Current advances in model-based development technology indicate that embedded-software developers can expect more tool support for the whole embedded-software spectrum, from applications to system software.

As processing power and memory costs have decreased, embedded-software applications have become pervasive. Over the past several years, strong demand for adding ever more features to software applications has led to much larger and more complex embedded systems. As the number and complexity of these applications grow, the safety-, mission-, life-, or business-critical aspects of many embedded applications continue to demand high levels of reliability and robustness in the face of evolving requirements and environments.

**MODEL-BASED DEVELOPMENT PROCESS**

In almost every engineering discipline, models have been used to manage system complexity. Developers also have employed them as reusable and analyzable artifacts to bridge the conceptual gap between requirements and target-system implementations.

The software engineering industry continues to witness a rapid growth in the use of model-based development (MBD) technologies in various vertical domains, with aerospace, transportation, medical, and the consumer electronics industries being the main drivers.

Although well supported by commercial tools, MBD is relatively new to embedded-software-engineering practitioners. In particular, we have recognized from our experience with leading embedded-automotive-software suppliers that applying MBD in embedded-software applications development encourages practitioners to use testing techniques that take another tack than that of traditional techniques.

While these techniques are not yet standardized, they have significant and commercially available tool support. They can thus provide some lessons and techniques that will help practitioners and aid tool vendors in making future improvements to their products.

A *model* is an abstract representation of a system with emphasis on some particular aspect of the development. Created to serve a particular purpose, models vary greatly in their level of abstraction and in the languages and notations employed to describe the specific concepts of the system being designed.

Although UML-based models have the lion’s share of current MBD practice in industry, they are rather general-purpose models and—in their pure format—not particularly suitable for describing and analyzing sophisticated systems. Other model types, such as mathematically based ones, have traditionally been used during the system-engineering design and analysis phases in embedded systems.
control systems. Here they describe the mathematical algorithms necessary for processing signals that control, for example, an electromechanical system. The industrial issues and challenges of developing such models, as well as implementing them in embedded-software applications, are particularly apropos to the automotive industry.

MBD FOR EMBEDDED-SOFTWARE SYSTEMS

The automotive systems of today exemplify complex, digitally controlled electromechanical systems accessible daily. Such systems involve up to hundreds of megabytes of code distributed over the electronic control unit (ECU) network and integrated into the vehicle’s electromechanical infrastructure. Every ECU comprises an embedded-software application that varies in sophistication from simple message-filtering applications to highly sophisticated control algorithms such as an engine-control ECU.

VERIFICATION AND VALIDATION

The basic rationale behind the verification and validation process involves doing the right things right. The key difference between V&V lies in the objective of the activity itself. Validation is about ensuring we are doing the right function, performing the right function, or demonstrating that we are performing the right function. A more detailed assessment might observe that “If we could just put this system description into machine-understandable format while preserving the same behavior we would be done,” something many software engineers think regularly.

MISSION NOT IMPOSSIBLE

While system designers focus on the design of controller algorithms and on adjusting control parameters, software developers focus on engineering this controller into a real-time software system. This involves the codification of this mathematically based model into appropriate software constructs suitable to be embedded in a specified target microcontroller.

Given the market pressure to deliver efficient and correct systems, meeting the software system’s delivery deadline with the specified requirements can be all but impossible. Consequently, many projects ultimately fail to deliver their products on time. However, as the growing maturity of MBD and its associated powerful automatic code-generation technologies become commercially available, complemented by innovative model-based V&V techniques, the software engineering mission will, theoretically, now be routinely possible. While this sounds attractive, the new
development paradigm comes with its own challenges.

SOFTWARE DESIGN SPECIFICATION

From the system engineers’ viewpoint, our model is no more than an executable diagram of signal flows and processing algorithms. There is nothing in the model that describes the software architecture in terms of the following:

- the mapping between different parts of the model’s algorithms and the software procedures, typically C-functions;
- the appropriate data structures—such as arrays, linked lists, and tables—used to store the signal values in memory; and
- how all these pieces of procedures and data-structure objects will be assembled in some abstraction of software components—such as a class in C++ or a module in C—that interact.

These points address the structural aspects of the software architecture. Another, more important aspect is the real-time architecture, which addresses issues such as

- the real-time tasks and their appropriate scheduling strategy (priority and periodicity),
- the mapping between the structural objects already described and these tasks, and
- the interrupt handling strategy and assignment to available interrupt vectors.

