In "Advances in Control, Communication Networks, and Transportation Systems: In Honor of Pravin Varaiya," E.H. Abed (Ed.), Systems and Control: Foundations and Applications Series, Birkhauser, Boston, 2005.

Engineering Education: A Focus on Systems

Edward A. Lee

Department of Electrical Engineering and Computer Science
University of California at Berkeley
Berkeley, CA 94720 USA
eal@eecs.berkeley.edu

1 Introduction

Engineers have a major advantage over scientists. For the most part, the systems we analyze are of our own devising. It has not always been so. Not long ago, the principle objective of engineering was to coax physical materials to do our bidding by leveraging their intrinsic physical properties. The discipline was one of "applied science." Today, a great deal of engineering is about coaxing abstractions that we have invented. The abstractions provided by microprocessors, programming languages, operating systems, and computer networks are only loosely linked to the underlying physics of electronics.

The rapid improvements in the capabilities of electronics during the last half of the 20-th century are, in part, the reason for this separation. The physical constraints imposed by limited memory, processing speed, and communication bandwidth appeared to evaporate with each new generation of computers. What appeared to one generation as luxuriously inefficient abstractions became the bread and butter of the next generation. The separation of "computer science" from "electrical engineering" is both a consequence and a cause, fueling the separation and reflecting it at the same time.

At the same time, the systems science that was incubated in the study of electronic circuits (control systems, communications theory, and signal processing) has also become more abstract. Although these disciplines were created by true "electrical engineers" ("true" means that they were engaged with electrical systems), many of the practitioners today rarely encounter electricity directly. Their techniques are often realized in "embedded" software, ironically building on the abstractions that are only loosely connected to the electronics that their theory originally helped to create. The theories, however, have not adapted as well as one might hope to world of software. Perhaps these theories remain too wedded to their physical heritage.

Computer scientists lament that the engineers who write embedded software use so few of the beautiful abstractions they have built. They write their

code in C (or even in assembly code), using low-level (less abstract) mechanisms. They ignore advances in object-oriented design, memory management, operating systems, and programming languages, and instead directly interact with memory-mapped registers that set up timer interrupts, provide interrupt service routines, and build application-specific task schedulers. Wouldn't it be nice if they would just learn to use the modern technology, and set up instead an HTTP server in Java? Or a peer-to-peer network of embedded sensor and actuator components that discover each other's capabilities via JXTA? The problem is that the modern technology does not talk about the properties of the system that they have to control, such as timing.

On the other hand, the information technology revolution of the late 20-th century was greatly accelerated by advancing computing abstractions. The Internet and the Web are not principally electronic systems. They are conceptual frameworks. The financial, economic, and social systems built on top of them have transformed our cultural landscape. But there is a weakness. While the computing abstractions we have built are extremely well suited to the management of information, their very divorce from the physics makes them less well suited to the management of our physical environment. This is the key reason that these abstractions have had less impact in embedded software.[1]

It seems likely that embedded computing is the next transformational revolution. Although it may seem that computers are already everywhere, the real potential is vastly greater than what we have today. The National Research Council's report *Embedded Everywhere* [4] summarizes this view in the introduction:

> "Information technology (IT) is on the verge of another revolution. Driven by the increasing capabilities and ever declining costs of computing and communications devices, IT is being embedded into a growing range of physical devices linked together through networks and will become ever more pervasive as the component technologies become smaller, faster, and cheaper... These networked systems of embedded computers ... have the potential to change radically the way people interact with their environment by linking together a range of devices and sensors that will allow information to be collected, shared, and processed in unprecedented ways. ... The use of [these embedded computers] throughout society could well dwarf previous milestones in the information revolution."

Sensor networks and "smart dust" [7] are only just breaking out of being laboratory curiosities, but their successes to date imply that the electronics tech-

[1] It is often assumed that real reason is that embedded software faces more severe resource constraints than general purpose software. But as I have pointed out, resource constraints have repeatedly evaporated with each new generation of computers, and yet the practice of embedded software has changed remarkably little.

nology scales, and that leveraging advances in sensors, actuators, and wireless networking will make possible (and probable) a pervasiveness of computing that we can only dream of today.

I am convinced, however, that the embedded revolution will require a reexamination, and probably a reinvention of some of the core abstractions of computing and systems engineering. All effective abstractions hide properties of the underlying systems, but the key to their effectiveness is that they hide the right properties. The divorce of computing from physics has to end for this embedded revolution to take hold.

