08AE-22

Automatic Code Generation – Technology Adoption Lessons Learned from Commercial Vehicle Case Studies

Tom Erkkinen
The MathWorks, Inc.

Scott Breiner
John Deere

Copyright © 2007 The MathWorks, Inc

ABSTRACT

Using Model-Based Design, engineers model complex systems and simulate them on their desktop environment for analysis and design purposes. Model-Based Design supports a wide variety of C/C++ code generation applications that include stand-alone simulation, rapid control prototyping, hardware-in-the-loop testing, and production or embedded code deployment.

Many of these code generation scenarios impose different requirements on the generated code. Stand-alone simulations usually need to run fast, for parameter sweep or Monte Carlo studies, but do not need to execute in true hard real-time. Hardware-in-the-loop tests by definition use engine control unit (ECU) component hardware that requires a hard real-time execution environment to protect the physical devices. Code generated for production ECUs must satisfy hard real-time, efficiency, legacy code, and other requirements involving verification and validation efforts.

With Model-Based Design, the functional behavior of the model needs to match that of the generated code. As a result the transformation of models into generated code must include necessary deployment and real-time artifacts to ensure that the code executes properly in the final software and hardware environments.

For example, in a typical commercial vehicle use case, a diesel engine control algorithm and engine plant model are simulated together as a hybrid system. The plant model is input into the code generator for deployment in a hard real-time HIL lab. Code generation for the engine control algorithm is often done in two, or even three, phases. First, the code is generated for real-time rapid control prototyping for algorithm assessment and refinement. Next, the code may be generated for execution on the actual embedded microprocessor during on-target rapid prototyping for algorithm assessment on the ECU hardware. Finally, the code is generated for production ECUs and several verification steps are employed, including software-in-the-loop (SIL), processor-in-the-loop (PIL), and finally hardware-in-the-loop (HIL) testing.

Organizations moving from traditional waterfall processes that involve paper documents and hand code to Model-Based Design face challenges familiar to those who have followed other technology migrations, such as drafting tables to CAD systems or Assembly language to C code. These challenges center on how to:

- best leverage the technology
- reuse existing process
- pace the transition
- develop necessary skills sets and training

This paper describes case studies on how John Deere adopted Model-Based Design for commercial vehicle development and discusses the benefits and lessons learned.

INTRODUCTION TO MODEL-BASED DESIGN

A model represents a dynamic system whose response at any time is a mathematical function based on its inputs, current state, and current time. Historically, system engineers have used block diagrams as shown in Figure 1 to create models and design algorithms within numerous engineering areas such as Feedback Control and Signal Processing. In recent years, graphical modeling environments consisting of block diagrams and state machines have been used to analyze, simulate, prototype, specify, and deploy software algorithms within a variety embedded systems and applications. Model-Based Design refers to the use of models and modeling environments as the basis for embedded system development.

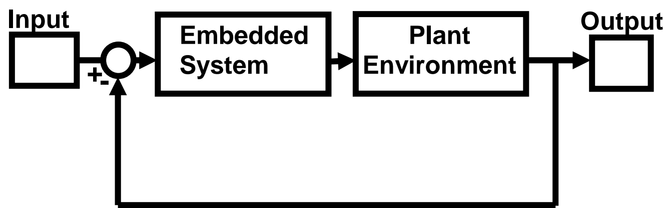

Figure 1: Feedback controller model.

Examples of systems developed using Model-Based Design include:

- Aircraft avionic systems
- Commercial vehicle electronics
- Power plant regulators
- Digital motor controllers
- Medical devices
- Audio and video signal processors

Model-Based Design is used throughout the system development life cycle because it focuses on producing design flows that lead toward continuous verification and validation of requirements, designs, and implementations. This approach is important for formal software processes as well as risk management of any project that finds solace in error prevention and early error detection.

