# What Is Real Time Computing? A Personal View

Edward A. Lee

July 17, 2017

**Author Institution:** University of California at Berkeley

**Abstract:** Today, real-time behavior of programs is a property that emerges from implementations rather than a property that is specified in models. Control over timing behavior of software is difficult to achieve, and timing behavior is neither predictable nor repeatable. This paper argues that this problem can be solved by making a commitment to deterministic models that embrace temporal properties as an integral part of the modeling paradigm.

**Keywords:** real-time systems, determinism, cyber-physical systems.

## A. Introduction

The most interesting cyber-physical system (CPS) applications today, such as medical devices, factory automation, and autonomous vehicles, necessarily include timing-sensitive safety-critical physical sensing and actuation. The software in such systems is called real-time software because the designer has to pay attention to the timing of actions taken by the software. But what do we mean by paying attention to timing? Fundamentally, time only matters in computing when software interacts with the world outside the software. Without such interaction, time is irrelevant. Of course, all useful software interacts with the world outside itself, or we wouldn't bother to have the software. Hence, time matters for all software. But the goal of this paper is to tease out the situations where timing becomes critically important. A slow program may be annoyance, but a missed deadline in a control system could be deadly.

In practice, when engineers talk about "real time," they may mean[1]

1. fast computation,
2. prioritized scheduling,
3. computation on streaming data,

[1] In 1988, Stankovic cataloged quite a few more possible (mis)interpretations of the term "real time" and laid out a research agenda that is dishearteningly valid today [9].

4. bounded execution time,
5. temporal semantics in programs, or
6. temporal semantics in networks.

These are different views, and which view dominates has a strong effect on the choice of technical approaches to the problem.

The first of these, fast computation, is useful in all computation, and therefore does not deserve our attention here. Nothing about fast computation can distinguish real-time problems from non-real-time problems. In fact, many real-time systems execute on decidedly slow computers, such as microcontrollers, and timing precision, predictability, and repeatability may be far more important than speed.

The second meaning, prioritized scheduling, is the centerpiece of much work in the real-time systems community. In this approach, the requirements of the physical world are reduced to deadlines and periods, and the temporal properties of software are reduced to execution times for tasks. In this paper, I will take a more CPS approach where we are interested in the closed-loop interactions of the physical and cyber parts of the system. I will argue that these reductions are oversimplifications and that a more holistic approach has great promise for advancing the field.

The third meaning, computation on streaming data, has become a hot area in recent years. The emerging Internet of Things (IoT) promises a flood of sensor data. The research and consulting firm Forrester defines "perishable insights" as "urgent business situations (risks and opportunities) that firms can only detect and act on at a moment's notice." Fraud detection for credit cards is one example of such perishable insights. This has a real-time constraint in the sense that once a fraudulent transaction is allowed, the damage is done. In CPS, a perishable insight may be, for example, a determination of whether to apply the brakes on a car, where a wrong or late decision can be quite destructive.

Computing on streaming data fundamentally means that you don't have all the data, but you have to deliver results. It differs from standard computation in that the data sets are unbounded, not just big. You can't do random access on input data, which constrains the types of algorithms you can use [1]. Because data keeps coming, programs that halt are defective, in contrast to the standard Turing-Church view of computation, where programs that fail to halt are defective.

Computing on streaming data demands different software architectures, using for example *actors* as software components rather than *objects*. Many subtle semantic questions arise. For example, when streams converge, what is the meaning of the interleaving of their elements? Are their elements simply nondeterministically interleaved, or is there more meaning to the relationship between an element of one stream and that of another? Can algorithms process potentially infinite data streams with bounded state? How is feedback handled, where processes send each other streams? What about streams that are partially ordered rather than totally ordered? And perhaps most interesting for this paper, is there any temporal semantics in the streams? That is, is there any notion of time associated with the elements of the stream, and what is the semantics of that notion of time?

The fourth meaning, bounded execution time, assumes that some deadline exists for a software execution, and that ensuring that the execution never oversteps that deadline is sufficient. This meaning is central to the sense-process-actuate programming models and is usually a basic assumption of the second meaning, prioritized scheduling. As I will discuss below, however, bounding the execution time of software is particularly problematic. The bound can only be determined for a particular implementation, where every detail of not only the software, but also the hardware on which it runs and the execution context are known. Moreover, by itself, bounding the execution time of software does not ensure predictable behavior, since it does little to ensure that the order of actions taken by the software is invariant.