It is not only the abstractions of computing that have to adapt. Embedded computing will also require a reexamination and reinvention of the core abstractions in the more physics-based engineering disciplines. The models that are used in civil, electrical, and mechanical engineering are deeply rooted in the interactions of physical devices, and poorly express the interactions of those physical devices with computing.

Consider a simple example. A physics-based model of a power distribution system will describe voltages and currents as a function of time, giving their dynamics as ordinary differential equations. The time variable, $t \in \mathbb{R}$, is universal. Its value in Schenectady is the same as its value in San Francisco. But the software embedded in the control system for the power network has tremendous difficulty maintaining a common time base across a distributed system. Even with technologies such as GPS (which provides atomic clock timing precision worldwide), building software that works tightly in concert over geographically distributed systems is extremely difficult. In fact, in the abstractions used to build the software, time is not a part of the ontology. No wonder the engineers who build this software are stuck working with very low-level mechanisms.

Another example is more technically difficult: dealing with random behavior. In standard computing abstractions, we have had the luxury of largely not having to worry about this. This is partly because electronics technology (with some algorithmic help from coding and communication theory) has delivered amazing reliability. Consider the fact that a 40 Gbyte hard disk can be copied flawlessly. This requires that the electronics process 320 billion bits without error. And operations like this occur by the millions on a daily basis. But when we switch our attention to embedded computers with energy scavenging and wireless communication, it is probably too much to expect such reliability. The computing abstractions will have to adapt.

The engineering of systems that are composed of both physical and computational components must be based on abstractions that embrace both physics and computation. There is huge potential for a new "systems science," and there are a few visionaries exploring it. But the cultural divide between computing and engineering is a major barrier to progress. We must break down that barrier.

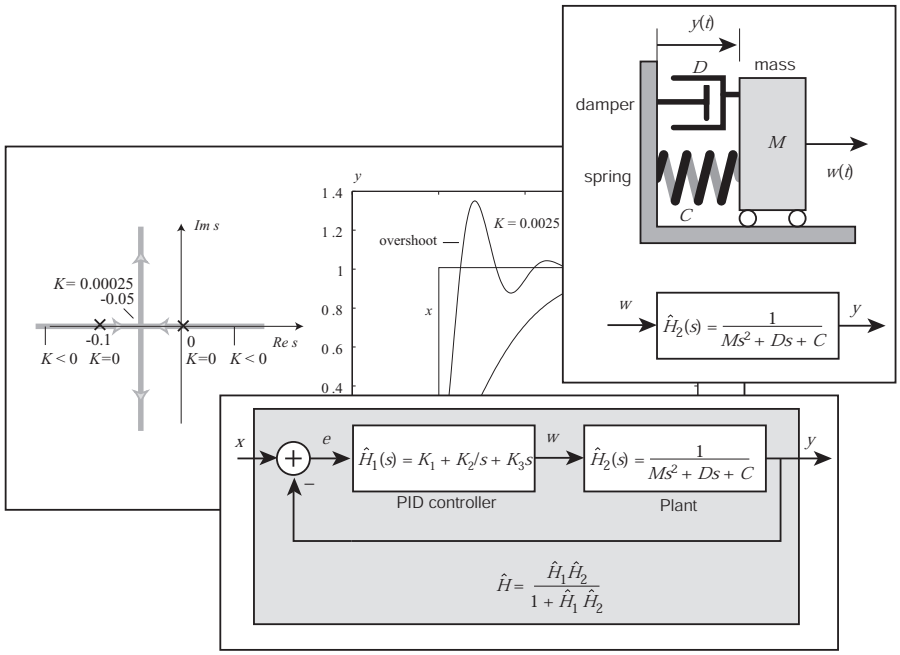

Fig. 1. Illustration of the mathematical tools of classical feedback control systems (from [14]).

2 Feedback Control, Hybrid Systems, and Beyond

A computational systems theory must, of course, build on both theories of computation and classical systems theories [3]. Ideally, it identifies the common foundations, like theories of composition of components. For example, classical feedback control theory, as illustrated in figure 1, builds on a key insight, dating back to the 1930s, that feedback systems can be effectively modeled self-referentially, using an abstraction of instantaneous feedback. At its roots, this principle rests on topological fixed point theory, the same foundation underlying recursion theory (a foundation for many modern programming languages) [6] and many theories of concurrency (e.g. the synchronous languages [2] and process networks [8]). It is extremely rare, however, for engineering students (or even engineering faculty) to be even aware of these commonalities. That these commonalities are not exploited in the curriculum is a consequence of the cultural divide that we created in the 20-th century between engineering and computer science.

