WHITE PAPER

Model-Based Design for Automotive Control Systems

MathWorks®

Imagine that your team is developing an advanced emergency braking system (AEBS) for a vehicle. When a collision is imminent, the AEBS warns the driver with an audio alarm. If the driver does not respond, it applies a warning brake. When the driver brakes, but with insufficient force to avoid a collision, the system calculates and then applies the required extra braking force. Before you begin the design, you want to address some key questions—for example:

- How do we size the brakes?
- What if the requirements change?
- How can we optimize the design to ensure the desired performance?
- How can we test the design thoroughly while minimizing risk?

Whether you're developing controls for an industrial robot, a wind turbine, a production machine, an autonomous vehicle, an excavator, or an electric servo drive, if your team is manually writing code and using document-based requirements capture, the only way to answer these questions will be through trial and error or testing on a physical prototype. And if a single requirement changes, the entire system will have to be recoded and rebuilt, delaying the project by days, or even weeks.

Using Model-Based Design with MATLAB® and Simulink®, instead of handwritten code and documents, you create a system model—in the case of the emergency braking system, a model comprising the brakes, vehicle dynamics, environment, and controls. You can simulate the model at any point to get an instant view of system behavior and to test out multiple what-if scenarios without risk, without delay, and without reliance on costly hardware.

This white paper introduces Model-Based Design and provides tips and best practices for getting started. Using real-world examples, it shows how teams across industries have adopted Model-Based Design to reduce development time, minimize component integration issues, and deliver higher-quality products.

What Is Model-Based Design?

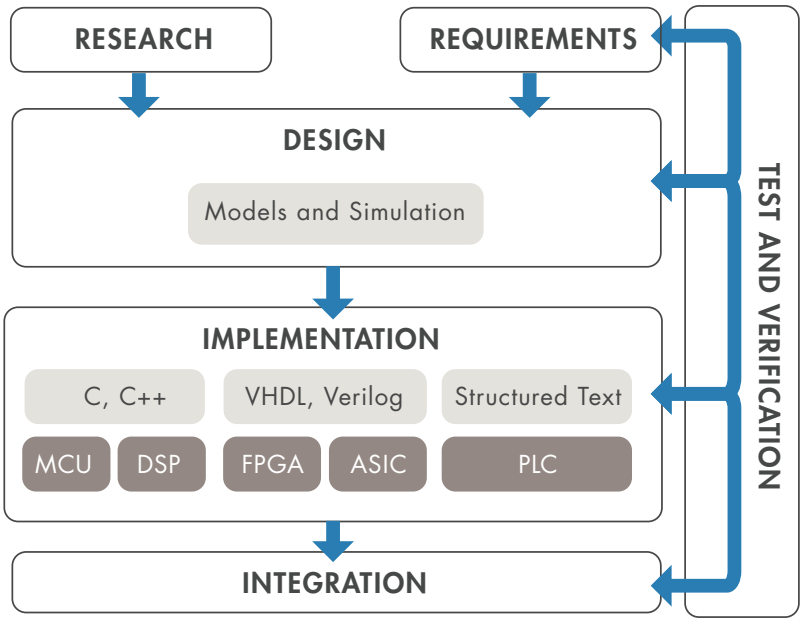
The best way to understand Model-Based Design is to see it in action:

> A team of automotive engineers sets out to build an engine control unit (ECU) for a passenger vehicle. Because they are using Model-Based Design, they begin by building an architecture model from the system requirements; in this case, the system is a four-cylinder engine. A simulation/design model is then derived. This high-level, low-fidelity model includes portions of the controls software that will be running in the ECU, and the plant—in this case, the engine and the operating environment.
>
> The team performs initial system and integration tests by simulating this high-level model under various scenarios to verify that the system is represented correctly and that it properly responds to input signals.
>
> They add detail to the model, continuously testing and verifying the system-level behavior against specifications. If the system is large and complex, the engineers can develop and test individual components independently but still test them frequently in a full system simulation.

Ultimately, they build a detailed model of the system and the environment in which it operates. This model captures the accumulated knowledge about the system (the IP). The engineers generate code automatically from the model of the control algorithms for software testing and verification. Following hardware-in-the-loop tests, they download the generated code onto production hardware for testing in an actual vehicle.

As this scenario shows, Model-Based Design uses the same elements as traditional development workflows, but with two key differences:

- Many of the time-consuming or error-prone steps in the workflow—for example, code generation—are automated.
- A system model is at the heart of development, from requirements capture through design, implementation, and testing.



