changes in the model or tests.

Recommendation 9.

Implement a consistent method for signal injection and logging with the model architecture.

Pass/Fail criteria of components are judged by monitoring signal values or model states. The model architecture must allow these values to be measured, which is especially critical if the model has global data that is overwritten by many functions. The model should be architected so that signal values can be recorded at certain points and that test input values can be injected into the model for stimulation and testing.

The mechanism to implement signal injection and signal logging is the same regardless of model size. In large-scale models, many tasks associated with testing must be heavily automated so that the engineer can quickly create, execute, and analyze many tests. Therefore it is important that these injection and logging methods are implemented in a consistent manner for all model files.

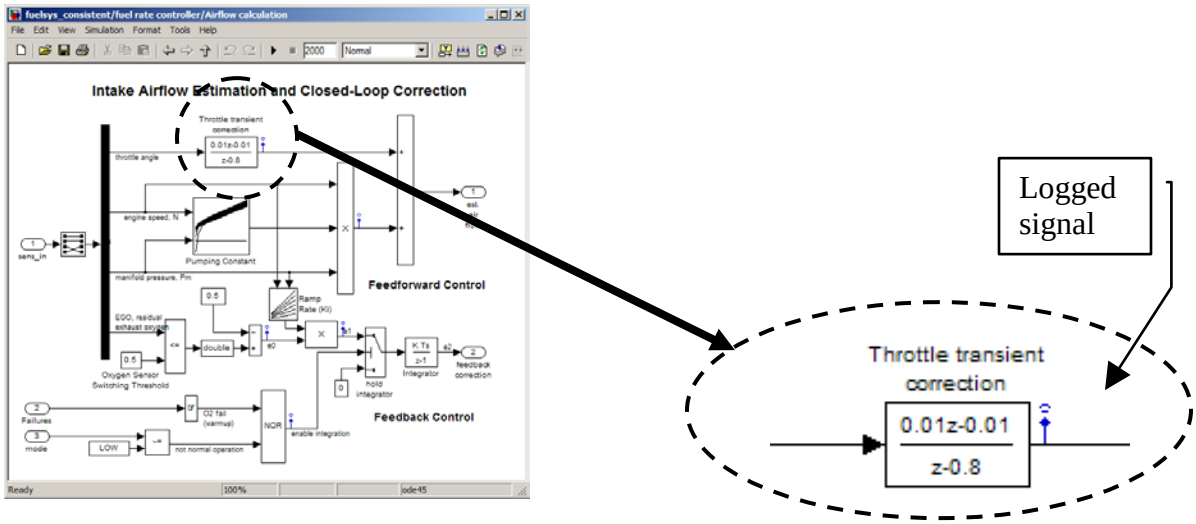


Figure 3. Consistent observable and controllable subsystem.

Figure 3 shows a model in which capturing intermediate signals (not model inport/outport signals) is implemented by using logged signals. Signal injection is implemented exclusively through the use of model inports. Having such a consistent method allows the engineer to properly set test input data and logged data in testing tools such as SystemTest.