The main activities that occur during Model-Based Design include:

- Modeling
- Simulation
- Rapid prototyping
- Embedded deployment
- In-the-loop testing
- Integral activities

MODELING – A block diagram model of a dynamic system is represented schematically as a collection of blocks interconnected by lines that represent signals. The signals are the inputs, outputs, and states of the blocks processed.

Blocks and lines can be real or virtual. Virtual blocks or lines have no effect on the simulation results but aid in constructing or understanding diagrams. Blocks and subsystems, whether they are real or virtual, can be stored in custom libraries to facilitate reuse and abstraction.

Models can be classified in many ways. One method involves whether the model contains continuous or discrete dynamics. A continuous-time system is a system in which the evolutions of the system results are continuously changing. Continuous system models are used to represent analog signals or real-world effects where time continues without interruption.

A discrete-time system is one in which the evolution of the system results are tracked at finite intervals of time. One example of a discrete-time system is an embedded microprocessor because it relies on clocks or interrupts to begin executing the software. A typical system model is hybrid and contains both continuous-time and discrete-time dynamics. Simulation and code generation are integral parts of Model-Based Design. It is important that hybrid models can be simulated and used for code generation.

SIMULATION - Integration is one of the key aspects of dynamic system simulation. During simulation, continuous-time signals are changed using numerical integration solvers. There are many solvers within Model-Based Design environments. They are classified into fixed-step and variable-step solvers.

As the name implies, fixed-step solvers use explicit methods to compute the next continuous state at fixed periodic intervals of time. A variable-step system uses explicit or implicit methods to compute the next continuous state at non-periodic intervals of time. A sample time also needs to be selected. For fixed-step solvers the sample time is the fixed step time. For variable-step solvers, the sample time is the maximum allowable sample time. Ultimately, the goal during simulation is to choose a sample time and integration method that will provide an accurate approximation of the continuous system's behavior but run reasonably fast. However, variable step solvers and continuous-time systems don't lend themselves to deterministic real-time executables, so this combination should be used sparingly on portions of the model that are targeted for embedded code generation.

Discrete systems, on the other hand, have their states explicitly updated and are well suited for code generation. They execute at the appropriate sample time, or interrupt, and generate outputs. If a system has only one sample time, it is single rate. If the system has multiple sample times, it is multirate. Multirate systems can be evaluated (executed) using either a single-tasking form of execution or a multitasking form. When multitasking execution is used, it often conforms to rate monotonic scheduling principals.

Finally, simulation can be accomplished in two ways. One way is to use an in-memory representation of the systems and execute the simulation in an interpretive mode. The other way is to generate code from the model and execute the code using a technique of simulation through code generation. Interpretive simulation provides users with more control of the execution environment and interaction capabilities, but it can be slow for large models. Simulation through code generation provides less user interaction but more speed. For this reason, it is also known as simulation acceleration.

RAPID PROTOTYPING - In bypass rapid prototyping, code is generated from the controller or algorithm model. The code is then cross-compiled and downloaded to a high-speed, often floating-point, rapid-prototyping computer where it executes in real time. I/O is typically managed by memory pod or emulation device that is connected to both the rapid prototyping computer and an existing ECU, perhaps still residing in a vehicle. Other I/O options include communication via buses, such as CAN, or other I/O devices, which may require some custom signal processing and power electronics. The controller parameters are tweaked "on-the-fly" during test drives or in the lab involving the actual plant (e.g., engine) and allowing for the insertion of new code to bypass existing ECU code. Success is declared when performance requirements are met, proving that the new algorithm is *feasible*. See Figure 2.

