The Embedded Systems Design Challenge*

Thomas A. Henzinger[1] and Joseph Sifakis[2]

[1] EPFL, Lausanne
[2] VERIMAG, Grenoble

Abstract. We summarize some current trends in embedded systems design and point out some of their characteristics, such as the chasm between analytical and computational models, and the gap between safety-critical and best-effort engineering practices. We call for a coherent scientific foundation for embedded systems design, and we discuss a few key demands on such a foundation: the need for encompassing several manifestations of heterogeneity, and the need for constructivity in design. We believe that the development of a satisfactory Embedded Systems Design Science provides a timely challenge and opportunity for reinvigorating computer science.

1 Motivation

Computer Science is going through a maturing period. There is a perception that many of the original, defining problems of Computer Science either have been solved, or require an unforeseeable breakthrough (such as the P versus NP question). It is a reflection of this view that many of the currently advocated challenges for Computer Science research push existing technology to the limits (e.g., the semantic web [4]; the verifying compiler [15]; sensor networks [6]), to new application areas (such as biology [12]), or to a combination of both (e.g., nanotechnologies; quantum computing). Not surprisingly, many of the brightest students no longer aim to become computer scientists, but choose to enter directly into the life sciences or nanoengineering [8].

Our view is different. Following [18, 22], we believe that there lies a large uncharted territory within the science of computing. This is the area of embedded systems design. As we shall explain, the current paradigms of Computer Science do not apply to embedded systems design: they need to be enriched in order to encompass models and methods traditionally found in Electrical Engineering. Embedded systems design, however, should not and cannot be left to the electrical engineers, because computation and software are integral parts of embedded systems. Indeed, the shortcomings of current design, validation, and maintenance processes make software, paradoxically, the most costly and least

* Supported in part by the ARTIST2 European Network of Excellence on Embedded Systems Design, by the NSF ITR Center on Hybrid and Embedded Software Systems (CHESS), and by the SNSF NCCR on Mobile Information and Communication Systems (MICS).

reliable part of systems in automotive, aerospace, medical, and other critical applications. Given the increasing ubiquity of embedded systems in our daily lives, this constitutes a unique opportunity for reinvigorating Computer Science.

In the following we will lay out what we see as the Embedded Systems Design Challenge. In our opinion, the Embedded Systems Design Challenge raises not only technology questions, but more importantly, it requires the building of a new scientific foundation —a foundation that systematically and even-handedly integrates, from the bottom up, computation and physicality [14].

2 Current Scientific Foundations for Systems Design, and their Limitations

2.1 The Embedded Systems Design Problem

What is an embedded system? An *embedded system* is an engineering artifact involving computation that is subject to physical constraints. The physical constraints arise through two kinds of interactions of computational processes with the physical world: (1) reaction to a physical environment, and (2) execution on a physical platform. Accordingly, the two types of physical constraints are *reaction constraints* and *execution constraints*. Common reaction constraints specify deadlines, throughput, and jitter; they originate from the behavioral requirements of the system. Common execution constraints put bounds on available processor speeds, power, and hardware failure rates; they originate from the implementation requirements of the system. Reaction constraints are studied in control theory; execution constraints, in computer engineering. Gaining control of the interplay of computation with both kinds of constraints, so as to meet a given set of requirements, is the key to embedded systems design.

Systems design in general. *Systems design* is the process of deriving, from requirements, a model from which a system can be generated more or less automatically. A *model* is an abstract representation of a system. For example, software design is the process of deriving a program that can be compiled; hardware design, the process of deriving a hardware description from which a circuit can be synthesized. In both domains, the design process usually mixes bottom-up and top-down activities: the reuse and adaptation of existing component models; and the successive refinement of architectural models in order to meet the given requirements.

