
ELECTRONIC-SYSTEM DESIGN IN THE AUTOMOBILE INDUSTRY

ELECTRONIC COMPONENTS ARE NOW ESSENTIAL TO CONTROL A CAR'S MOVEMENTS AND CHEMICAL, MECHANICAL, AND ELECTRICAL PROCESSES; TO PROVIDE ENTERTAINMENT AND COMMUNICATION; AND TO ENSURE SAFETY. A NEW, PLATFORM-BASED METHODOLOGY CAN REVOLUTIONIZE THE WAY A CAR IS DESIGNED.

**Alberto Sangiovanni-
Vincentelli**
University of California at
Berkeley

..... With the advent of highly powerful microprocessors, the explosion of wireless communication, and the development of new generations of integrated sensors and actuators, the way electronic products are conceived, designed, and implemented has undergone a revolution. In addition, the electronics industry is undergoing a major restructuring that favors horizontal integration and vertical disintegration. In this framework, collaboration among different industry segments is essential to bringing new products to market. In particular, system companies are shifting electronic-component research and development costs to semiconductor companies, which therefore must significantly increase their system design competence. Recent International Business Strategies market studies show that more than 50 percent of design activities that move to the 0.09-micron technology node will be in software.¹ Thus, the semiconductor design problem becomes a system solution problem.

Today, European automobile manufacturers provide specifications to first-tier subsystem suppliers such as Bosch, Siemens, and Magneti-Marelli, which design software and hardware subsystems that include mechanical

parts such as injectors and throttle bodies. These subsystems contain ICs from second-tier suppliers such as Motorola, Texas Instruments, Hitachi, and ST Microelectronics. They also contain intellectual property (IP) from various second-tier suppliers such as the WindRiver and ETAS software companies. In general, the subsystem volumes are large, cost being a major driving force.

Once car manufacturers receive the subsystems, they must integrate them in the car and then test the overall system. If they detect errors through extensive testing, which includes driving under extreme conditions, they initiate a chain of engineering changes that often causes major delays in the design process. The problems are traceable to software errors, misunderstanding of the specifications, and unpredictable side effects of interconnecting the subsystems. This design process loop is particularly painful because it occurs when the car is almost ready for its market launch.

Car manufacturers increasingly realize the importance of electronics in their business. According to Daimler-Chrysler sources, more than 90 percent of the innovation (and hence value added) in a car is in electronics. Accord-

ing to BMW, electronic components comprise more than 30 percent of a car's manufacturing cost. The trend in the car manufacturing industry is to acquire more in-house electronics competence to capture added value that previously went to subsystem suppliers. The strategy calls for software and hardware standards that will facilitate plug-and-play subsystems, reducing the strategic importance of any single subsystem supplier. The Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug (open systems and corresponding interfaces for automotive electronics), or OSEK, operating system requirements are an example of this policy.² Clearly, however, without an overall understanding of the interplay of subsystems and the difficulties of integrating highly complex parts, system integration is increasingly a nightmare for car manufacturers. In addition, subsystem suppliers are trying to enlarge their perimeter of competence to capture more added value.

Automobile electronics comprises three basic domains:

- power train management—for example, electronic control units (ECUs) that control ignition timing and the amount of fuel injected into the cylinders;
- body electronics—for example, ECUs that control dashboard displays, suspension settings, and temperature; and
- information processing, communication with the outside world, and entertainment (often called the telematics or infotainment system).

The first domain is typical of any transportation system and is characterized by tight safety and efficiency constraints. Its core competence is control algorithms, along with software and mechanical-electrical hardware design and implementation.

The body control domain involves the management of a distributed system that increasingly resembles a network with protocols likely to have different requirements than standard communication protocols. Guaranteed services are the essence of this domain.

A car's infotainment system is the product of industrial domains that are progressing in the technology race at a faster rate than the automotive domain. This domain can reap

the most short- to medium-term profits. Customers now often base buying decisions on the infotainment environment more than engine performance and handling. Hence, the question arises: What will constitute a car company's core competence? Will the electronic components be the car and the mechanical components an accessory?

For car manufacturers, system design is definitely the most important technology they must master to improve the quality and increase the value of their cars' electronic components. Designers of automotive electronic systems need a methodology that focuses on two main principles: separation of concerns and platform-based design.

Electronic-system design issues

To support the electronic-design chain, system designers in the automobile industry must establish a new design flow. Clean interfaces and unambiguous specifications are essential parts of this design flow. In addition, the design flow must address the thorny issue of IP protection. This is even more important in the automotive domain than in other industrial segments because the automotive-supplier chain is deeper.