The fifth meaning, temporal semantics in programs, has a long history with little practical impact. Although many experimental programming languages with some notion of time have been created, none has survived. As I will discuss below, temporal semantics is absent in programs from the lowest level of abstraction, in the instruction-set architecture (ISA), and reintroducing it at higher levels has proved persistently problematic. We believe that for real progress to be made, temporal semantics needs to appear throughout the abstraction stack.

The final meaning, temporal semantics in networks, is present only in specialized networks, such as those in safety critical systems including factory automation, avionics, and automotive electronics. Recent work shows, however, that temporal semantics is not entirely incompatible with commodity networks. I will say more about this below.

## B. Controlling Timing in Software

All of the above interpretations of real-time computing require some measure of control over the timing of software. However, controlling timing of software is difficult. More important, achieving *repeatable* timing is difficult, and systems do not necessarily behave in the field in a manner similar to the test bench.

At the microarchitecture level, ISAs define the behavior of a microprocessor implementation in a way that makes timing irrelevant to correctness. Timing is merely a performance metric, not a correctness criterion. In contrast, arithmetic and logic *are* correctness criteria. A microprocessor that fails to take a branch when the condition evaluates to true is simply an incorrect implementation. But a microprocessor that takes a long time to take the branch is just slow. Computer architects have long exploited this property, that timing is irrelevant to correctness. They have developed clever ways to deal with deep pipelines, such as speculative execution and instruction reordering, and with memory heterogeneity, such as multi-level caches. These techniques, however, introduce highly variable and unpredictable timing. The goal is to speed up a typical execution, not to make executions repeatable.

The design of modern programming languages reflects the microarchitectural choice, so timing is again irrelevant to correctness. Hence, programmers have to step outside the programming abstraction to control timing, for example by writing to memory-mapped registers to set up a timer interrupt, or more indirectly, by making operating system calls

to trigger context switches. The result is timing granularity that is much more coarse than what is achievable in the hardware. More important, since interrupts occur unpredictably relative to whatever is currently executing, these techniques inevitably make behavior nonrepeatable. I contend that ISAs for CPS need to be rethought to provide repeatable and precise timing.

Today, most CPS applications stand to benefit enormously from being networked, an observation that underlies the current enthusiasm around IoT. But the networking technology that underlies the Internet also ignores timing. When mainstream networking pays attention to timing, the problem is put under the header "quality of service," which emphasizes that timing is viewed as a *quality* metric, not a *correctness* criterion. In contrast, reliable eventually delivery, realized by the widely used TCP protocol, is a correctness criterion.

Despite these challenges, engineers have managed to make reliable real-time systems. How? Some of the techniques used are:

1. overengineering,
2. using old technology,
3. response-time analysis,
4. real-time operating systems (RTOSs),
5. specialized networks, and
6. extensive testing and validation.

Overengineering is common because Moore's law has given us impressively fast processors. If the execution of software is essentially instantaneous with respect to the physical processes with which it is interacting, then the timing of the software becomes irrelevant. However, overengineering is becoming increasingly difficult as the complexity of CPS applications increases and as technology no longer tracks Moore's law. Moreover, many CPS applications are extremely cost sensitive or energy constrained, making overengineering a poor choice.

Using old technology is also common. Safety-critical avionics software, for example, rarely uses modern programming languages, operating systems, or even interrupts. Software is written at a very low level, I/O is done through polling rather than interrupts, and multitasking is avoided. Programmable logic controllers (PLCs), widely used in industrial automation, are often programmed using ladder logic, a notation that dates back to the days when the logic of digital controllers was entirely controlled with mechanical relays. And the software executes without the help of a modern operating system, sacrificing useful capabilities such as network stacks. Many embedded systems designers avoid multicore chips, a strategy that is becoming increasingly difficult as single-core chips become more rare. And programmers often disable or lock caches, thereby getting little advantage from the memory hierarchy.

