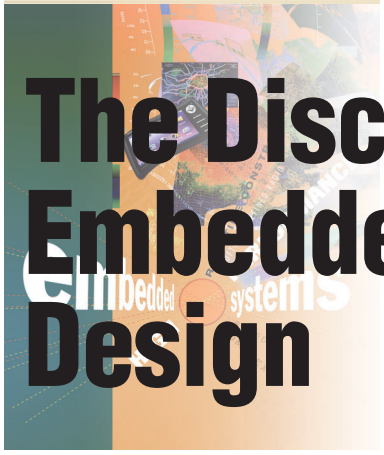

The Discipline of Embedded Systems Design

Thomas A. Henzinger, EPFL
Joseph Sifakis, Verimag

The wall between computer science and electrical engineering has kept the potential of embedded systems at bay. It is time to build a new scientific foundation with embedded systems design as the cornerstone, which will ensure a systematic and even-handed integration of the two fields.

Computer science is maturing. Researchers have solved many of the discipline's original, defining problems, and many of those that remain require a breakthrough that is impossible to foresee. Many current research challenges—the Semantic Web, nanotechnologies, computational biology, and sensor networks, for example—are pushing existing technology to the limits and into new applications. Many of the brightest students no longer aim to become computer scientists, but choose to enter directly into the life sciences or nanoengineering.[1] At the same time, computer technology has become ubiquitous in daily life, and embedded software is controlling communication, transportation, and medical systems.

From smart buildings to automated highways, the opportunities seem unlimited, yet the costs are often prohibitive, and dependability is generally poor. The automotive industry is a good example. As each car receives an ever-increasing number of electronic control units, software complexity escalates to the point that current development processes and tools can no longer ensure sufficiently reliable systems at affordable cost. Paradoxically, the shortcomings of current design, validation, and maintenance processes make software the most costly and least reliable part of embedded applications. As a result, industries cannot capitalize on the huge potential that emerging hardware and communication technologies offer.

We see the main culprit as the lack of rigorous techniques for embedded systems design. At one extreme, computer science research has largely ignored embedded systems, using abstractions that actually remove physical constraints from consideration. At the other, embedded systems design goes beyond the traditional expertise of electrical engineers because computation and software are integral parts of embedded systems.

Fortunately, with crises comes opportunity—in this case, the chance to reinvigorate computer science research by focusing on embedded systems design. The embedded systems design problem certainly raises technology questions, but more important, it requires building a new scientific foundation that will systematically and even-handedly integrate computation and physicality from the bottom up.[2] Support for this foundation will require enriching computer science paradigms to encompass models and methods traditionally found in electrical engineering.[3,4]

In parallel, educators will need to renew the computer science curriculum. In industry, trained electrical engineers routinely design software architectures, and trained computer scientists routinely deal with physical constraints. Yet embedded systems design is peripheral to both computer science and electrical engineering curricula. Much of the cultural wall between the two fields can be traced to differences between the discrete mathematics of computer science and the continuous

Published by the IEEE Computer Society
0018-9162/07/$25.00 © 2007 IEEE

mathematics of traditional engineering. The industry desperately needs engineers who feel equally at home in both worlds. The embedded systems design discipline has the potential to produce such integrated talent. But defining its scientific foundation will take a concerted, coordinated effort on the part of research, academia, industry, and policy makers.

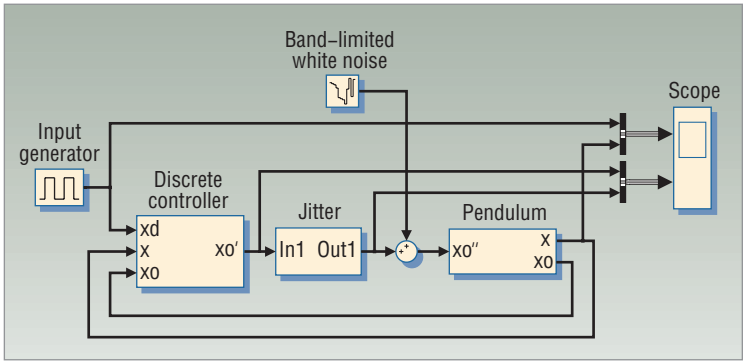

Figure 1. An analytical model. The block diagram models an inverted pendulum controlled by a discrete controller, described in Matlab's Simulink. From feedback signals and a periodic input, the controller generates an ideal control signal, which jitter and noise functions transform.