The following issues are likely to determine the preferred approaches to the design and implementation of complex embedded systems in the automotive domain (and others):

Reuse. Design time and cost will dominate system designers' decision-making process. Therefore, design reuse of all kinds, as well as just-in-time, low-cost design debugging techniques, will be highly important. Design flexibility is essential to mapping an ever-growing functionality onto a continuously evolving set of associated hardware implementation options.

High levels of abstraction. Designers must capture designs at the highest abstraction level to exploit all the available degrees of freedom. This abstraction level should not distinguish hardware from software, since this distinction is the consequence of a design decision.

Concurrency. The implementation of efficient, reliable, and robust approaches to the design, implementation, and programming of con-

current systems is essential. Whether implementing the silicon as a single, large chip or a collection of smaller chips interacting across a distance, designers must deal with concurrent processing and communication in a uniform and scalable manner. In any large-scale embedded system, designers must consider concurrency a first-class citizen at all abstraction levels and in both hardware and software.

Separation of concerns: communication and behavior. Concurrency implies communication among design components. Communication is too often intertwined with design component behavior, making it difficult to separate the two. Separating communication and behavior is essential to handling system design complexity. It is difficult to reuse components if their behavior depends on communication with other components of the original design. In addition, because designers can describe communication at various abstraction levels, they can potentially implement communication behavior in many forms according to the available resources. Today, designers seldom exploit this freedom.

Multiple chips. Next-generation systems will probably use a few highly complex (Moore's law-limited) part types, and many more energy- and power-efficient, medium-complexity chips (with 10 million to 100 million gates in 50-nm technology). All these chips will work concurrently to solve complex sensing, computing, signaling, and actuating problems.

Platform-based design. System developers will most likely develop these chips as instances of a particular platform. That is, rather than assembling the chips from a collection of independently developed blocks of silicon functionality, system developers will derive them from a specific microarchitecture family, or platform, possibly aimed at a particular class of problems. System developers can modify (extend or reduce) these platforms. The platforms will support extensibility mainly through the use of large blocks of functionality (for example, in the form of coprocessors), but they will support extensibility in the memory and communication architecture as well. When selecting a platform, designers must consider cost, size, energy consumption, and

flexibility. A platform has much wider applicability than an application-specific IC (ASIC), so design decisions are crucial. A less than excellent choice can result in an economic debacle. Hence, design methods and tools that optimize the platform selection process are very important.

Software programmability. Platforms will be highly programmable at various granularity levels—at instruction level for microprocessors, at gate level for field-programmable gate array (FPGA) blocks. Consequently, mapping an application into a platform efficiently will require a set of software design tools that resemble logic synthesis tools. This will be a fruitful research area.

Current automotive system-level design

Automobiles' real-time and safety-critical electronics systems are implemented as distributed architectures that typically include several ECUs communicating via one or more networked broadcast buses. These buses are controlled by communication protocols such as controller area network (CAN), time-triggered protocol (TTP), local interconnect network (LIN), and FlexRay.

Each ECU includes

- application and diagnostic software;
- base software—for example, real-time operating system (RTOS) and communication layers;
- one or more microcontrollers with local memories and bus controllers with one or more channels, to support redundancy for fault-tolerant systems and complex bus architectures such as constellations and star couplers; and
- optional dual-ported RAMs for communications between bus controllers and microcontrollers and between CPUs within the same ECU.

Automotive applications (such as control systems for steering and braking) have introduced a new design dimension—the system's distributed nature—that adds complexity yet also provides the potential for optimizations such as reducing the number of ECUs needed. In fact, better use of each ECU potentially reduces the number of ECUs in a distributed

architecture. (Redistribution is not always possible because some applications tie the software to a specific ECU.) In a nutshell, the design problem for automotive applications consists of distributing a pool of functions over the target architecture to satisfy cost, safety, and real-time operating requirements. Because these applications are distributed, designers must accurately model the communication protocol. A by-product of the methodology described here is that designers can experiment with new protocol configurations.

Figure 1 illustrates the typical “V” design flow for automotive distributed systems. The development process starts with the functional system analysis phase, in which the system designer develops a functional network (the overall system behavior). The system design partitioning phase determines the distribution of functions in an architectural network. In the software design specification phase, the system designer defines algorithms for each functional component. The implementation phase maps a composition of functional components onto the target hardware. The upward part of the “V” design flow represents the verification process, including testing the implemented architecture’s software integration, verifying the communication subsystem, and calibrating the system in the car.

Specification

The car manufacturer is responsible for overall functionality, whereas the first-tier suppliers deliver control algorithms and hardware. The carmaker specifies system functionality on the basis of an overall analysis of desired car performance and features and decomposes this functionality into subsystem specifications sent to first-tier suppliers. Expert designers perform this decomposition, using their experience and sometimes prototypes (lab cars). They almost always write specifications informally in natural language in a contract.