The third approach, response-time analysis, includes execution-time analysis, which puts bounds on the time it takes for sections of code to execute [11], and analysis of factors such as operating system scheduling and mutual exclusion locks. Even just the subproblem of execution-time analysis is fundamentally hard because for all modern programming

languages, whether a section of code even terminates is undecidable. However, even when the execution paths through the code can be analyzed, sometimes with the help of manual annotations such as bounds on loops, the microarchitectural features mentioned above make analysis extremely difficult. The analysis tools need a detailed model of the particular implementation of the processor that will run the code, including every minute (and often undocumented) detail. As a result, a program that has been validated using execution-time analysis is only validated for the particular piece of silicon that has been modeled. With any change in the hardware, all bets are off; even though the new hardware will *correctly* execute the code, there is no longer any assurance that the *system* behavior is correct. Manufacturers of safety-critical embedded systems, therefore, are forced to stockpile the hardware that they expect to need for the entire production run of a product. This runs counter to most basic principles in modern supply chain management for manufacturing, and it makes it impossible to take advantage of technology improvements for cost reduction, improved safety, or reduced energy consumption.

Moreover, execution-time analysis tools often need to make unrealistic assumptions, such as that interrupts are disabled, in order to get reasonable bounds. But while interrupts are disabled, the software does not react to stimulus from the outside world, so the variability in reaction time may be significantly increased, undermining the value of execution-time analysis.

In practice, designers either avoid interrupts altogether (as commonly done in avionics) or attempt to keep program segments short so that the time during which interrupts are disabled is small. Both strategies are increasingly difficult as we demand more functionality from these programs. As execution time increases, either the polling frequency decreases or the variability of the timing of other tasks that get locked out by disabled interrupts increases.

The fourth technique, RTOSs, provides real-time scheduling policies in a multitasking operating system. At the core, RTOSs use timer interrupts and priorities associated with tasks. There is a long history of strategies that can be proven optimal under (often unrealistic) assumptions, such as bounds on execution time and well-known deadlines. In simple scenarios, these strategies can yield repeatable behaviors, but in more complex scenarios, they can even become chaotic [10], which makes behavior impossible to predict. Moreover, because of the reliance on interrupts, RTOSs violate the typical assumptions made for execution-time analysis, and thereby invalidate their own optimality proofs, which assume known execution times. A consequence is that when RTOSs deliver predictable timing, the precision of the resulting timing is several orders of magnitude coarser than what is in principle achievable with the underlying digital hardware.

The specialized networks that constitute the fifth approach use methods such as synchronized clocks and time-division multiple access (TDMA) to provide latency and bandwidth guarantees. Examples include CAN busses, ARINC busses, FlexRay, and TTEthernet. With the possible exception of TTEthernet, these networks are hard to integrate with the open Internet, so these systems cannot benefit from Internet connectivity nor from the economies of scale of Internet hardware and software.

The final approach, extensive testing and validation, is a laborious, brute-force engineering method. One automotive engineer described to me what he called “the mother

of all test drives," where you literally drive the car a million miles in as many conditions as you can possibly muster and hope that you have comprehensively covered all the behaviors that the cyber-physical system may exhibit in the field. But as the complexity of these systems (and their environments) increases, the likelihood that testing will be comprehensive becomes more remote.

Taken together, these techniques do make it *possible* to design safety-critical real-time embedded software, but their weaknesses suggest that it may be time to step back and reexamine the problem of real-time computing with fresh eyes. After all, microprocessors are realized in a technology, synchronous digital logic, that is capable of realizing sub-nanosecond timing precision with astonishing reliability and repeatability. It is the layers of abstraction overlaid on this technology, ISAs, programming languages, RTOSs, and networks, that discard timing. I contend that it is time for a paradigm shift where we make a commitment to deterministic models that include timing properties.

## C. What is Time?

Time, as a physical phenomenon, is poorly understood. CPS engineers mostly adopt a Newtonian view, where time is a continuum that advances uniformly and identically to all observers, even though we know from relativity that the flow of time depends on the observer, and some physicists suspect from quantum field theories that time may be discrete rather than a continuum. The Newtonian view is pragmatic and has proved effective for a wide range of physical system design problems. However, it does not translate easily to the cyber world, where everything is discrete and the dynamics of programs is a sequence of steps rather than a continuous flow.