THE DESIGN PROBLEM

An embedded system is an engineering artifact involving computation that is subject to physical constraints. The physical constraints arise through the two ways that computational processes interact with the physical world: reaction to a physical environment and execution on a physical platform. Common *reaction* constraints specify deadlines, throughput, and jitter and originate from behavioral requirements. Common *execution* constraints bound available processor speeds, power, and hardware failure rates and originate from implementation choices. Control theory deals with reaction constraints; computer engineering deals with execution constraints. The key to embedded systems design is gaining control of the interplay between computation and both kinds of constraints to meet a given set of requirements on a given implementation platform.

General versus embedded systems design

Systems design derives an abstract system representation from requirements—a model—from which a system can be generated automatically. Software design, for example, derives a program from which a compiler can generate code; hardware design derives a hardware description from which a computer-aided design tool can synthesize a circuit. In both domains, the design process usually mixes bottom-up activities, such as the reuse and adaptation of component libraries, and top-down activities, such as successive model refinement to meet a set of requirements.

Although they are similar to other computing systems because they have software, hardware, and an environment, embedded systems differ in an essential way: Since they involve computation that is subject to physical constraints, the powerful separation of computation (software) from physicality (platform and environment)—traditionally, a central enabling concept in computer science—does not work for embedded systems. Instead, embedded systems design requires a more holistic approach that integrates essential paradigms from hardware and software design and control theory.

Differing design principles

Embedded systems design is not a straightforward extension of either hardware or software design. Rather, design theories and practices for hardware and software are tailored toward the individual properties of these two domains, often using abstractions that are diametrically opposed.

Hardware systems designers, for example, compose a system from interconnected, inherently parallel building blocks, which can represent transistors, logic gates, functional components such as adders, or architectural components such as processors. Although the abstraction level changes, the building blocks are always deterministic, or probabilistic, and their composition is determined by how data flows among them. A building block's formal semantics consist of a transfer function, typically specified by equations. Thus, the basic operation for constructing hardware models is the composition of transfer functions. This type of equation-based model is an *analytical* model, such as the example in Figure 1. Examples of analytical models include netlists, dataflow diagrams, and other notations for describing system structure.

Software systems designers, in contrast, use sequential building blocks, such as objects and threads, whose structure often changes dynamically. Designers can create, delete, or migrate blocks, which can represent instructions, subroutines, or software components. An abstract machine, also known as a virtual machine or automaton, defines a block's formal semantics operationally. Abstract machines can be nondeterministic, and designers define the blocks' composition by specifying how control flows among them. For example, the atomic blocks of different threads are typically interleaved, possibly using synchronization operations to constrain them. Thus, the basic operation for constructing software models is the product of sequential machines. This type of machine-based model is a *computational* model, such as the state diagram fragment[5] in Figure 2. Examples of computational models include programs, state machines, and other notations for describing system dynamics.

