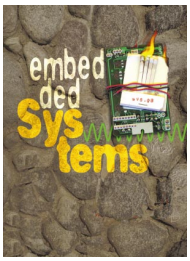
Cover Feature

Model-Based Codesign



The authors describe a type of hardware-software codesign in which developers model a system specification independently of implementation and use simulation-based design to assess virtual prototypes before the system is built. In an experimental design, the authors produced a control unit directly from the model-based specification.

Stephan Schulz
Jerzy W. Rozenblit
University of Arizona

Michael Mrva
Klaus Buchenrieder
Siemens AG

Hardware-software codesign has been a research topic since the beginning of this decade, but only now are structured methods emerging that focus on automating design. Unfortunately, to date most codesign approaches simply leverage performance from individual hardware and software tools, rather than enforcing a structured integration of hardware and software systems simultaneously.[1,2] A few frameworks have successfully done this integration and have the potential for significant benefits, including reduced time to market, smaller scale design, better likelihood of component reuse, and maximum use of processing power.

In this article, we describe a codesign approach that lets developers create models of a formal system representation independently of the hardware and software implementation. In our framework, which targets embedded systems, developers use simulation-based modeling to explore the feasibility of virtual prototypes and then interactively map the specification onto a software-hardware architecture. Our approach has several benefits:

- *It addresses model continuity*. Our approach makes it easier to convert system specifications into a hardware-software architecture because designers simply map simulatable models onto hardware, software, and interface implementations. Our position is that formal specification techniques can only do so much without a systematic modeling methodology to guide their use.
- *It has the potential to speed model development*. Although we have not done formal benchmarking and therefore cannot offer quantitative comparisons, we believe our framework's extensive reliance on modularity and hierarchy, model reuse, and the separation of model descriptions from simulation experiments has great potential to streamline design.
- *It promotes reuse*. Designers can reuse elementary components regardless of their technology, aggregating them into more complex design structures.
- *It supports design decision-making*. Designers can trade off design alternatives and select a design solution that best meets the specifications and requirements.

We have used this approach to design and build a prototype of an embedded system, the control unit of an autonomous, intelligent cruise controller (AICC). In this article, we step through this design, presenting the virtual design, the simulation results that validate our specifications, the model mapping, and a physical realization of the model-based design. Our laboratory experience at both the University of Arizona and Siemens AG indicates that moderately experienced designers can master this design framework in a short time.

FRAMEWORK OVERVIEW

Figure 1 shows the fundamental phases of our approach.

Specification

In the first phase, the designer converts the system's requirements and constraints into a formal specification. The initial specification defines the interface between the system and its environment as well as the system's functionality. Nonfunctional requirements (size, weight, architecture restrictions) are also recorded. Because our modeling approach is implementation-independent, designers can refine the specification without modifying any physically designed components.

Modeling

A model is a set of instructions for generating structural and behavioral data. Valid model-generated data is a subset of the system's behavioral data—a subset because no one can build models that account for *all* behavior. A specification of the system and its environment forms the basis for building models that correspond to a set of questions about the design, including its objectives and reason for being.[2]

The *structural* model explains the design's decomposition into its components. The *functional* model describes the overall functionality of the system and how it will integrate into its environment. The *dynamic* model assigns timing constraints to the internal func-

0018-9162/98/$10.00 © 1998 IEEE

tions and shows the details of state changes within the system model. These three descriptions are generally sufficient to generate a simulatable system description.

The type of specification language used in modeling is extremely important. The specification must accommodate different levels of granularity so that the developer can map components at different levels of abstraction to corresponding hardware or software modules. There could be descriptions at the register-transfer, gate, or chip levels for hardware components and at the instruction, function, or program level for software components. These modules must be specified in an abstract description language, to achieve implementation independence.

In our framework, designers encode system behavior using the Discrete Event System Specification formalism.[3] DEVS is a general, formal specification language that lets developers specify a system as a mathematical object. A system has a time base, inputs, states, and outputs, as well as functions for determining next states and outputs. Because models encode behavior, they become the design blueprints. Developers make no decisions about how to build the components at this stage; they simply connect elementary blocks hierarchically until they arrive at an acceptable preliminary model that conforms to the initial specifications.

Designers combine the structural and behavioral models specified in DEVS and encode them in the DEVS-Java environment. Building models in this hierarchical, modular manner permits a systems-oriented approach not possible with popular simulation languages.

As Figure 1 shows, the model base—a collection of model components—is another input into the modeling phase. The stored models serve as a repository of knowledge about existing designs. As developers modify a design, either by augmenting it or refining it, they can store new model units for potential reuse.

Simulation and verification

In our approach, simulation is a computational process that generates data in response to suitably encoded model instructions. Simulation is thus a way to verify the design specifications given by the model. Developers support this process by retrieving model test scenarios encoded in *experimental frames*, which are stored in the experimental frame base. They can then quickly set up simulation by retrieving the relevant experiments.