Under the Newtonian model, an instant in time can be represented as a real number. In software, real numbers are almost always approximated by floating-point numbers. This works well when modeling continuous systems because for continuous systems, by definition, small perturbations have bounded effects, so the small errors introduced in floating-point arithmetic can often be safely neglected. However, when dealing with discrete systems or with mixed discrete and continuous systems, these same errors can have much bigger effects. With discrete behaviors, and hence with software, the order in which events occur, no matter how small the time difference between them, can drastically affect an outcome.

Broman *et al*. show that floating-point representations of time are incompatible with basic requirements for modeling hybrid systems, which mix discrete and continuous behaviors [3]. They offer an alternative, a superdense model of time with quantized resolution that exhibits a clean semantic notion of simultaneity. The key is to adopt a *model* of time, not one that attempts to solve the physics problem of what is time in the physical world, but rather one that expresses properties of real-time systems that we care about and that can be physically realized with high confidence at a reasonable cost. In other words, we need a useful temporal semantics.

What do we mean by "temporal semantics"? Consider a program that wishes to take two distinct orchestrated actions $A$ and $B$ at 100$\mu$s intervals. We can argue that it is physically impossible for these actions to be simultaneous to all observers, but

that would be missing the point. Even the meaning of "100$\mu$s intervals" is questionable in physics. Instead, we should admit that what we want is to have these actions be *logically* simultaneous and *reasonably* precise. What does this mean? It could mean that any observer of these actions within our system will at all times have counted the same number of actions $A$ and $B$ that have occurred. That is, if the observer has seen $n$ $A$ actions, then it has also seen $n$ $B$ actions. Note that this requirement is independent of timing precision and is most certainly physically realizable. It gives a clean semantic notion to simultaneity.

Another example of a useful temporal semantics property is reaction time. Suppose that we have a system that reacts to sporadic discrete events, and that we wish it to react to each event with a latency no greater than 100$\mu$s. Here, "sporadic" has a technical meaning without which we could never provide such an assurance. A sporadic stream of events is one where the time between events has a lower bound. Consider a scenario where we have two sporadic streams into a software system running on single CPU, where in each stream, the lower bound between events is 100 $\mu$s. Events arrive no more frequently than once per 100 $\mu$s, but possibly less frequently. The interleaving of events from these two streams is arbitrary, and events could even arrive simultaneously. Nevertheless, we wish to react to each event within 100 $\mu$s.

Today, we can solve this problem with interrupts, but since interrupts disrupt timing analysis, each event handler will have to disable interrupts while it handles its event. Because the timing of events in each stream is arbitrary, this interrupt-driven strategy will introduce considerable timing jitter. Suppose for example that the hander for events from stream $A$ requires 95 $\mu$s to complete, whereas the handler for stream $B$ requires only 5 $\mu$s. In this case, reactions to events from stream $B$ may occur in 5 $\mu$s or in 100 $\mu$s or anything in between, depending on whether an event from stream $A$ is being handled. This is a huge jitter compared to the reaction time.

A promising solution, not yet been widely adopted, is PRET machines [12]. PRET machines can give a deterministic temporal semantics to interrupt-driven reactions without any loss of performance. I will discuss next this idea of a deterministic temporal semantics and explain how it can overcome the limitations in today's real-time computing technologies.

## D. A Commitment to Models

All of engineering is built on models. For the purposes of this paper, I will define a "model" of a system to be any description of the system that is not Kant's thing-in-itself (*das Ding an sich*). Mechanical engineers use Newton's laws as models for how a system will react to forces. Civil engineers use models of materials to understand how structures react to stresses. Electrical engineers model transistors as switches, logic gates as networks of switches, and digital circuits as networks of logic gates. Computer engineers model digital circuits as instruction set architectures (ISAs), programs as executions in an ISA, and applications as networks of program fragments.

Every one of these models rests on a modeling paradigm. The Java programming language, for example, is just such a modeling paradigm. What constitutes a well-formed Java program is well defined, as is the meaning of the execution of such a program.

The program is a model of what a machine does when it executes the program. Synchronous digital circuits constitute another such modeling paradigm. They model what an electronic circuit does. Under the synchrony hypothesis, the latencies of logic gates are ignored, and the behavior of a network of logic gates and latches is given by Boolean algebra. Models abstract away details, and layers of models may be built one on top of another [8].