It seems that we design the software system retrospectively, from the bottom up. Although this somewhat contradicts the traditional view of best practice in software design, the retrospective design approach can be enhanced if the model itself is constructed modularly and the modeling language has sufficient expressiveness and abstractions to create a structured description of the model. In addition, standardized modeling guidelines and best practices can aid the system engineer in using the modeling language’s features.

Specifying this software architecture appropriately requires techniques such as annotations (extra information added to the model to guide code generators or other model processing tools) and adding extra—possibly user-defined—objects to the model itself. These specifications should be integrated in the model so that code-generation tools can exploit them and reflect them in the generated code.

Describing and specifying the software architecture in this way is tool-specific and is partially supported in commercially available tools. Hence, we can skip the description of this activity and assume that we have a model with all the software architecture specification in place. Thus, we assume that the model is ready for code generation.

In this next step, the software engineer must verify the autogenerated software to be embedded in the final target ECU. This is not a one-click step. Rather, developers verify the software in sequential testing stages. At every stage, the verification test targets a different kind of possible defect and its accompanying design flaws. This powerful separation of concerns accelerates the detection and debugging of defects.

X-IN-THE-LOOP TESTING

Model-, software-, processor-, and hardware-in-the-loop (MIL, SIL, PIL, and HIL) tests—which we call X-in-the-loop tests—provide four testing configurations that developers can apply to autogenerated code. The model, software, processor, and hardware terms refer to the different target system configurations in the testing environment, each of which adds value to the verification process.

Model-in-the-loop

The MIL is the basic simulation that system engineers originally perform to analyze the controller model along with the simulated plant model. System engineers derive the set of test scenarios that covers the system requirements. Once the MIL test results are proven satisfactory they are considered a valid reference for comparison with any subsequent or further non-MIL tests. The basic idea behind this trustworthiness is that in MIL test simulation the computations are performed in floating-point arithmetic with high precision, dependent on the word length of the underlying simulation machine. Thus, the controller’s MIL test output values provide the best representatives of the physical values needed to control the plant.

This shows clearly that, apart from the analysis in the system engineering phase, the MIL test’s purpose is to generate the reference test result values for subsequent tests. The only problem that could occur during an MIL execution session is that the model would fail to execute, which might reveal a problem with the simulation environment and tool, or even with syntax in the model itself. This is much less likely to happen, however, because system engineers should have already fixed any such problems.

Software-in-the-loop

Here, we replace the model in the MIL test with the executable object code of the corresponding autogenerated software. The code generator, typically integrated in the
modeling and simulation tool environment, provides a pretest step to generate the source code from the model. Modularity, data structures, and the sizes of the generated source code depend on the model’s software architecture.

The system then compiles the source code for the underlying host PC on which the simulation environment is running and links with the code necessary to provide the stubs for setting the software inputs to the appropriate data values of test vectors, which records the output test values for later comparison.

MIL performs these underlying computations in floating-point arithmetic, using tool-specific libraries that provide the necessary functionality and calculations for the elementary model blocks. However, the SIL test performs the computations in fixed-point with the real code, which reflects the actual computations that would be done in the final target system. Not all tools provide fixed-point execution in SIL.

This approach enables the detection of any possible overflow and division-by-zero fault occurrences caused by the inadequate design choices of the variable data sizes during the software architecture’s specification. This is important because software designers always seek to optimize the overall code size and RAM usage by selecting the minimum sizes that will satisfy the value ranges of the input and output variables estimated by the system engineers.

The estimated ranges, however—particularly in control systems’ mathematical algorithms—can be determined incorrectly. Once the system detects a defect, the testing environment should provide the tester with a suitable navigation tool to easily jump to the erroneous data variable, which will be reported to the designer for correction.

Up to now, we have accepted that the testing environment runs MIL and SIL tests entirely on a normal PC. In the following tests, the work will take more realistic configurations of the system as the generated code executes—in the case of the PIL test—on the target processor in a non-real-time environment. Then, in the case of the HIL test, it will execute on the final real-time environment ECU.

**Processor-in-the-loop**

PIL provides the first step taken to test the real object code on the target processor. It generates the cross-compiled source code and executes it on the target processor machine, with all the predefined optimization options applied by the compiler, and then executes the test scenarios on the embedded software in non-real-time simulation sequential steps.

At this point, the code generator itself should share responsibility with the cross-compiler for generating the final optimized real object code. Indeed, the code generator, acting like an automatic programmer, should generate optimized source code constructs and compiler-specific keywords that guide the compiler itself to produce efficient object code, just as the embedded-software programmer does in the manual coding activity. Not surprisingly, this imposes several challenges on the code-generating tools that exploit commercial compilers to provide the means for designers to tailor the code generator to a specific compiler suite. Note that the correctness of the generated code should not be compromised by excessive optimizations, and code-generator vendors should consider mechanisms to ensure that.