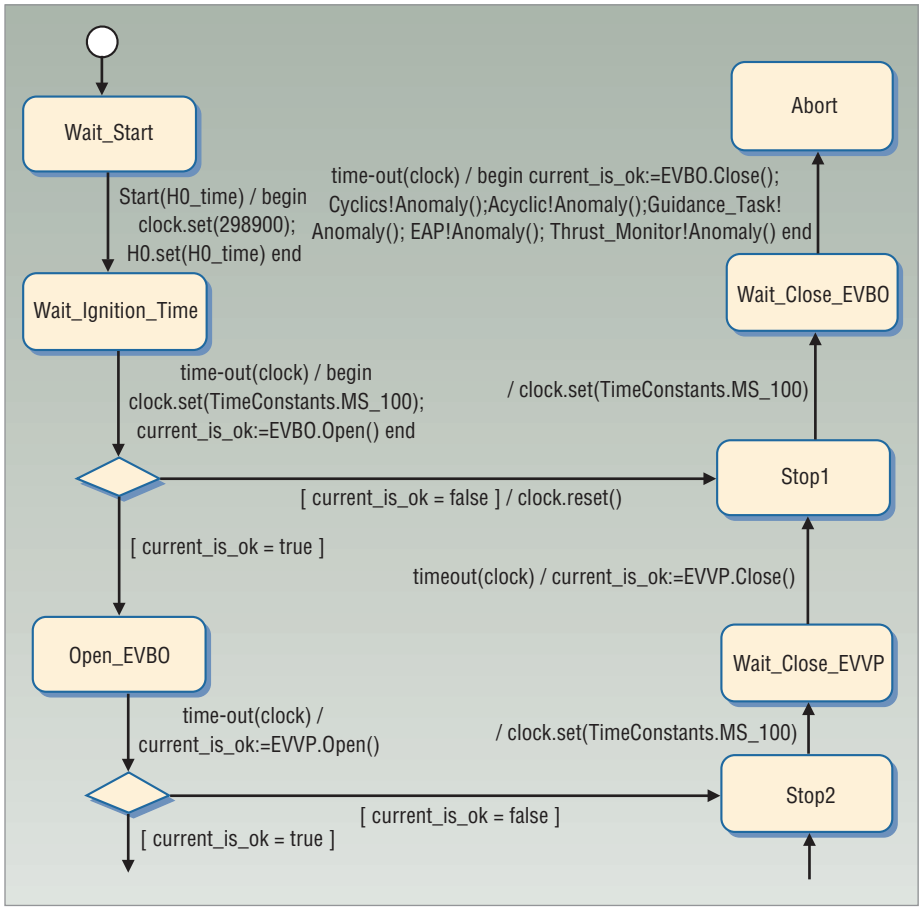

Figure 2. A computational model. The state diagram fragment models a part of the flight controller for the Ariane 5 rocket, described in Rational Rose UML. State diagrams are extended automata that specify the behavior of objects. The transitions are labeled with guarded commands involving two types of interaction between objects: synchronous function calls and asynchronous message passing.

Analytical and computational models embody orthogonal abstractions. Analytical models deal naturally with concurrency and with quantitative constraints, but struggle with partial and incremental specifications (nondeterminism) and computational complexity. Equation-based models and associated analytical methods are used not only in hardware design and control theory, but also in scheduling and performance evaluation. Computational models, on the other hand, naturally support nondeterministic abstraction hierarchies and a rich theory of computational complexity, but taming concurrency and incorporate physical constraints is difficult.

Many major computer science paradigms, such as the Turing machine and the thread concurrency model, have succeeded precisely because they abstract away all physical notions of concurrency and all physical constraints on computation. Indeed, entire computer science subfields are built on and flourish because of such abstractions. In operating systems and distributed computing, both time sharing and parallelism are famously abstracted to the same concept, namely, nondeterministic sequential computation. In algorithms and complexity theory, real time is abstracted to big-O time, and physical memory to big-O space. These powerful abstractions are inadequate for embedded systems design.

Differing system requirements

Analytical and computational models aim to satisfy different system requirements. *Functional* requirements specify the expected services, functionality, and features—independent of the implementation. *Extrafunctional* requirements specify performance and robustness. The former demand the efficient use of real-time and implementation resources, while the latter require the ability to deliver some minimal functionality under circumstances that deviate from nominal. Functional requirements are naturally expressed in discrete, logic-based formalisms. However, to express many extrafunctional requirements, designers need real-valued quantities to represent physical constraints and probabilities.

For software, the dominant driver is correct functionality, and designers often discretely specify even performance and robustness, such as the number of messages exchanged or failures tolerated. For hardware, continuous performance and robustness measures are more prominent and refer to physical resource levels, such as clock frequency, energy consumption, latency, mean time to failure, and cost. Critical to embedded systems design is the ability to quantify tradeoffs among functionality, performance, and robustness under given technical and economic constraints.

Differing design processes

Analytical and computational models support different design processes. Equation-based modeling yields rich analytical tools, especially if stochastic behavior is present. Moreover, for a system with only a few basic building block types, as in circuit design, automatic synthesis techniques have proved extraordinarily successful in designing very large systems. Indeed, they have basically spawned the elec-

tronic design automation industry. Machine-based models, on the other hand, while sacrificing powerful analytical and synthesis techniques, are directly executable. They also give the designer more fine-grained control and provide a greater space for design variety and optimization. Indeed, robust software architectures and efficient algorithms are still designed individually, not automatically generated, and this will likely remain so for some time. The emphasis, therefore, shifts from design synthesis to design verification—the proof of correctness.

CURRENT DESIGN PRACTICES

The sidebar "Three Generations of Embedded Systems Design" describes the historical progression of embedded systems design practices. Current trends are moving away from the specificity that particular programming languages and implementation platforms offer toward greater genericity. Practices typically use higher levels of abstraction, as in model-based design, and apply either critical or best-effort systems engineering.

Three Generations of Embedded Systems Design

The evolution of embedded systems design shows how design practices have moved from a close coupling of design and implementation levels to relative independence between the two.