Properties of the modeling paradigm are fundamental when an engineer attempts to build confidence in a design. A synchronous digital circuit, as a model, realizes a deterministic function of its input, despite the fact that we have no useful deterministic model of the underlying physics comprising individual electrons sloshing in silicon and metal. A single-threaded Java program is also a deterministic function of its inputs. The determinism of these modeling paradigms is assumed without question by the engineer building these models. Without such determinism, we would not have billion-transistor chips and million-line programs handling our banking.

Does this mean that the execution of a Java program on a particular microprocessor chip is deterministic? This question, by itself, makes no sense. Determinism is a property of models, not of physical systems [8]. If the chip overheats or get submersed in salt water, the program will very likely not behave as expected. The physical realization has many properties that the model does not have.

More to the point for this paper, the timing exhibited by the Java program is not specified in the model (the Java program itself). Whether an execution of the program is correct does not depend on the timing, so within this model, an infinite number of timing behaviors are permitted. Nevertheless, we assert that the model (the single-threaded Java program) is deterministic because the model does not include timing in its notion of the behavior of the program.

The notion of determinism is not a simple one. Earman, in his *Primer on Determinism*, admits defeat in getting a "real understanding" of the concept [4, p. 21]. Earman insists that "determinism is a doctrine about the nature of the world," but I believe that a more useful view is that determinism is a property of *models* and not a property of the physical world. As a property of models, determinism is relatively easy to define:

> A model is deterministic if given an initial *state* of the model, and given all the *inputs* that are provided to the model, the model defines exactly one possible *behavior*.

In other words, a model is deterministic if it is not possible for it to react in two or more ways to the same conditions. Only one reaction is possible. In this definition, the italicized words must be defined within the modeling paradigm to complete the definition, specifically, "state," "input," and "behavior." Precise definitions of these words necessarily circumscribe the assumptions made by the designer. For example, if the timing of the execution of a Java program is included in the notion of "behavior," then no Java program is deterministic. For an example of a deterministic model, if the state of a particle is its position $x(t)$ in a Euclidean space at a Newtonian time $t$, where both time and space are continuums, and if the *input* $F(t)$ is a force applied to the particle at each instant $t$, and the *behavior* is the motion of the particle through space, then Newton's second law provides a

deterministic model.

One reason that this simple concept has been so problematic is that all too often, when speaking of determinism, the speaker is confusing the map for the territory (the model for the thing-in-itself). To even speak of determinism, we must define "input," "state," and "behavior." How can we define these things for an actual physical system? Any way we define them requires constructing a model. Hence, an assertion about determinism will actually be an assertion about the model not about the thing being modeled. Only a model can be unambiguously deterministic, which underscores Earman's struggle to pin down the concept.

Consider that any given physical system has more than one valid model. For example, a particle to which we are applying a force exhibits deterministic motion under Newton's second law but not under quantum mechanics, where the position of the particle will be given probabilistically. However, under quantum mechanics, the evolution of the particle's wave function is deterministic, following the Schrödinger equation. If the "state" and "behavior" of our model are the wave function, then the model is deterministic. If instead the state and behavior are the particle's position, then the model is nondeterministic. It makes no sense to assign determinism as a property to the particle. It is a property of the model.

If we have a deterministic model that is faithful to some physical system, then this model *may* have a particularly valuable property: the model may predict how the system will evolve in time in reaction to some input stimulus. This predictive power of a deterministic model is a key reason to seek deterministic models.

But a model can only predict aspects of behavior that lie within its modeling paradigm. My essential claim in this paper is that we should make a commitment to using models that include aspects of behavior that we care about. If we care about timing, we should use models that *do* include timing in their notion of behavior. Today, with real-time systems, we do not do that.

Instead, today, timing properties emerge from a physical implementation. When we map a particular program onto a particular microprocessor, a real physical chip embedded in a real board, with real memory chips and peripherals sharing the bus, only then do we get timing properties. Timing is a property of the thing-in-itself not of the model. We have conflated the map and the territory.

Determinism is a key property of many of the most successful modeling paradigms in engineering. Logic gates, synchronous digital circuits, ISAs, single-threaded programs, and Newtonian mechanics are all deterministic modeling paradigms. Should we insist on deterministic modeling paradigms for CPS?