Returning to the PIL test, the resulting object code links with other test-management functionality and is then downloaded, typically to an off-the-shelf evaluation board of the target processor. The simulation tool, running on the PC machine, then communicates with the downloaded software, typically via a serial communication link. Therefore, we have two pieces of distinct embedded software: the test management code and the code for the software under test.

During the PIL test session, for each test scenario the simulation tool sends the test values to the embedded software on the evaluation board via the communication link, then waits for the test output results. The test management code receives the test values and sends them to the input variables of the software under test. It then triggers the test software’s code to execute once. Next, the test management code collects the test output results from the output variables and sends them to the awaiting simulation tool. Finally, the simulation tool records the results for later comparison. The simulation tool repeats this process automatically until all the test scenarios execute.

With the simulation tool involved in the testing flow, the software’s real-time behavior cannot be tested in PIL. Although this seems to be a limitation in the PIL test, it is actually desirable because, essentially, the PIL test’s objective is to detect non-real-time-related defects such as code-generator bugs caused by inadequate target-specific code, undesirable side effects of the compiler setting and optimization options, and bugs in the compiler itself. Some demonstrations show the role PIL testing plays in bridging the gap between the simulation-based testing in MIL and SIL tests and the real-time testing of HIL.

**Hardware-in-the-loop**

In the last line of defense during the verification process, HIL testing, the system integrates the application software
generated from the model with the real-time software infrastructure (I/O and network management drivers, real-time operating system [RTOS], services, and so on), then downloads it to the hardware processor or microcontroller, which is integrated into the ECU hardware.

The plant model originally developed by system engineers no longer simulates the physical environment signals; rather, a dedicated hardware setup specially designed for this purpose simulates them. This programmable simulator generates physical signals that mimic the real environment’s inputs to the ECU connected to the simulator through a controller area network (CAN) or local interconnect network (LIN) cable as if it were connected to the real vehicle system.

The test engineer programs the timed test scenarios, originally devised by system engineers, which simulate the actual scenarios that would happen in the real vehicle environment. During the HIL test session, the system sets the signal-simulator to automatically apply scenario-by-scenario tests successively and record the time-stamped results for later analysis. The HIL test simulator might take hours or days of free running time for the HIL test session to finish because of the possible long and accumulated delay intervals imposed in test scenarios for simulating the real context—for example, the test designer might insert an interval of several seconds or minutes between two steps in the test scenario. This is crucial for validating performance requirements and timing constraints. Figure 1 summarizes the X-in-the-loop testing phases.

**Timing issues**

System engineers who understand the whole system’s processes and behavior originally designed the test scenarios’ timing. This timing aspect does not make sense in any previous test configuration except the HIL test’s because the time itself was being simulated in non-real-time: The HIL test’s purpose is to confirm the real-time functionality, performance, and system behavior as a black box with effects from all the involved elements, depending on the applied tests.

With this kind of black-box end-to-end testing, the tester will be unable to probe the internals of the system under test. Although this is intended, any defect that pops up in this test will be hard to debug. Most probably, this type of defect is of “the system not doing the right thing” type, not “the system is not doing the thing in the right way” type. If the defect were of the latter type, it would have been caught in either SIL or PIL tests, thanks to the X-in-the-loop testing strategy of separating different defect detection routines in different test configurations.

We found significant value in conducting HIL tests in a lab setting using a dedicated simulator rather than a real vehicle system. This offers the following pros:

- **Repeatability.** We can repeat the test anytime and for any number of sessions.
- **Automatic execution.** The test session executes automatically, with facilities that record the results for offline analysis and comparison; hence, this can be done overnight without allocating a dedicated engineer to run or monitor the process.
- **Economy.** Despite the simulator’s potential cost, it is still cheaper than setting up a real vehicle environment to test the system. The hardware’s precision in generating accurate time and high-fidelity signals emphasizes this benefit.

![Figure 1. Summary of X-in-the-loop testing phases. In this process, the test reference results are generated in the MIL test and then compared with the corresponding results output from all subsequent tests.](image-url)
Managing complexity
Software developers frequently face the challenge of managing model complexity. From the code-level testing viewpoint, the generated code must meet certain code-coverage criteria that depend on the underlying development process followed. Modified condition/decision coverage (MC/DC) is a typical coverage criterion required by safety-related software development standards such as DO-178B. For large models, the generated code is too complex to pass such coverage criteria using only the test scenarios designed by system engineers, which generally cover typical system operations. One intuitive way to eliminate this complexity decomposes the model into smaller units, generates their corresponding code, and designs test scenarios that achieve the required coverage within the scope of these smaller units. Once we have the set of test vectors (scenarios) that achieve the required coverage level for every unit, we can ensure the proper and safe operation of these units when involved in any higher-level scenario, at least as far as the coverage level is concerned.