Language- and synthesis-based origins

The first generation of methodologies traced their origins to one of two sources: *Language-based* methods lie in the software tradition, and *synthesis-based* methods stem from the hardware tradition. A language-based approach is centered on a particular programming language with a particular target runtime system (often fixed-priority scheduling with preemption). Early examples include Ada and, more recent, RT-Java. Synthesis-based approaches have evolved from circuit design methodologies. They start from a system description in a tractable, often structural, fragment of a hardware description language such as VHDL and Verilog and automatically derive an implementation that obeys a given set of constraints.

Implementation platform independence

The second generation of methodologies introduced a semantic separation of the design level from the implementation level to gain maximum independence from a specific execution platform during early design phases. There are several examples. The synchronous programming languages embody an abstract hardware semantics (synchronicity) within software; implementation technologies are available for different platforms, including bare machines and time-triggered architectures. SystemC combines a synchronous hardware semantics with asynchronous execution mechanisms from software (C++); implementations require partitioning into components that will be realized in hardware on the one side and in software on the other. The semantics of common dataflow languages such as Matlab's Simulink are defined through a simulation engine, as is the controller specification in Figure 1; implementations focus on generating efficient code. Languages for describing distributed systems, such as the Specification and Description Language (SDL), generally adopt an asynchronous semantics.

Execution semantics independence

The third generation of methodologies is based on modeling languages such as the Unified Modeling Language (UML) and Architecture Analysis and Design Language (AADL) and go a step beyond implementation independence. They attempt to be generic not only in the choice of implementation platform, but even in the choice of execution and interaction semantics for abstract system descriptions. This leads to independence from a particular programming language, as well as to an emphasis on system architecture as a means of organizing computation, communication, and resource constraints.

Much recent attention has focused on frameworks for expressing different models of computation and their interoperation.[1-4] These frameworks support the construction of systems from components and high-level primitives for their coordination. They aim to offer not just a disjoint union of models within a common metalanguage, but also to preserve properties during model composition and to support meaningful analyses and transformations across heterogeneous model boundaries.

References

1. F. Balarin et al., "Metropolis: An Integrated Electronic System Design Environment," *Computer*, Apr. 2003, pp. 45-52.
2. J. Eker et al., "Taming Heterogeneity: The Ptolemy Approach," *Proc. IEEE*, vol. 91, no. 1, 2003, pp. 127-144.
3. K. Balasubramanian et al., "Developing Applications Using Model-Driven Design Environments," *Computer*, Feb. 2006, pp. 33-40.
4. J. Sifakis, "A Framework for Component-Based Construction," *Proc. Software Eng. and Formal Methods*, IEEE Press, 2005, pp. 293-300.

Model-based design

The goal in any model-based design approach is to describe system components within a modeling language that does not commit the designer early on either to a specific execution and interaction semantics or to specific implementation choices.

Central to all model-based design is an effective theory of model transformations. Design often involves the use of multiple models that represent different system views at different granularity levels. Usually, design proceeds neither strictly top-down, from the requirements to the implementation, nor strictly bottom-up, by integrating library components, but in a less directed fashion, by iterating model construction, analysis, and transformation. Model transformation must preserve essential properties. Some transformations can be automated; for others, the designer must guide the model construction.

Model-based design seeks independence from specific execution semantics or implementation choices.

The ultimate success story in model transformation is compilation theory. It is difficult to manually improve on the code a good optimizing compiler produces from computational models written in a high-level language. On the other hand, code generators often produce inefficient code from equation-based models: They can compute the solutions of fixed-point equations or approximate them iteratively, but a designer must supply more efficient algorithmic insights and data structures.

For extrafunctional requirements, such as timing, the separation of human-guided design decisions from automatic model transformations is even less well understood. Indeed, engineering practice often relies on a trial-and-error loop of code generation, followed by test, followed by redesign. An alternative is to develop high-level programming languages that can express reaction constraints, together with compilers that guarantee the satisfaction of reaction constraints on a given execution platform.[6] Such a compiler must mediate between program-specified reaction constraints, such as time-outs, and the platform's execution constraints, typically provided as worst-case execution times.

Systems engineering

Today's systems engineering methodologies are either critical or best effort. Critical methods try to guarantee system safety at all costs, even when the system operates under extreme conditions. Best-effort methods try to optimize system performance (and cost) when the system operates under expected conditions. One views design as a constraint-satisfaction problem; the other views it as an optimization problem.

