Multilevel Testing for Design Verification of Embedded Systems

Stephan Schulz
Nokia

Klaus J. Buchenrieder
Infineon Technologies

Jerzy W. Rozenblit
University of Arizona

A multilevel testing approach for embedded systems addresses mixed hardware and software implementations. Contrary to conventional approaches, it provides consistent generation of scenarios throughout all levels of testing, an early assessment of alternative design implementations, integrated system and component testing, and performance assessments of design specifications starting from the system level.

According to recent surveys, approximately 90% of all processors are part of embedded systems, computing systems that continually and autonomously control and react to the environment. The embedded system itself is an information processing system that consists of hardware and software components. Nowadays, the number of embedded computing systems—in areas such as telecommunications, automotive electronics, office automation, and military applications—is steadily growing.

This market expansion arises from greater memory densities as well as improvements in embeddable processor cores, intellectual-property modules, and sensing technologies. At the same time, these improvements have increased the amount of software needed to manage the hardware components, leading to a higher level of system complexity. Designers can no longer develop high-performance systems from scratch but must use sophisticated system modeling tools.[1,2]

A continuing increase in system complexity, diminishing design cycles, tightly integrated mixed hardware and software components, and the growing use of reconfigurable devices characterize the current generation of embedded systems. Software tends to be customized, and programmers code it using low-level programming languages to achieve predictable, high performance. The processing environment reflects restricted budgets and physical limitations, and uses only a minimal set of hardware components.

Conventionally, hardware and software development groups design and test these systems separately, and then integrate them into a system prototype. This late integration tends to require many design iterations on the application prototype. For applications with high performance requirements and safety constraints, this high number of late design iterations is a major concern because design teams must guarantee a well-tested and fully debugged final product. Thus, for these applications, we advocate the use

0740-7475/02/$17.00 © 2002 IEEE

of sound design methodologies and development environments that emphasize early design assessment.

We developed one such methodology, *model-based codesign*,[2] which uses system modeling[3] to prototype systems under design. Our work focuses on the development of design techniques in which models can be synthesized and tested for several objectives, taking these objectives individually or in tradeoff combinations. Model-based codesign lets developers create computer models of embedded systems independently of their eventual hardware and software implementation, enforcing a late partitioning of the system design. Designers use simulation to explore the feasibility of virtual prototypes and then interactively map the specifications onto a mixed hardware-software architecture. In several publications, we have elaborated on the fundamental concepts supporting model-based codesign.[2,4] Figure 1 provides an abstract representation of the methodology, which has six basic components:

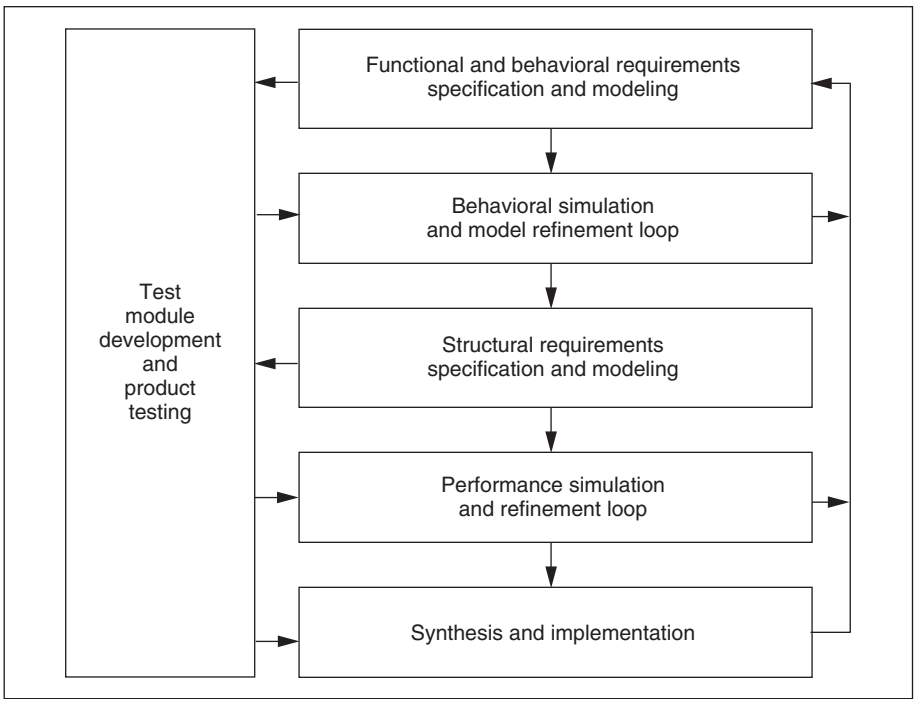

Figure 1. Abstract design flow in model-based codesign.