The idea behind experimental frames is that you can design a system from multiple perspectives. The experiments reflect these perspectives and help orient model building by drawing system boundaries and determining the model components of relevance. The experimental frame has several elements:

- It specifies when (under what circumstances) a model or the real system is to be observed and experimented with.[3]
- It defines the input data used in the experiment.

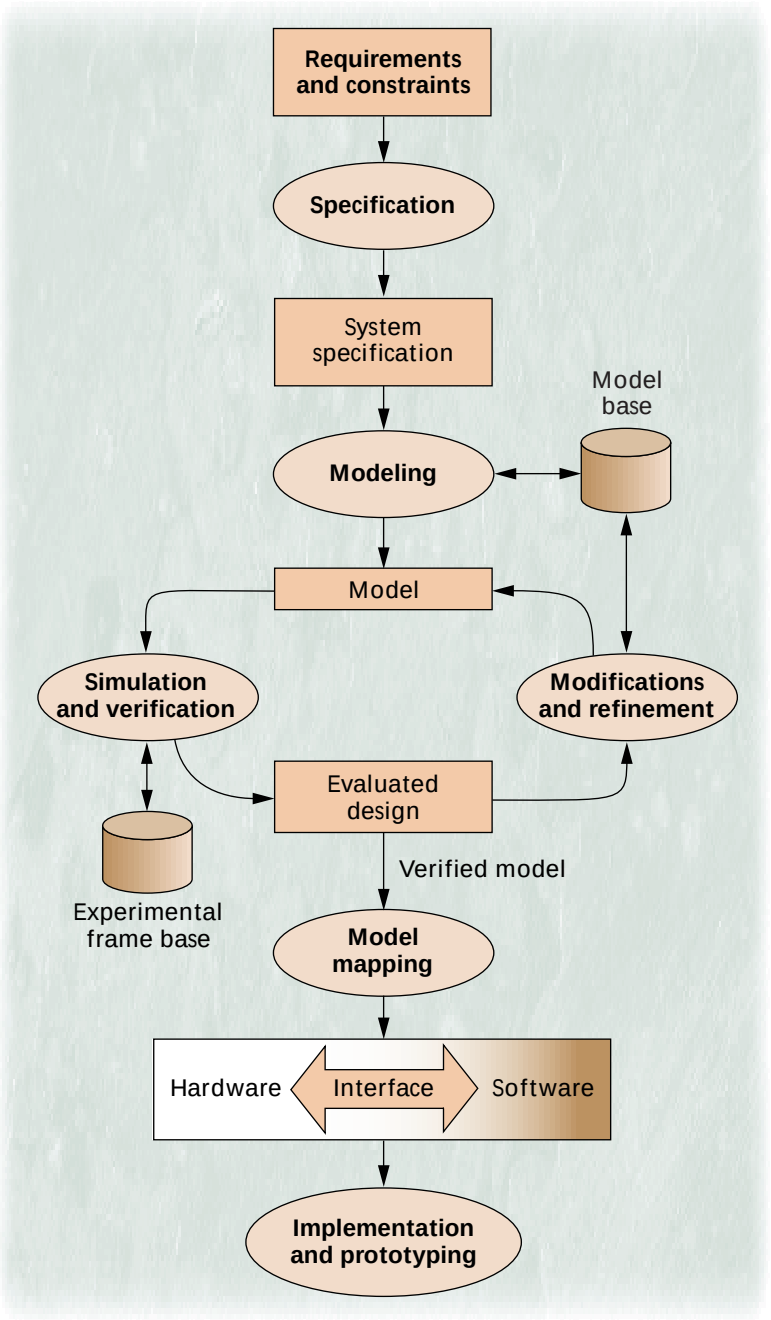


Figure 1. Phases of model-based codesign. The goal of each phase (ovals) is to provide developers with a way to independently evaluate system design through simulation-based modeling. The models are then mapped to a software-hardware architecture. The approach emphasizes a structured integration, as opposed to simply leveraging individual hardware and software tools.

We clearly distinguish among what drives the model, what is observed as its output, and the model itself. Developers can trade off design alternatives to arrive at the most suitable solution.

- It observes and collects data generated by the model.
- It controls experimentation by placing constraints on the values of the model's state variables and monitoring the constraints.

Developers use the data collected from such experiments to evaluate the proposed design solutions. By associating various experimental frames with design alternatives in the form of models, developers can trade off design alternatives to arrive at the most suitable solution.

Note that we clearly distinguish among what drives the model, what is observed as its output, and the model itself. We avoid incorporating data-gathering facilities into the actual model, which would make the model not only more complex but also unsuitable for reuse. With these distinctions, developers are free to associate a model with different experimental frames, each corresponding to a particular performance specification.