Embedded systems design. Embedded systems consist of hardware, software, and an environment. This they have in common with most computing systems. However, there is an essential difference between embedded and other computing systems: since embedded systems involve computation that is subject to physical constraints, the powerful separation of computation (software) from physicality (platform and environment), which has been one of the central ideas enabling the science of computing, does not work for embedded systems. Instead, the

design of embedded systems requires a holistic approach that integrates essential paradigms from hardware design, software design, and control theory in a consistent manner.

We postulate that such a holistic approach cannot be simply an extension of hardware design, nor of software design, but must be based on a new foundation that subsumes techniques from both worlds. This is because current design theories and practices for hardware, and for software, are tailored towards the individual properties of these two domains; indeed, they often use abstractions that are diametrically opposed. To see this, we now have a look at the abstractions that are commonly used in hardware design, and those that are used in software design.

2.2 Analytical versus Computational Modeling

Hardware versus software design. *Hardware systems* are designed as the composition of interconnected, inherently parallel components. The individual components are represented by analytical models (equations), which specify their transfer functions. These models are deterministic (or probabilistic), and their composition is defined by specifying how data flows across multiple components. *Software systems*, by contrast, are designed from sequential components, such as objects and threads, whose structure often changes dynamically (components are created, deleted, and may migrate). The components are represented by computational models (programs), whose semantics is defined operationally by an abstract execution engine (also called a virtual machine, or an automaton). Abstract machines may be nondeterministic, and their composition is defined by specifying how control flows across multiple components; for instance, the atomic actions of independent processes may be interleaved, possibly constrained by a fixed set of synchronization primitives.

Thus, the basic operation for constructing hardware models is the composition of transfer functions; the basic operation for constructing software models is the product of automata. These are two starkly different views for constructing dynamical systems from basic components: one *analytical* (i.e., equation-based), the other *computational* (i.e., machine-based). The analytical view is prevalent in Electrical Engineering; the computational view, in Computer Science: the netlist representation of a circuit is an example for an analytical model; any program written in an imperative language is an example for a computational model. Since both types of models have very different strengths and weaknesses, the implications on the design process are dramatic.

Analytical and computational models offer orthogonal abstractions. Analytical models deal naturally with concurrency and with quantitative constraints, but they have difficulties with partial and incremental specifications (nondeterminism) and with computational complexity. Indicatively, equation-based models and associated analytical methods are used not only in hardware design and control theory, but also in scheduling and in performance evaluation (e.g., in networking).

Computational models, on the other hand, naturally support nondeterministic abstraction hierarchies and a rich theory of computational complexity, but they have difficulties taming concurrency and incorporating physical constraints. Many major paradigms of Computer Science (e.g., the Turing machine; the thread model of concurrency; the structured operational semantics of programming languages) have succeeded precisely because they abstract away from all physical notions of concurrency and from all physical constraints on computation. Indeed, whole subfields of Computer Science 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, however, are largely inadequate for embedded systems design.

Analytical and computational models aim at different system requirements. The differences between equation-based and machine-based design are reflected in the type of requirements they support well. System designers deal with two kinds of requirements. *Functional requirements* specify the expected services, functionality, and features, independent of the implementation. *Extra-functional requirements* specify mainly performance, which characterizes the efficient use of real time and of implementation resources; and robustness, which characterizes the ability to deliver some minimal functionality under circumstances that deviate from the nominal ones. For the same functional requirements, extra-functional properties can vary depending on a large number of factors and choices, including the overall system architecture and the characteristics of the underlying platform.

Functional requirements are naturally expressed in discrete, logic-based formalisms. However, for expessing many extra-functional requirements, real-valued quantities are needed to represent physical constraints and probabilities. For software, the dominant driver is correct functionality, and even performance and robustness are often specified discretely (e.g., number of messages exchanged; number of 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. For embedded systems integrated in mass-market products, the ability to quantify trade-offs between performance and robustness, under given technical and economic constraints, is of strategic importance.