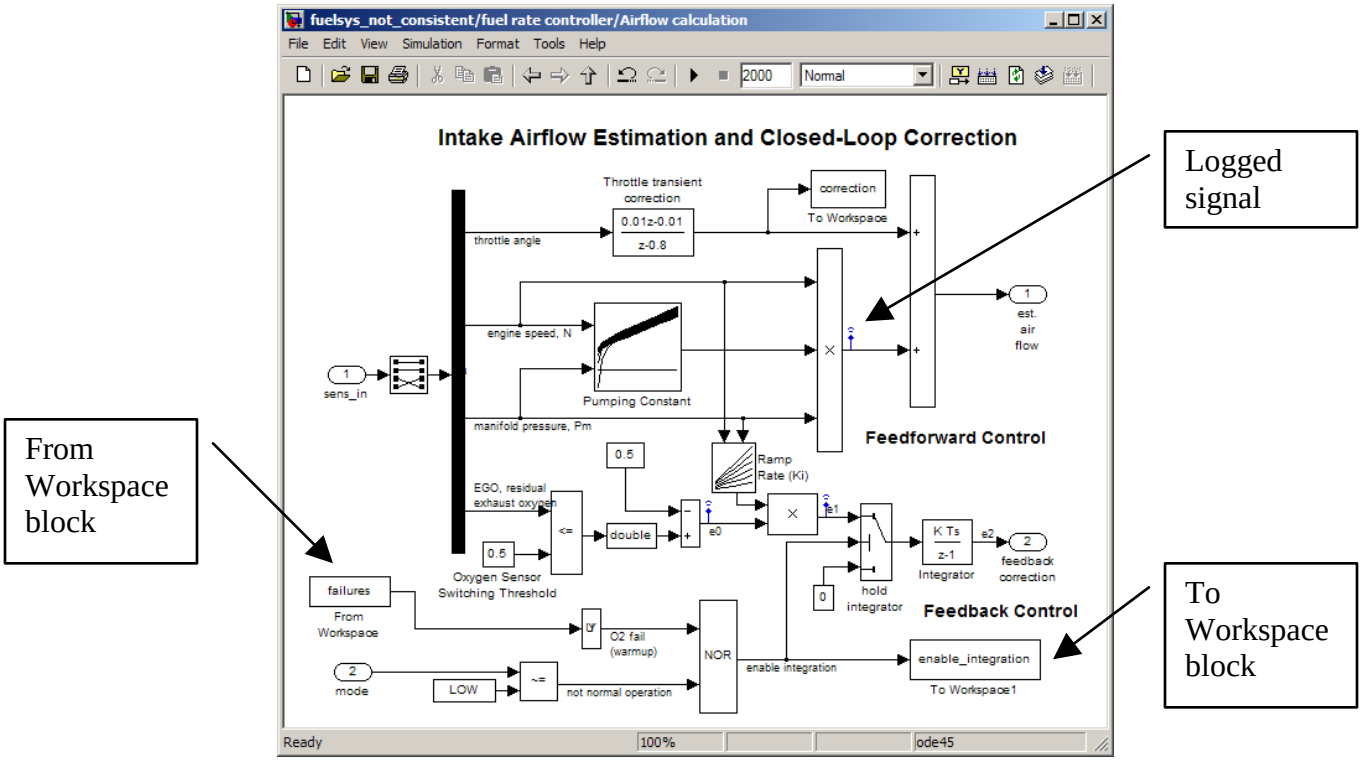


Figure 4. Inconsistent observable and controllable subsystem.

In Figure 4, data capture is implemented using logged signals and To Workspace blocks. Signal injection of the failures input is implemented with a From Workspace block, and the other inputs are model inports. Although this type of model construct is perfectly valid, setting up these disparate methods in the testing tool will most likely need manual intervention, and therefore be prone to error. Furthermore, it is not recommended to use To and From Workspace blocks within a model component. Signal injection and logging through the use of To and From Workspace blocks should happen outside the model with a test harness.

DESIGN AND IMPLEMENTATION

Too often the design and implementation considerations for collaborative development are not fully understood or are implemented late in the development phase forcing rework. This section addresses four areas of design and implementation that impact large-scale modeling for embedded applications: consistent modeling environments to support collaboration, model modularity, selecting a modeling language (Simulink, Stateflow, and Embedded MATLAB), and selecting a production code generation approach.

CONSISTENT MODELING ENVIRONMENTS

When deploying code from a large model it is extremely important that the collaborative modeling environment have consistent settings between the MATLAB environment, MATLAB workspace, model configuration set, and embedded target preferences. Failure to maintain consistent settings usually results in compile or run-time errors in the final target code generation.

Recommendation 10.

Create a consistent modeling environment by using startup.m, cached MAT-file loading, referenced configuration sets, sl_customization.m, and system target file.

Recommendation 11.

Architect the model and scripting to support parallel model reference accelerator builds and code generation target builds using Parallel Computing Toolbox.