Modifications and refinement

During this phase, developers refine a validated system model into elementary submodels, couple the model's components, and define component interactions. Our framework replaces the traditional design partitioning and integration phases with stepwise refinement based on an abstract simulation model and its coupling. Developers also generate interfaces according to component interrelations they derive from the refined model.

Thus, developers refine the system model into the final, validated specification iteratively—a strategy that is especially advantageous in large-scale designs because any reallocation (or shift between hardware and software) requires numerous interface changes.

Moreover, a stepwise refinement process, in which technology is assigned late in design, fosters component reuse regardless of the technology eventually selected.

Model mapping

In model mapping, developers create an environment for subsequent prototyping by mapping simulation models onto specific components (hardware, software, interface), guided by performance estimates derived in the previous phase. In traditional design, the partitioning scheme is tied to the target architecture. In our approach, mapping model components to hardware and software is not as limited because the design is independent of the implementation until this phase, which is relatively late in design. The DEVS-Java model specification is translated into C and VHDL code. Once C and VHDL code segments have been generated, the designers realize the respective software and hardware components through virtual prototyping (which we later illustrate in the AICC application example).

The final model satisfies all imposed requirements. Our ultimate goal is to develop a *model compiler*—a system that automatically translates the simulatable model specifications into software and hardware description languages from which physical elements can be built.

Implementation and prototyping

In the last phase, the developer builds a prototype. Because prototyping is application-specific, we defer describing this phase in detail until the application section. Basically, the developer begins by searching the design space for the best implementation fit. The hardware platform is typically selected first, which serves as a reference architecture for defining the parameters of the software modules. To aid his search, the developer uses a library of hardware and software components, which helps foster the reuse of implemented, tested, and verified designs.

Three types of interfaces must also be chosen: hardware with hardware, software with software, and hardware with software. Choosing the appropriate interface ensures correct program execution, handles synchronization, and protects shared variables when components are to execute in parallel. Techniques used to implement interfaces include interrupt handling, handshake protocols, semaphores, and busy waiting.

To further preserve model continuity in the design flow, we developed the Simple Modeling Operating System, a general prototyping tool that provides software support on the target architecture and schedules the synthesized model components. SMOS is based on centralized dynamic scheduling. It is preemptive in the sense that it removes components that run past a certain timeout limit. A timer interrupt stops the component and issues an alarm. The unit then kills the process and identifies the malfunctioning component.

In SMOS, either a currently running process or incoming data can cause a model component to execute. The data is sent by either the running process or by a data interrupt triggered by an external source, such as the human-machine interface. An interrupt handler puts the data into the correct format.

When a component finally gets its turn as the running process, it sends out its process number to the hardware port, which activates the supporting hardware functions. The software body of the component then obtains from its mailbox the values necessary to execute its process body, then it executes, and then it sends out results to other components. Finally, it schedules the components to be placed on the SMOS waiting list. The process body can be implemented either as a software routine, a hardware function, or

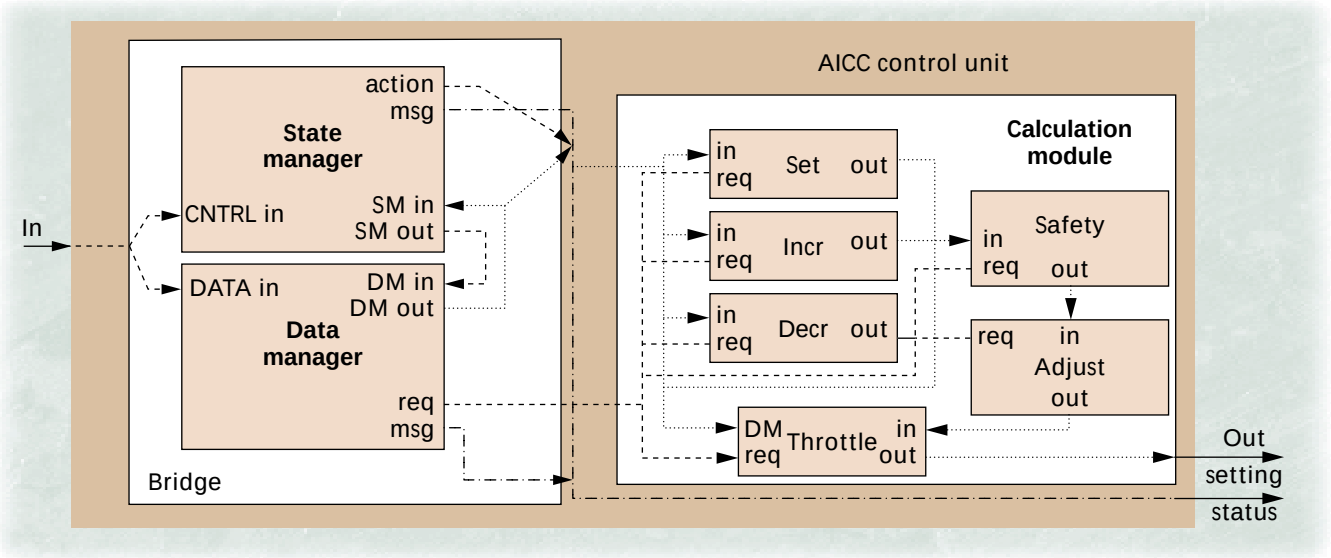


Figure 2. Final AICC model. The model shows the structure as two main components: bridge and calculation module. In the figure, DM is data manager, SM is state manager, req is request, and msg is message.

a hybrid, in which some hardware functions support and accelerate a software routine.