Analytical and computational models support different design processes. The differences between models based on data flow and models based on control flow have far-reaching implications on design methods. Equation-based modeling yields rich analytical tools, especially in the presence of stochastic behavior. Moreover, if the number of different basic building blocks is small, as it is in circuit design, then automatic synthesis techniques have proved extraordinarily successful in the design of very large systems, to the point of creating an

entire industry (Electronic Design Automation). Machine-based models, on the other hand, while sacrificing powerful analytical and synthesis techniques, can be executed directly. They 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 individually designed, not automatically generated, and this will likely remain the case for some time to come. The emphasis, therefore, shifts away from design synthesis to design verification (proof of correctness).

Embedded systems design must even-handedly deal with both: with computation and physical constraints; with software and hardware; with abstract machines and transfer functions; with nondeterminism and probabilities; with functional and performance requirements; with qualitative and quantitative analysis; with booleans and reals. This cannot be achieved by simple juxtaposition of analytical and computational techniques, but requires their tight integration within a new mathematical foundation that spans both perspectives.

3 Current Engineering Practices for Embedded Systems Design, and their Limitations

3.1 Model-based Design

Language-based and synthesis-based origins. Historically, many methodologies for embedded systems design trace their origins to one of two sources: there are language-based methods that lie in the software tradition, and synthesis-based methods that come out of the hardware tradition. A *language-based* approach is centered on a particular programming language with a particular target run-time system. Examples include Ada and, more recently, RT-Java [5]. For these languages, there are compilation technologies that lead to event-driven implementations on standardized platforms (fixed-priority scheduling with preemption). The *synthesis-based* approaches, on the other hand, have evolved from hardware 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, ideally automatically, derive an implementation that obeys a given set of constraints.

Implementation independence. Recent trends have focused on combining both language-based and synthesis-based approaches (hardware/software codesign) and on gaining, during the early design process, maximal independence from a specific implementation platform. We refer to these newer aproaches collectively as *model-based*, because they emphasize the separation of the design level from the implementation level, and they are centered around the semantics of abstract system descriptions (rather than on the implementation semantics). Consequently, much effort in model-based approaches goes into developing efficient code generators. We provide here only a short and incomplete selection of some representative methodologies.

Model-based methodologies. The synchronous languages, such as Lustre and Esterel [11], embody an abstract hardware semantics (synchronicity) within different kinds of software structures (functional; imperative). Implementation technologies are available for several platforms, including bare machines and time-triggered architectures. Originating from the design automation community, SystemC [19] also chooses a synchronous hardware semantics, but allows for the introduction of asynchronous execution and interaction mechanisms from software (C++). Implementations require a separation between the components to be implemented in hardware, and those to be implemented in software; different design-space exploration techniques provide guidance in making such partitioning decisions. A third kind of model-based approaches are built around a class of popular languages exemplified by MATLAB Simulink, whose semantics is defined operationally through its simulation engine.

More recent modeling languages, such as UML [20] and AADL [10], attempt to be more generic in their choice of semantics and thus bring extensions in two directions: independence from a particular programming language; and emphasis on system architecture as a means to organize computation, communication, and constraints. We believe, however, that these attempts will ultimately fall short, unless they can draw on new foundational results to overcome the current weaknesses of model-based design: the lack of analytical tools for computational models to deal with physical constraints; and the difficulty to automatically transform noncomputational models into efficient computational ones. This leads us to the key need for a better understanding of relationships and transformations between heterogeneous models.

Model transformations. Central to all model-based design is an effective theory of model transformations. Design often involves the use of multiple models that represent different views of a system at different levels of granularity. 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, model analysis, and model transformation. Some transformations between models can be automated; at other times, the designer must guide the model construction. The ultimate success story in model transformation is the theory of compilation: today, it is difficult to manually improve on the code produced by a good optimizing compiler from programs (i.e., computational models) written in a high-level language. On the other hand, code generators often produce inefficient code from equation-based models: fixpoints of equation sets can be computed (or approximated) iteratively, but more efficient algorithmic insights and data structures must be supplied by the designer.