The problem with this is that software engineers often lack the requisite knowledge of algorithms developed by system engineers, but must still understand every piece of functionality and their composition into these model units to determine sufficient tests to achieve the desired coverage. Trying to comprehend the nuts and bolts of these model units is laborious and inefficient. So, developers sometimes use ad hoc techniques such as random tests, which in our experience did not scale linearly with the coverage percentage. Table 1 shows a real example of this problem.

After adding around 200 more vectors and using the same random process to re-execute the whole test session of about 600 vectors, we attained 40 percent coverage. These results provided a diminishing return in increased coverage percentage as the test session added more vectors, with saturation at an unacceptable 56 percent.

We tried a more regular technique of test-vector generation whereby we changed the value of one parameter only from a selected value in the range while holding the other parameters fixed. This proved to be less efficient than random test generation, with regard to our example.

Without achieving the coverage criterion, developers cannot make full use of SIL and PIL tests, which depend mainly on verifying code constructs. Practitioners expect tool vendors to provide smart solutions to these problems by, at least, giving guidance to developers regarding suitable test vectors.

For example, the tools might analyze the model parameters’ coverage, trace back the uncovered points to the I/O parameters in the model, then devise a set of values for these parameters to take to reach the uncovered branch. These values can be an input to the random test generator that specifies the effective test vectors. Such smart tools would save weeks of the developer’s time, and let the testing process be repeatable with almost no effort. This approach also mitigates a serious problem in the automotive software engineering industry of frequent requirements and design changes.

A more protective approach to mitigate this problem is designing code generators to be customizable in order to limit the complexity of the generated code. If code generators can be adjusted to generate as “flat” as possible

<table>
<thead>
<tr>
<th>Model size (input/output signals)</th>
<th>Random vectors</th>
<th>Code coverage</th>
<th>Total branches</th>
<th>Reached branches</th>
<th>Unreached branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>19/14</td>
<td>393</td>
<td>32.2 percent</td>
<td>264</td>
<td>85</td>
<td>179</td>
</tr>
</tbody>
</table>

• **Negative option.** Last and most important, the real vehicle environment provides a negative option given the difficulty—if not impossibility—of applying certain signals with specific timings to the inputs of the ECU under test.

The negative option occurs in this case mainly because the ECU would then be deeply embedded in the vehicle networks, and testers would be unable to devise the vehicle-system scenario that can achieve the required signals at the ECU inputs, such as driving the car at a certain speed, for example, on a certain road or in certain climate conditions. Even if testers could achieve such results, however, they would not be able to control all other relevant factors from the vehicle’s other ECUs.

**LESSONS LEARNED**

The emerging technology of X-in-the-loop verification confirms the fidelity of embedded-software systems, especially data-intensive signal-processing applications. Ideally suited to these kinds of applications, it can also apply to application types such as network data packet switching in network routers and access servers.

Other analysis techniques, however, such as queue theory and security-related analysis techniques, must be applied to these kinds of control-intensive applications. The use of models as a lingua franca of development between system and software engineers facilitates the communications between them and lets each seamlessly play a managed role in the process.

**Managing complexity**

Software developers frequently face the challenge of managing model complexity. From the code-level testing viewpoint, the generated code must meet certain code-coverage criteria that depend on the underlying development process followed. Modified condition/decision coverage (MC/DC) is a typical coverage criterion required by safety-related software development standards such as DO-178B. For large models, the generated code is too complex to pass such coverage criteria using only the test scenarios designed by system engineers, which generally cover typical system operations. One intuitive way to eliminate this complexity decomposes the model into smaller units, generates their corresponding code, and designs test scenarios that achieve the required coverage within the scope of these smaller units. Once we have the set of test vectors (scenarios) that achieve the required coverage level for every unit, we can ensure the proper and safe operation of these units when involved in any higher-level scenario, at least as far as the coverage level is concerned.

The problem with this is that software engineers often lack the requisite knowledge of algorithms developed by system engineers, but must still understand every piece of functionality and their composition into these model units to determine sufficient tests to achieve the desired coverage. Trying to comprehend the nuts and bolts of these model units is laborious and inefficient. So, developers sometimes use ad hoc techniques such as random tests, which in our experience did not scale linearly with the coverage percentage. Table 1 shows a real example of this problem.