The final design implementation then consists of a complete description of a system to be designed, preferably using hardware and software description languages. To implement the design, developers can use commercial synthesis tools, such as those offered by Synopsis or Cadence, or C or Java compilers.

APPLICATION

The model-based codesign approach lends itself well to electronic automotive applications. The automotive industry needs reduced design cycles and is keen on saving costs. At the same time, the electronic components used in automobiles have become highly complex embedded systems. Designing hardware and software components separately is difficult to justify.

For these reasons, we elected to apply our codesign approach to the design of the autonomous, intelligent cruise controller. The AICC is part of a complex vehicle communication and control system. Our task was to design the AICC control unit.

Specification

We used recent AICC literature[4] to guide the system specification. The AICC is an extension of regular cruise control that not only maintains a fixed speed, but also adapts to the speed of the vehicle ahead. It has no lateral control. It is autonomous, in that it does not rely on communication between its vehicle and another. It can be activated for speeds above 56 km/hour only.

The *safe speed* is the maximum speed that keeps the car within a safe distance of the vehicle ahead. The *set speed* is the speed the driver requests. Thus, the circuitry of a safety unit must satisfy real-time constraints, including returning control to the driver if the system fails. The unit must keep the speed of the vehicle within ± 2 km/hour of the safe speed or set speed, whichever is lower.

Modeling

As described earlier, modeling involves creating one structural and two behavioral models (functional and dynamic).

Structural model. To develop the structural model, we analyzed the domain of automotive safety and selected an instance of the AICC. In previous work, we used an OMT (Object Modeling Technique)-like notation[5] to develop the complete structural model in an AICC instance.[6] In this article, we skip the evolution of the structural model and describe only the functional and dynamic models to demonstrate continuity from system specification to physical design.

As Figure 2 shows, the AICC has three basic elements. The *state manager* keeps track of the cruise control's state. The *data manager* collects data from the driver, computational routines, and sensors and distributes them to the state manager or to components within the *calculation module* that request it. These components process specific computations. The req port of each component in Figure 2 indicates a bidirectional communication line that handles data requests. Thus, from the informal description of the AICC control unit, we derived a structural model that has a bridge for internal and external communication and a calculation module for data processing.

Because we expected to implement the AICC control unit in a single-processor environment, we added a scheduler to our modeling environment. The scheduler executes the components' functions as a process. We needed the scheduler because, although concurrency is reflected in the simulation time, it cannot be realized in a single-processor environment. The scheduler let us develop a model that can be tested for functionality and timing constraints.

Functional and dynamic models. We defined the functional and dynamic models by converting assumptions and functional requirements into informal specifications, including