A startup.m file placed in your MATLAB startup folder executes commands when MATLAB starts, which enables you to add required folder paths to the MATLAB path through the `addpath` MATLAB command. Because this method supports the MATLAB session it can be used with multiple projects. This startup file should be used with parallel simulation and code generation target builds as well because the MATLAB path must be consistent for the local workers.

The MATLAB workspace needs to be consistent when developing a single component or using production code generation for the final target. Typical MATLAB workspace objects include signal objects, parameter objects, configuration set objects, and parameters. Two issues could arise here. First, inconsistent parameter files could be used for component and system model development. Second, the loading of multiple MATLAB files (note MATLAB files are textual and MAT files are binary) with large data arrays could take a significant amount of time and CPU memory. The use of a scripted cached loading mechanism [8] can be used to eliminate these two issues. When the required MATLAB workspace objects are needed, the cache mechanism loads the binary MAT file if the timestamp is newer than the MATLAB file; loading a binary file can improve load time and reduce CPU memory consumption.

For completeness the concept of a model workspace needs to be mentioned here. Although it is common to rely entirely on the MATLAB workspace to manage data, the base workspace does not provide a private data architecture since every model has access to MATLAB workspace data. Other requirements may exist such that the data needs to be encapsulated with the model due to configuration management. This is where the model workspace can be used. The model workspace allows for three sources of data: the model file, a MAT binary file, or a MATLAB text file. It needs to be noted here that even though the model workspace is effective at handling private data, Simulink data objects need to be moved to the MATLAB workspace prior to code generation. Simulink also offers data management methods: `Simulink.saveVars` is used to save workspace variables to a MATLAB file and `Simulink.findVars` is used to discover which workspace variables are used by the model.

A configuration set is a named set of values for a model's parameters, including solver type and simulation start and stop times. By default, a configuration set resides in a single model. When differences occur between component configuration sets, the production code generation process may be interrupted with an error. The issue of dissimilar configuration sets can be easily managed by using a referenced configuration set, which is defined in the model, and a freestanding configuration set exists in the base MATLAB workspace. Freestanding configuration sets can be instantiated by creating a `Simulink.ConfigSet`. At a minimum use two freestanding configuration sets to incorporate single and multi-instance model reference components. In most cases, these configuration settings are similar except for the property entitled: "Total number of instances allowed per top model."

Modeling style guidelines provide a foundation for project success and teamwork with projects that are conducted both in-house and with partners and subcontractors [9]. It is recommended that project or enterprise-wide modeling style guidelines be established and enforced. Establishing a custom environment for Simulink user interfaces and connection to external tools is also desirable. The Simulink `sl_customization.m` file can register and apply a variety of customizations, including custom model checks.

The system target file (STF) provides control over the code generation stage of the build process and the presentation of the target to the end user. It provides:

- Definitions of variables that are fundamental to the build process, such as code format to be generated
- The main entry point to the top-level TLC program that generates code
- Target information for display in the System Target File Browser
- A mechanism for defining target-specific code generation options
- A mechanism for inheriting options from another target

At a minimum the target-specific code generation options in the configuration set must be locked down to ensure a consistent build process. This can be easily accomplished using a custom STF that inherits properties from the Embedded Real-Time (ERT) target of Real-Time Workshop Embedded Coder.

SELECTING A MODELING LANGUAGE

A high-quality software design minimizes the time required to create, modify, test, and maintain the software while achieving acceptable run-time performance. Every design decision must be made in the context of the entire system and should be expressed in precise design language. The MATLAB product family provides a rich set of modeling choices – Simulink, Stateflow, and Embedded MATLAB – to design, simulate and generate production code for multi-domain systems. Each of these modeling choices is appropriate for expressing certain aspects of the design, and an improper choice, while fully functional, may lead to a low-quality design. Although a subpar design might not expose design integrity problems in small-scale system, it will result in significant burden in a large-scale system, requiring undesirable redesign in the later stages of a project.