For extra-functional 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 (e.g., priority tweaking when deadlines are missed). An alternative is to develop high-level program-

ming languages that can express reaction constraints, together with compilers that guarantee the preservation of the reaction constraints on a given execution platform [13]. Such a compiler must mediate between the reaction constraints specified by the program, such as timeouts, and the execution constraints of the platform, typically provided in the form of worst-case execution times. We believe that an extension of this approach to other extra-functional dimensions, such as power consumption and fault tolerance, is a promising direction of investigation.

3.2 Critical versus Best-Effort Engineering

Guaranteeing safety versus optimizing performance. Today's systems engineering methodologies can be classified also along another axis: critical systems engineering, and best-effort systems engineering. The former tries to guarantee system safety at all costs, even when the system operates under extreme conditions; the latter tries to optimize system performance (and cost) when the system operates under expected conditions. Critical engineering views design as a constraint-satisfaction problem; best-effort engineering, as an optimization problem.

Critical systems engineering is based on worst-case analysis (i.e., conservative approximations of the system dynamics) and on static resource reservation. For tractable conservative approximations to exist, execution platforms often need to be simplified (e.g., bare machines without operating systems; processor architectures that allow time predictability for code execution). Typical examples of such approaches are those used for safety-critical systems in avionics. Real-time constraint satisfaction is guaranteed on the basis of worst-case execution time analysis and static scheduling. The maximal necessary computing power is made available at all times. Dependability is achieved mainly by using massive redundancy, and by statically deploying all equipment for failure detection and recovery.

Best-effort systems engineering, by contrast, is based on average-case (rather than worst-case) analysis and on dynamic resource allocation. It seeks the efficient use of resources (e.g., optimization of throughput, jitter, or power) and is used for applications where some degradation or even temporary denial of service is tolerable, as in telecommunications. The 'hard' worst-case requirements of critical systems are replaced by 'soft' QoS (quality-of-service) requirements. For example, a hard deadline is either met or missed; for a soft deadline, there is a continuum of different degrees of satisfaction. QoS requirements can be enforced by adaptive (feedback-based) scheduling mechanisms, which adjust some system parameters at run-time in order to optimize performance and to recover from deviations from nominal behavior. Service may be denied temporarily by admission policies, in order to guarantee that QoS levels stay above minimum thresholds.

A widening gap. The two approaches —critical and best-effort engineering— are largely disjoint. This is reflected by the separation between 'hard' and 'soft'

real time. They correspond to different research communities and different practices. Hard approaches rely on static (design-time) analysis; soft approaches, on dynamic (run-time) adaptation. Consequently, they adopt different models of computation and use different execution platforms, middleware, and networks. For instance, time-triggered technologies are considered to be indispensable for drive-by-wire automotive systems [17]. Most safety-critical systems adopt very simple static scheduling principles, either fixed-priority scheduling with preemption, or round-robin scheduling for synchronous execution. It is often said that such a separation is inevitable for systems with uncertain environments. Meeting hard constraints and making the best possible use of the available resources seem to be two conflicting requirements. The hard real-time approach leads to low utilization of system resources. On the other hand, soft approaches take the risk of temporary unavailability.

We believe that, left unchecked, the gap between the two approaches will continue to widen. This is because the uncertainties in embedded systems design keep increasing for two reasons. First, as embedded systems are deployed in a greater variety of situations, their environments are less perfectly known, with greater distances between worst-case and expected behaviors. Second, because of the rapid progress in VLSI design, embedded systems are implemented on sophisticated, hardware/software layered multicore architectures with caches, pipelines, and speculative execution. The ensuing difficulty of accurate worst-case analysis makes conservative, safety-critical solutions ever more expensive, in both resource and design cost, in comparison to best-effort solutions. The divide between critical and best-effort engineering already leads often to a physical separation between the critical and noncritical parts of a system, each running on dedicated hardware or during dedicated time slots. As the gap between worst-case and average-case solutions increases, such separated architectures are likely to become more prevalent.

