

Computing Foundations and Practice for Cyber-Physical Systems: A Preliminary Report

Edward A. Lee



Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-72

<http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-72.html>

May 21, 2007

Copyright © 2007, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

This ongoing study is supported by the National Science Foundation (CNS-0647591).

Computing Foundations and Practice for Cyber-Physical Systems: A Preliminary Report

Edward A. Lee *

Department of EECS, UC Berkeley, eal@eecs.berkeley.edu

May 21, 2007

Abstract

Cyber-Physical Systems (CPS) are integrations of computation and physical processes. Embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computations and vice versa. The economic and societal potential of such systems is vastly greater than what has been realized, and major investments are being made worldwide to develop the technology. There are considerable challenges, particularly because the physical components of such systems introduce safety and reliability requirements qualitatively different from those in general-purpose computing. This report examines the potential technical obstacles impeding progress, and in particular raises the question of whether today's computing and networking technologies provide an adequate foundation for CPS. It concludes that it will not be sufficient to improve design processes, raise the level of abstraction, or verify (formally or otherwise) designs that are built on today's abstractions. To realize the full potential of CPS, we will have to rebuild computing and networking abstractions. These abstractions will have to embrace physical dynamics and computation in a unified way.

1 Introduction

Cyber-Physical Systems (CPS) are integrations of computation with physical processes. Embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computations and vice versa. In the physical world, the passage of

*This ongoing study is supported by the National Science Foundation (CNS-0647591).

time is inexorable and concurrency is intrinsic. Neither of these properties is present in today's computing and networking abstractions. This report examines this mismatch of abstractions.

Applications of CPS arguably have the potential to dwarf the 20-th century IT revolution. They include high confidence medical devices and systems, assisted living, traffic control and safety, advanced automotive systems, process control, energy conservation, environmental control, avionics, instrumentation, critical infrastructure control (electric power, water resources, and communications systems for example), distributed robotics (telepresence, telemedicine), defense systems, manufacturing, and smart structures. It is easy to envision new capabilities, such as distributed micro power generation coupled into the power grid, where timing precision and security issues loom large. Transportation systems could benefit considerably from better embedded intelligence in automobiles, which could improve safety and efficiency. Networked autonomous vehicles could dramatically enhance the effectiveness of our military and could offer substantially more effective disaster recovery techniques. Networked building control systems (such as HVAC and lighting) could significantly improve energy efficiency and demand variability, reducing our dependence on fossil fuels and our greenhouse gas emissions. In communications, cognitive radio could benefit enormously from distributed consensus about available bandwidth and from distributed control technologies. Financial networks could be dramatically changed by precision timing. Large scale services systems leveraging RFID and other technologies for tracking of goods and services could acquire the nature of distributed real-time control systems. Distributed real-time games that integrate sensors and actuators could change the (relatively passive) nature of on-line social interactions. Tight integration of physical devices and distributed computing could make "programmable matter" a reality.

The positive economic impact of any one of these applications areas would be enormous. Today's computing and networking technologies, however, may have properties that unnecessarily impede progress towards these applications. For example, the lack of temporal semantics and adequate concurrency models in computing, and today's "best effort" networking technologies make predictable and reliable real-time performance difficult, at best. Many of these applications may not be achievable without substantial changes in the core abstractions.

If the US fails to lead the development of these applications, we would almost certainly find our economic and military leadership position compromised. To prevent that from happening, this report will identify the potential disruptive technologies and recommend research investments to

ensure that if such technologies are successfully developed, that they are developed in the US.

2 Requirements for CPS

Embedded systems have always been held to a higher reliability and predictability standard than general-purpose computing. Consumers do not expect their TV to crash and reboot. They have come to count on highly reliable cars, where in fact the use of computer controller has dramatically improved both the reliability and efficiency of the cars. In the transition to CPS, this expectation of reliability will only increase. In fact, without improved reliability and predictability, CPS will not be deployed into such applications as traffic control, automotive safety, and health care.

The physical world, however, is not entirely predictable. Cyber physical systems will not be operating in a controlled environment, and must be robust to unexpected conditions and adaptable to subsystem failures.

An engineer faces an intrinsic tension; designing predictable and reliable components makes it easier to assemble these components into predictable and reliable systems. But no component is perfectly reliable, and the physical environment will manage to foil predictability by presenting unexpected conditions. Given components that are predictable and reliable, how much can a designer depend on that predictability and reliability when designing the system? How does she avoid brittle designs, where small deviations from expected operating conditions cause catastrophic failures?

This is not a new problem in engineering. Digital circuit designers have come to rely on astonishingly predictable and reliable circuits. Circuit designers have learned to harness intrinsically stochastic processes (the motions of electrons) to deliver a precision and reliability that is unprecedented in the history of human innovation. They can deliver circuits that will perform a logical function essentially perfectly, on time, billions of times per second, for years. All this is built on a highly random substrate. Should system designers rely on this predictability and reliability?

In fact, every digital system we use today relies on this to some degree. There is considerable debate in the circuit design community about whether this reliance is in fact impeding progress in circuit technology. Circuits with extremely small feature sizes are more vulnerable to the randomness of the underlying substrate, and if system designers would rely less on the predictability and reliability of digital circuits, then we could progress more rapidly to smaller feature sizes.