Recommendation 12.

Consider using:
- Simulink for signal flow and feedback control algorithms
- Stateflow for combinatorial logic, schedulers, and finite-state machines
- Embedded MATLAB for matrix and single line equations.

Simulink is ideal for designing signal flow centric control algorithms while Stateflow is ideal for combinatorial logic, schedulers and state machines. While you could use only one modeling choice to express the entire design, it may result in an unintuitive design that has unacceptable run-time performance and is difficult to test and maintain. For example, Simulink has blocks called “For Iteration Subsystem” and “While Iterator Subsystem” for implementing simple for loops and while loops, respectively. If you have more complicated loops, such as nested loops or ones in which the loop index is modified, it may be easier to implement using Stateflow.

Embedded MATLAB is useful when you are deriving your algorithms from existing MATLAB code or when developing algorithms using matrix mathematics. Some advanced computations are more naturally expressed as

equations or in a procedural language, rather than a block diagram. You could implement equations in Simulink using several multiply and divide blocks, but it may be better to simply type a single-line equation, resulting in improved readability and maintainability. Embedded MATLAB facilitates migrating algorithms from MATLAB code to Simulink. To improve testability, scalability, maintainability, and performance aspects of the design, however, we recommend that appropriate portions or all of the MATLAB code ultimately be migrated to Simulink and Stateflow. For example, for debugging purposes it is trivial to log an intermediate signal in Simulink by creating a test point. This approach doesn't require any restructuring of the design other than changing the signal properties.

ALGORITHM EXPORT AND FULL REAL-TIME TARGET

Traditional activities in Model-Based Design that incorporate large-scale modeling for production code generation follow either an algorithm export or turnkey build for the full real-time executable. There are many considerations prior to the selection as well as during the setup of the production code generation (PCG) environment including a legacy software base, code reuse with different target processors, and the need for a turnkey build.

Recommendation 13.

Determine the PCG deployment strategy along with architecture design.

Figure 5 demonstrates an algorithm export approach that uses production code generation. With this approach the developer generates code for the controller model, which is highlighted. The generated C code will need to be integrated with a real-time operating system (RTOS) and with other handwritten software components, such as the input and output device drivers and a downstream embedded development environment. The compile and linking processes are handled via an external make environment or integrated development environment. Automation to handle the compile and link process can be managed within the simulation environment if desired. This is the most common approach for production deployment especially when hand written drivers and schedulers already exist.

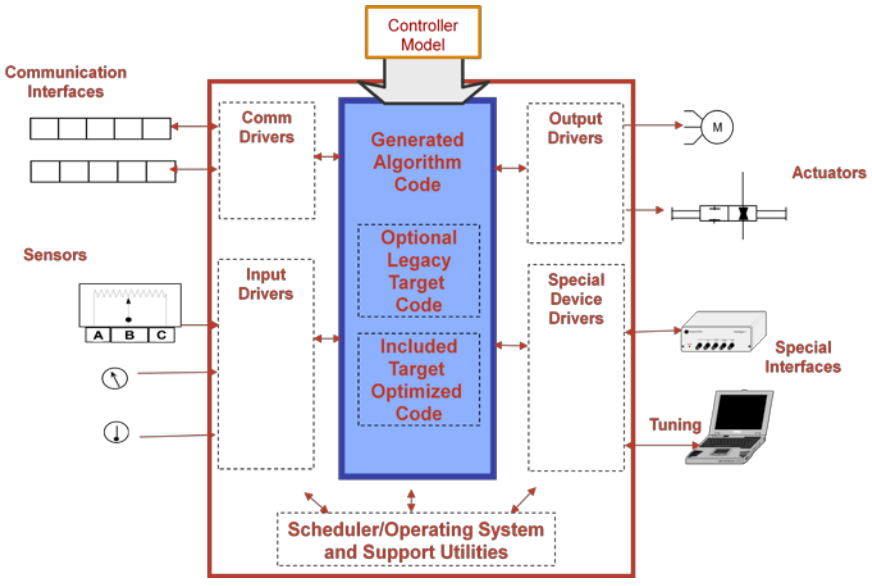


Figure 5. Algorithm export approach that uses production code generation.