Critical. Critical systems engineering is based on conservative approximations of system dynamics and on static resource reservation. Tractable conservative approximations often require simple execution platforms, such as bare machines without operating systems and processor architectures that allow time predictability for code execution. Typical examples of such approaches are those used for safety-critical systems in avionics. In these systems, real-time constraint satisfaction is guaranteed on the basis of worst-case execution time analysis and static scheduling. The maximum necessary computing power is available at all times, and designers achieve dependability by using massive redundancy and statically deploying all equipment for failure detection and recovery.

The time-triggered architecture (TTA)[7] is an example of critical systems engineering. Developers use it for distributed real-time systems in certain safety-critical applications, such as brake-by-wire or drive-by-wire systems in cars. A TTA node consists of two subsystems: the communication controller and host computer. All communication among nodes follows a predetermined static schedule. Each communication controller has a message description list that determines at what point a node is allowed to send a message, and when it is expected to receive a message from another node. A fault-tolerant clock synchronization protocol provides each node with global time ticks.

Best effort. Best-effort systems engineering is based on average-case, not worst-case, analysis and on dynamic rather than static resource allocation. It seeks the efficient use of resources, as in optimizing throughput or power, and is useful in applications that can tolerate some degradation or even temporary denial of service. Instead of the worst-case, or *hard*, requirements applied to critical systems, best-effort systems have *soft* quality-of-service (QoS) requirements, such as jitter and error rate in telecommunications networks. Hard deadlines must be met; soft deadlines can occasionally be missed. QoS requirements are enforced by adaptive scheduling mechanisms that use feedback to adjust some system parameters at runtime to optimize performance and recover from behavioral deviations.

Many networks and multimedia systems are examples of best-effort engineering. These systems often use control mechanisms that, under different workloads, provide different priorities to different users or data flows and, in that way, guarantee certain performance levels. Deviations from a nominal situation, such as accepted error or packet loss rate, guide the control mechanisms.

Bridging the gap. Critical and best-effort engineering are largely disjoint, since meeting hard constraints and making the best possible use of available resources work against each other. Critical systems engineering can lead to the underutilization of resources, best-effort systems engineering to temporary unavailability.

We believe that the gap between the two approaches will continue to widen, as the uncertainties in embedded systems design increase. First, as embedded systems become more widespread, their environments are known less perfectly, with greater distances between worst-case and expected behaviors. Second, because of VLSI design's rapid progress, developers are implementing embedded systems on sophisticated, layered multicore architectures with caches, pipelines, and speculative execution. The ensuing difficulty of accurate worst-case analysis makes conservative, safety-critical designs ever more expensive, in both resource and design cost, relative to best-effort designs.

As the gap between critical and best-effort designs increases, partitioned architectures are likely to become more prevalent. Partitions physically separate critical and noncritical system parts, letting each run in dedicated memory space during dedicated time slots. The result is a guarantee of minimal-level worst- and average-case performance. Such designs can find support in the ever-growing computing power of system-on-chip and network-on-chip technologies, which reduce communication costs and increase hardware reliability, thereby allowing more rational and cost-effective resource management. Corresponding design methodologies must guarantee a sufficiently strong separation between critical and noncritical components that share resources.

There must be a way to quantify tradeoffs between critical and best-effort engineering decisions.

THE RESEARCH CHALLENGE

Embedded systems design must deal even-handedly with

- computation and physical constraints,
- software and hardware,
- abstract machines and transfer functions,
- nondeterminism and probabilities,
- functional and performance requirements,
- qualitative and quantitative analysis, and
- Boolean and real values.

The solution is not simply to juxtapose analytical and computational techniques. It requires their tight integration within a new mathematical foundation that spans both perspectives. There must also be a way to methodically quantify tradeoffs between critical and best-effort engineering decisions.

In arriving at a solution, research must address two opposing forces—the ability to cope with heterogeneity and the need for constructivity during design. *Heterogeneity* arises from the need to integrate components with different characteristics. It has several sources and manifestations, and the existing body of knowledge is largely fragmented into unrelated models and corresponding results. *Constructivity* is the capacity for building complex systems from building blocks and glue components with known properties. It is achievable through algorithms, as in compilation and synthesis, as well as through architectures and design disciplines.

Heterogeneity and constructivity pull in different directions. Encompassing heterogeneity looks outward, toward the integration of multiple theories to provide a unifying view. Constructivity looks inward, toward developing a tractable theory for system construction. Since constructivity is most easily achieved in restricted settings, a scientific foundation for embedded systems design must provide a way to intelligently balance and trade off both ambitions.

Finally, the resulting systems must not only meet functional and performance requirements, they must also do so robustly. Robustness includes resilience and measured degradation in the event of failures, attacks, faulty assumptions, and erroneous use. Ensuring robustness is, in our view, a central issue for the embedded systems design discipline.