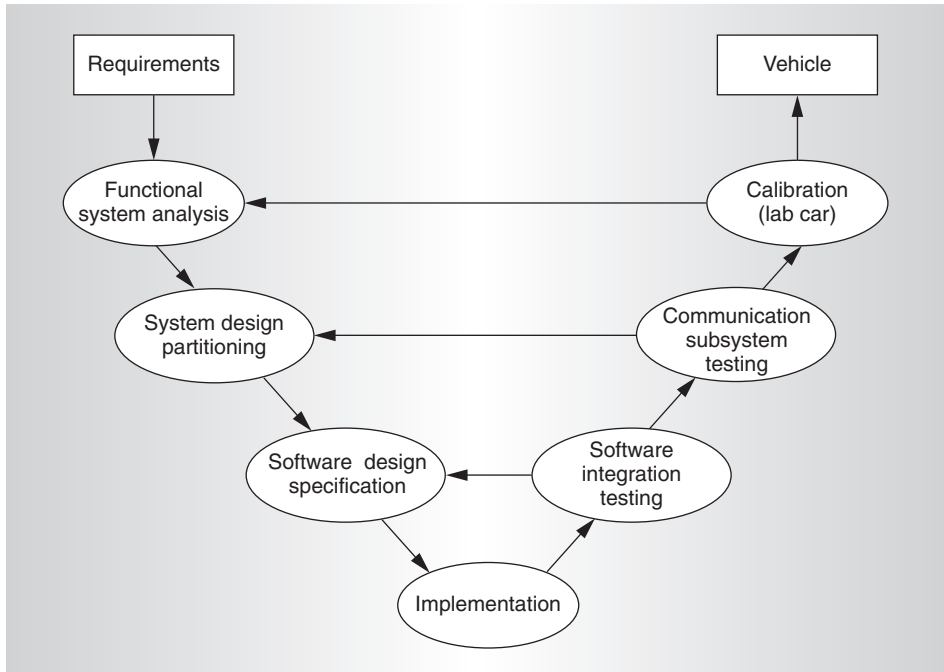


Figure 1. Typical “V” design flow for automotive distributed systems.

The first-tier suppliers analyze the specifications and negotiate the terms of the contract. The car manufacturer’s specifications might also include implementation requirements, thus restricting the suppliers’ design space. For example, sometimes the contract lists particular microcontrollers to use. In addition, there is a growing trend for car manufacturers to use internally developed software instead of relying fully on first-tier suppliers. To ease integration, communication standards such as TTP and FlexRay provide clean semantics and guaranteed behavior. An OSEK-compliant operating system also eases integration.

Without a rigorous design methodology that can deal with heterogeneity, specifications at different abstraction levels always cause problems. For first-tier suppliers, integrating other suppliers’ software modules is a severe problem, especially for hard real-time systems.

Algorithm development

For safety-critical applications, the design of control algorithms that satisfy the functional requirements is a critical step for both car manufacturers and first-tier suppliers. Recently, manufacturers and first-tier suppliers have developed control algorithms using

Table 1. Characteristics of embedded software for automotive subsystems.

Characteristic	Power train unit	Body gateway	Instrument cluster	Telematic unit
Memory (Kbytes)	256	128	184	8,000
Lines of code	50,000	30,000	45,000	300,000
Productivity (lines per day)	6	10	6	10*
Residual defect rate (ppm)	3,000	2,500	2,000	1,000
Rate of subsystem change (years)	3	2	1	<1
Development effort (staff-years)	40	12	30	200
Validation time (months)	5	1	2	2
Time to market (months)	24	18	12	<12

* C++ code

languages such as C or mathematical equations. Typically, designing an algorithm requires modeling the relevant part of the environment—the part the algorithm will control—and abstracting the behavior of the remaining part of the system.

The result of the software design specification phase is an algorithm described as a single block or a hierarchical subnetwork. Algorithm development proceeds in either a top-down (designing new algorithms) or bottom-up (use of previously defined IP) fashion. Given the same system requirements, different algorithms can correctly implement the system functionality. Designers explore these different solutions during this phase. Using functional design tools such as the MathWorks toolset (Matlab and Simulink)³ to capture the algorithms and perform simulation on a mathematical model is a growing trend.

Implementation

Designers implement the algorithms in a selected architecture as software modules or hardware components. Architecture selection is often an ad hoc process based on experience and extrapolation from existing products. Selecting the ICs for an ECU involves a search among IC providers active in the automotive arena. The selection often depends more on commercial relations among companies than on a technical assessment of performance-price ratio.