Figure 6 demonstrates a turnkey build for a full real-time executable that uses production code generation. With a turnkey approach the developer will generate code for the entire controller model. Input/output drivers and RTOS interfaces can be generated as well but are not required if handwritten software exists. If a comprehensive target is created, the developer has complete control of their input/output driver configurations, build options, and linker settings all from within the simulation model.

Another consideration for a turnkey build is the effort to develop and maintain a full real-time executable PCG environment. Commercial-of-the-shelf solutions exist to eliminate this effort and should be considered first. Otherwise resources need to be allocated to support the development activity of a custom target support package, which includes peripheral simulation blocks, RTOS integration target language compiler (TLC) files, code profiling TLC files, the Build/Make components, and required verification support options, such as processor-in-the-loop.

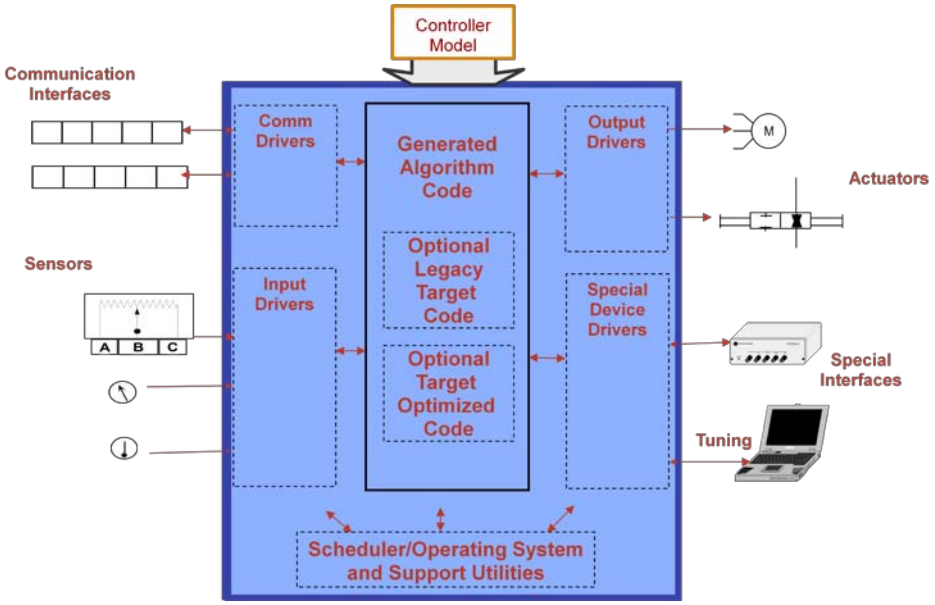


Figure 6. Full real-time executable that uses production code generation.

Figure 7 summarizes some typical decisions that are made when migrating into production code generation. Algorithm Export PCG is typically chosen when:

- Code reuse with multiple hardware platforms or software architectures
- Large legacy software base exists
- No need for a turnkey full real-time executable solution

Setting up the most appropriate Algorithm Export PCG environment for your situation requires some additional decisions. First, the decision needs to be made whether to use Export Functions or complete model build mechanism. With Export Functions, the subsystem model is architected with function calls and code is generated that produces no scheduler or step functions. This approach allows the user to model RTOS functionality without impacting the generated code, and atomic subsystems are then used to specify the software entry points. Next there is the decision regarding legacy code inclusion. If simulation capability of existing software is required then the Legacy Code Tool can be used to automate this process using MATLAB APIs. The APIs will automate the block creation through the use of an inline S-function/TLC pair as well as the

creation of a build dependency file (`rtwmakecfg.m`) used to specify the source code inclusion to the build process. Embedded processors and associated compilers often have specialized instructions to support certain operations that are used frequently in typical embedded applications. Such processor-specific instructions execute much faster than their C-equivalents and can improve code performance significantly. The target function library enables processor-specific code generation that takes advantage of processor-specific instructions. If the model size warrants, there is some minor setup to enable the parallel build feature of Real-Time Workshop Embedded Coder. The user can package the generated software for inclusion in their target build environment using the MATLAB instruction `packNGo`.