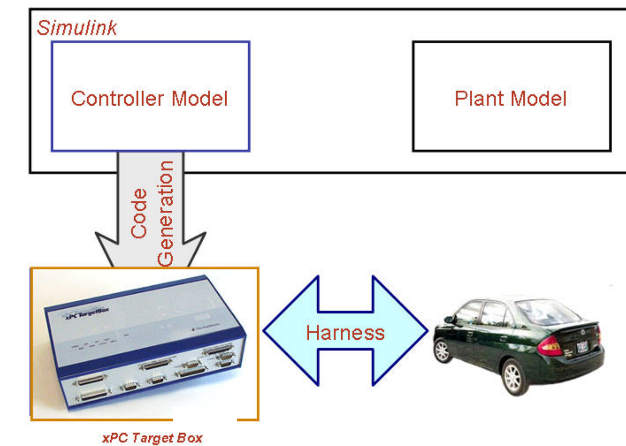

Figure 2: Bypass rapid prototyping.

In on-target rapid prototyping, as with bypass rapid prototyping, code is generated just for the controller portion of the model. However, the code is then cross-compiled and downloaded to the embedded microprocessor or ECU used in production, or perhaps to a close approximation of it configured with a little more memory and I/O. On-target rapid prototyping often uses an integer processor and thus needs a more detailed, fixed-point model, as opposed to the floating-point processors and models used for bypass rapid prototyping. I/O is managed via standard ECU devices.

The host computer then interfaces directly with the on-target rapid-prototyping ECU hardware, perhaps residing in fleet vehicles. Controller parameters are tweaked "on-the-fly." Success is declared when performance requirements are met, proving that the new algorithm is both feasible and *practical*; it will work in a production, resource-constrained, environment. See Figure 3.

EMBEDDED DEPLOYMENT - After rapid prototyping, a detailed software design activity is often undertaken to convert the controller model to a detailed, executable software specification.

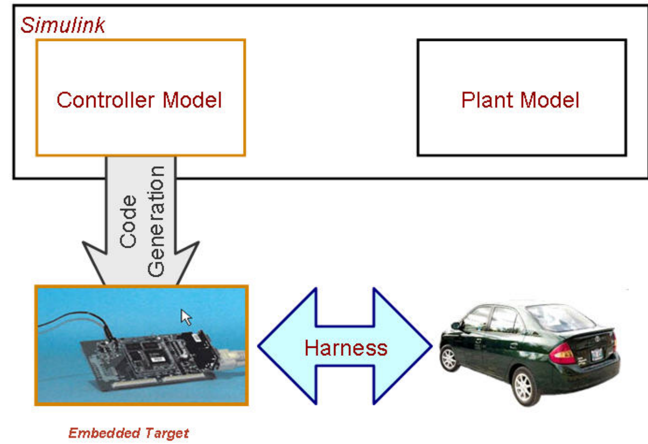

Figure 3: On-target rapid prototyping.

There are a number of topics involved here, such as fixed-point data types, function and file partition, defensive programming measures, startup and shutdown procedures, and diagnostics and built-in test routines. The model is in effect constrained and elaborated to perform properly on embedded system hardware.

Embedded code is then generated for the detailed controller model and downloaded to the actual embedded microprocessor or ECU as part of the production software build. No simulation activity is associated with this step. The key here is to ensure that the final build has fully integrated the automatically generated code with existing legacy code, I/O drivers, and real-time operating system (RTOS) software.

There are two approaches to code-generation embedded deployment. The first approach is to generate code for the functions and then integrate into the overall hand written application. A second, emerging approach is to use the model to generate the entire application. The first approach is more commonly used today. See Figure 4.