If the architecture has problems meeting the constraints, designers can adjust it during the design phase. Developers provide new software needed for novel features by “growing” it over existing modules. Starting from an existing, proven module and tweaking it to provide the

new features limits the risk of malfunction. Extensive experimentation on rapid-prototyping systems or actual cars is the preferred way to verify the system’s correctness.

Software architectures are often old-fashioned and difficult if not impossible to port from one platform to another. The software is not cleanly partitioned into application code, communication, design drivers, and basic I/O system (BIOS). The exponentially growing complexity of features for software implementation makes the problem of embedded-software design a serious obstacle to new car development. Table 1 presents a typical set of productivity and quality indicators for automotive embedded software.

The most advanced first-tier suppliers have restructured their code so that porting is no longer difficult and expensive, thus opening new possibilities for cost reduction and performance improvement. In addition, many of these companies are using capture tools such as Simulink, Statecharts, and ASCET-SD (Advanced Simulation and Control Engineering Tool-Software Development)⁴ for automatic code generation from algorithmic specifications given in structured form. Although automatic code generation solves the problem of designing software that correctly represents a given functionality, it does not solve the timing problem. The code’s timing depends on the tasks to be handled by the RTOS, the scheduling policy, and the ECU’s performance. Several companies offer scheduling-analysis tools, which compare different scheduling policies (for example, cooperative and preemptive versus preemptive only) to determine a usable software architecture. Designers can perform static scheduling-policy

analysis offline—for example, using rate monotonic analysis—or dynamic analysis online via interactive simulation. The analysis relies on time budgets (task periodicity and task execution times) provided by the designer.

Integration

Once the first-tier suppliers deliver their subsystems, the carmaker integrates them in the car. This step would be difficult without tools that help analyze a subsystem's behavior and performance before a prototype car is available. Such tools are mainly company internal. For example, the BMW flow exports design data to a proprietary database, Boardnet, a customized version of the Oracle database. Designers then use the Boardnet data to configure downstream tools for emulation and measurement of the communication protocols (for example, a TTP-cluster prototype board).

Calibration

In the calibration phase, designers or engineers tune a subset, or calibration set, of the behavior IP's control and regulation parameters (typically, control algorithms) to obtain the controlled system's required performance. This phase also involves tuning the overall system functionality parameters. Designers define the calibration set by selecting the tunable parameters (characteristic values, curves, and maps) during the control algorithm specification phase.

Calibration performed on test cells (instrumented benches) and test tracks (autodromes where drivers test prototype cars) is a very expensive process. Today, calibration engineers are more numerous than designers, an indication of the state of the design methodology currently in use. Expensive tools to facilitate calibration are available from companies such as ETAS and dSpace.

Problems

This design flow poses several problems:

- *Lack of continuity.* There is a communications gap between requirements analysis and functional network definition, and between software development and overall architecture netlist definition.
- *Long turnaround time.* Designers validate the solution only on the car or (at best)

with prototyping hardware very late in the design cycle. Software development can start only when a hardware prototype is available, and then only for a single ECU.

- *Suboptimal and overly conservative solutions.* Because the flow supports a per-ECU design style, design exploration concentrates on different scheduling policies, not on the overall distributed system, including communication protocols.

Obviously, these problems seriously affect development and production costs. According to OSEK:

Vehicle manufacturers traditionally focus on production cost rather than on development cost—the sensors and the actuators, along with the bare ECU, represent almost the entire cost for electronics in the car. However, although software does not have a “production” cost, it is not for free! The software development costs are skyrocketing: today, they are about twice as much as the development costs for hardware.²

Overall system design exploration is possible only if the integration step takes place at the virtual level, not on the car as is the current method. Indeed, the entire automotive industry is trying to move tests from cars to labs that can emulate or simulate real conditions at a much lower cost. (The cost of setting up an experiment on a car is \$120 to \$500 per hour. The setup time is about one hour. Engineers can perform about two tests each day.)