Coping with heterogeneity

Systems designers deal with a variety of components from many viewpoints. Each component has different characteristics, and each viewpoint highlights different system dimensions. Such complexity gives rise to two central problems. One is how to compose heterogeneous components to ensure their correct interoperation; the other is how to transform and integrate heterogeneous viewpoints during the design process. Superficial classifications distinguish hardware and software components, or continuous-time (analog) and discrete-time (digital) components, but heterogeneity has two more fundamental sources: the composition of subsystems with different execution and interaction semantics, and the abstract view of a system from different distances and perspectives.

Execution and interaction semantics. At one extreme of the semantic spectrum are fully synchronized components, which proceed in lockstep with a global clock and interact in atomic transactions. Such a tight component coupling is the standard model for most synthesizable hardware and for synchronous real-time software. The synchronous model considers a system's execution as a sequence of global steps. It assumes that the environment does not change during a step, or equivalently, that the system is infinitely faster than its environment. In each execution step, all system components contribute by executing some quantum of computation. The synchronous execution paradigm, therefore, has a built-in strong assumption of fairness: In each step all components can move forward.

At the other extreme are completely asynchronous components, which proceed at independent speeds and interact nonatomically. Such a loose component coupling is the standard model for multithreaded software. The lack of built-in mechanisms for sharing computation among components can be compensated through scheduling constraints, such as priorities and fairness, and through interaction mechanisms, such as messages and shared resources. Between the two extremes are a variety of intermediate and hybrid models. The fragment of the Ariane 5 rocket flight controller specification in Figure 2 uses both synchronous interaction through function calls and asynchronous interaction through message passing.

Abstraction levels and viewpoints. System design involves the use of system models that have varying degrees of detail and are related to each other in an abstraction or refinement hierarchy. Heterogeneous abstractions, which relate diverse model styles, are often the most powerful. A notable example is the Boolean-valued gate-level abstraction of real-valued transistor-level models for circuits. In embedded systems, a key abstraction relates application software to its implementation on a given platform. Application software refers to real time through reaction constraints, such as time-outs. The application code running on a given platform is subject to real-time implementation constraints, such as worst-case execution times. It is possible to model both as dynamic systems that are specified, for example, by timed or hybrid automata, but formal refinement relations between such automata are both unnecessarily precise and prohibitively expensive.

Scalable design techniques require rules for building correct systems from correct components.

Another cause of heterogeneity in abstractions is the use of different viewpoints, such as computational or analytical perspectives. Heterogeneous models also arise when representing different extrafunctional system dimensions, such as timing, power consumption, and fault tolerance. In certain settings, these dimensions might be tightly correlated; in others, these dimensions might allow independent solutions.

Achieving constructivity

The system construction problem is basically "Build a system that meets a given set of requirements from a given set of building blocks." This problem is at the root of systems design activities such as modeling, architecting, programming, synthesis, upgrading, and reuse. Because a general formulation of the problem is intractable, the goal must be limited to practical success in common scenarios. The main obstacle to reaching this goal is scalability, which is rarely achievable through algorithmic verification and synthesis techniques alone. It requires a combination of two fundamental techniques: compositionality—design rules and disciplines for building correct systems from correct components—and the use of architectures and protocols that ensure global system properties.

Compositionality. Correct-by-construction bottom-up design is based on component interfaces and noninterference rules. A well-designed *interface* exposes exactly the component information needed to check for composability with other components. In a sense, an interface formalism is a type theory for component composition.[8] Recent trends have been toward rich interfaces, which expose not only functional but also extrafunctional component information, such as resource consumption levels. The composition of two or more interfaces then specifies the combined resources consumed by putting together the underlying components.

A *noninterference* rule guarantees that all essential component properties are preserved during system construction. However, often the requirements for different resources are interdependent, as in timeliness and power efficiency. In such cases, concerns cannot be completely separate, and construction methods must meet multiple requirements.

Architectural properties. Correct-by-construction top-down design is based on the preservation of architectural properties through refinement—ideally, the independent refinement of different components. Scalable, lightweight analyses exist for very specific architectures and properties. TTAs, for example, ensure timely and fault-tolerant communication for distributed real-time systems; a token-ring protocol guarantees mutual exclusion for strongly synchronized processes that are connected in a ring. It is essential to study the interplay between architectures and properties in more general terms.