After adding around 200 more vectors and using the same random process to re-execute the whole test session of about 600 vectors, we attained 40 percent coverage. These results provided a diminishing return in increased coverage percentage as the test session added more vectors, with saturation at an unacceptable 56 percent.

We tried a more regular technique of test-vector generation whereby we changed the value of one parameter only from a selected value in the range while holding the other parameters fixed. This proved to be less efficient than random test generation, with regard to our example.

Without achieving the coverage criterion, developers cannot make full use of SIL and PIL tests, which depend mainly on verifying code constructs. Practitioners expect tool vendors to provide smart solutions to these problems by, at least, giving guidance to developers regarding suitable test vectors.

For example, the tools might analyze the model parameters’ coverage, trace back the uncovered points to the I/O parameters in the model, then devise a set of values for these parameters to take to reach the uncovered branch. These values can be an input to the random test generator that specifies the effective test vectors. Such smart tools would save weeks of the developer’s time, and let the testing process be repeatable with almost no effort. This approach also mitigates a serious problem in the automotive software engineering industry of frequent requirements and design changes.

A more protective approach to mitigate this problem is designing code generators to be customizable in order to limit the complexity of the generated code. If code generators can be adjusted to generate as “flat” as possible
code structures, it would keep the cyclomatic complexity as low as possible (and hence the code coverage could be achieved by fewer test vectors). Not only the cyclomatic complexity metric but other code metrics should also be controllable. On the other hand, at higher levels, models should be modularized enough in a design-for-test strategy to help the code generators do their job.

Model-level tool support

Some tool support is commercially available for the test coverage and test autogeneration problems, but at the model level, not the code level. Even more confusing, the tools apply code-level coverage criteria (such as MC/DC) to model blocks such as conditional blocks.

We do not, however, know how the code generator will implement these conditional blocks. It could use any of the implementation language’s conditional constructs—such as if-else, while-loop, or for-loop—each of which might need different testing to achieve the target coverage criteria. Model coverage thus provides the interesting challenge of finding the redundant blocks that are not covered or partially covered by system test scenarios, and their contribution to the system functionality is questionable. However, using code-level metrics to measure the model coverage is Infeasible, at least from a verification viewpoint. Models need a different focus when used for V&V.

FUTURE CHALLENGES

Undoubtedly, models have provided a common vocabulary between domain system and software engineering teams in this multidisciplinary development process. However, this is half the story for software engineers. The kind of models and associated testing we have described mainly focus on the ECU software’s application implementation.

The infrastructure software that comprises different abstraction layers of I/O and device drivers, network management drivers, real-time OS services, and so on must still be dealt with. This infrastructure software is at least as complex as the application part. It requires a similar approach to modeling and analysis, accompanied by effective techniques for verification, such as X-in-the-loop.

Looking carefully at the application software part, we recognize a solid theoretical foundation behind its models of control theory, digital signal processing, and so on that gives the models trustworthiness. Likewise, the modeling tools provide semantic analyses to capture these foundational concepts, such as ordinary differential equations and solvers.

The infrastructure software part also involves concepts that have theoretical foundations, such as scheduling and network analyses. However, developers have very little tool support to provide modeling and analyses of these concepts in embedded systems.

T he state of practice in developing embedded-infrastructure software uses component-based architectures such as the AUTOSAR initiative (www.autosar.org), along with static code analysis tools to capture design flaws. However, these still fail to address the dynamic and real-time aspect of the infrastructure software. Fortunately, current advances in MBD technology indicate that embedded-software developers can expect more tool support for the whole ECU software spectrum.

Acknowledgment

This work was supported in part by Science Foundation Ireland grant 03/CE2/I303_1 to Lero—the Irish Software Engineering Research Centre (www.lero.ie).

References


Hesham Shokry is a PhD student at Lero—the Irish Software Engineering Research Centre. His research interests include runtime-reconfigurable software systems for applications in avionics and automotive domains, supported by domain-specific languages and model-driven development tools. He received a BSc in computer engineering from Assiut University, Egypt. He is a student member of the IEEE. Contact him at hesham.shokry@lero.ie.

Mike Hinchey is codirector of Lero and a professor of software engineering at the University of Limerick, Ireland. His research interests include self-managing software and formal methods for system development. Hinchey received a PhD in computer science from the University of Cambridge. He is a senior member of the IEEE and is currently chair of the IFIP Technical Assembly. Contact him at mike.hinchey@lero.ie.