Bridging the gap. We think that technological trends oblige us to revise the dual vision and separation between critical and best-effort practices. The increasing computing power of system-on-chip and network-on-chip technologies allows the integration of critical and noncritical applications on a single chip. This reduces communication costs and increases hardware reliability. It also allows a more rational and cost-effective management of resources. To achieve this, we need methods for guaranteeing a sufficiently strong, but not absolute, separation between critical and noncritical components that share common resources. In particular, design techniques for adaptive systems should make flexible use of the available resources by taking advantage of any complementarities between hard and soft constraints. One possibility may be to treat the satisfaction of critical requirements as minimal guaranteed QoS level. Such an approach would require, once again, the integration of boolean-valued and quantitative methods.

4 Two Demands on a Solution

Heterogeneity and constructivity. Our vision is to develop an Embedded Systems Design Science that even-handedly integrates analytical and computational views of a system, and that methodically quantifies trade-offs between critical and best-effort engineering decisions. Two opposing forces need to be addressed for setting up such an Embedded Systems Design Science. These correspond to the needs for encompassing heterogeneity and achieving constructivity during the design process. *Heterogeneity* is the property of embedded systems to be built from components with different characteristics. Heterogeneity has several sources and manifestations (as will be discussed below), and the existing body of knowledge is largely fragmented into unrelated models and corresponding results. *Constructivity* is the possibility to build complex systems that meet given requirements, from building blocks and glue components with known properties. Constructivity can be achieved by algorithms (compilation and synthesis), but also by architectures and design disciplines.

The two demands of heterogeneity and constructivity pull in different directions. Encompassing heterogeneity looks outward, towards the integration of theories to provide a unifying view for bridging the gaps between analytical and computational models, and between critical and best-effort techniques. Achieving constructivity looks inward, towards developing a tractable theory for system construction. Since constructivity is most easily achieved in restricted settings, an Embedded Systems Design Science must provide the means for intelligently balancing and trading off both ambitions.

4.1 Encompassing Heterogeneity

System designers deal with a large variety of components, each having different characteristics, from a large variety of viewpoints, each highlighting different dimensions of a system. Two central problems are the meaningful composition of heterogeneous components to ensure their correct interoperation, and the meaningful refinement and integration of heterogeneous viewpoints during the design process. Superficial classifications may distinguish between hardware and software components, or between 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 perspectives.

Heterogeneity of 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 coupling of components is the standard model for most synthesizable hardware and for hard real-time software. At the other extreme are completely asynchronous components, which proceed at independent speeds and interact nonatomically. Such a loose coupling of components is the standard model for most multithreaded software. Between the two extremes, a variety of intermediate and hybrid models

exist (e.g., globally-asynchronous locally-synchronous models). To better understand their commonalities and differences, it is useful to decouple execution from interaction semantics [21].

Execution semantics. Synchronous execution is typically used in hardware, in synchronous programming languages, and in time-triggered systems. It considers a system's execution as a sequence of global steps. It assumes synchrony, meaning 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. Asynchronous execution, by contrast, does not use any notion of global computation step. It is adopted in most distributed systems description languages such as SDL [16] and UML, and in multithreaded programming languages such as Ada and Java. The lack of built-in mechanisms for sharing computation between components can be compensated through constraints on scheduling (e.g., priorities; fairness) and through mechanisms for interaction (e.g., shared variables).

