Two Challenges in Embedded Systems Design: Predictability and Robustness

Thomas A. Henzinger

EPFL Station 14, 1015 Lausanne, Switzerland
EECS, University of California, Berkeley, CA 94720-1770, U.S.A.

Abstract. We discuss two main challenges in embedded systems design: the challenge to build predictable systems, and the challenge to build robust systems. We suggest how predictability can be formalized as a form of determinism, and robustness, as a form of continuity.

1 Introduction

An embedded software system is sometimes defined as a computing system that interacts with the physical world. This definition is incomplete, because every software system, once it is up and running, interacts with the physical world. More precisely, what is meant is that an embedded software system has nonfunctional requirements, which concern the system's interaction with the physical world.

There are two interfaces of a software system with the physical world: the environment and the platform. The environment includes the human users of the system, possibly a physical plant that is controlled by the system, and other application software processes that interact with the system. The platform consists of software and hardware components that implement a virtual machine on which the system is executed; it includes the operating system and network, with specific scheduling and communication mechanisms. Correspondingly, the nonfunctional requirements of an embedded software system can be classified into [9]

- *reaction requirements*, which concern the interaction of the system with the environment, and
- *execution requirements*, which concern the interaction of the system with the platform.

The source of reaction requirements is user expectation, where the user may be a human, a physical plant, or some other software. A common reaction requirement is response time, which bounds the worst-case or average-case delay

between an external stimulus of the system and its response. The source of execution requirements are resource constraints, which may be hardware imposed —such as limits on available machine cycles, memory space, battery capacity, and channel bandwidth— or software imposed —such as the use of a specific scheduling algorithm or communication protocol. Like reaction requirements, execution requirements —e.g., a bound on the power consumption of the system— may be hard (worst-case) or soft (average-case). While reaction requirements are independent of the platform, execution requirements change from platform to platform.

By contrast, a system that is not embedded has only functional requirements. For example, the functional requirement on a sorting program is that it outputs a sorted permutation of the input. For sorting programs, usually there is neither a reaction requirement (it is not specified when the sorted output must be provided) nor an execution requirement (it is not specified how much memory space or energy the sorting program may consume). The reaction and execution requirements are absent not because there are no corresponding user expectations or resource constraints —in practice the patience of a user to wait for the response of a sorting program is limited, and so is the available memory space— but these constraints are neglected because they are secondary to the functional requirements. In other words, to control design complexity, it is useful to abstract reaction and execution constraints whenever this can be done, i.e., when writing non-embedded software. Indeed, the success of high-level programming languages is built to a large degree on their ability to relieve the programmer from worrying about execution details such as memory management. This is why introductory programming is best taught in a language with garbage collection. It is, however, not prudent to abstract the response time of the electronic braking system in automobiles, nor the power consumption of remote sensor nodes that scavenge their energy from the environment. In these examples, the reaction and execution requirements are not secondary to the functional requirements, but they are integral to the correct operation of the system. It is for this reason that conventional high-level languages are not suitable for programming safety-critical real-time and highly resource-constrained software systems.

This raises what we consider to be the grand challenge in embedded software design: to identify high-level programming models that expose the reaction and execution properties of a system in a way that

1. permits the programmer to express desired reaction and execution requirements and

2. permits the compiler and run-time system to ensure that these requirements are satisfied.

While there have been many proposals to incorporate, say, real time into high-level programming languages [2], we believe that none of these attempts has successfully achieved both objectives. For example, the implicit specification of timing properties through the use of task priorities renders objective (2) trivial

on a preemptive platform, but fails with respect to objective (1), because it is often difficult for the programmer to foresee the effect that a given priority assignment has on the real-time behavior of the system. As a result, task priorities are often tuned during the testing phase of the system, only to be adjusted again whenever the environment or platform changes.

We submit that a successful solution to the grand challenge in embedded software design has to exhibit two key characteristics. First, the programming model must have the property that all software written in the model is *predictable*, in particular, not only predictable in its functional properties (what output does a program compute?) but also predictable in its reaction properties (e.g., when does the program provide the output?) and in its execution properties (e.g., what resources does the computation consume?). We will formalize the notion of predictability as a form of determinism. Second, the programming model must have the property that all software written in the model is *robust*, in the sense that its reaction properties change only slightly if the environment changes slightly, and that its execution properties change only slightly if the platform changes slightly. We will formalize the notion of robustness as a form of continuity. The remainder of this article will, in turn, discuss these two universal meta-requirements on embedded software systems: predictability and robustness.