- *Functional and behavioral requirements specification and modeling* encompasses the solicitation and documentation of requirements and the development of an executable model.
- The *behavioral simulation and model refinement loop* iteratively refines the design model until it is functionally correct.
- *Structural requirements specification and modeling* relates physical design constraints to a proposed processing architecture.
- In the *performance simulation and model refinement loop*, designers enhance the model with performance measures for computation and communication. They obtain performance measures from a preliminary, reconfigurable system prototype that implements the chosen architecture.
- *Synthesis and implementation* involves extracting design specifications from the models to produce a physical prototype.
- *Test module development and product testing* creates a set of test scenarios from the system requirements specification, which designers use to assess the design at all levels of the design process.

Here, we propose a multilevel approach to testing, which complements the development of embedded systems using model-based codesign. This approach has several benefits: It provides

- early evaluations of alternative system configurations,
- a consistent set of test scenarios that follow the iterative system refinement,
- real system performance measurements as opposed to estimates,
- the ability to test system components individually or in a compound structure,
- debugging as a part of each design level, and
- reliable model component reuse.

Embedded-systems testing

Testing methods and objectives differ in the hardware and software domains. Embedded

software development uses specialized compilers and development software that offer means for debugging. Programmers develop application software on more powerful computers and eventually test the application in the target processing environment. Future on-chip debugging support promises to improve software performance, and estimation and analysis.[5]

In contrast, hardware component testing concerns itself mainly with functional verification and self-test after chip manufacturing. Hardware developers use tools to simulate or formally prove the correct behavior of circuit models. Vendors design chips for self-test,[3] which mainly ensures proper operation of circuit models after their implementation. Test engineers—not the original hardware developers—test the integrated system.

This conventional, divided approach to software and hardware development does not address the embedded system as a whole during the system design process. It instead focuses on these two critical facets of testing separately. New problems arise when developers integrate the components from these different domains.

In theory, unsatisfactory performance of the system under test should lead to a redesign. In practice, a redesign is rarely feasible because of the cost and delay involved in another complete design iteration. A common engineering practice is to compensate for problems within the integrated system prototype by using software patches. These changes can unintentionally affect the behavior of other parts in the computing system.[5]

At a higher abstraction level, executable specification languages[1] provide an excellent means to assess embedded-systems designs. Developers can then test system-level prototypes with either formal verification techniques[6] or simulation. A current shortcoming of many approaches is, however, that the transition from testing at the system level to testing at the implementation level is largely ad hoc. To date, system testing at the implementation level has received attention in the research community only as coverification,[7] which simulates both hardware and software components conjointly. Coverification runs simulations of specifications on powerful computer systems. Commercially available coverification tools link hardware simulators and software debuggers in the implementation phase of the design process.

While working on a design methodology for tightly integrated embedded systems, we noted that research in system-level design and test for such systems has not identified a need for a gradual transition of test specifications to the implementation level. This gradual transition allows a consistent assessment of application design specifications with various levels of detail.

To provide this gradual transition, we developed a multilevel testing approach for mixed-system prototype implementations. Our approach uses a software-based real-time testing environment for system testing.

Multilevel testing approach

In the design of complex embedded systems, we encounter multifaceted requirements. The assessment and verification of these requirements is complex and even impossible in some cases because of the design's abstract specification at the model level. We advocate a structured multilevel approach to testing, which follows the system development through its various stages. Figure 2b depicts our approach.

In this approach, we start by deriving a set of test scenarios from a textual system requirements specification.[2,8] As previously outlined, we initially develop an abstract system model for the application, which we then decompose into model components in a top-down design approach. In these models, we can isolate specific aspects of the system under design, thus reducing initial design complexity. Gradual refinement adds more detail to the models. We continue to improve our system model until its behavior cannot be distinguished from the desired behavior of the specified system.