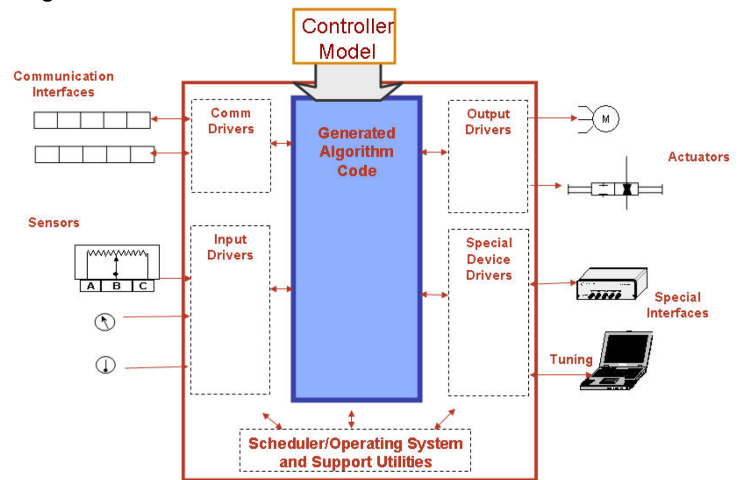

Figure 4: Embedded code deployment using function generation.

IN-THE-LOOP TESTING -Simulation of models is one of the first verification and validation (V&V) steps. Testing models via simulation requires a more rigorous approach than the ad-hoc simulation runs that are often used in early algorithm development. Model testing involves a formal approach to the creation and execution of test cases. Special blocks, such as signal builders and assertions, facilitate this type of formal test procedure. New tools are emerging to assist with model V&V based on V&V techniques applied on the code, such as structural coverage analysis and test generation.

Software-in-the-loop (SIL) testing involves executing the production code for the controller within the modeling environment for non-real-time execution with the plant model and interaction with the user. The code executes on the same host platform that is used by the modeling environment. A code wrapper of the generated code provides the interface between the simulation and the generated code. See Figure 5.

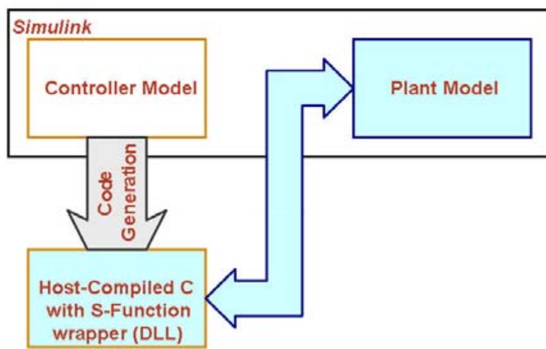

Figure 5: Software-in-the-loop testing.

Processor-in-the-loop (PIL) testing is similar to SIL in that it too executes the production code for the controller. However the code executes on the actual embedded processor or an instruction set simulator, so that this verifies the code behavior on the actual target. Real I/O via CAN or serial devices are used to pass data between the production code executing on the processor and a plant model executing in the modeling environment. As with SIL, PIL testing is a non-real-time execution scenario as shown in Figures 6-7.

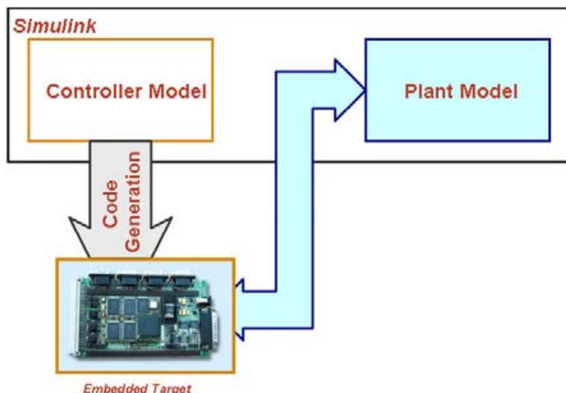

Figure 6: Processor-in-the-loop testing using direct hardware connection.