Using a virtual environment rather than prototyping hardware for design and test can significantly reduce development and production costs. If designers could simulate the distributed application on their host workstations rather than a test track, they could repeat tests after every design change. Flexibility is another advantage: A virtual environment would more easily support derivative designs (variants), and waiting for the next hardware prototype to run the application software would become unnecessary. Hence, car manufacturers would achieve their time-to-market and cost reduction goals. According to a BMW management source in a personal communication:

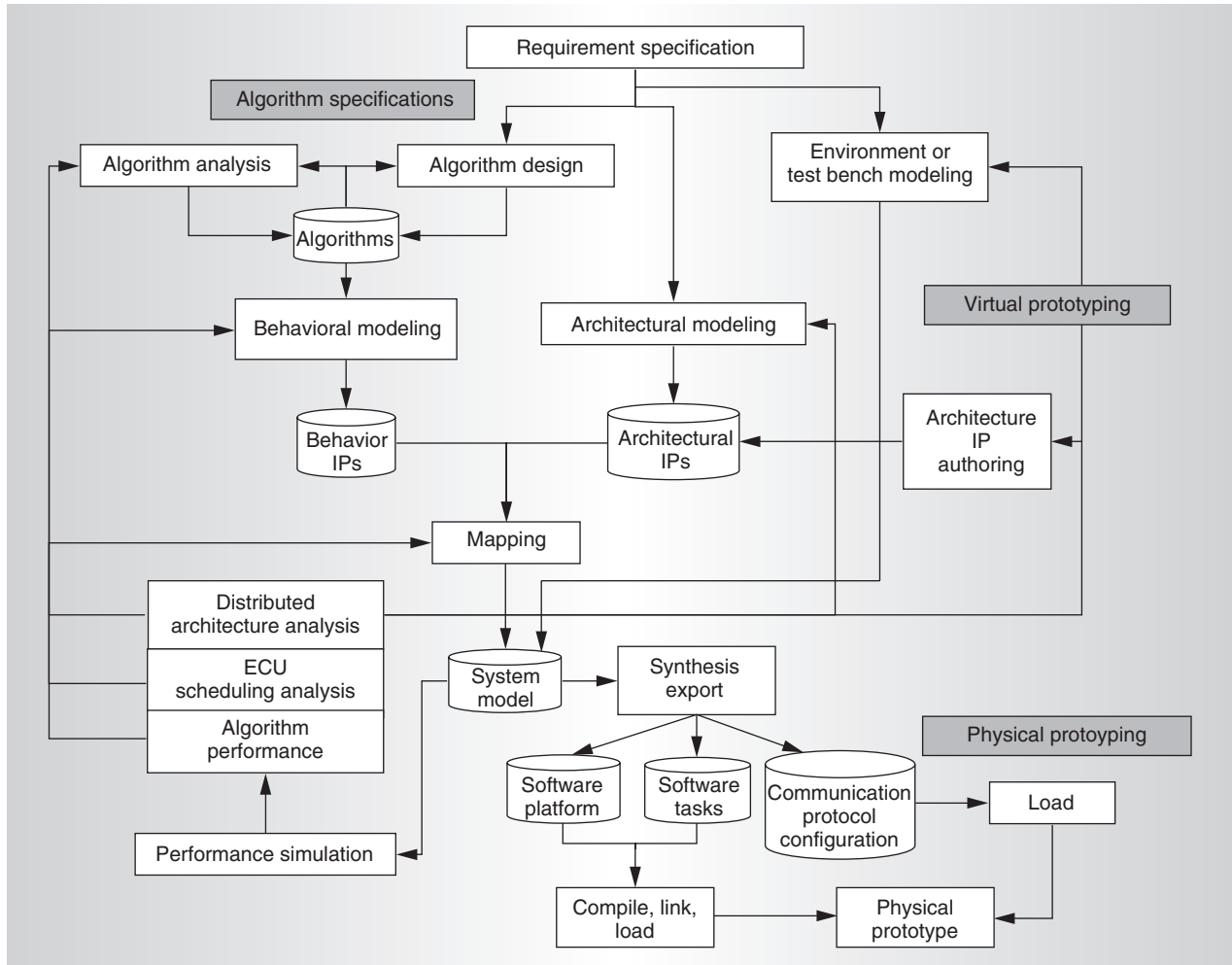


Figure 2. Design flow of the new methodology.

One of the focuses and values of a system-level design methodology and tool set is that redundancy and fail-safe system tests can be repeated after every change in the design. However, a valuable use of any methodology and tool set is only possible if interfaces to the approved and existing BMW development methods and tool chains (from specifying functionality to implementing it onto an ECU) are supported by the flow.

This comment summarizes well why a design methodology must accommodate existing tools that have become de facto standards.

Finding design errors and near-optimal functional networks as early as possible in the design process requires novel design methodologies and integrated tool environments that

embody the concept of virtual integration platforms.⁵ (A functional network includes the overall system functionality with the definition of subsystems and their interfaces independent of the target architecture and hardware and software architectures.)