At the system level, we use test modules to validate the system model. To assess an application design, we connect these modules to and simulate them in conjunction with the system model. Test modules are directly derived from one or more behavioral and structural application requirements, and follow the incremental refinement of the application design model. The

modular construction also enables us to monitor specific model components during the simulation and identify possible performance bottlenecks early in the design process.

During the application design, we map the system model onto a selected hardware architecture and integrate it into a reconfigurable prototype. We follow this transition by converting our testing modules to a set of test processes for a real-time system-testing platform. At the integrated-system level, we can reuse test scenarios from the set developed during the modeling phase. The test processes then create the corresponding scenarios in real time on the mapped model of the embedded computing system. During a test run, the system testing environment (STE) records and analyzes the design implementation's performance.

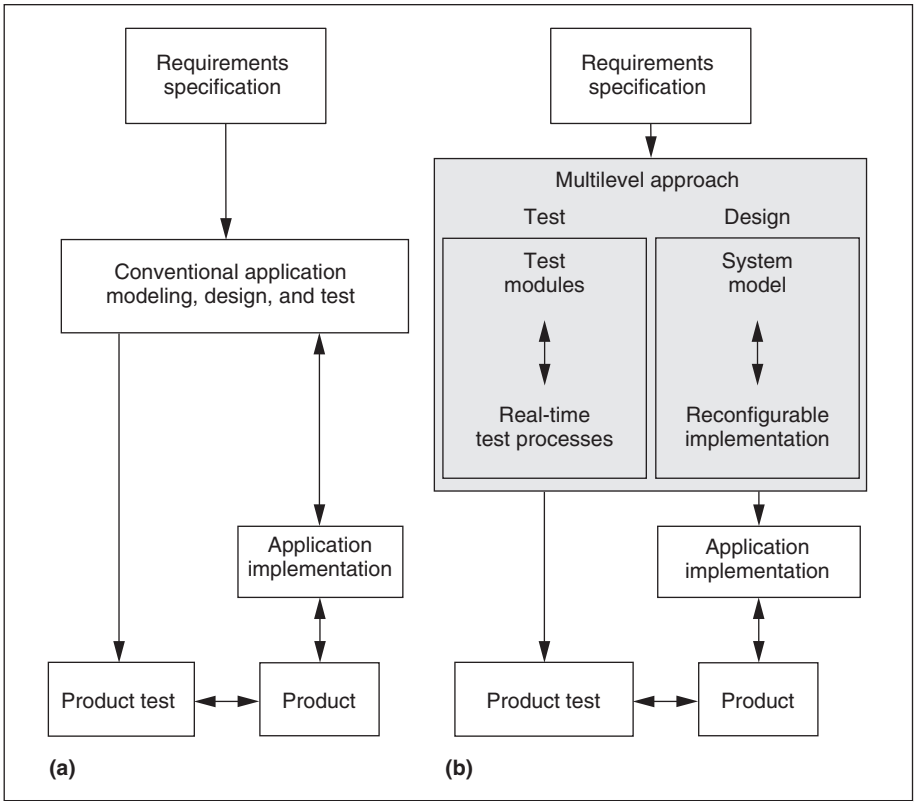

Figure 2. Conventional (a) and multilevel (b) design and test approaches.

Compared to the conventional approach, multilevel design and test permits the specification of test scenarios at a high level of abstraction—the system level. It encourages a gradual refinement of these abstract test scenarios from the system model level to the integrated-implementation level. As the abstraction level drops, the test scenarios' abilities increase to serve the different testing objectives of system- and implementation-level testing. That is, these scenarios become useful for functional versus performance testing. Test scenarios remain consistent throughout system development, and changes can easily be propagated between lower and higher abstraction levels. In addition, it clearly distinguishes system testing from system modeling early in the design process.