On the engineering side, we often misrepresent to our students that the connection between the physical world and the world of software is simply a matter of discretizing time. As long as we respect the Nyquist sampling theorem, everything will be fine. Regrettably, software does not perform with the clock-tick regularity of discrete-time abstractions. And even if it did (or if

we use tricks to achieve a reasonable approximation), the systems we build in software are far more complex than those we used to build with resistors, capacitors, and inductors. The linear-time-invariant abstraction that underlies so much of the pertinent systems theory is simply not applicable. No wonder engineers using embedded software are stuck with bench testing as their principal analysis tool.

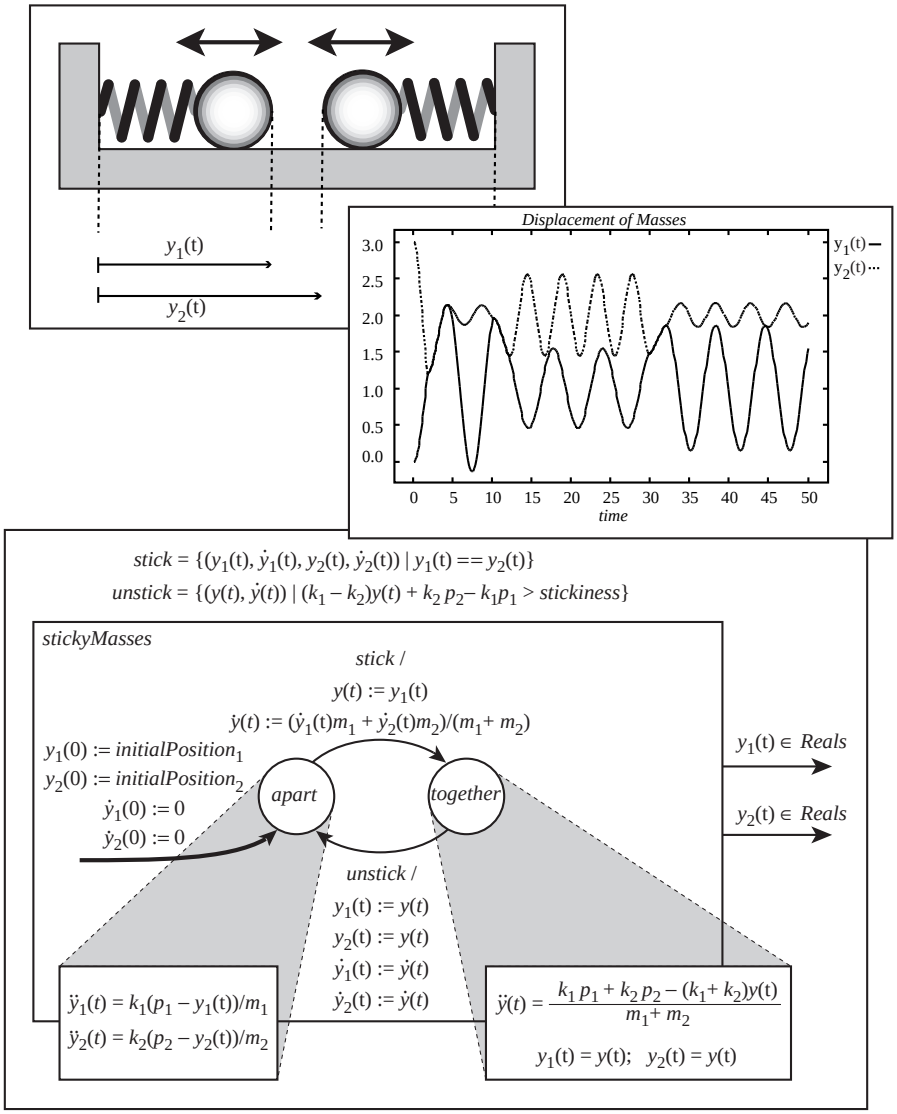

Fig. 2. Illustration of the mathematical tools of hybrid systems (from [14]).

The theory of hybrid systems (see for example [16, 10, 5]) is relatively recent example of a modern systems theory, one that combines computation with classical systems theory. It combines the continuous-time world (or its discretized versions) with the world of irregularly timed mode transitions. It provides analytical tools that are rooted in both linear-time-invariant systems and automata (see figure 2). It leverages theories of computation to achieve decidability results (or, more commonly, undecidability results) [9], and theories of feedback control to study dynamics and stability. Much of the pioneering work in this area was carried out by teams that included both computer scientists and electrical engineers.