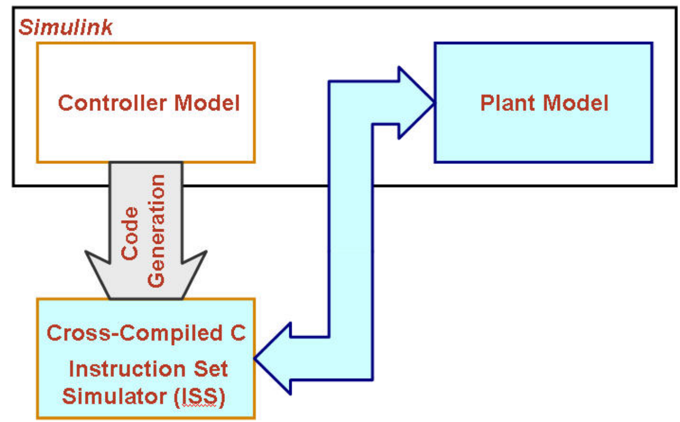

Figure 7: Processor-in-the-loop testing using Instruction Set Simulator.

For hardware-in-the-loop (HIL) testing, the code is generated just 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 ECU inputs (sensors) and receive the ECU outputs (actuator commands). Whereas rapid prototyping is often a development or design activity, HIL serves as a final lab test phase before final system integration and field tests commence. See Figure 8.

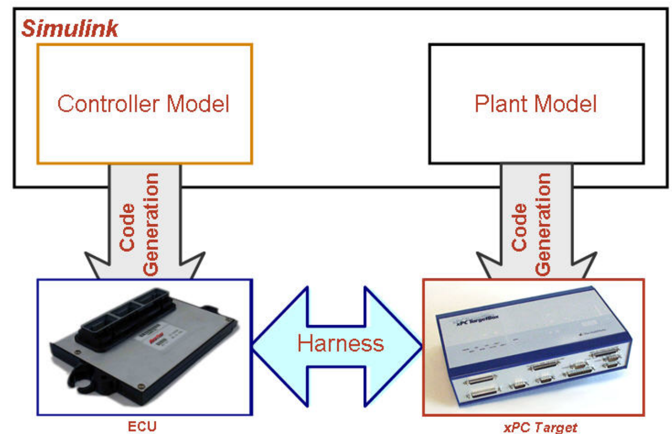

Figure 8: Hardware-in-the-loop testing.

INTEGRAL ACTIVITIES - Model-Based Design environments automate the generation of documentation from models. In one case, documentation is done in template form letting users specify the content of each documentation section. Requirements traceability is accomplished using interfaces between blocks in the model and existing requirement management sources. The code generated from the model can also be traced back to the block, letting auditors trace high-level requirements all the way to the code. As with requirements management, source control for a model is accomplished outside the modeling environment using existing source control products. Interfaces are provided that let developers check in and check out models as well as document the changes.

CASE STUDY – JOHN DEERE CONSTRUCTION AND FORESTRY

The Construction and Forestry Division of John Deere is employing Model-Based Design on several 2008 production programs, including a crawler tractor program. A significant aspect of these projects is that 100% of the application code is automatically being generated using Real-Time Workshop® Embedded Coder. Real-Time Workshop Embedded Coder generates the entire algorithm portion of the application (85-90%) and automatically includes the required interfaces to the rest of the operating system code. Unlike function-based code generation in Figure 4, application-based code generation outputs the complete executable including John Deere Operating System (JDOS) services and fault codes, as shown in Figure 9.

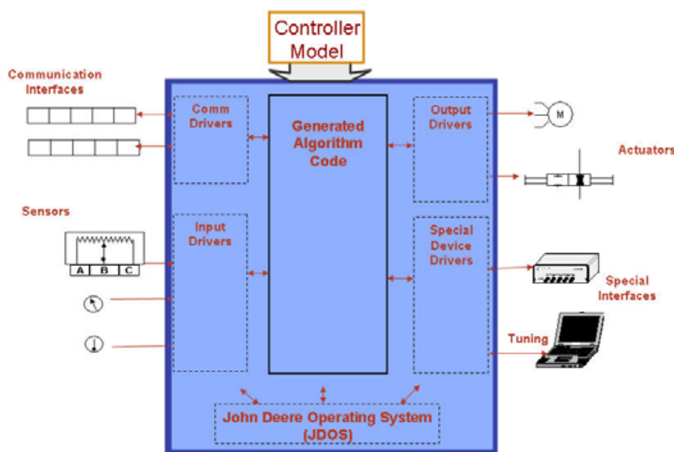

Figure 9: Full embedded application deployment using JDOS.