Model testing

Model-based embedded systems design has recently gained a lot of attention in the embedded-systems community.[1,2,7] One of its advantages is implementation-independent system design, which fosters late integration of hardware and software components. Another advantage is that developers can easily analyze time-critical applications by varying model execution parameters for the simulation platform. Simulation time imposes an ordering on the occurrence of events instead of acting as a hard processing constraint. Developers can introduce delay estimates to identify possible system bottlenecks.

Gradual development. In the modeling phase, the abstract system design gradually evolves into a virtual system prototype that closely resembles the final implementation. The model can then be converted into detailed design descriptions that allow the physical prototyping of a mixed hardware-software system design. This late partitioning into a detailed, implementation-level design specification lets developers easily produce reconfigurable and customized implementations.

Although several system modeling tools exist, published research about corresponding design methodologies does not directly address testing at the system level. In our methodology,

we address system model testing with a concept called *experimental frames*—coupled test modules that, in their entirety, model the environment in which the application is embedded.[4]

Our methodology specifies models using a formal, discrete, event-based specification that enforces a separate and modular specification of design and test models. This formalism, called DEVS (Discrete Event System Specification), also facilitates a conjoint execution and evaluation of these model components over a specified simulation interval.[9]

Test modules. Test modules, which compose experimental frames, mainly serve the function of a test event generator, test monitor, or performance analyzer. These model components are created separately from the system model. Each component represents a part of the environment and reflects certain behavioral requirements of the system design. Simulation of the test modules together with the system model represents an experiment where the application interacts with its environment. In essence, we create a test bench at the model level, which we use to validate various aspects of either the entire application or its components. Test modules encode test scenarios, either fixed or interactive, from specified design requirements. Interactive modules allow an early integration of a human system user into the application development. The modularity of the test modules enables reuse in different applications or in design alternatives for the same application.

We base the development of test modules on a stepwise refinement process. This process requires the prior identification of behavioral requirements in a textual requirement specification. We then incrementally introduce requirements into event-generating test module specifications and into the actual design component specifications. Each newly introduced requirement also requires a refinement of conformance criteria in monitoring test module specifications. Notice that a model-based design approach even allows performance assessments of abstract design models by using our concept of computational complexity for model component specifications.[2]

Product testing

At the implementation level, software components should be fully debugged, and the target architecture should have been tested for integrity. In product testing, developers verify the final application prototype by either using specialized test equipment that emulates parts of the target architecture or, more commonly, the real environment. Here, in contrast to the modeling phase, the execution time is fixed, and developers can obtain true performance results. They can also observe the structural properties introduced by the system design, board design, hardware configurations, software components, and their interactions with one another.

Developers integrate embedded-systems designs at the implementation level. Conventionally, they develop application-specific software and iterations on the board design separately, integrate them, and for the first time, test the entire system for compliance with the originally specified requirements in this product testing phase.

In our model-based approach, assessment of the integrated system prototype occurs much earlier in the design, as it is directly derived from the design model specification and assessed as a system implementation. Developers can still perform product testing for conformance purposes, as in the conventional approach.

Real-Time STE

Our multilevel testing approach takes advantage of the already accumulated repository of test scenarios, which are in the form of test modules. Our STE provides a smooth transition from simulation to real time, and inserts another level of testing between model and product testing. We consider this STE as a step toward real-time simulation—that is, real-time execution of application models in their environment.

The modeling level allows an early assessment of design requirements using performance estimates. This assessment might not suffice to accurately verify applications with high-performance constraints. To obtain true performance measures, we apply test scenarios generated by the test modules to a physical realization of the system model implemented in a reconfigurable