However, the current intellectual formulation of hybrid systems has its limitations. It still relies on a model of time that poorly fits what software does. Consider a simple example, due to Jie Liu [15], where two software-based controllers execute on a single computer under the control of a real-time operating system (RTOS). A model of such a system is shown in figure 3. A typical RTOS will offer scheduling policy alternatives, such as preemptive or non-preemptive multitasking, and will support the assignment of priorities to tasks. Under the formulation in figure 3, if the scheduling policy is preemptive multitasking, only one of the two feedback loops can be made stable (which one depends on the relative priorities). Under non-preemptive multitasking, both can be made stable. This difference is extremely hard to explain using classical control theory. If such a simple system renders our analytical tools useless, then engineers are forced to reject either the implementation technology or the analytical tools. In the former case, an engineer might choose to not share the same computer for the two control loops in order to be able to rely on the analytical results. In the latter case, an engineer will bench test the system to verify stability, tweaking priorities and scheduling policies until the desired behavior is achieved experimentally. Neither outcome is particularly attractive.

When our analytical tools break down even for such small, localized systems, how can we expect them to perform for large-scale distributed systems? The lack of an effective temporal abstraction in software is a major limitation. The tight binding of a universal time continuum with control theory is an equally major limitation. The future of systems theory is going to have to offer better time and concurrency abstractions that yield to both formal analysis and distributed and concurrent software realizations.

3 A Focus on Systems

A few years ago, Pravin Varaiya, David Messerschmitt and I led an effort at Berkeley that started down the road of updating the curriculum in the EECS department. We began with our outdated introductory curriculum in EE, where an "introduction to electrical engineering" was principally about passive analog circuits. The rationale for the changes is described in [11], where

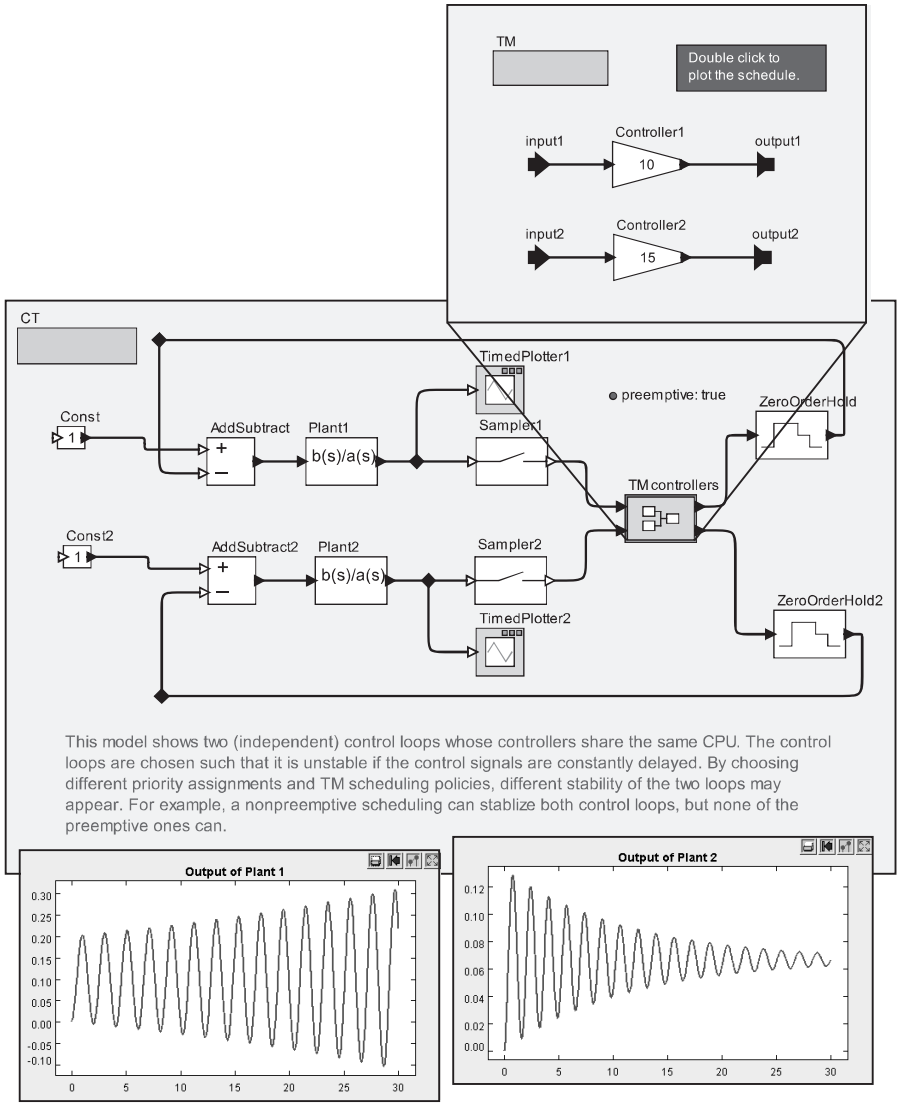

Fig. 3. Model of two software-based controllers executing on a single computer under the control of a real-time operating system, where the controllers are attempting to each stabilize an unstable plant (after [15]).