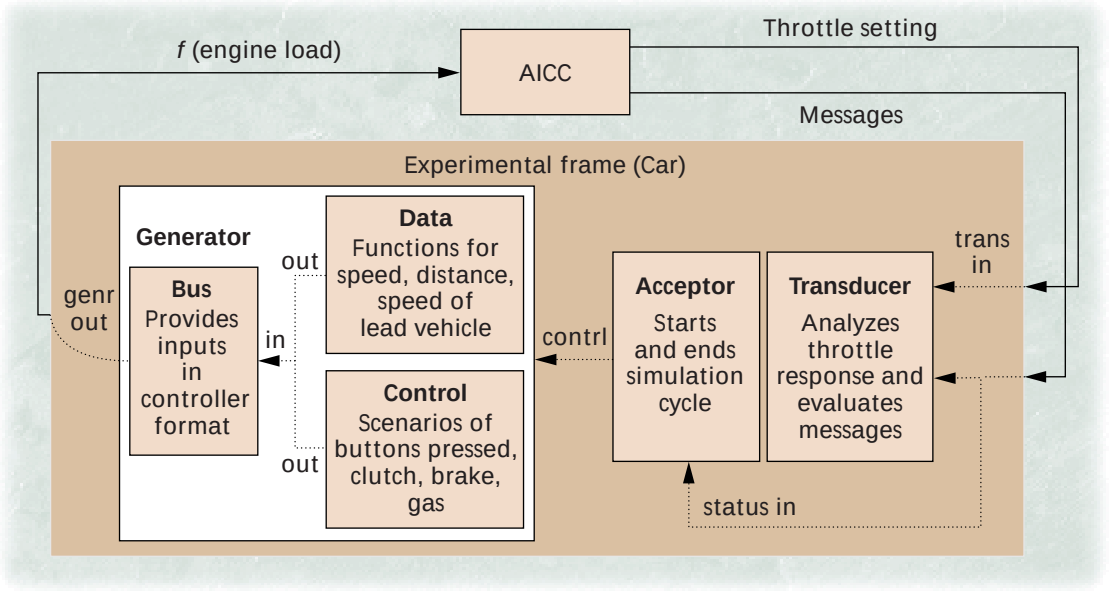


Figure 3. An experimental frame to test the AICC control unit. Experimental frames let developers easily set up simulation sessions. This experimental frame tests the unit's ability to maintain speed under varying engine loads and tests its response to interrupt signals such as braking and throttle setting. The generator *produces the input segments sent to a model, the* acceptor *continually tests the run-control segments to satisfy a set of constraints, and a* transducer *collects input/output data and computes mappings.*

- The driver transmits data using switches on the human-machine interface. These switches are typically found on a regular cruise controller.
- The control unit obtains data from internal sensors, which can include radar units and devices that measure such aspects as steering angle, momentum, speed, and throttle angle.
- Data arrives with a specific protocol at a rate that depends on the sensor type. From this data, the control unit gains information about the distance and relative speeds of surrounding vehicles.

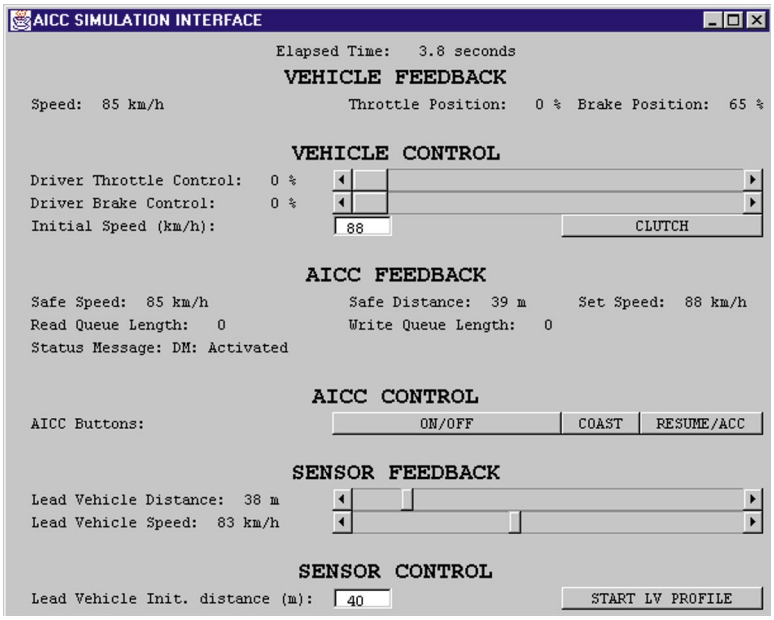


Figure 4. AICC control unit's GUI during a safety test run. The interface is based on the model of a real vehicle.

We captured these informal specifications using Statecharts.[6,7]

Specification representation. To obtain a simulatable specification, we encoded the behavior of the AICC control unit in DEVS, as described earlier.

Refinement and modifications

From the informal description of the AICC control unit, we decomposed the overall structural model into a bridge for internal and external communication and a calculation module for data processing. As Figure 2 shows, the bridge itself decomposes into a data manager, which manages data collected from sensors, and the *state manager*, which handles the control inputs of the AICC sent by the user interface.

From the requirements, we first described the AICC's functionality and dynamics—the driver must activate the cruise control by pressing the On/Off button and after that activate it by pressing the Coast button. In the DEVS model, the state manager is created and coupled to the experimental frame as well as to the corresponding calculation function ports. Because the internals of the component exhibit the required functionality, we also had to activate the correct computation functions for the respective control

data inputs. We then integrated the DEVS-Java representation of the state manager into the overall AICC model. In this way, we were able to test and assess the AICC control unit in its entirety as well as its subcomponents in simulation experiments using different road scenarios.

Simulation and verification

DEVS modules are instrumented with experimental frames, which are given concrete form. Employing the concepts of automata theory and the DEVS formalism, DEVS originator Bernard P. Zeigler[3] defines a *generator*, which produces the input segments sent to a model; an *acceptor*, a device that continually tests the run-control segments to satisfy given constraints; and a *transducer*, which collects the input/output data and computes the summary mappings. Figure 3 shows an instance of an experimental frame design to test the AICC unit's ability to maintain speed given varying engine loads and the unit's response time to signals that interrupt its operation, such as brake, clutch, throttle, and control buttons.