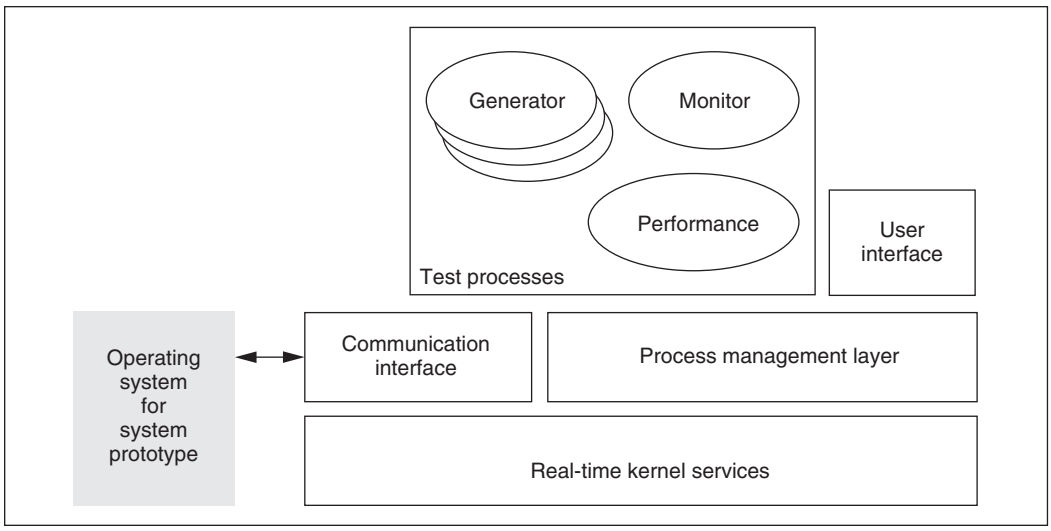

Figure 3. STE organization.

processing architecture. This prototype consists of standard processing elements, reconfigurable hardware components, a flexible operating system to coordinate the software components' communication, and an efficient interface with the testing environment.

The real-time STE provides the foundation for our performance tests of application prototypes. The environment is written in C and runs on a standard PC. The program minimizes processing overhead in the generation of external stimuli for physical prototypes and allows an accurate evaluation of the prototype's response. As Figure 3 shows, the STE software consists of

- test processes;
- a kernel based on minimal real-time operating system μC/OS;[10]
- a process management layer, which handles scheduling, interprocess communication, test analysis, real-time compliance of experiments, and so on;
- an efficient communication channel to the system prototype; and
- a user interface for test data analysis.

The STE supports test processes generated from test modules with a real-time operating system platform. The direct mapping results in a consistent set of test scenarios for performance evaluations of the prototype under system testing conditions. In addition, the execution environment remains application independent.

Test processes fall into categories similar to those for test modules: generator, monitor, or performance processes.

Generator processes

During an experiment, these processes reproduce test scenarios in real time from test scripts that match a specified behavior of corresponding test modules in previous simulation runs.

Contrary to the conventional approach of manually specifying implementation-level test scenarios, we automate this process by leveraging our formal specification of test modules. Our approach generates test scripts for each test module by recording time-stamped events at the output and input ports of each module during simulation. Such an approach is feasible for high-performance test scenarios that require an efficient tester implementation. Implementing generator processes as test script interpreters minimizes processing overhead in the STE. Notice that we can also use the same techniques to transform our test module specifications into software implementations for the STE as we use for design model components.[2] Here, we transform each test module specification into a process description that uses operating system communication primitives and preserves the original test module's behavior. This

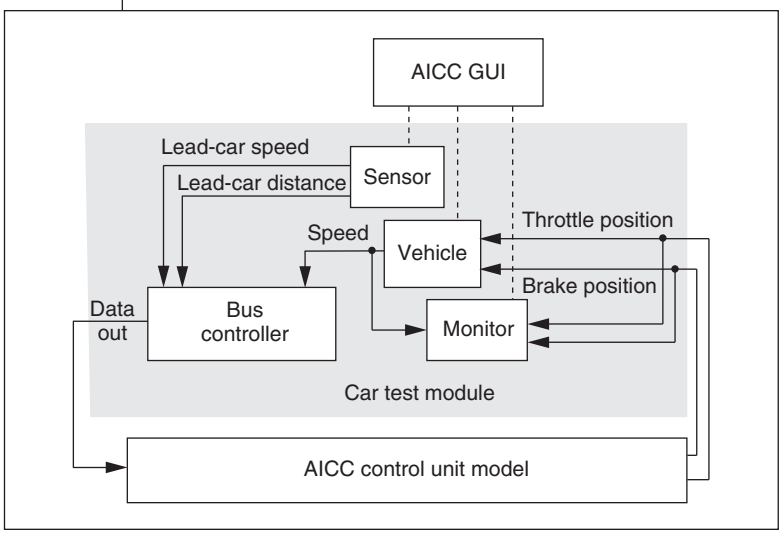

Figure 4. AICC test modules.

approach would be of interest for embedded systems that interact with human users via many possible test scenarios.