2 Predictability through Determinism

The first universal challenge in systems design is the construction of systems whose behavior can be predicted. In embedded systems, the interesting aspects of system behavior emcompass not only functionality, but also reaction and execution properties such as timing and resource consumption. For the sake of illustration, we will concentrate in the following on timing. For this purpose we will use a notion of system behavior that includes, in addition to the values that are being computed, also the times at which the computed values become available. If other nonfunctional dimensions of behaviors are of interest, such as power consumption, then similar arguments can be made.

Predictability, at first glance, seems at odds with the concept of nondeterminism. A nondeterministic process is a process that may continue in several different ways, and thus, to predict the outcome of the process, all possible continuations need to be considered. This is often a difficult and prohibitively expensive task. Nondeterminism, on the other hand, is one of the central, defining notions of Computer Science and lies at the very heart of several fundamental questions: Is nondeterministic computation more efficient than deterministic computation (P v. NP)? Is there an observable difference between two actions being executed concurrently and the same two actions being executed sequentially in a nondeterministic order? Etc.

One approach to the building of predictable systems is to build them entirely from deterministic parts. This would require to use only processors for which

the execution time of each instruction is predictable, in particular, independent of cache and memory accesses; communication channels for which the delivery time of each message is predictable; etc. We believe that such a fully deterministic approach cannot provide a generally acceptable solution for two reasons. First, some residual sources of nondeterminism, such as the possibility of hardware failure, are difficult if not impossible to remove. Indeed, as circuit elements become smaller, they also become less predictable. Second, the fully deterministic approach voluntarily foregoes one of the most successful design principles, namely, that certain forms of nondeterminism can be controlled by hiding them underneath a higher, deterministic layer of abstraction. Such a deterministic layer is typically a programming model (but it could also be a processor model [6]). Thus we believe that the key question before us is not how we can build complex embedded systems from deterministic parts, but how we can build deterministic abstraction layers from nondeterministic parts.

To approach this question, we need to distinguish between several sources of nondeterminism:

1. input nondeterminism,
2. unobservable implementation nondeterminism,
3. don't-care nondeterminism, and
4. observable implementation nondeterminism.

We will argue that the fourth kind of nondeterminism, and only the fourth kind, needs to be avoided in order to design predictable systems. Since we focus on software systems, we generally refer by the term "system" to a program written in a high-level programming language, and by the term "implementation," to a lower layer of abstraction, such as machine code. Hence, when we classify a system as deterministic (or in the next section, continuous), we appeal to the semantics of the system as defined by the programming model (i.e., the definition of the programming language). Other, more hardware-centric views reserve the term "system" for the concrete, deployed artifact; in their terminology, what we call a deterministic (or continuous) system, would be referred to as an executable, deterministic (or continuous) "model" [10].

The first source of nondeterminism is the environment of the system, which is free to behave in many different ways. This kind of nondeterminism, called *input nondeterminism*, is uncontrollable; it must be present whenever a system reacts to environment stimuli. Therefore, when we refer to a reactive system as *deterministic*, what we mean is not that there is a unique stream of input and output values, but that for every stream of input values that is provided by the environment, the stream of output values that is computed by the system is unique. For embedded systems, the input and output streams must include time stamps. More precisely, a timed input stream is a sequence of time-stamped input values, such as sensor readings or user commands; and a timed output stream is a sequence of time-stamped output values, such as actuator updates

and other generated events that are observable by the environment. We say that an embedded system is *time-deterministic* if for every timed input stream, the timed output stream that is computed by the system is unique [8, 10]. Note that time-determinism refers not only to input and output values, but also to the times at which input values are given to the system and the times at which output values are made available to the environment. If an embedded system computes a unique output value, but may make the value available to the environment (say, by updating an actuator) at different time instants, then the system is not time-deterministic. Obviously, time determinism is essential for safety-critical real-time systems such as those deployed to control automobiles or aircraft.

The second source of nondeterminism is the omission of implementation details that do not influence the observable behavior of the system. For example, we may specify a sorting algorithm by saying that as long as there are two adjacent values x and y with $x > y$ in the input sequence, the two values are swapped, without specifying how the input sequence is scanned in order to find such values. No matter whether the input sequence is scanned left-to-right (as in conventional bubble sorting) or right-to-left or in any other way, the result is the same sorted output sequence. We call this kind of nondeterminism *unobservable implementation nondeterminism*, because the resolution of nondeterministic choices does not affect the uniqueness of the outcome. Unobservable implementation nondeterminism is not a source of unpredictability because, by definition, it has no bearing on the observable behavior of the system. Unobservable implementation nondeterminism is nonetheless helpful as it prevents overspecification, permits efficient implementations, and often simplifies correctness arguments. Since our definition of time-determinism refers only to input and output values and their times, it does allow unobservable implementation nondeterminism.