The experimental frame represents the vehicle, the environment in which the AICC control unit is embedded. The unit may experience different responses depending on the type of vehicle it is installed in, so not only is the car represented by an experimental frame, but there must also be a model of that car that depicts its reaction to the AICC control unit's response. The generator's output function corresponds to an engine load, which is obtained from the underlying model of the car dynamics. The AICC responds with a throttle setting. This is evaluated by the transducer, which is also directed to the model of the car. Finally, the acceptor is used to observe behavior during different cruise control states and to test for correct state transitions.

Simulation interface. Figure 4 is a snapshot of the AICC control unit's graphical user interface during a safety test run. While the simulation is running, the user can operate the various cruise controller buttons and run a profile for the driving pattern of the car ahead. The GUI is based on the model of an actual vehicle. It was created for this application to assist the designers in testing the simulation model.

The GUI brings together information about the dynamic characteristics of a car,[8] the radar control unit, and the user interface component. The car component generates speed data influenced by the throttle setting while the control signals from the GUI are passed directly to the AICC control unit. In addition, the designer can generate data values from a radar control unit to simulate different scenarios when approaching lead vehicles. Before the data arrives at the AICC unit, it passes through the bus controller component. This emulates a real-world application in which the board must be connected to the vehicle's communication bus.

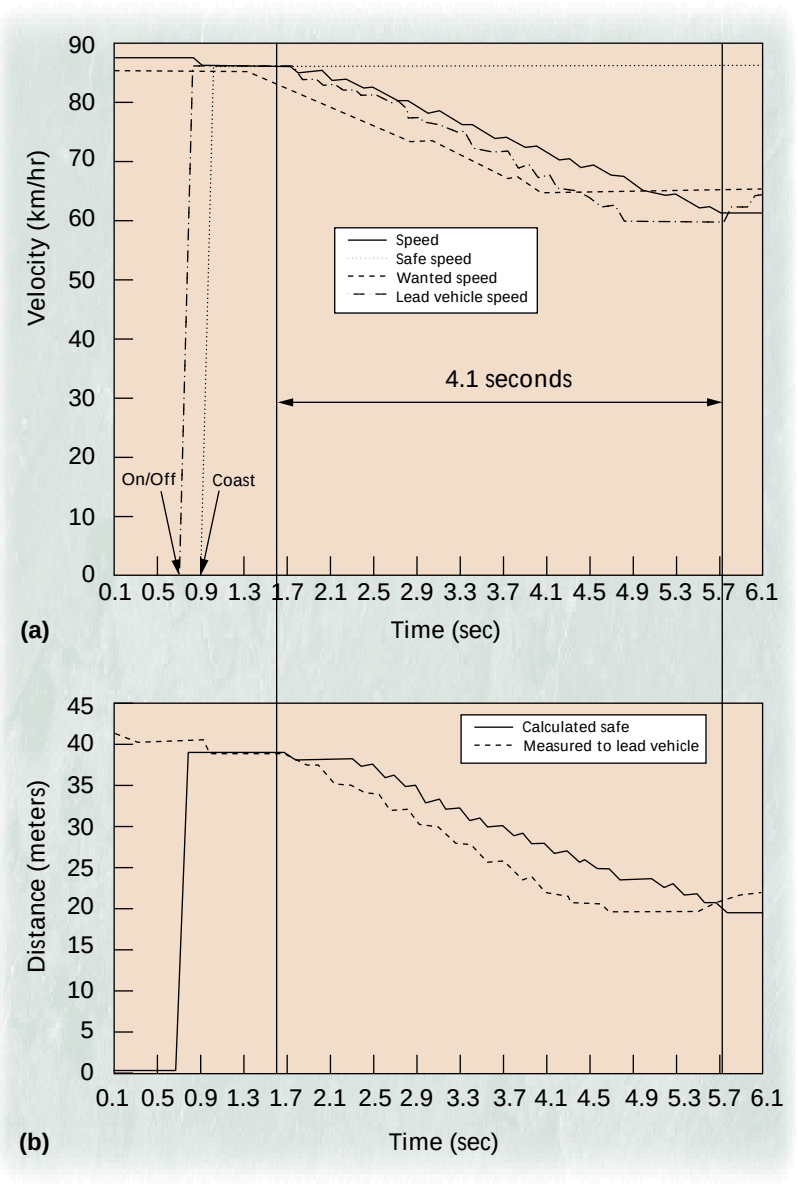


Figure 5. Simulation results of a safety test run. In (a), the test is measuring the ability of the vehicle with the AICC to adjust to the speed of the lead vehicle. In (b) the test is measuring the elapsed time to adjust relative to the safe distance parameter. The AICC is constrained to always be within 2 km/hour of the lead vehicle speed or set speed, whichever is lower. Here the lead vehicle slows to 65 km/hour after the driver of the AICC vehicle has set the cruise speed to 86 km/hour. The AICC vehicle takes 4.1 seconds to adjust its speed to stay a safe distance from the lead vehicle.