Workflow for Model-Based Design.

Requirements Capture and Management

In a traditional workflow, where requirements are captured in documents, handoff can lead to errors and delay. Often, the engineers creating the design documents or requirements are different from those who design the system. Requirements may be "thrown over a wall," meaning there's no clear or consistent communication between the two teams.

In Model-Based Design, you author, analyze, and manage requirements within your Simulink model. You can create rich text requirements with custom attributes and link them to designs, code, and tests. Requirements can also be imported and synchronized from external sources such as requirements management tools. When a requirement linked to the design changes, you receive automatic notification. As a result, you can identify the part of the design or test directly affected by the change and take appropriate action to address it. You can define, analyze, and specify architectures and compositions for systems and software components.

Case Study: *Iveco*

An Iveco heavy-duty vehicle.

> "Our system engineers work directly with our software engineers on the Simulink model. This speeds development because there is no misinterpretation of requirements. When we're confident that the model is right, we save even more time by generating code from it, with no implementation errors."
>
> *— Demetrio Cortese, Iveco*

To capitalize on a market opportunity in Latin America for a range of medium- to heavy-duty vehicles, Iveco had to design, implement, test, and deliver a shift range inhibitor system for vehicles with 9- and 16-speed transmissions in about six weeks. The aggressive deadline required a compressed software development schedule that left no room for specification or implementation errors.

Due to the tight time constraints on the project, the team planned to use a preexisting hardware configuration that included a PLC. The software engineers, however, had no experience in writing structured text for PLCs. To avoid implementation errors and increased development time, Iveco needed to generate structured text automatically.

Iveco's traditional approach, in which system engineers define requirements and specifications that are handed off to software engineers, was not feasible given the project's short timeline. Working together, system engineers and software engineers developed a preliminary model of the system in Simulink.

Software engineers refined and customized the model, adding constraints, data types, built-in tests, and diagnostics. They simulated the model to verify the design's integrity and to identify overflow conditions, unexercised blocks, and other potential issues.

The engineers conducted real-time lab tests using the PLC and the actual transmission, quickly adjusting the model, regenerating the code, and rerunning tests until the management system met its functional and performance requirements.

Design

In a traditional approach, every design idea must be coded and tested on a physical prototype. As a result, only a limited number of design ideas and scenarios can be explored because each test iteration adds to the project development time and cost.

In Model-Based Design, the number of ideas that can be explored is virtually limitless. Requirements, system components, IP, and test scenarios are all captured in your model, and because the model can

be simulated, you can investigate design problems and questions long before building expensive hardware. You can quickly evaluate multiple design ideas, explore tradeoffs, and see how each design change affects the system.

Case Study: *Ather Energy*

The Ather 450 intelligent electric scooter.

> "We had lots of promising ideas, but as a small startup, we did not have the time, money, or people to build prototypes to test each one. With Model-Based Design, we identified and validated the best ideas through simulation, making it possible to deliver a more full-featured scooter in less time."
>
> *— Shivaram N.V., Ather Energy*

Most of Bangalore's more than 5 million two-wheeled scooters—about 70% of all vehicles in the city—are petrol-powered, resulting in high levels of noise pollution and CO_2 emissions. To meet the demand for cleaner alternatives, startup company Ather Energy has built India's first intelligent electric scooter. Capable of accelerating from 0 to 40 km/h in less than 4 seconds, the Ather 450 has a top speed of 80 km/h and a range of up to 75 km on a single charge.

Because the 450 would be one of the first of its kind in the market, the team faced numerous unknowns. After building a plant model comprising the scooter and its main components, they ran simulations to evaluate riding and usage scenarios—for example, operating the 450 on an incline, with multiple riders, in extreme temperatures, and with a mostly depleted battery.

They used the simulation results to make informed design tradeoffs; the results showed, for example, that increasing battery capacity would provide better range, but it would also increase cost and size, as well as alter the scooter's center of gravity. They refined the design until they had identified a motor and battery configuration that met the target acceleration and range requirements while satisfying cost, size, and temperature constraints.

In addition to designing the scooter itself, the engineers developed embedded control algorithms for battery charging, temperature management, and other key functions. In the absence of detailed component data, the team took an empirical approach to modeling the battery cells. They tested the battery at various temperatures and state-of-charge levels and used the measured input-output data to create a black-box model of the cell's electrical and thermal characteristics.