New design methodology

Figure 2 illustrates a new design methodology developed by automotive and tool companies and academic institutions including BMW, Cadence, ETAS, dSpace, the Project for Advanced Research of Architecture and Design of Electronic Systems (Parades), Magneti-Marelli, and the University of California at Berkeley.⁵ The methodology includes three main steps: algorithm specification, virtual prototyping, and physical prototyping. It assumes that the designers, given an informal

specification of the subsystems, can specify the requirements in a semiformal language such as the Unified Modeling Language (UML). The overall behavior (functional network) and architecture netlist of the distributed system constitutes the output of this specification phase.

This methodology supports the entire automotive electronic design chain. Carmakers specify the overall system to the first-tier subsystem suppliers, who develop algorithms and implementations in software and hardware platforms. The second-tier suppliers, semiconductor and software companies, develop components of the hardware platform and software. Carmakers use the virtual platform of the overall electronic system to validate their partitioning and specification, and subsystem suppliers use it to ensure that their components work as expected when integrated.

In this methodology, the model-based software design approach is important. Through synthesis export, a model described in functional terms migrates to an efficient software implementation. The subsystem supplier develops most of the software, but carmakers are developing an increasing amount of critical software and can benefit substantially from this approach.

The new methodology's key distinctions are its use of the following:

- *Virtual platform.* A virtual platform supports system testing and prototyping (hardware-software architecture) via simulation.
- *Virtual application models.* Virtual models of the application software and the target hardware-software architecture (bus controllers, CPUs, RTOS schedulers, and communication protocols) let designers create a virtual prototype of the entire distributed application. Designers import the application software models from previous designs or write new software for the system under development. Designers develop the architectural models within the virtual models (for example, a communication protocol model), using a standard C++ application programming interface.
- *Other virtual models.* Besides virtual application models, virtual models of the

environment, of complex human-machine interactions, and of test benches that provide stimuli to the system under test are also necessary. Designers either import these models from tools such as MathWorks Simulink or author them within the system.

Advantages

The new methodology makes a major shift to an integrated design style in which designers model the entire ECU network along with the application and base software used to customize the platform for a particular car series.

Integration takes place at the virtual level. Automatic configuration of tools for protocol analysis and implementation is based on the results of simulations of the virtual model. For example, once a designer has decided how to distribute the pool of functions among the ECUs, a downstream code generation tool can use this information (number of tasks needed, scheduling policies, and so on) to generate the RTOS scheduler. At the same time, designers configure the downstream tools for communication protocol analysis, using the configuration data determined at the virtual level (type of protocol, frame packaging, communication cycle, and redundancy management policies). Thus, a step that is currently manual or that requires intensive user intervention (for example, the designer must explicitly specify messages sent over the network bus) is automatic in the new flow.

In the new methodology, timing estimation takes place in the earliest design phases, before implementation. Typical estimates are for software task execution times and network communication latencies. These estimates might shorten algorithm and platform (single ECU or network) exploration considerably. Estimation models can be provided by the subsystem supplier or developed by the system designer for use as a specification to the supplier.

Platform-based design paradigm

Figure 3 depicts the core of the new methodology as an instance of platform-based design, a paradigm shift in design, verification, and testing. The platform-based design paradigm is a meet-in-the-middle approach. It leverages the power of top-down methods and the efficiency of bottom-up methods. The

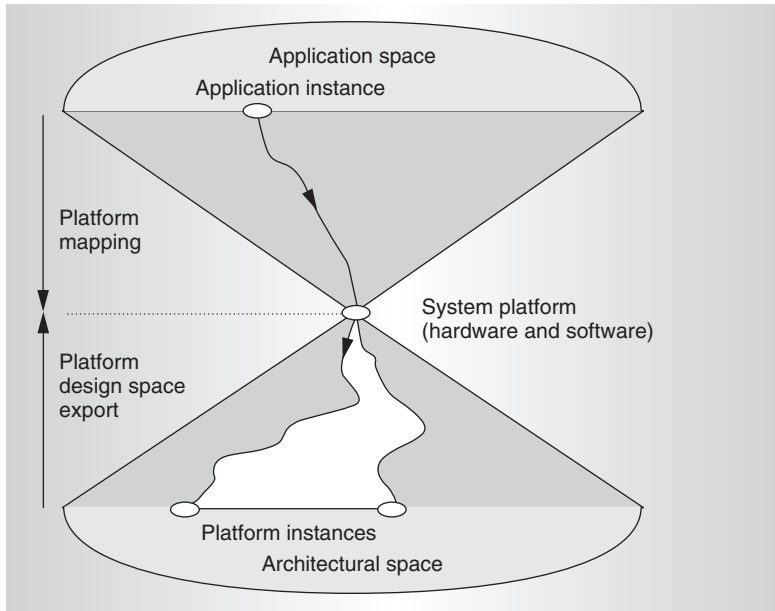


Figure 3. Platform-based design.