Monitor and performance processes

The monitor process focuses on verifying that the system prototype's response aligns with previous simulation results or other specified constraints. The performance process tracks the system prototype performance in the testing environment. We can also remotely instantiate a second performance process on the reconfigurable system prototype to gather data about system components at runtime. This additional process then reports basic performance measures during the system prototype's idle time. All of these performance processes can interrupt the experiment in cases of significant deviation from specified behavior or invalid system response.

During the experiment, the central process manager coordinates the STE process communication within the testing environment and with the system prototype, recording incoming and outgoing data. The monitor process tracks the system response, and the performance process collects runtime information about specified components on the physical prototype.

Sample application

In a case study, we designed an autonomous, intelligent cruise controller (AICC) using our model-based codesign methodology.[2] The AICC system is an extension of the regular automotive cruise control. It not only enforces a fixed speed, but also adapts to a lead vehicle's speed. Our example focuses on the design of this device's control unit, which interacts with sensors and actuators in the vehicle. Behavioral requirements for the control unit include keeping the vehicle speed within a narrow margin of error. The control unit must also meet a minimum response time requirement when interacting with other AICC system components.

We applied multilevel testing in the design process for this unit, creating a model of the environment to represent the vehicle under different driving conditions. This model also provided a user interface for the driver. We also directly derived testing scenarios from the STE test modules. After converting these scenarios into test processes, we used them to evaluate the performance of three selected, mixed hardware-software implementations of the control unit.

AICC test modules

The test modules for the AICC system create the environment in which we embed the unit. One approach is to decompose the computing environment into the five model components in Figure 4: a sensor, vehicle, monitor, graphical user interface (GUI), and bus controller.The sensor provides the model with information about the lead vehicle's speed and distance. The vehicle component models the car engine's behavior. It derives the current speed from throttle and brake positions computed by the AICC control unit. The monitor checks the validity of the AICC response—that is, whether the measured speed is within the specified margin. The GUI and bus controller are interfaces to the driver and the car; they convert information into appropriate internal data formats for other model components.

In this example, the vehicle component is more than a mere generator module that models the environment. It uses the AICC control unit's response as feedback in its calculation of the next speed value. At the system level, the vehicle component can assume the properties of any car engine, such as acceleration or deceleration patterns. We could enhance the car test module by introducing additional model com-

ponents describing, for example, the road's incline or curvature, or weather conditions.

Simulation results from our first design attempt[4] indicated that our AICC design was unstable: Instead of a smooth acceleration to the desired coasting speed, the control unit oscillated between acceleration and deceleration when the vehicle approached the desired speed. Tracking the system state to locate the design flaw would have taken considerably more effort in the implementation phase. In this case, it was fairly easy to perform a system-level simulation of the application design model.

System prototype testing results

We continue with the mapping of the test modules into test processes. For most test modules, the conversion is trivial. In the case of the vehicle component, we realized the module in both generator and monitor processes. We converted the stimulus produced by this component into a test script and encoded its recorded response into the monitor process. Specified real-time constraints of the AICC were included in the performance process.

We used the reconfigurable prototype to test three different processing configurations for the control unit: an all-software solution employing a Motorola 68HC11 microcontroller; a mixed Altera MAX9320 FPGA and 68HC11 solution; and an all-software solution based on a Siemens C161O microcontroller.

We used the proposed STE to record the data for performance analysis of these AICC design alternatives, obtaining results for two selected test scenarios:

- simple test scenario A with 11 input events, and
- more computationally intense scenario B involving 108 test messages.

We averaged the observed data over consecutive test runs for each test scenario.

Software-only solutions. In these configurations, we based the system prototype on two different microcontroller architectures and implemented the system entirely in software. Table 1 shows the performance of the two processing environments for both test scenarios. Average response time refers to the average time the AICC control unit took to return throttle and brake positions. Initially we constrained this response time to be less than 100 ms.

Table 1. Software-only implementation comparison.