Setting up a full real-time executable PCG environment also requires some additional decision. First, the determination must be made whether commercial target support package and embedded IDE link products exist for the embedded target and compiler pair. This determination also includes reviewing the peripheral, linking, and memory map requirements against the commercial products. If these commercial products meet the program requirements for the embedded target, we recommend using the commercial products. If not, the user has to create a custom target support package, which includes the system target file, custom device driver blocks, a template make file, and any required build process file hook mechanisms [10]. The user also has to consider legacy code inclusion and intrinsic support through the use of Legacy Code Tool and target function library respectively. Finally, there is some minor setup to enable the parallel build feature within Real-Time Workshop Embedded Coder which entails defining the MATLAB path, dynamic Java path, and MATLAB workspace on the local workers.

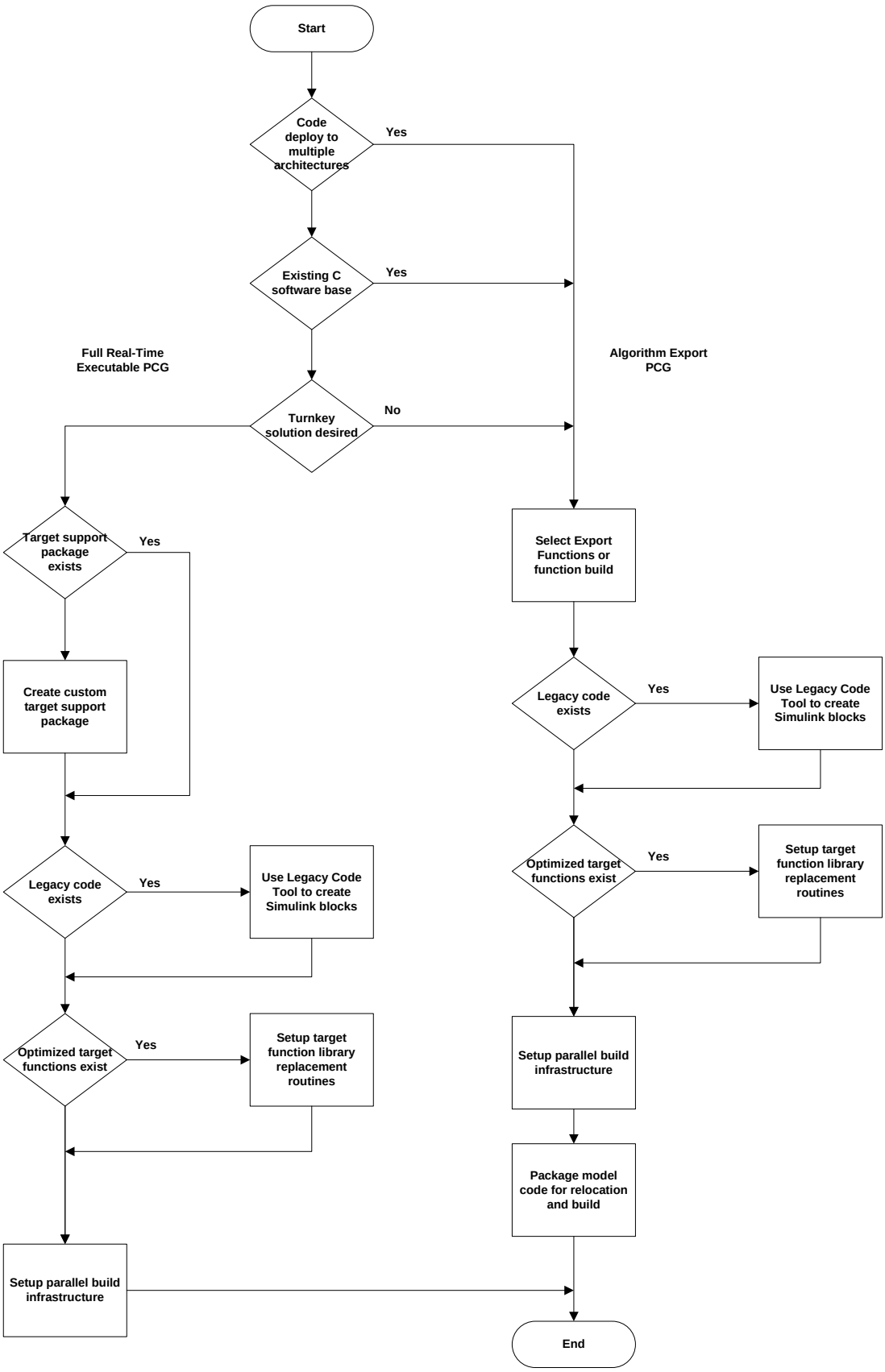


Figure 7 - Algorithm export/full real-time PCG decision chart.

SUMMARY/CONCLUSIONS

The breadth and the scope of projects applying Model-Based Design for development of embedded systems continues to increase rapidly resulting in models that are exceptionally large and complex. Due to the ability of engineers to communicate using models, create frequent and faster iterations of the design, and generate production code, Model-Based Design for large embedded systems creates both opportunities and challenges. This paper presents some proven recommendations based on accumulated industry experience with large-scale modeling. It is critical to address these challenges early in the design phase to eliminate an inefficient or suboptimal model design that does not effectively support production code generation.