For the crawler tractor, John Deere used Model-Based Design to implement advanced functions that synchronize the left and right crawler speeds for even tracking and include added functions that will smooth the starting and stopping motion.

By developing an application-based automatic code generation environment, John Deere was able to reduce the amount of hand integration required for auto-code generation and remove unnecessary wrapper code. As part of this effort, they were also able to keep the automatically generated code better synchronized with the production software while reducing maintenance issues. Finally, John Deere was able to develop a single model for the complete application that was used for both on-target rapid prototyping and production code deployment as described earlier in this paper.

A basic JDOS Library was created in Simulink® to read and write variables, plus handle fault codes. It uses a VAR Manager that was constructed with help from The MathWorks™ consulting services. The VAR Manager gives access to all existing I/O variables. These variables are read into the model from a header file and are available to the model developer via the ReadVar, WriteVar Mask. If a new variable is required in the model, the user can add the variable to the list via the VAR Manager interface. A header file is automatically recreated during the code generation process that keeps the model algorithm code synchronized with the operating system and I/O code.

Figure 10 shows an example model that uses several JDOS Library blocks and shows the VAR Manager interfaces. Code generation and the header file creation are done via the custom menu selection in Figure 11.

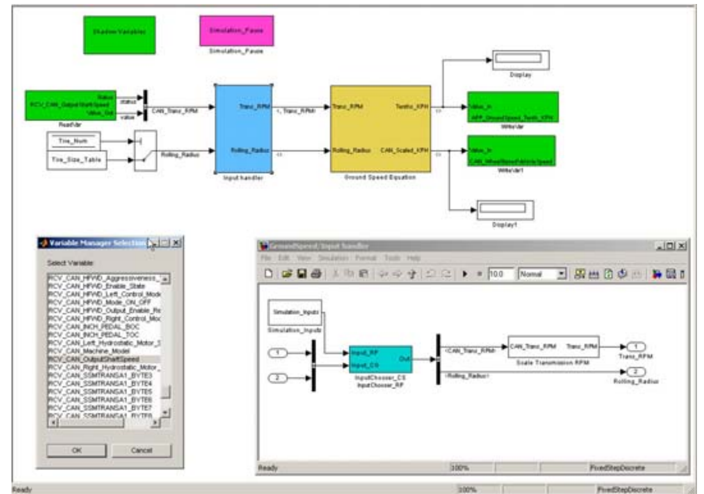

Figure 10: Top-level model showing example JDOS library blocks.

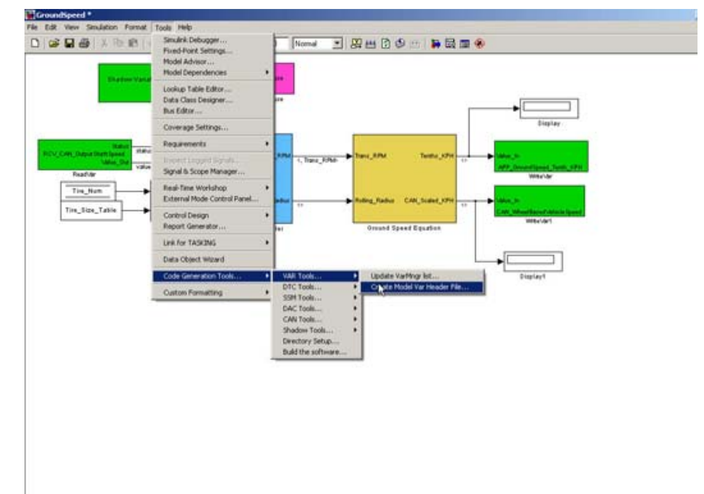

Figure 11: Custom menu for generating production code and custom headers.