Test scenario	A		B	
Microcontroller	68HC11	C161O	68HC11	C161O
Average response time (μs)	28	3	33	2
Total idle time (s)	2.665	2.824	2.791	2.804
Total test time (s)	8.705	10.015	9.931	10.006
Microcontroller utilization (%)	5.62	0.49	13.08	0.76

The results show that the Siemens microcontroller dominates both performance tests in all categories. Its computational advantage becomes apparent when you compare average response times; the C161O is 10 times faster than the 68HC11. This is not surprising, because the C161O is a 16-bit pipelined microcontroller running at twice the speed of the 8-bit 68HC11. Our data suggests that the Siemens microcontroller is powerful enough to handle additional computational tasks, a capability that would help reduce the number of processors in a vehicle.

Mixed hardware-software solution. In this test configuration, we implemented one system prototype component on an Altera MAX9320 FPGA to speed up the computation in the 68HC11 implementation. This move required converting a 25-line C function into a 19-cell FPGA design. The function, converted into hardware, handles driver control inputs. Table 2 (next page) summarizes the 68HC11 test data for both test scenarios. The categories in Table 2 are the same as those in Table 1.

The data indicates a performance improvement over the previous software-only solutions in terms of a decrease in microcontroller utilization for the mixed hardware-software configuration. Hardware acceleration causes this decrease in utilization. Although there is little difference in the average response times for the control unit, test scenario A shows a larger drop in utilization because it has a higher percentage of control versus sensor data messages in its message mix. The response time decreased

Table 2. Mixed hardware-software implementation using the 68HC11.

Test scenario	A	B
Average response time (µs)	28	32
Total idle time (s)	2.687	2.819
Total test time (s)	8.726	10.015
Microcontroller utilization (%)	4.69	12.86

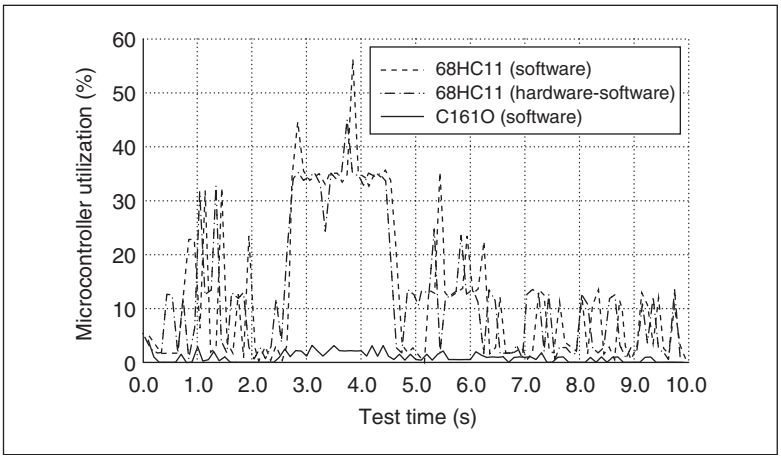

Figure 5. Microcontroller utilization over time for scenario B.

only a little because the implemented hardware function does not directly affect the most computationally intensive task, calculating throttle and brake position values.

Analysis of test results over time

We can also use the test data to show performance differences over time, that is, how the processor load varies over time for each alternative design. Figure 5 plots microcontroller utilization against the test time for test scenario B and covers all configurations.

Test scenario B activates the AICC control unit at 2.7 seconds of test time. The car reaches its coasting speed 1.8 seconds later, resulting in a decrease in computation. The control unit is disabled at 6.1 seconds. When active, the control unit frequently performs throttle and brake position computations, leading to an increase in microcontroller utilization. When disabled, it only performs basic data management tasks.

Again, the C161O outperforms any 68HC11 configuration because its utilization stays consistently below 4.2% during the entire test scenario. The 68HC11 data series for the software-only implementation indicates a performance increase after activation of the control unit, which reaches a maximum of 56% processor utilization. The hardware-software implementation improves application performance during the AICC's active period, in which the maximum processor utilization is only 45%. Though providing a noticeable increase in performance, the mixed hardware-software configuration cannot compete with the more powerful C161O microcontroller, software-only configuration.

IN OUR MULTILEVEL TESTING approach for embedded systems, testing follows the gradual refinement of the system design from the first abstract model down to the final application implementation. This approach's unique feature is the ability to translate simulation-based design experiments (test modules) into a set of real-time test processes.