Verification. During this phase, we used experimental frames like the one in Figure 3 to generate data from the bus controller to a microcontroller and analyze output from the control unit. Figure 5 shows a subset of the simulation results from verifying a safety test run of the AICC control unit. In this sce-

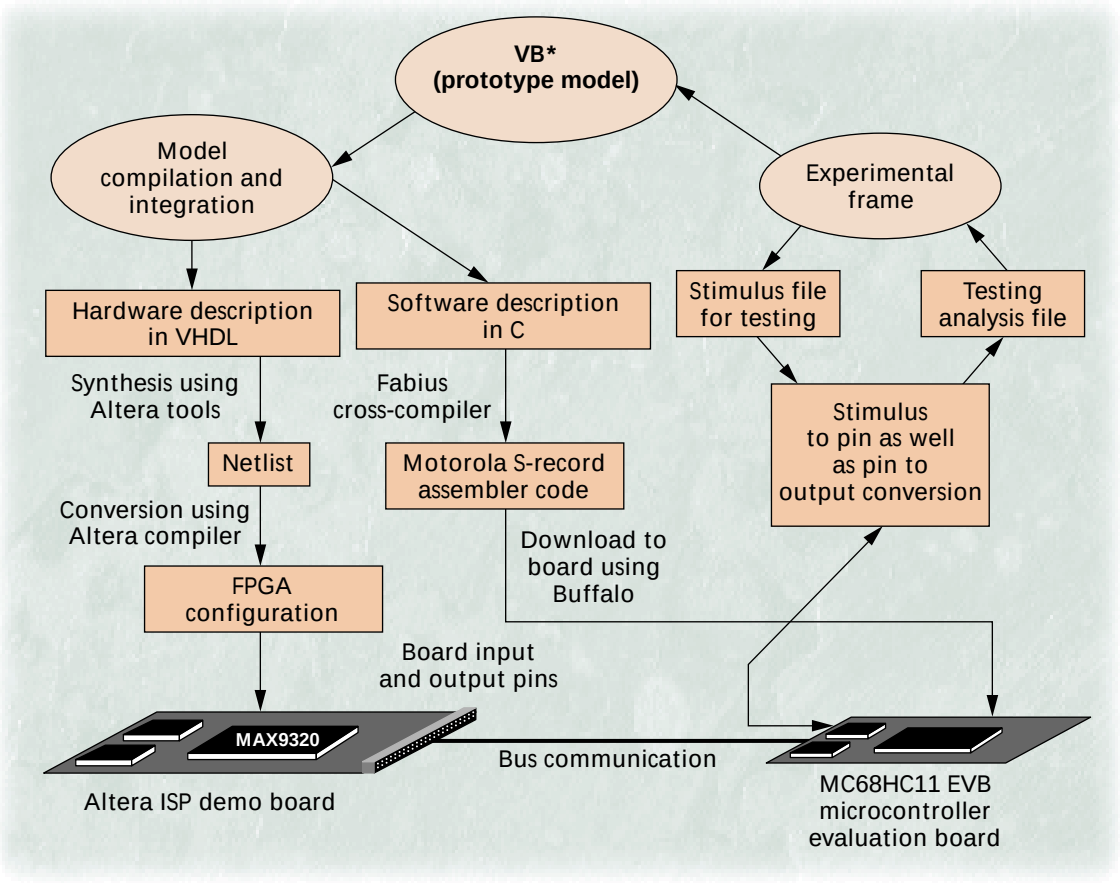


Figure 6. Prototyping of the AICC application, from the final model (VB) to the reference architecture (MC68HC11 microcontroller and Altera ISP board).*

nario, the two cars start out with the same set speed. The car under test then engages the AICC at 0.7 second and sets the wanted speed to 86 km/hour. Initially, the vehicle speed remains in the specified range of ±2 km/hour (safe environment). The lead vehicle slows to 65 km/hour. The vehicle with the AICC slows down accordingly, violating the safe distance constraint slightly for 4.1 seconds. Once a safe distance is established, the car tries to approach the same speed as the lead vehicle. The test case also shows the correct response of the control unit to the corresponding AICC control buttons.

This is just one of many possible simulation exercises. Experimental frames let developers flexibly and rapidly assess the design specification.[9] Because our approach separates the model description from the specification of its simulation experiment, a new simulation test simply requires attaching a new experimental frame to the model.

Model mapping

We called the final model VB* (virtual board). As described earlier, we interactively partitioned VB* by searching the design space for the best implementation fit. Our search was limited by hard implementation constraints. For example, only a certain chip was available for part of the design, so our choice of bus controller was limited to this chip. The rest of the assignment was guided mostly by the timing parameters we got from model simulation.