Interaction semantics. Interactions are combinations of actions performed by system components in order to achieve a desired global behavior. Interactions can be atomic or nonatomic. For atomic interactions, the state change induced in the participating components cannot be altered through interference with other interactions. As a rule, synchronous programming languages and hardware description languages use atomic interactions. By contrast, languages with buffered communication (e.g., SDL) and multithreaded languages (e.g., Java) generally use nonatomic interactions. Both types of interactions may involve strong or weak synchronization. Strongly synchronizing interactions can occur only if all participating components agree (e.g., CSP rendezvous). Weakly synchronizing interactions are asymmetric; they require only the participation of an initiating action, which may or may not synchronize with other actions (e.g., outputs in synchronous languages).

Heterogeneity of abstractions. System design involves the use of models that represent a system at varying degrees of detail and are related to each other in an abstraction (or equivalently, refinement) hierarchy. Heterogeneous abstractions, which relate different styles of models, are often the most powerful ones: a notable example is the boolean-valued gate-level abstraction of real-valued transistor-level models for circuits.

In embedded systems, a key abstraction is the one relating application software to its implementation on a given platform. Application software is largely untimed, in the sense that it abstracts away from physical time. References to physical time may occur in the parameters of real-time statements, such as timeouts, which are treated as external events. The application code running on a given platform, however, is a dynamical system that can be modeled as a timed or hybrid automaton [1]. The run-time state includes not only the variables of the application software, but also all variables that are needed to characterize

its dynamic behavior, including clock variables. Modeling implementations may require additional quantitative constraints, such as probabilities to describe failures, and arrival laws for external events. We need to find tractable theories to relate the application and implementation layers. In particular, such theories must provide the means for preserving, in the implementation, all essential properties of the application software.

Another cause of heterogeneity in abstractions is the use of different abstractions for modeling different extra-functional dimensions (or 'aspects') of a system. Some dimensions, such as timing and power consumption in certain settings, may be tightly correlated; others, such as timing and fault tolerance, may be achievable through independent, composable solutions. In general we lack practical theories for effectively separating orthogonal dimensions, and for quantifying the trade-offs between interfering dimensions.

Metamodeling. We are not the first to emphasize the need for encompassing heterogeneity in systems design. Much recent attention has focused on so-called 'metamodels,' which are semantic frameworks for expressing different models and their interoperation [2, 9, 3]. We submit that we need a metamodel which is not just a disjoint union of submodels within a common (meta)language, but one which preserves properties during model composition and supports meaningful analyses and transformations across heterogeneous model boundaries. This leads to the issue of constructivity in design.

4.2 Achieving Constructivity

The system construction problem can be formulated as follows: "build a system meeting a given set of requirements from a given set of components." This is a key problem in any engineering discipline; it lies at the basis of various systems design activities, including modeling, architecting, programming, synthesis, upgrading, and reuse. The general problem is by its nature intractable. Given a formal framework for describing and composing components, the system to be constructed can be characterized as a fixpoint of a monotonic function which is computable only when a reduction to finite-state models is possible. Even in this case, however, the complexity of the algorithms is prohibitive for real-world systems.

What are the possible avenues for circumventing this obstacle? We need results in two complementary directions. First, we need construction methods for specific, restricted application contexts characterized by particular types of requirements and constraints, and by particular types of components and composition mechanisms. Clearly, hardware synthesis techniques, software compilation techniques, algorithms (e.g., for scheduling, mutual exclusion, clock synchronization), architectures (such as time-triggered; publish-subscribe), as well as protocols (e.g., for multimedia synchronization) contribute solutions for specific contexts. It is important to stress that many of the practically interesting results require little computation and guarantee correctness more or less by construction.

Second, we need theories that allow the incremental combination of the above results in a systematic process for system construction. Such theories would be particularly useful for the integration of heterogeneous models, because the objectives for individual subsystems are most efficiently accomplished within those models which most naturally capture each of these subsystems. A resulting framework for incremental system construction is likely to employ two kinds of rules. *Compositionality* rules infer global system properties from the local properties of subsystems (e.g., inferring global deadlock-freedom from the deadlock-freedom of the individual components). *Noninterference* rules guarantee that during the system construction process, all essential properties of subsystems are preserved (e.g., establishing noninterference for two scheduling algorithms used to manage two system resources). This suggests the following action lines for research.