we cite the considerable work of others that influenced our thinking. A truly long term (and highly speculative) vision is laid out in [12]. The first concrete outcome of this work was a new introductory course on systems [13] and a supporting textbook [14]. Despite this modest progress, the vision remains incompletely unfulfilled. Academic institutions have considerable inertia.

A unifying theme in these efforts is an increased focus on systems rather than technologies. From [11],

> "First, we must prepare students to select abstractions, not just technologies. Second, just as designs can be built on top of higher level abstractions, so can courses."

Selecting abstractions requires being able to reason about the properties of those abstractions. All too often, engineering abstractions are presented as immutable facts ("this is how computers work," or "this differential equation describes that feedback circuit") rather than as human ideas ("this is how VonNeumann proposed that we control automatic machines," or "ignoring the intrinsic randomness and latency in this circuit, Black proposed that we could idealize its behavior in this way.") When we present these ideas as immutable facts, we are doing it out of a genuine believe that the methods are useful to engineers. But we are failing to convey that in a rapidly changing technological climate, engineers must be prepared to think critically about the engineering methods, not just about the engineering designs. When we teach modeling, we must also teach meta-modeling, where we discuss the modeling choices.

4 Conclusion

Abelson and Sussman describe computer science as "procedural epistemology" [1]. Indeed, 20-th century computing was about procedure as knowledge. I believe that 21-st century computing will transform into a system science that subsumes procedure, but also embraces concurrency, time, randomness, and physicality. 21-st century computing will be an epistemology of concurrent interacting components. And the highly valued engineering education will be that which focuses on systems rather than on technologies.

References

1. H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, second edition edition, 1996.
2. A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.
3. R. Boute. Integrating formal methods by unifying abstractions. In E. Boiten, J. Derrick, and G. Smith, editors, *Fourth International Conference on Integrated*

Formal Methods (IFM), volume LNCS 2999, page 441460, Canterbury, Kent, England, 2004. Springer-Verlag.
4. C. S. Committee on Networked Systems of Embedded Computers, D. o. E. Telecommunications Board, and N. R. C. Physical Sciences. *Embedded, Everywhere - A Research Agenda for Networked Systems of Embedded Computers*. National Academy Press, Washington DC, 2001.
5. A. Deshpande and P. Varaiya. Information structures for control and verification of hybrid systems. In *American Control Conference (ACC)*, 1995.
6. J. Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, 1967.
7. T. Hoffman. Smart dust - mighty motes for medicine, manufacturing, the military and more. *Computerworld*, March 24 2003.
8. G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.
9. P. Kopke, T. Henzinger, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *27th Annual ACM Symposioum on Theory of Computing (STOCS)*, pages 372–382, 1995.
10. M. Kourjanski and P. Varaiya. Stability of hybrid systems. volume Hybrid Systems III, LNCS 1066, pages 413–423. Springer-Verlag, 1995.
11. E. A. Lee and D. G. Messerschmitt. Engineering an education for the future. *IEEE Computer Magazine*, 31(1), 1998.
12. E. A. Lee and D. G. Messerschmitt. A highest education in the year 2049. *Proceedings of the IEEE*, 87(9), 1999.
13. E. A. Lee and P. Varaiya. Introducing signals and systems - the berkeley approach. In *First Signal Processing Education Workshop*, Hunt, Texas, 2000.
14. E. A. Lee and P. Varaiya. *Structure and Interpretation of Signals and Systems*. Addison Wesley, 2003.
15. J. Liu. Responsible frameworks for heterogeneous modeling and design of embedded systems. Ph.D. Thesis Technical Memorandum UCB/ERL M01/41, December 20 2001.
16. A. Puri and P. Varaiya. Verification of hybrid systems using abstractions. In *Hybrid Systems Workshop*, volume Hybrid Systems II, LNCS 999, pages 359–369. Springer-Verlag, 1994.