Next, they developed algorithms for battery charging, power control, and temperature control. They ran closed-loop simulations with the plant model to validate their control design. They generated code from the controller model and deployed it either to an ARM® Cortex® processor on the scooter or to a TI C2000™ microcontroller in the charging stations.

The Ather 450 is now in production, with an initial release in Bengaluru as well as 31 charging stations and 7 stations in Chennai.

Code Generation

In a traditional workflow, embedded code must be handwritten from system models or from scratch. Software engineers write control algorithms based on specifications written by control systems engineers. Each step in this process—writing the specification, manually coding the algorithms, and debugging the handwritten code—can be both time-consuming and error-prone.

With Model-Based Design, instead of writing thousands of lines of code by hand, you generate code directly from your model, and the model acts as a bridge between the software engineers and the control systems engineers. The generated code can be used for rapid prototyping or production.

Rapid prototyping provides a fast and inexpensive way to test algorithms on hardware in real time and perform design iterations in minutes rather than weeks. You can use prototype hardware or your production ECU. With the same rapid prototyping hardware and design models, you can conduct hardware-in-the-loop testing and other test and verification activities to validate hardware and software designs before production.

Production code generation converts your model into the actual code that will be implemented on the production embedded system. The generated code can be optimized for specific processor architectures and integrated with handwritten legacy code.

Case Study: *Ponsse*

Ponsse's Scorpion wood harvester.

> "The ability to generate error-free code from our Simulink models made it possible for our control engineers to focus on algorithm design and our software engineers to focus on programming the firmware layer. The result was faster development, better quality, and lower costs."
>
> — *Juha Inberg, Ponsse*

The Ponsse Scorpion is an eight-wheeled wood harvester designed to work in rugged forest terrain. The machine's distinctive frame consists of three segments linked by joints that rotate by as much as 12%. This frame enables the cabin on the center segment to remain level while the wheeled front and rear segments adjust to changes in the terrain.

On previous projects, Ponsse control engineers developed and debugged algorithms in MATLAB, and software engineers translated the algorithms manually into C. As the complexity of the control algorithms increased from project to project, this approach became difficult to sustain. The risk of introducing human errors into the C code grew, and there was a lengthy interval between the initial design of the algorithm and its verification on hardware. Ponsse wanted to shorten this interval, minimize coding errors, and reduce overall development time.

The engineers developed a control model that processes input from accelerometers and gyroscopes and actuates hydraulic valves to keep the Scorpion's center frame level, and built a prototype controller to generate a real-time application from their model and deploy it to Speedgoat target computer hardware.

The team used this real-time prototype to conduct tests on the actual Scorpion hardware. Based on the results, they made minor changes to the control model before regenerating and retesting an updated prototype. They then generated C code from their model for the Scorpion's ECU.

They integrated the generated code with firmware and other low-level interface code for the ECU, and tested it, first in a third-party simulator and later on the actual Scorpion harvester.

Since their successful completion of the Scorpion project, Ponsse engineers have used Model-Based Design to develop embedded controllers for other harvesters in the Ponsse product line, reusing filters and model components from the Scorpion control design.

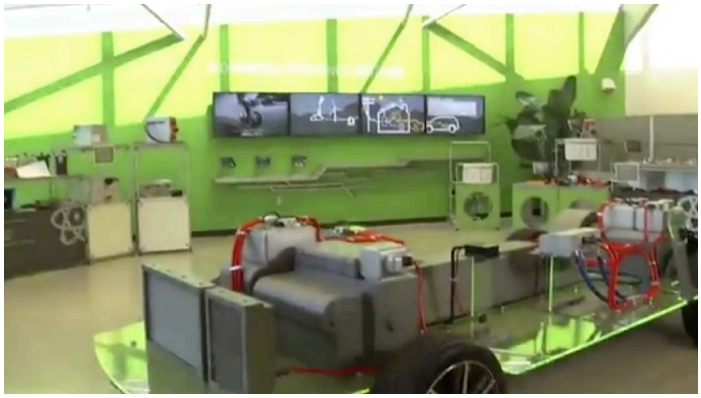
Test and Verification

In a traditional development workflow, test and verification typically occur late in the process, making it difficult to identify and correct errors introduced during the design and coding phases.

In Model-Based Design, test and verification occur throughout the development cycle, starting with modeling requirements and specifications and continuing through design, code generation, and

integration. You can author requirements in your model and trace them to the design, tests, and code. Formal methods help prove that your design meets requirements. You can produce reports and artifacts and certify your software to functional safety standards.