Future research will focus on the integration of the STE in our model-based codesign environment at the University of Arizona and the automatic generation of STE processes. In addition, this environment is expected to evolve in further performance analysis of other mixed hardware-software implementations. ■

Acknowledgments

This work has been supported by the research laboratories of Infineon Technologies/Siemens, Munich. We thank Altera Corp. for providing an Altera ISP demo board and development software for the development of the AICC system prototype. We also thank Steve J. Cunning and Sanjaya K. Wijeratne for their help with the implementation of this project.

■ References

1. D. Gajski et al., *Specification and Design of Embedded Systems*, Prentice Hall, Englewood Cliffs, N.J., 1994.
2. S. Schulz, *Model-Based Codesign for Real-Time Embedded Systems*, doctoral dissertation, UMI No. 3002539, Dept. of Electrical and Computer

Eng., University of Arizona, Spring 2001.
3. S.J. Cunning, S. Schulz, and J.W. Rozenblit, "An Embedded System's Design Verification Using Object-Oriented Simulation Techniques," *Simulation*, vol. 72, no. 4, Apr. 1999, pp. 238-249.
4. G. Sheedy and G. Martin, "Siemens OCDS: The Next Generation On-Chip Debug Support," *Contact*, vol. 1, no. 4, Mar. 1999, pp. 53-57.
5. H.P.E. Vranken, M.F. Witteman, and R.C. van Wuijtswinkel, "Design for Testability in Hardware-Software Systems," *IEEE Design & Test of Computers*, vol. 13, no. 3, Fall 1996, pp. 79-87.
6. S. Schneider, *Concurrent and Real-Time System: The CSP Approach*, John Wiley & Sons, Chichester, UK, 2000.
7. G.R. Hellestrand, "The Revolution in Engineering a Chip," *IEEE Spectrum*, vol. 36, no. 9, Sept. 1999, pp. 43-51.
8. S.J. Cunning and J.W. Rozenblit, "Automatic Test Case Generation from Requirements Specifications for Real-Time Embedded Systems," *Proc. 1999 IEEE Systems, Man, and Cybernetics Conf.*, IEEE Press, Piscataway, N.J., 1999, pp. 784-789.
9. B.P. Zeigler, H. Praehofer, and T.G. Kim, *Theory of Modeling and Simulation*, 2nd edition, Academic Press, Burlington, Mass., 2000.
10. J.J. Labrosse, *µC/OS: The Real-Time Kernel*, R&D Books, Gilroy, Calif., 1992.

Stephan Schulz is a research engineer at the Nokia Research Center in Helsinki. His research interests include protocol testing, the Testing and Test Control Notation 3 (TTCN-3) test language, discrete-event modeling and simulation, real-time operating systems, and hardware-software codesign. Schulz has an MS and PhD in electrical and computer engineering from the University of Arizona, and a BSEE from the State University of New York, Binghamton. He is a member of the IEEE.

Jerzy W. Rozenblit is a professor of electrical and computer engineering at the University of Arizona. His research interests include complex systems design and simulation modeling. Rozenblit has an MS and PhD in computer science from Wayne State University, Michigan, and an MSc in computer engineering from the Technical University of Wroclaw, Poland. He is a senior member of the IEEE and the Society for Computer Simulation, and a member of the ACM.

Klaus J. Buchenrieder leads the Research Department for Hardware/Software Codesign, part of the corporate R&D group at Infineon Technologies AG, Munich. He is also a professor of computer science at the University of Tübingen and an adjunct professor of computer and electrical engineering for the University of Arizona. His research interests include system-level design for microelectronics and dynamically reconfigurable computing structures for sophisticated embedded hardware/software systems. Buchenrieder has a PhD in computer and information science from Ohio State University in Columbus, Ohio. He is the founding chair of the Codes/Cashe workshop series on codesign and of the Consyse workshop on conjoint systems engineering.

■ Direct questions and comments to Stephan Schulz, Mobile Networks Laboratory, Nokia Research Center, PO Box 407, 00045 Nokia Group, Helsinki, Finland; stephan.schulz@nokia.com.