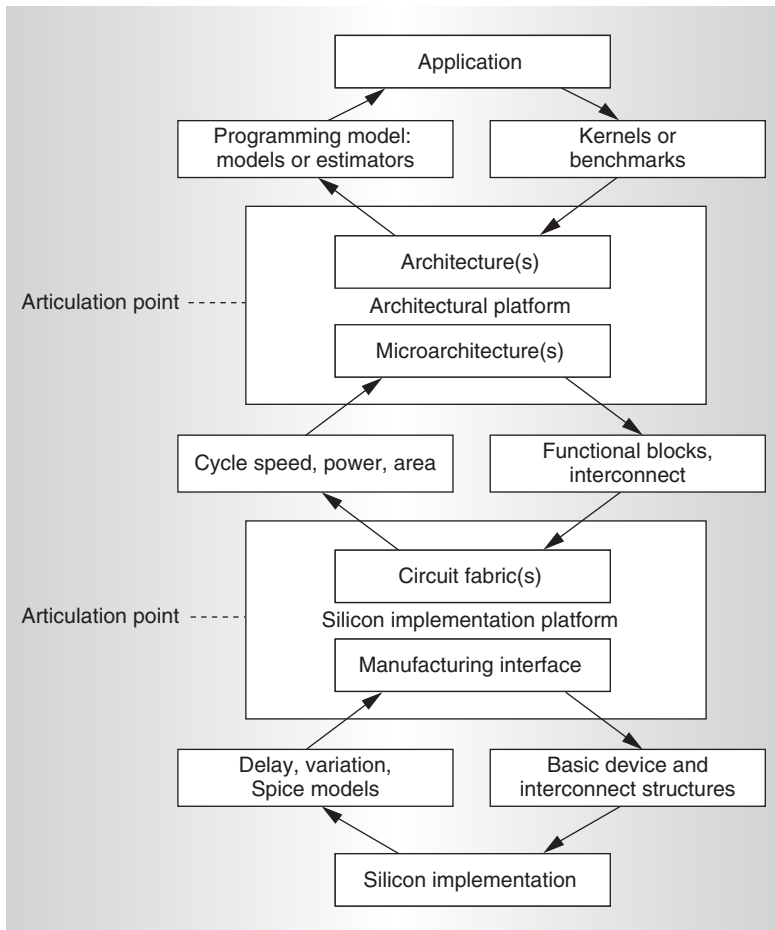


Figure 4. Platform stack.

design process is the stepwise refinement of a specification into a lower-level abstraction chosen from a restricted library, or platform, of available components. Components are computational blocks and interconnections.

In this view, a platform is a design family, not a single design. A platform defines the explorable design space. Once we select a particular collection of platform components, we obtain a platform instance. As Figure 3 shows, we can obtain multiple platform instances in refining a platform. The choice of the platform instance and the mapping of the specification components defining the application of interest into the platform instance components constitute the top-down process. This process maps constraints accompanying the specification into constraints on the platform instance components. Platform mapping often involves budgeting because we might have to distribute a global constraint among a set of components.

Stepwise refinement continues by defining the selected platform instance as a specification and using a lower-level platform to march toward implementation. When a component is fully instantiated, stepwise refinement stops, because the designer then has an implementation for that component.

In selecting a platform instance and mapping constraints using budgeting, it is important to guide the selection with parameters that summarize the platform component characteristics. Delay, power consumption, size, and cost are examples of such parameters. When selecting a platform instance, the designer must quickly and accurately evaluate the design's potential performance. Thus, the selection of the guiding parameters is a critical part of platform-based design. In Figure 3, this process is called *platform design space export*.

It is possible to automate the selection of components and the verification of consistency between the specification behavior and the platform instance behavior. To do so, designers must find a common semantic domain between the behavior and the platform so that the selection process becomes a covering problem—that is, the selection of platform components is an effort to cover the entire design functionality with one or more platform elements. The concepts of platform-based design can apply to the entire design process, even

with an ASIC design style. The framework is the same; the platforms are different. The number and location of platforms in the design abstractions, the number and type of components that constitute a platform, and the choice of parameters to represent the components are critical aspects of this method.

As Figure 4 shows, platforms form a stack from the design specification defined in the application domain to implementation. Some platforms demarcate boundaries critical in the electronics supply chain. These articulation points warrant particular attention. We call an architecture platform the articulation point between system architecture and microarchitecture. Microarchitecture is a platform whose components are architectural elements such as microprocessors, memories, and interfaces. This articulation point is where the application engineer maps a design into a physical structure. To find a common semantic domain, the application engineer must abstract these components via an operating system, device drivers, and a communication mechanism. The hardware components support the execution of the specification behavior. Another essential platform is the one that corresponds to the layer between design and manufacturing—the implementation platform.