If there is anything we can be sure about, it is that we can never be sure about cyber-physical systems. We cannot know everything about them, and particularly their possible behaviors in all environments. Does this mean that we can never build confidence in a CPS realization? No, because we can build confidence in *models* of the system. If the system itself, *das Ding an sich*, matches the model with high fidelity, then our confidence in the model translates into confidence in the system. We rely on such matching when we assume that a chip will correctly realize an ISA and correctly execute a program.

It is important to understand the distinction between the engineering and scientific uses

of models [8]. An engineer seeks a physical system to match a model, whereas a scientist seeks a model to match a physical system. For these two uses, determinism plays different roles. For an engineer, the determinism of a model is useful because it facilitates building confidence *in the model*. Logic gates, for example, are deterministic models of electrons sloshing around in silicon. The determinism of the logic gate model is valuable: it enables circuit designers to use Boolean algebra to build confidence in circuit designs that have billions of transistors. The model predicts behaviors *perfectly*, in that an engineer can determine how a logic gate model will react to any particular input, given any initial state.

Of course, the usefulness of the logic gate model also depends on our ability to build silicon structures that are extremely faithful to the model. We have learned to control the sloshing of electrons in silicon so that, with high confidence, a circuit will emulate the logic gate model billions of times per second and operate without error for years.

For a scientist, fundamentally, when considering the use of deterministic models, it matters quite a lot whether the physical system being modeled is also deterministic. The value of a deterministic logic gate model, however, does not depend at all on whether the sloshing of electrons in silicon is deterministic. It depends only on whether we can build silicon structures that emulate the model with high confidence. We do not need and cannot achieve perfection. As Box and Draper say, all models are wrong, but some models are useful [2], and logic gates have proved extremely useful.

I claim that for real progress to occur, we must make a commitment to deterministic models of timing and concurrency in cyber-physical systems. My essential claim is that we *can* build systems that match the behavior of such models with high confidence. This is not the same as a claim that we can construct useful deterministic models of today's cyber-physical systems. I am not making the latter claim.

What about adaptability, resilience, and fault tolerance? Any cyber-physical system will face the reality of unexpected behaviors and failures of components. Using deterministic models does not prevent us from making fault-tolerant and adaptive systems. On the contrary, it *enables* it. A deterministic model defines unambiguously what a *correct* behavior is. This enables detection of *incorrect* behaviors, an essential prerequisite to fault-tolerant adaptive systems.

The reader may protest that a deterministic model of time may be foiled by the fact that timing of programs is difficult to control. As I have pointed out, ISAs have no temporal semantics at all, and computer architects have developed a plethora of clever techniques that make timing difficult to control. Timing of programs is in practice controlled by external timer hardware that uses interrupts to react to timed requests. Interrupts are imprecise and they disrupt the hardware (cache, branch predictors, etc.) in ways that invalidate timing analysis of the programs that get interrupted.

I have already mentioned PRET machines, which are microarchitectures that deliver repeatable and controllable timing down to the precision of a clock cycle [12]. They are capable of interrupt-driven I/O that does not disrupt the timing of timing-critical tasks. I believe that PRET machines will eventually become widely available because their benefits to safety-critical systems are enormous and their performance is competitive with conventional architectures. They deliver repeatable behavior, where the behavior in the field is assured of matching their behavior on the test bench with extremely high precision

and probability (at the same level of confidence as we currently get from synchronous digital logic circuits). In my expectation, it is just a matter of time before the world accepts the paradigm shift that they entail.

For distributed systems, we know from industrial practice that networks with controllable timing are realizable. Time-triggered architectures [6], TTEthernet, FlexRay, ARINC busses, and CAN bus networks all deliver some measure of controllable timing. These have been successful in specialized industrial settings, but they (mostly) don't adapt well to the open Internet. A promising development, however, is time-sensitive networking (TSN), a task group of the IEEE 802.1 working group that is developing standards that extend Internet protocols to support high-precision clock synchronization and other technologies that can enable networks with deterministic latencies and reliability delivery that are compatible with the Internet.