Constructivity for performance and robustness. The focus must shift from compositional methods and architectures for ensuring only functional properties, to extra-functional requirements such as performance and robustness.

Performance. The key issue is the construction of components (schedulers) that manage system resources so as to meet or optimize given performance requirements. These cover a large range of resource-related constraints involving upper and lower bounds, averages, jitter, and probabilities. Often the requirements for different resources are antagonistic, for instance, timeliness and power efficiency, or respecting deadlines and maximizing utilization. Thus we need construction methods that allow the joint consideration of performance requirements and the analysis of trade-offs.

Another inherent difficulty in the construction of schedulers comes from uncertainty and unpredictability in a system's execution and external environments. In this context, poor precision for time constants used in static scheduling techniques implies poor performance [23]. One approach is to build adaptive schedulers, which control execution by dynamically adjusting their scheduling policies according to their knowledge about the system's environment. However, currently there is no satisfactory theory for combining adaptive techniques for different kinds of resources. Such an approach must address the concerns of critical systems engineering, which currently relies almost exclusively on static techniques. The development of a system construction framework that allows the joint consideration of both critical and noncritical performance requirements for different classes of resources is a major challenge for the envisioned Embedded Systems Design Science.

Robustness. The key issue is the construction of components performing as desired under circumstances that deviate from the normal, expected operating environment. Such deviations may include extreme input values, platform failures, and malicious attacks. Accordingly, robustness requirements include a broad spectrum of properties, such as safety (resistance to failures), security (resistance to attacks), and availability (accessibility of resources). Robustness is a transversal issue in system construction, cutting across all design activities and

influencing all design decisions. For instance, system security must take into account properties of the software and hardware architectures, information treatment (encryption, access, and transmission), as well as programming disciplines. The current state of the art in building robust systems is still embryonic. A long-term and continuous research effort is necessary to develop a framework for the rigorous construction of robust systems. Our purpose here is only to point out the inadequacy of some existing approaches.

In dynamical systems, robustness can be formalized as continuity, namely, 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. Worse, many of our models for embedded systems are nonrobust even in the continuous domain. For example, in timed automata, an arbitrarily small change in the arrival time of an input may change the entire behavior of the automaton.

In computer science, redundancy is often the only solution to build reliable systems from unreliable components. We need theories, methods, and tools that support the construction of robust embedded systems without resorting to such massive, expensive overengineering. One hope is that continuity can be achieved in fully quantitative models, where quantitative information expresses not only probabilities, time, and other resource consumption levels, but also functional characteristics. For example, if we are no longer interested in the absolute (boolean-valued) possibility or nonpossibility of failure, but in the (real-valued) mean-time to failure, we may be able to construct continuous models where small changes in certain parameters induce only small changes in the failure rate.

Incremental construction. A practical methodology for embedded systems design needs to scale, and overcome the limitations of current algorithmic verification and synthesis techniques. One route for achieving scalability is to rely on compositionality and noninterference rules which require only light-weight analyses of the overall system architecture. Such correct-by-construction techniques exist for very specific properties and architectures. For example, time-triggered architectures 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 extend the correct-by-construction paradigm by studying more generally the interplay between architectures and properties.

A related class of correct-by-construction techniques is focused on the use of component interfaces [7]. A well-designed interface exposes exactly the information about a component which is necessary to check for composability with other components. In a sense, an interface formalism is a 'type theory' for component composition. Recent trends have been towards rich interfaces, which expose functional as well as extra-functional information about a component, for example, resource consumption levels. Interface theories are especially promising for incremental design under such quantitative constraints, because the composition

of two or more interfaces can be defined as to calculate the combined amount of resources that are consumed by putting together the underlying components.