Case Study: *Lear*



Lear hardware-in-the-loop testing.

"For the BCM project, we used virtual integration and test with executable functional models in Simulink to identify more than 95% of requirements issues before implementation—compared with just 30% before we started using Model-Based Design."

— Jason Bauman, Lear

Automotive OEMs are pushing suppliers to deliver more functionality in ECU software. As vehicle electronics and electrical distribution systems become increasingly complex, requirements need to be clear, complete, and consistent. In traditional hand-coding workflows, ambiguous or conflicting requirements are often discovered late in the development process, leading to schedule or cost overruns. Engineers at Lear Corporation addressed these challenges by using Model-Based Design to develop, verify, and implement body control electronics systems.

Engineers analyzed customer requirements and partitioned the overall system into components such as lighting, battery management, and vehicle starting control. They then developed functional behavioral models and plant models for functional and unit testing. Engineers analyzed model coverage and continued refining test cases, designs, and requirements until they reached satisfactory model coverage levels, including decision coverage and modified condition/decision coverage (MC/DC).

After verifying nearly 400 unit models, the team generated C code and verified this code via software-in-the-loop (SIL) tests that reused the test cases generated for the unit model tests.

Lear engineers integrated the generated code for each unit model into 20–30 feature-level components, which were in turn integrated into a complete system model. The team met with the customer and ran simulations of the components and the complete model to resolve ambiguities in the original design specification.

The group used MATLAB scripts to automate the conversion of test cases into test vectors for hardware-in-the-loop (HIL) and vehicle-based testing. Overall, the team generated about 700,000 lines of code and reused test cases throughout the development cycle, reducing overall development time.

Getting Started

While your team might see the benefits of moving to Model-Based Design, they might also be concerned about the risks and challenges—organizational, logistical, and technical—that could be involved. This section addresses questions frequently asked by engineering teams considering adopting Model-Based Design and provides tips and best practices that have helped many of these teams manage the transition.

Q. How are engineering roles affected by the introduction of Model-Based Design?

A. Model-Based Design does not replace engineering expertise in control design and software architecture. With Model-Based Design, control engineers' roles expand from providing paper requirements to providing executable requirements in the form of models and code. Software engineers spend less time coding application software and more time on modeling architecture; coding OS, device driver, and other platform software; and performing system integration. Both control and software engineers influence the system-level design from the earliest stages of the development process.

Q. What happens to our existing code?

A. It can become part of the design; your system model can contain both intrinsically modeled and legacy components. This means that you can phase in legacy components while continuing to perform system simulation, verification, and code generation.

Q. Is there a recommended way to adopt Model-Based Design?

A. Trying new approaches and design tools always carries an element of risk. Successful teams have mitigated this risk by introducing Model-Based Design gradually, taking focused steps that help a project along without slowing it down. Organizations of all sizes begin their initial adoption of Model-Based Design at the small group level. They usually start with a single project that will provide a quick win and build on that early success. After gaining experience, they roll out Model-Based Design at the department level so that models become central to all the group's embedded systems development.

These **four best practices** have worked well for many teams:

- **Experiment with a small piece of the project.** A good way to start is to select a new area of the embedded system, build a model of the software behavior, and generate code from the model. A team member can make this small change with a minimal investment in learning new tools and techniques. You can use the results to demonstrate some key benefits of Model-Based Design:
  - High-quality code can be generated automatically.
  - The code matches the behavior of the model.
  - By simulating a model you can work out the bugs in the algorithms much more simply. and with greater insights than by testing C code on the desktop.
- **Build on your initial modeling success by adding system-level simulation.** As previous sections of this paper have shown, you can use system simulation to validate requirements, investigate design questions, and conduct early test and verification. The system model does

not need to be high-fidelity; it just needs to have enough detail to ensure that interfacing signals have the right units and are connected to the right channels, and that the dynamic behavior of the system is captured. The simulation results give you an early view of how the plant and controller will behave.

- **Use models to solve specific design problems.** Your team can gain targeted benefits even without developing full-scale models of the plant, environment, and algorithm. For example, suppose your team needs to select parameters for a solenoid used for actuation. They can develop a simple model that draws a conceptual "control volume" around the solenoid, including what drives it and what it acts upon. The team can test various extreme operating conditions and derive the basic parameters without having to derive the equations. This model can then be stored for use on a different design problem or in a future project.
- **Begin with the core elements of Model-Based Design.** The immediate benefits of Model-Based Design include the ability to create component and system models, use simulations to test and validate designs, and generate C code automatically for prototyping and testing. Later, you can consider advanced tools and practices and introduce modeling guidelines, automated compliance checking, requirements traceability, and software build automation.