The third source of nondeterminism are don't-care values in the output behavior of a system. For example, we may not care about the output value y whenever the input value x satisfies a certain condition, or we may not care about the precise time of an output event. This kind of nondeterminism, which we call *don't-care nondeterminism*, is again useful as it prevents overspecification. Yet don't-care nondeterminism is not allowed by our definition of time-determinism, which requires a unique output value and output time for each input value and time. However, we can accommodate don't-care nondeterminism easily by modifying the value and time domains for outputs. If explicit "don't-care" values are added to these domains, then the notion of time-determinism does not need to be modified. For example, if on input value 0 the system may output either 0 or 1, then $\{(0,0),(0,1)\}$ is a nondeterministic representation of the system behavior (there are two possible outputs for the same input), while $\{(0,\perp)\}$, where $\perp$ denotes a "don't-care" value, is a deterministic representation of the same system behavior (the unique possible output is $\perp$). This small mathematical trick, which allows don't-care nondeterminism in deterministic systems, shows that don't-care nondeterminism, while useful, is not essential.

The last source of nondeterminism is the omission of implementation details that do influence the observable behavior of the system. For example, a system with several tasks may compute different output values depending on the order in which the tasks are scheduled. If the scheduler is nondeterministic, or deterministic but not observable, then there are many possible output streams generated by the system for the same input stream. We submit that this kind of *observable implementation nondeterminism* is harmful and must be avoided because it leads to unpredictable behavior. While unobservable implementation nondeterminism can be hidden underneath a deterministic programming model, and don't-care nondeterminism is shallow, observable implementation nondeterminism is often difficult to control and prohibitively expensive to analyze. Design in the presence of observable implementation nondeterminism requires the exhaustive consideration of all nondeterministic choices and their consequences. Hence, in order to build predictable systems, a designer must strive to minimize all observable implementation nondeterminism.

Consider the case of non-embedded programming. Unobservable implementation nondeterminism has created some of the most powerful programming abstractions by freeing the programmer from managing inconsequential implementation details and, at the same time, enabling the compiler to perform optimizations. For example, the exact memory layout of data structures does not influence the results of most conventional programs. Thus, memory management is unobservable implementation nondeterminism better left to the compiler than to the programmer. In stark contrast, observable implementation nondeterminism is the result of some of the most problematic programming abstractions. The thread model for concurrent programming falls into this class. Concurrent threads may be interleaved in any way, thus producing highly nondeterministic results. The programmer is left to manually reduce this nondeterminism by introducing locks and other synchronization constructs, which can have other unwanted consequences such as deadlocks. The current research in transactional memories is aimed at addressing the problem exactly by removing observable implementation nondeterminism, by implementing the slogan "think sequential, run parallel" [1]. Actor models of concurrency offer a different approach to achieve the same goal [12].

In embedded programming, the challenge is to build time-deterministic systems. This is difficult because the scheduler has direct impact on the times at which output values are computed and, due to race conditions, may have indirect impact on the computed values themselves. As a consequence, scheduled systems are often unpredictable. While there have been successes in removing this observable implementation nondeterminism —most prominently, the synchronous programming languages [7]— we have no widely acceptable solution, especially for soft (average-case) real-time requirements. A key question is if scheduling can be turned into unobservable implementation nondeterminism, similar to memory management: Is it possible to have the times of output events specified by the programmer and ensured by the compiler? One approach in this direction has the compiler generate a schedule that guarantees that each output value is

computed before it is needed and then withheld until the specified time [8]. The withholding of computed output values until the times specified by the program yields a time-deterministic system.

3 Robustness through Continuity

A second universal challenge in systems design is the construction of systems whose behavior is robust in the presence of perturbations. While robustness is understood for most physical engineering artifacts, it is rarely heard of in connection with software. This is because computer programs can be readily idealized as discrete mathematical objects —a perspective that has been advocated in Computer Science since the 1960s [13]. If a program is studied as a discrete mathematical object, then correctness is a boolean notion and can be established by proof: the program either satisfies its requirements, or it does not. This prevalent view of programs as discrete partial functions on values or states has led to tremendous successes: it enabled the most fundamental paradigms of the science of computing, including the theories of computability, complexity, and semantics.