5 Summary

We believe that the challenge of designing embedded systems offers a unique opportunity for reinvigorating Computer Science. The challenge, and thus the opportunity, spans the spectrum from theoretical foundations to engineering practice. To begin with, we need a mathematical basis for systems modeling and analysis which integrates both abstract-machine models and transfer-function models in order to deal with computation and physical constraints in a consistent, operative manner. Based on such a theory, it should be possible to combine practices for critical systems engineering to guarantee functional requirements, with best-effort systems engineering to optimize performance and robustness. The theory, the methodologies, and the tools need to encompass heterogeneous execution and interaction mechanisms for the components of a system, and they need to provide abstractions that isolate the subproblems in design that require human creativity from those that can be automated. This effort is a true grand challenge: it demands paradigmatic departures from the prevailing views on both hardware and software design, and it offers substantial rewards in terms of cost and quality of our future embedded infrastructure.

Acknowledgments. We thank Paul Caspi and Oded Maler for valuable comments on a preliminary draft of this manuscript.

References

1. R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.
2. F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A.L. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Computer*, 36(4):45–52, 2003.
3. K. Balasubramanian, A.S. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema. Developing applications using model-driven design environments. *IEEE Computer*, 39(2):33–40, 2006.
4. T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.
5. A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, third edition, 2001.
6. D.E. Culler and W. Hong. Wireless sensor networks. *Commununications of the ACM*, 47(6):30–33, 2004.
7. L. de Alfaro and T.A. Henzinger. Interface-based design. In M. Broy, J. Grünbauer, D. Harel, and C.A.R. Hoare, editors, *Engineering Theories of Software-intensive Systems*, NATO Science Series: Mathematics, Physics, and Chemistry 195, pages 83–104. Springer, 2005.

8. P.J. Denning and A. McGettrick. Recentering Computer Science. *Commununications of the ACM*, 48(11):15–19, 2005.
9. J. Eker, J.W. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity: The Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.
10. P.H. Feiler, B. Lewis, and S. Vestal. The SAE Architecture Analysis and Design Language (AADL) Standard: A basis for model-based architecture-driven embedded systems engineering. In *Proceedings of the RTAS Workshop on Model-driven Embedded Systems*, pages 1–10, 2003.
11. N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.
12. D. Harel. A grand challenge for computing: Full reactive modeling of a multicellular animal. *Bulletin of the EATCS*, 81:226–235, 2003.
13. T.A. Henzinger, C.M. Kirsch, M.A.A. Sanvido, and W. Pree. From control models to real-time code using Giotto. *IEEE Control Systems Magazine*, 23(1):50–64, 2003.
14. T.A. Henzinger, E.A. Lee, A.L. Sangiovanni-Vincentelli, S.S. Sastry, and J. Sztipanovits. *Mission Statement: Center for Hybrid and Embedded Software Systems*, University of California, Berkeley, http://chess.eecs.berkeley.edu, 2002.
15. C.A.R. Hoare. The Verifying Compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.
16. ITU-T. *Recommendation Z-100 Annex F1(11/00): Specification and Description Language (SDL) Formal Definition*, International Telecommunication Union, Geneva, 2000.
17. H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.
18. E.A. Lee. Absolutely positively on time: What would it take? *IEEE Computer*, 38(7):85–87, 2005.
19. P.R. Panda. SystemC: A modeling platform supporting multiple design abstractions. In *Proceedings of the International Symposium on Systems Synthesis (ISSS)*, pages 75–80. ACM, 2001.
20. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, second edition, 2004.
21. J. Sifakis. A framework for component-based construction. In *Proceedings of the Third International Conference on Software Engineering and Formal Methods (SEFM)*, pages 293–300. IEEE Computer Society, 2005.
22. J.A. Stankovic, I. Lee, A. Mok, and R. Rajkumar. Opportunities and obligations for physical computing systems. *IEEE Computer*, 38(11):23–31, 2005.
23. L. Thiele and R. Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, 2003.