Case Study: *Danfoss*

The Danfoss VLT® AutomationDrive FC302.

> "We completed our first solar inverter project with Model-Based Design on schedule, despite ramping up new engineers and adopting a new design process. For our second project, we actually reduced development time by 10–15%."
>
> — *Jens Godbersen, Danfoss*

To help meet increased demand for its products, the Danfoss power electronics group hired new engineers and re-evaluated its embedded software development processes, which up to then had relied on manual coding. With a traditional development process and manual coding, errors remained undetected until hardware prototype and certification testing.

While Danfoss knew they needed a new process, they were worried that adopting Model-Based Design would jeopardize deadlines. Bringing the team up to speed would take time. In addition, work on a new product, a solar inverter, had already begun. Model-Based Design would have to be introduced during development, and without affecting project deadlines.

Working with MathWorks consultants, Danfoss first developed a plan to ensure successful adoption of Model-Based Design. Danfoss engineers attended on-site training courses on Simulink, Stateflow®, and Embedded Coder® led by MathWorks engineers.

The team then completed a pilot project in which they rebuilt an existing software component that had been coded by hand. For the pilot, they decided to focus on three core capabilities of Model-Based Design: modeling, simulation, and code generation. After completing the pilot project, the team fully transitioned to Model-Based Design for development of the new solar inverter.

In weekly phone calls, MathWorks consultants advised them on the best way to get started, provided feedback on early versions of the models, and helped the team apply industry best practices to maximize model reuse and improve generated code performance.

The team completed development on schedule, and the test and certification campaign progressed smoothly due to the extensive simulations the team had performed in preparation.

Now that they have demonstrated the success of the new workflow, more engineers are involved in Model-Based Design across the organization, and they have built a library of models and a knowledge base that can be reused on future projects.

Summary

A system model is at the center of development, from requirements capture to design, implementation, and testing. That's the essence of Model-Based Design. With this system model you can:

- Link designs directly to requirements
- Collaborate in a shared design environment
- Simulate multiple what-if scenarios
- Optimize system-level performance
- Automatically generate embedded software code, reports, and documentation
- Detect errors earlier by testing earlier

> "Three years ago, SAIC Motor did not have rich experience developing embedded control software. We chose Model-Based Design because it is a proven and efficient development method. This approach enabled our team of engineers to develop highly complex HCU control logic and complete the project ahead of schedule."
>
> *— Jun Zhu, SAIC Motor Corporation*

Tools for Model-Based Design

Foundation Products

MATLAB®

Analyze data, develop algorithms, and create mathematical models

Simulink®

Model and simulate embedded systems

Requirements Capture and Management

Simulink Requirements™

Author, manage, and trace requirements to models, generated code, and test cases

System Composer™

Design and analyze system and software architectures

Design

Simulink Control Design™

Linearize models and design control systems

Stateflow®

Model and simulate decision logic using state machines and flow charts

Simscape™

Model and simulate multidomain physical systems

Code Generation

Simulink Coder™

Generate C and C++ code from Simulink and Stateflow models

Embedded Coder®

Generate C and C++ code optimized for embedded systems

HDL Coder™

Generate VHDL and Verilog code for FPGA and ASIC designs

Test and Verification

Simulink Test™

Develop, manage, and execute simulation-based tests

Simulink Check™

Verify compliance with style guidelines and modeling standards

Simulink Coverage™

Measure test coverage in models and generated code

Simulink Real-Time™

Build, run, and test real-time applications

Polyspace® Products

Prove the absence of critical run-time errors

Learn More

mathworks.com has a range of resources to help you ramp up quickly with Model-Based Design. We recommend that you begin with these:

Interactive Tutorials

MATLAB Onramp

Simulink Onramp

Stateflow Onramp

Webinars

Simulink for New Users (36:05)

Model-Based Design of Control Systems (54:59)

Accelerating the Pace and Scope of Control System Design (51:03)

Modeling, Simulating, and Generating Code for a Solar Inverter (45:00)

Onsite or Self-Paced Training Courses

MATLAB Fundamentals

Simulink for System and Algorithm Modeling

Control System Design with MATLAB and Simulink

Additional Resources

Consulting Services