However, in Computer Science, unlike in other engineering disciplines, we often lose sight of the fact that the mathematical representation of a software system is just a model, and that the actual system is physical, executing on a physical, imperfect platform and interacting with a physical, unknowable environment. The realization that programs are ultimately physical shatters the boolean illusion: Of two mathematically correct programs, one may be preferable to the other because of the way it behaves if the platform or environment deviates from the nominal expectations, be it because of resource limitations, failures, attacks, or simply erroneous specifications. To some degree this observation guides the design of robust non-embedded software, for example, by having the system check if the input values lie within expected ranges. Moreover, one program may be more fault-tolerant than another, functionally equivalent program; secure against a larger class of attacks; etc. But the incompleteness of the clean, boolean view becomes most apparent in embedded programming, where computing explicitly meets the physical world. This is because, often, the reaction and execution properties of a system —such as response time or power consumption— are best measured in terms of continuous quantities, and they may satisfy a design specification to different degrees.

In order to study computation as an imperfect, physical process, we must develop metrics for preferring one system over another. Given a set of requirements, a preference metric provides a measure of how close a system comes to meeting the requirements, and how robust it is against small changes in the requirements [3]. Both reaction and execution requirements may refer to several dimensions, such as the precision of the output values, their timeliness, the physical cost of the system, and its expected lifetime. Hard, worst-case requirements must be given more weight than soft, best-effort requirements, and in general

it will be unlikely that a single design point is to be preferred according to all criteria. In such cases, design decisions need to be based on a Pareto analysis that illuminates the trade-offs [14]. In short, the boolean constraint-satisfaction problem of building a system that satisfies the requirements is to be replaced by a multi-dimensional optimization problem.

In a quantitative setting, we can formalize robustness properties as mathematical continuity: A system is *continuous* if continuous changes in the environment or platform cannot cause discontinuous changes in the system behavior. For a continuous system, for every positive real epsilon there must be a positive real delta such that, for example, delta changes in the values and times of sensor readings cause at most epsilon changes in the values and times of actuator updates; or, delta changes in the processor speeds and loads cause at most epsilon changes in output values and times. Such continuity properties can hold, of course, only for continuous variables such as real time and other physical quantities; for discrete variables such as switches, the flipping of a single bit may completely alter the behavior of a system. However, embedded systems and their requirements refer to many continuous quantities, yet we have no design theory or methodology for building continuous systems. For example, a system that reads a sensor value, adds a constant, and outputs the result after some constant time delay behaves continuously, both in output value and time. On the other hand, a system that checks if the sensor value is greater than a threshold, and depending on the result of the check, invokes two different control algorithms with two potentially different execution times to compute the output, does not behave continuously. Since branching is not continuous, it is usually infeasible to build continuous software systems entirely from continuous parts; so the question before us is how to build continuous systems from non-continuous parts. In the above example, a preferred way to combine the two control algorithms may be to execute both procedures if the input value is near the threshold, to average the result, and to withhold the output until a specific time that is independent of the computation times.

Continuity properties are essential not only because we expect the system performance to degrade smoothly if the environment or platform changes in unforeseen ways, accidentally or maliciously, but also because physical quantities cannot be measured with infinite precision. In this sense, continuity is a natural requirement with respect to real-valued variables. There is also an altogether more radical approach, which looks at entire discrete transition systems through a continuous lense. For example, for a given program, the boolean value of whether a functional requirement is satisfied forever may be replaced by a quantitative value that measures for how many transitions the requirement is satisfied [5]. The goal would be to define a continuous mathematics of programs where the resulting notion of robustness, formalized as continuity, naturally corresponds to our intuition of a system behaving well under disturbances.

Before concluding, let us have a brief look at a potential way of designing reliable systems which touches both on determinism and on robustness. Suppose that

an embedded control system updates an actuator value x at periodic times. Not all of the updates of x are going to be correct, because sensors may give faulty readings, processors may be too overloaded to compute new actuator values on time, hardware may fail, etc. We give the programmer the option to specify the desired reliability of x as a real number between 0 and 1. In particular, a reliability of 0.999 means that in the long-run, at most 1 out of 1000 actuator updates writes an invalid value; or more precisely, if a valid update has value 1 and an invalid update value 0, then the limit average of the infinite number of updates generated by the system is at least 0.999 [4]. This reliability requirement is nondeterministic, as there are many different infinite sequences of 0's and 1's with a limit average of at least 0.999. However, we can make the reliability requirement deterministic by strengthening it slightly. For this purpose, we specify the output of the system as a pair consisting of a stream of actuator values x_i, for $i \geq 0$, and a probability p (in our example, $p = 0.999$). The behaviors of the system are precisely those streams that are generated by an infinite repetition of a random coin toss that, in the ith round, yields the specified value x_i with probability p and a special "don't-care" value with probability $1-p$. By the law of large numbers, the system behaviors satisfy the reliability requirement with probability 1. Moreover, the reliable system is now specified by a unique stochastic process generating outputs, which makes the definition deterministic.