Ensuring robustness

A robust system maintains certain performance levels even under circumstances that deviate from the normal operating environment. Such deviations can include extreme input values, platform failures, malicious attacks, wrong modeling assumptions, and human error. Consequently, robustness requirements include a broad spectrum of properties, such as safety (resistance to failures), security (resistance to attacks), and availability. Robustness cuts across all design activities and influences all design decisions in system construction. System security, for example, must account for software and hardware architectures, information treatment—such as encryption, access, and transmission—and programming disciplines.

In dynamic systems, robustness can be formalized as continuity, so that small perturbations of input values

cause small perturbations of output values. No such formalization is available for discrete systems, where the change of a single input or state bit can lead to a completely different output behavior. Therefore, in computer science, redundancy is often the only solution for building reliable systems from unreliable components.

There is a critical need for theories, methods, and tools that support the construction of robust embedded systems without forcing designers to resort to such massive, expensive over-engineering. Continuity might be achievable in fully quantitative models, where quantitative information expresses not only probabilities, time, and other resource-consumption levels, but also functional characteristics. For example, if the interest is in mean time to failure, not the Boolean-valued possibility or impossibility of failure, designers should be able to build continuous models, in which small changes in certain parameters induce only small changes in the failure rate.

The lack of adequately trained bicultural engineers causes fragmentation and inefficiencies in industry.

THE EDUCATION CHALLENGE

Embedded systems are not central to the computer science curriculum or to the electrical engineering or computer engineering curricula. When embedded computing is taught at all, it is often as a project course that emphasizes tinkering with sensors and actuators, not as an engineering discipline with a solid scientific foundation. A commendable exception is the recent textbook by Edward Lee and Pravin Varaiya,[9] which attempts a bicultural education of freshmen in both computer science and electrical engineering. The text goes beyond traditional bicultural education, which covers both software and hardware. Rather, it combines computational and analytical thinking.

The lack of adequately trained bicultural engineers impacts industrial practice, creating fragmentation, inefficiencies, and difficulties in communication and sharing experiences within a company. For example, aeronautics and space engineering—two closely related fields that address similar problems—use different methodologies, design flows, and tools for reasons that seem more cultural than technical.[10] Often, engineers untrained in software abstractions or control theory are asked to architect complex embedded systems that critically rely on both disciplines.

The lack of a common cultural background also results in fragmented research. Different communities use different terminologies. Even commonly used terms such as "timed" and "untimed" or "synchronous" and "asynchronous" do not have the same meaning across communities. This is a circular phenomenon: The teachers do not understand each other, so neither do the students. Conferences are separate, communities do not interact, and papers do not have the desired impact.

The educational system has significant inertia and will take time to adapt to evolving industrial and cultural needs. It is particularly difficult to gather specialists who are spread among several university departments to design and implement new curricula. We recommend, at a minimum, exposing computer science students to both computational and analytical thinking and corresponding approaches to embedded systems design. Another near-term goal should be to define and distribute a computer science reference curriculum with an optional track for specializing in embedded systems. The curriculum can serve as a basis for certifying courses and materials.

Advances in hardware components technology are creating an enormous potential for the widespread application of embedded systems in all economic sectors, but without some well-grounded design paradigm, society cannot fully benefit from this potential. Unfortunately, many groups and the public at large fail to grasp the importance of embedded software. Unlike many innovations, such as the Web, that are tangible and thus well known and appreciated, embedded systems are visible only through the improved function and performance of devices and products. Indeed, the more seamlessly embedded computers and software are integrated into the products and the less often they fail, the less visible they are. It is thus important to raise awareness about how vital embedded computing is to society.

What is needed in research? A first step would be to derive a mathematical basis for systems modeling and analysis that integrates both abstract-machine models and transfer-function models. Such a theory could unite two sets of practices: those from critical systems engineering that guarantee hard requirements and those from best-effort systems engineering that optimize performance. From there, the theory, methodologies, and tools must

- encompass heterogeneous execution and interaction mechanisms for system components,
- provide abstractions that isolate the design subproblems requiring human creativity from those that can be automated,
- scale by supporting compositional, correct-by-construction techniques, and
- ensure the robustness of the resulting systems.

This effort is a true Grand Challenge, demanding departures from the prevailing views on both hardware and software design, but if successful, it will substantially reduce the cost and increase the quality of tomorrow's embedded infrastructure.