To take advantage of such networks, we can leverage a deterministic programming model for distributed real-time systems called PTIDES [5]. PTIDES assumes a bound on clock synchronization error and a bound on network latency, both of which can be reliably delivered with TSN. *Every* deterministic model makes assumptions about the underlying implementation, and violations of those assumptions must be treated as faults, not as performance degradations. PTIDES enables *detection* of these faults, some of which are fundamentally undetectable without a coordinated notion of time.

Despite the value of deterministic models, the real world is full of uncertainty. And even deterministic models have limitations. Chaos, complexity, and undecidability mean that deterministic models may not lead to predictable or analyzable behaviors, and incompleteness means that no set of deterministic models can cover all possible circumstances [7]. Moreover, nondeterminstic models are the only reasonable option when unknown or unknowable properties are central to the model. Hence, probabilistic and nondeterministic models will be needed. But this does not in any way undermine the value of determinism. When deterministic models work, they work spectacularly. Consider the fact that we know how to design silicon chips with billions of transistors that work as predicted the first time they are made. This simply would not be possible without the power of deterministic models.

# E Conclusion

So what is real-time computing? Today, it is an ad-hoc emergent property of physical realizations of cyber-physical systems. Tomorrow, it will be a model used by engineers to build high-confidence, safety-critical systems.

# F Acknowledgement

The author thanks Reinhard Wilhelm and three anonymous reviewers for helpful suggestions on an earlier version of this paper.

This paper reflects research conducted with support from the National Science Foundation (NSF).

# References

[1] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. "Regular programming for quantitative properties of data streams." In European Symposium on Programming Languages and Systems (ESOP), volume LNCS 9632, pages 15–40. Springer, 2016. doi:10.1007/978-3-662-49498-1_2.

[2] George E. P. Box and Norman R. Draper. Empirical Model-Building and Response Surfaces. Wiley Series in Probability and Statistics. Wiley, Hoboken, NJ, 1987.

[3] David Broman, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. "Requirements for hybrid cosimulation standards." In Hybrid Systems: Computation and Control (HSCC), 2015. doi:10.1145/2728606.2728629.

[4] John Earman. A Primer on Determinism, volume 32 of The University of Ontario Series in Philosophy of Science. D. Reidel Publishing Company, Dordrecht, Holland, 1986.

[5] John Eidson, Edward A. Lee, Slobodan Matic, Sanjit A. Seshia, and Jia Zou. "Distributed real-time software for cyber-physical systems." Proceedings of the IEEE (special issue on CPS), 100(1):45–59, 2012. doi:10.1109/JPROC.2011.2161237.

[6] Hermann Kopetz and Günter Bauer. "The time-triggered architecture." Proceedings of the IEEE, 91(1):112–126, 2003.

[7] Edward A. Lee. "Fundamental limits of cyber-physical systems modeling." ACM Transactions on Cyber-Physical Systems, 1(1):26, 2016. doi:10.1145/2912149.

[8] Edward Ashford Lee. Plato and the Nerd — The Creative Partnership of Humans and Technology. MIT Press, 2017.

[9] John A. Stankovic. "Misconceptions about real-time computing: a serious problem for next-generation systems." Computer, 21(10):10–19, 1988.

[10] Lothar Thiele and Pratyush Kumar. "Can real-time systems be chaotic?" In EMSOFT, pages 21–30. ACM, 2015.

[11] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika

Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenstr. "The worst-case execution-time problem - overview of methods and survey of tools." ACM Transactions on Embedded Computing Systems (TECS), 7(3):1–53, 2008.

[12] Michael Zimmer, David Broman, Chris Shaver, and Edward A. Lee. "FlexPRET: A processor platform for mixed-criticality systems." In Real-Time and Embedded Technology and Application Symposium (RTAS), 2014. URL: http://chess.eecs.berkeley.edu/pubs/1048.html.

**Author bio:** Edward A. Lee is the Robert S. Pepper Distinguished Professor in EECS at UC Berkeley, where he has been on the faculty since 1986. He is the author of eight books and some 300 papers and has delivered more than 170 invited talks worldwide. Lee's research group studies cyber-physical systems.

**Contact the author:**
Edward A. Lee
545Q Cory Hall
UC Berkeley
Berkeley, CA 94720-1770
eal@eecs.berkeley.edu