This distinction between nondeterministic and probabilistic outcomes is important, as in situations where we cannot hope to build fully reliable systems, we still need to strive for building systems whose failures are probabilistically predictable. For instance, if on input value 0 the system outputs 0 with probability 0.6 and 1 with probability 0.4, then even though it is probabilistic, the system behavior is fully determined, and in that sense predictable. Intuitively, while drawing a random ball from an urn with unknown numbers of red and black balls has a nondeterministic outcome, depending on the ratio of red and black balls, such a drawing from an urn with known numbers of red and black balls has an outcome that, albeit stochastic, is completely specified. Technically, this is similar to our handling of don't-care nondeterminism in deterministic systems: there, several possible output values are specified by a unique don't-care value; here, several possible output values are specified by a unique stochastic process, that is, by the value of a random variable. In other words, probabilistic behavior is a general case of deterministic behavior.

Back to our example: While the programmer specifies the reliability requirement, the compiler must ensure it, based on assumptions about the failure rates of individual sensors, processors, actuators, and communication links. In particular, if the probability of a failing computation on a single processor exceeds 0.001, then the compiler has to replicate the computation of each actuator value x_i on several processors and use a voting or averaging procedure to compute the result. Furthermore, to achieve continuity, the compiler must ensure that if the failure assumptions are slightly incorrect, then the reliability requirement is only missed by a small amount; or precisely, for every positive real

epsilon there exists a positive real delta such that if the failure assumptions are wrong by delta, then the system reliability lies within epsilon of the target.

4 Conclusion

We outlined what we believe to be two major challenges in embedded systems design:

1. the challenge to build, on top of nondeterministic system implementations, system abstractions that are deterministic with regard to nonfunctional properties such as time and resource consumption; and

2. the challenge to build, on top of noncontinuous system implementations, system abstractions that are continuous with regard to physical quantities.

Of course, to be of help in systems design, the abstractions need to support compilers and other techniques for synthesizing implementations, usually from a given collection of building blocks such as a specified instruction set. Both challenges require a rethinking of the conventional, purely discrete, purely functional (i.e., nonphysical) foundation of computing. Embedded systems design, therefore, offers a prime opportunity to reinvigorate Computer Science [9, 11].

Acknowledgments. The author thanks Edward Lee, Joseph Sifakis, and the two anonymous referees for many valuable comments. Support for this work was provided by the National Science Foundation under the ITR grant CCR-0225610 and by the ArtistDesign and COMBEST projects of the European Union.

References

[1] F. Allen. *The Challenge of the Multi-Cores: Think Sequential, Run Parallel.* Regents' Lecture, University of California, Berkeley, 2008.

[2] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages.* Addison-Wesley, 2001.

[3] K. Chatterjee, et al. Compositional quantitative reasoning. In *Proc. IEEE Conference on Quantitative Evaluation of Systems* (QEST), pp. 179–188, 2006.

[4] K. Chatterjee, et al. Logical reliability of interacting real-time tasks. In *Proc. Design, Automation, and Test in Europe* (DATE), 2008.

[5] L. de Alfaro, T.A. Henzinger, and R. Majumdar. Discounting the future in systems theory. In *Proc. International Colloquium on Automata, Languages, and Programming* (ICALP), pp. 1022–1037, 2003.

[6] S.A. Edwards and E.A. Lee. The case for the precision-timed PRET machine. In *Proc. Design Automation Conference* (DAC), pp. 264–265, 2007.

[7] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, 1993.

[8] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE* 91:84–99, 2003.

[9] T.A. Henzinger and J. Sifakis. The discipline of embedded systems design. *IEEE Computer* 40(10):36-44, 2007.

[10] H. Kopetz. *The Complexity Challenge in Embedded System Design.* Tech. Rep. 55, Technical University of Vienna, 2007.

[11] E.A. Lee. Absolutely positively on time: What would it take? *IEEE Computer* 38(7):85–87, 2005.

[12] E.A. Lee. The problem with threads. *IEEE Computer* 39(5):33–42, 2006.

[13] J. McCarthy. Towards a mathematical science of computation. In *Proc. IFIP Congress*, pp. 21–28, 1962.

[14] R. Szymanek, F. Catthoor, and K. Kuchcinski. Time-energy design space exploration for multi-layer memory architectures. In *Proc. Design, Automation, and Test in Europe* (DATE), pp. 318–323, 2004.