What is needed in education? To adequately train new generations of engineers and researchers, institutions must focus on embedded systems design as a scientific discipline and as a specialization area within existing curricula. This requires taking down the cultural wall that exists between many computer science and electrical engineering departments. Embedded systems design is not computer engineering. It centrally includes software, which most view as pure computer science, and control, which most view as pure electrical engineering. In spite of the recent efforts by research societies such as the IEEE and the ACM to promote and integrate the field of embedded systems, much inertia is preventing greater integration of the concerned communities. It is essential to create a critical mass by colocating and fusing existing conferences. A promising step in this direction is the recent organization of the annual Embedded Systems Week as an umbrella for several research conferences that are related to embedded systems design.

What is needed from industry? Industry tends to stay with available technologies, optimizing existing investments and competencies. Moreover, the trend is to build systems incrementally by modifying existing solutions, so the emphasis is on backward compatibility to reduce costs. This design rigidity often leads to overly complex systems that new design and analysis techniques could radically improve and simplify.

On the other hand, the inherent limits of ad hoc approaches to manage system complexity and the resulting cost explosion provide strong incentives for industry to look for alternatives. New standardization efforts, such as the Automotive Open System Architecture (AUTOSAR) initiative in the automotive industry, are appearing. It is important to seize this opportunity and develop new technologies through joint academic-industrial pilot projects. Researchers must reinvent many software abstraction layers, such as high-level languages, operating systems, middleware, and network layers to allow better control of system resources. Current practices to improve system dependability by over-engineering and using massive component redundancy must be replaced with lightweight techniques that are based on solid scientific foundations.

What is needed from policy makers? Embedded technologies have enormous impact on societal and economic transformations, but without a firm scientific basis, current practices in embedded systems design cannot fulfill the public needs and expectations in system dependability. Significant resources for basic research are required to face the embedded systems design challenge, which requires policy makers and funding agencies to become fully aware of the issues.

Acknowledgments

An earlier version of this article appeared as "The Embedded Systems Design Challenge," *Proc. 14th Int'l Symp. Formal Methods*, LNCS 4085, Springer, 2006, pp. 1-15.

The work reported in this article is supported in part by the Artist2 European Network of Excellence on Embedded Systems Design, by the National Science Foundation's Information Technology Research (ITR) project on Foundations of Hybrid and Embedded Software Systems, and by the Swiss National Science Foundation's National Center of Competence in Research (NCCR) on Mobile Information and Communication Systems (MICS). We thank Paul Caspi for valuable comments on a draft of this article.

References

1. P.J. Denning and A. McGettrick, "Recentering Computer Science," *Comm. ACM*, vol. 48, no. 11, 2005, pp. 15-19.
2. T.A. Henzinger et al., "Mission Statement: Center for Hybrid and Embedded Software Systems," Univ. of California, Berkeley; http://chess.eecs.berkeley.edu, 2002.
3. E.A. Lee, "Absolutely Positively on Time: What Would It Take?" *Computer*, July 2005, pp. 85-87.
4. J.A. Stankovic et al., "Opportunities and Obligations for Physical Computing Systems," *Computer*, Nov. 2005, pp. 23-31.
5. S. Graf, I. Ober, and J. Ober, "Validating Timed UML Models by Simulation and Verification," *Software Tools for Technology Transfer*, vol. 8, no. 2, 2006, pp. 128-145.
6. T.A. Henzinger et al., "From Control Models to Real-Time Code Using Giotto," *IEEE Control Systems Magazine*, Feb. 2003, pp. 50-64.
7. H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, 1997.
8. L. de Alfaro and T.A. Henzinger, "Interface-Based Design," *Engineering Theories of Software Intensive Systems*, M. Broy, et al., eds., NATO Science Series: Mathematics, Physics, and Chemistry, vol. 195, Springer, 2005, pp. 83-104.
9. E.A. Lee and P. Varaiya, *Structure and Interpretation of Signals and Systems*, Addison-Wesley, 2003.
10. P. Caspi et al., "Guidelines for a Graduate Curriculum on Embedded Software and Systems," *ACM Trans. Embedded Computing Systems*, vol. 4, no. 3, 2005, pp. 587-611.

Thomas A. Henzinger *is a professor of computer and communication sciences at EPFL in Lausanne, Switzerland, and an adjunct professor of electrical engineering and computer sciences at the University of California, Berkeley. He received a PhD in computer science from Stanford University and is a fellow of the IEEE and the ACM and a member of the German Academy of Sciences (Leopoldina) and of Academia Europaea. Contact him at tah@epfl.ch.*

Joseph Sifakis *is a research director at CNRS and the founder of Verimag laboratory. He received a PhD in computer science from the University of Grenoble. Contact him at Joseph.Sifakis@imag.fr.*