The top-level model in Figure 10 shows the basic model architecture of reading in measurements and variables using CAN, doing input signal handling, performing the algorithm calculation, and writing outputs and commands using CAN.

For this model, the Input Handler subsystem scales the Shaft Speed signal appropriately for use later in the

Ground Speed Equation subsystem. Prior to scaling, the Input Handler uses a custom block (Input Chooser), which was built by John Deere to determine whether the application or environment is a simulator or embedded system. If the application is for simulation or rapid prototyping, then specially prepared simulation inputs are accessed and used to execute and test the algorithm (Input_Chooser_RP). If the application is for code generated and running on the embedded system, then the actual CAN messages are accessed and processed (Input_Choser_CG). By using an input selector technique, John Deere is able to use a single model for multiple purposes: simulation, rapid prototyping, and production.

The custom tool menu in Figure 11 shows a variety of tools and applications in addition to the custom header file creation. Creating custom menus and dialogs in Simulink is straightforward and requires a customization file (sl_customization.m) that is placed on the MATLAB® path. At the bottom of the menu is a "Build the software …" menu option.

By selecting this option, the full production build process begins as follows:

1. Simulink Model Advisor is invoked to check if model is suitable for production.
2. C Code is generated for the algorithm with JDOS interfaces and header files.
3. JDOS code is linked in.
4. The complete executable is cross-compiled and built.

By using this master build technique; John Deere can use the model as the primary maintenance mechanism for their entire production software process. The code builds the same way every time by generating the same number of files with the same interfaces (e.g., *modelname*_step, *modelname*_initialize, …).

One of the main benefits to using Model-Based Design is that it lets John Deere reduce the need for expensive rapid-prototyping hardware ($30,000 vs. $200). For example, John Deere's rapid prototyping can mainly be done on a simple ST10 development board as opposed to using a powerful real-time system simulator. Prototyping directly on the target eliminates the need for creating custom bypass controllers, which not only add delays to the program, but also add delays (latencies) to the I/O and data processing, making it difficult to estimate the algorithm's true target performance.

By using application-focused Model-Based Design, John Deere now ensures that the early algorithm designs are quickly tested in a production hardware environment, which imposes CPU and memory limitations on the model designer. This testing promotes good modeling practices and helps ensure that algorithms can make it into the field.

The Model-Based Design focus at John Deere currently also includes:

- Improving functionality of the JDOS library
- Establishing modeling design guidelines (using MAAB as the starting point)
- Automatically configuring the automatic code generation environment with existing code and I/O drivers
- Integrating CCP protocol into JDOS, using John Deere's own CCP
- Utilizing more model validation and verification tools

CONCLUSION

Automatic code generation with Model-Based Design is an important technology that offers embedded system developers a number of advanced options for prototyping, deploying, and verifying production software. It is important to understand the potential applications of code generation. But technology alone is not going to improve production processes. Embedded system developers must also establish a production workflow that leverages code generation technologies and yet fits within well-established software engineer principals, such as reducing complexity and establishing proper configuration management and version control.

In the case study, John Deere focused their development process on using single models for multiple purposes and controlling the entire production build process with a single source (the model). Thus, John Deere leverages Model-Based Design with automatic code generation but does so in way that greatly facilitates maintenance and configuration management for their production workflows.

John Deere is currently using on-target rapid prototyping and production code generation for several production-intent programs. Using on-target rapid prototyping instead of traditional rapid prototyping on real-time computers has saved time and reduced costs. Reusing the models for production deployment on embedded ECUs provides additional returns on their Model-Based Design investments.

MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, SimBiology, SimHydraulics, SimEvents, and xPC TargetBox are registered trademarks and The MathWorks, the L-shaped membrane logo, Embedded MATLAB, and PolySpace are trademarks of The MathWorks, Inc. Other product or brand names are trademarks or registered trademarks of their respective holders.