Implementation and prototyping

We built the VB* component by component, from bottom to top, by verifying real-time timing constraints. Figure 6 shows the reference architecture we eventually implemented: a Motorola MC68HC11 microcontroller, which we chose because it is commonly used by the automotive industry for engine control applications. We realized the hardware functions on an Altera Max9320 FPGA (field programmable gate array). The synthesis tool for this chip can process either VHDL descriptions or graphical designs to create the desired circuits. The two chips interface via a 16-bit communication bus. Because I/O pins were not always available, we multiplexed data transfers to increase communication bandwidth.[10]

Our model-based approach has proved successful in several ways. First, we were able to produce the AICC control unit directly from the model-based specification, validating our claim that the framework supports model continuity. The resulting code is very compact (800 lines of C), and the hardware platform is small (an 8-bit microcontroller and a 320-cell FPGA).

We plan to continue developing model-refinement techniques and define the model granularity levels necessary for optimized model mapping and implementation. We also plan to implement heuristic techniques to obtain the optimized model mapping by assigning the model components to hardware, software, and

interface components. As a basic research issue, we will investigate the real-time scheduling issues that arise when mapping the model-based specifications onto a mixed hardware-software implementation. Finally, we plan to develop libraries of basic hardware, software, and interface components for the AICC. In this way, we hope to demonstrate general techniques for reusing models. ❖

Acknowledgments

This work is supported by Siemens AG, Central Research and Development Laboratories, München, Germany. We thank Joe Hanson at the Altera University Program for providing the Altera ISP demo board and corresponding development software.

References

1. G. De Micheli and R. Gupta, "Hardware/Software Co-Design," *Proc. IEEE*, Mar. 1997, pp. 349-365.
2. *Codesign: Computer-Aided Software/Hardware Engineering*, J. Rozenblit and K. Buchenrieder, eds., IEEE Press, New York, 1994.
3. B. Zeigler, *Multifacetted Modelling and Discrete Event Simulation*, Academic Press, London, 1984.
4. U. Palmquist, "Intelligent Cruise Control and Roadside Information," *IEEE Micro*, Feb. 1993, pp. 20-28.
5. J. Rumbaugh et al., *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, N.J., 1991.
6. S. Schulz, J. Rozenblit, and K. Buchenrieder, "Towards Model-Based Codesign: An Intelligent, Autonomous Cruise Controller Application," *Proc. Conf. and Workshop Eng. Computer-Based Systems*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 73-80.
7. D. Harel, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Trans. Software Eng.*, Apr. 1980, pp. 403-414.
8. P. Ioannou and C. Chien, "Autonomous Intelligent Cruise Control," *IEEE Trans. Vehicular Technology*, Nov. 1993, pp. 657-672.
9. M. Mrva, M. Heuchling, and W. Ecker, "The Shall-Prototype-Test Development Model," *Proc. Conf. and Workshop Eng. Computer Based Systems*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 385-391.
10. S. Schulz et al., "A Prototyping Environment for Model-Based Codesign," *Proc. Conf. and Workshop Eng. Computer-Based Systems*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 145-150.

Stephan Schulz is a PhD candidate in electrical and computer engineering at the University of Arizona. His research interests include embedded systems applications, model-based design, real-time operating systems, continuous and discrete-event simulation, and hardware-software codesign. He received a BS from the State University of New York, Binghamton, and an MSc in electrical and computer engineering from the University of Arizona. He is a student member of the IEEE.

Jerzy W. Rozenblit is a professor of electrical and computer engineering at the University of Arizona. His research and teaching are in complex systems design and simulation modeling. He received a PhD and an MSc in computer science from Wayne State University and an MSc in computer engineering from the Technical University of Wroclaw. He is editor of Computer*'s Integrated Engineering department and an associate editor of* ACM Transactions on Modeling and Computer Simulation *and* IEEE Transactions on Systems, Man and Cybernetics. *He is a senior member of the IEEE and the Society for Computer Simulation and a member of the ACM.*

Michael Mrva is head of the High-Level Design Techniques Department at Siemens AG and an adjunct professor in electrical and computer engineering at the University of Arizona. His research interests include hardware-software codesign, high-level specification and verification of hardware systems, object-oriented codesign, VHDL-based modeling, and hardware design reuse. He received a PhD in mathematics from the University of Vienna. He is a member of Gesellschaft für Informatik.

Klaus Buchenrieder heads research in hardware-software codesign at the Central Technology Laboratories of Siemens AG. He received a Dipl.Ing. in electrical engineering from the Fachhochschule in München, and an MS and a PhD in computer science from Ohio State University. He is the founding chair of the Codes/Cashe workshop series on codesign and of the Consyse workshop on conjoint systems engineering. He is also a professor of computer science at the University of Tübingen and an adjunct professor of computer and electrical engineering at the University of Arizona.

Contact Rozenblit at the University of Arizona, ECE Dept., Tucson, AZ 85721-0104; jr@ece.arizona.edu. Contact Mrva at Siemens AG, ZT ME 5, Otto-Hahn-Ring 6, 81730 Munich, Germany; michael.mrva@mchp.siemens.de.