REFERENCES

1. The MathWorks, “BAE Systems Achieves 80% Reduction in Software-Defined Radio Development Time with Model-Based Design,” http://www.mathworks.com, May 2006.
2. The MathWorks, “GM Standardizes on Model-Based Design for Hybrid Powertrain Development,” http://www.mathworks.com, May 2009.
3. Smith, Paul, Prabhu, Sameer, Friedman, Jonathan. “Best Practices for Establishing a Model-Based Design Culture,” 2007-01-0777, The MathWorks, Natick, MA, 2007.
4. Walker, Gavin, Friedman, Jonathan, Aberg, Rob. “Configuration Management of the Model-Based Design Process,” 2007-01-1775, The MathWorks, Natick, MA, 2007.
5. MATLAB Central, “Simulink Modeling: Buses Best Practices,” http://www.mathworks.com/matlabcentral/fileexchange/23480, March 2009.
6. MathWorks Technical Solutions, What are Data Store blocks best practices for modeling and code generation using Simulink and Real-Time Workshop Embedded Coder?, June 2009.
7. MATLAB Central, “Two Methods for Breaking Data Dependency Loops in System Level Models,” http://www.mathworks.com/matlabcentral/fileexchange/15368, Sept 2009.
8. MATLAB Central, “Fast Parameter Loading for MATLAB/Simulink,” http://www.mathworks.com/matlabcentral/fileexchange/14898, May 2007.
9. The MathWorks, “Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow Version 2.1,” http://www.mathworks.com/automotive/standards/maab.html, July 2007.
10. The MathWorks, “Real-Time Workshop Embedded Coder 5 - Developing Embedded Targets,” http://www.mathworks.com, September 2000.
11. Eric Dillaber, Larry Kendrick, Wensi Jin and Vinod Reddy, The MathWorks, Inc. "Pragmatic Strategies for Adopting Model-Based Design for Embedded Applications," SAE Paper 2010-01-0935.

CONTACT INFORMATION

Kerry Grand
Kerry.Grand@mathworks.com

Vinod Reddy
Vinod.Reddy@mathworks.com

Gen Sasaki
Gen.Sasaki@mathworks.com

Eric Dillaber
Eric.Dillaber@mathworks.com