Application example

Fellow researchers and I are testing the new methodology in advanced industrial environments. For example, consider an ECU design carried out in collaboration by Parades, Magneti-Marelli, and STMicroelectronics.⁶ This design has a strong control component and tight safety constraints. In addition, the application contained a large portion of legacy design. The electronic engine control subsystem's functionality comprises the following elements:

- failure detection and recovery of input sensors;
- computation of engine phase, status, and angle; crankshaft revolution speed; and acceleration;
- injection and ignition control law; and
- injection and ignition actuation drivers.

The existing implementation had 135,000 lines of C source code without comments.

Our first task was to extract the precise functionality from the implementation. To do this, we used a representation based on the Code-sign Finite-State Machine network, the computation model used in the Virtual Component Codesign (VCC) development environment,⁷ resulting in 89 CFSMs and 56 timers. We completely rewrote the behavior of the actuators and some of the sensors in the formal model. For the ignition and injection control law, we encapsulated the legacy C code into 18 CFSMs representing concurrent processes. We redesigned the software to make mapping into different microarchitectures relatively easy. In particular, we tested three different CPUs and, for each, two different software partitionings, to verify functionality and real-time behavior. In addition, we explored three different architectures for the I/O subsystem: one with a full software implementation, one that used a peripheral (provided by the CPU vendor) for timing functions, and one with a newly designed, highly optimized full-hardware peripheral.

Performance estimation based on VCC resulted in an error of only 11 percent, compared with a prototype board containing real hardware. We implemented functionality on three platforms, resulting in software reusability of more than 86 percent. We also used the functionality captured in semiformal terms to design a new dual-processor architecture being sampled at STMicroelectronics.⁸

In considering the future of electronic devices and infrastructures as they affect the car manufacturing and design world, the issues related to choice and design of integrated platforms are crucial. A rigorous design methodology based on separation of concerns (function and architecture, function and communication) and platforms is essential. The challenges ahead are great, but the opportunities are enormous. The automobile industry is on the edge of a revolution in the way it conceives and designs a car.

Acknowledgments

I thank the Parades researchers and the Cadence Automotive Tiger Team—in particular, A. Ferrari, M. Chiodo, P. Giusto, L. Lavagno, and J.Y. Brunel—for their contributions to the vision presented here. This research

was supported in part by Parades, the Gigascale Silicon Research Center, and the IST Columbus Project of the European Community.

References

1. H. Jones, *Monthly Report on Semiconductor Industry*, Jan. 2003, International Business Strategies, Inc.
2. OSEK, www.osek-vedx.org.
3. MathWorks/Simulink, www.mathworks.com.
4. ETAS ASCET-SD Homepage, www.etas.de.
5. P. Giusto et al., "Automotive Virtual Integration Platforms: Why's, What's, and How's," *Proc. Int'l Conf. Computer Design (ICCD 02)*, IEEE CS Press, 2002, pp. 370-378.
6. G. Bombarda, G. Gaviani, and P. Marceca, *Power-Train System Design: Functional and Architectural Specifications*, SAE Tech. Paper 2000-01-C049, Society of Automotive Engineers, 2000.
7. Cadence Design Systems, Virtual Compo-

nent Codesign (VCC), www.cadence.com.

8. A. Ferrari et al., *The Design and Implementation of a Dual-Core Platform for Power-Train Systems*, SAE Tech. Paper 2000-01-C050, Society of Automotive Engineers, 2000.

Alberto Sangiovanni-Vincentelli is a guest editor of this issue. His biography appears on page 9.

Direct questions and comments about this article to Alberto Sangiovanni-Vincentelli, University of California at Berkeley, 515 Cory Hall, Berkeley, CA 94720-1770; alberto@eecs.berkeley.edu.

For further information on this or any other computing topic, visit our Digital Library at <http://computer.org/publications/dlib>.

Look for these topics in IEEE Computer Society magazines this year

Computer

Agile Software Development
Piracy & Privacy

IEEE Computer Graphics & Applications

3D Reconstruction & Visualization

Computing in Science & Engineering

The End of Moore's Law

IEEE Design & Test

Clockless VLSI Design

IEEE Intelligent Systems

AI & Elder Care

IEEE Internet Computing

The Semantic Web

IT Professional

Financial Market IT

IEEE Micro

Hot Chips 14

IEEE MultiMedia

Computational Media Aesthetics

IEEE Software

Software Geriatrics:
Planning the Whole Life Cycle

IEEE Security & Privacy

Digital Rights Management

IEEE Pervasive Computing

Smart Spaces



computer.org/publications