No major semiconductor foundry has yet taken the plunge and designed a circuit fabrication process that delivers logic gates that work as specified 80% of the time. Such gates are deemed to have failed completely, and a process that delivers such gates routinely has a rather poor yield.

But system designers do, sometimes, design systems that are robust to such failures. The purpose is to improve yield, not to improve reliability of the end product. A gate that fails 20% of the time is a failed gate, and a successful system has to route around it, using gates that have not failed to replace its functionality. The gates that have not failed will work essentially 100% of the time. The question, therefore, becomes not whether to design robust systems, but rather at what level to build in robustness. Should we design systems that work with gates that perform as specified 80% of the time? Or should we design systems that reconfigure around gates that fail 20% of the time, and then assume that gates that don't fail in yield testing will work essentially 100% of the time?

I believe that the value of being able to count on gates that have passed the yield test to work essentially 100% of the time is enormous. Such solidity at any level of abstraction in system design is enormously valuable. But it does not eliminate the need for robustness at the higher levels of abstraction. Designers of memory systems, despite the high reliability and predictability of the components, still put in checksums and error-correcting codes. If you have a billion components (one gigabit RAM, for example) operating a billion times per second, then even nearly perfect reliability will deliver errors upon occasion.

The principle that we need to follow is simple. Components at any level of abstraction should be made predictable and reliable if this is technologically feasible. If it is not technologically feasible, then the next level of abstraction above these components must compensate with robust design.

Successful designs today follow this principle. It is (still) technically feasible to make predictable and reliable gates. So we design systems that count on this. It is not technically feasible to make wireless links predictable and reliable. So we compensate one level up, using robust coding schemes and adaptive protocols.

The obvious question, therefore, is whether it is technically feasible to make software systems predictable and reliable. At the foundations of computer architecture and programming languages, software is essentially perfectly predictable and reliable, if we limit the term “software” to refer to what is expressed in simple programming languages. Given an imperative programming language with no concurrency, like C, designers can count on a computer to perform exactly what is specified in the program with nearly

100% reliability.

The problem arises when we scale up from simple programs to software systems, and particularly to cyber-physical systems. The fact is that even the simplest C program is not predictable and reliable in the context of CPS because *the program does not express aspects of the behavior that are essential to the system*. It may execute perfectly, exactly matching its semantics, and still fail to deliver the behavior needed by the system. For example, it could miss timing deadlines. Since timing is not in the semantics of C, whether a program misses deadlines is in fact irrelevant to determining whether it has executed correctly. But it is very relevant to determining whether the *system* has performed correctly. A component that is perfectly predictable and reliable turns out not to be predictable and reliable in the dimensions that matter. This is a failure of abstraction.

The problem gets worse as software systems get more complex. If we step outside C and use operating system primitives to perform I/O or to set up concurrent threads, we immediately move from essentially perfect predictability and reliability to wildly nondeterministic behavior that must be carefully reigned in by the software designer [31]. Semaphores, mutual exclusion locks, transactions, and priorities are some of the tools that software designers have developed to attempt to compensate for this loss of predictability and reliability.

But the question we must ask is whether this loss of predictability and reliability is really necessary. I believe it is not. If we find a way to deliver predictable and reliable software (that is predictable and reliable with respect to properties that matter, such as timing), then we do not eliminate the need to design robust systems, but we dramatically change the nature of the challenge. We must follow the principle of making systems predictable and reliable if this is technically feasible, and give up only when there is convincing evidence that this is not possible or cost effective. There is no such evidence for software. Moreover, we have an enormous asset: the substrate on which we build software systems (digital circuits) is essentially perfectly predictable and reliable with respect to properties we care about (timing and functionality).

Let us examine further the failure of abstraction. Figure 1 illustrates schematically some of the abstraction layers on which we depend when designing embedded systems. In this three-dimensional Venn diagram, each box represents a set. E.g., at the bottom, we have the set of all microprocessors. An element of this set, e.g., the Intel P4-M 1.6GHz, is a particular microprocessor. Above that is the set of all x86 programs, each of which can run on that processor. This set is defined precisely (unlike the previous set,

which is difficult to define) by the x86 instruction set architecture (ISA). Any program coded in that instruction set is a member of the set, such as a particular implementation of a Java virtual machine. Associated with that member is another set, the set of all JVM bytecode programs. Each of these programs is (typically) synthesized by a compiler from a Java program, which is a member of the set of all syntactically valid Java programs. Again, this set is defined precisely by Java syntax.

Each of these sets provides an abstraction layer that is intended to isolate a designer (the person or program that selects elements of the set) from the details below. Many of the best innovations in computing have come from careful and innovative construction and definition of these sets.

However, in the current state of embedded software, nearly every abstraction has failed. The instruction-set architecture, meant to hide hardware implementation details from the software, has failed because the user of the ISA cares about timing properties the ISA does not guarantee. The programming language, which hides details of the ISA from the program logic, has failed because no widely used programming language expresses timing properties. Timing is merely an accident of the implementation. A real-time operating system hides details of the program from their concurrent orchestration, yet this fails because the timing may affect the result. The RTOS provides no guarantees. The network hides details of electrical or optical signaling from systems, but many standard networks provide no timing guarantees and fail to provide an appropriate abstraction. A system designer is stuck with a system *design* (not just implementation) in silicon and wires.

All embedded systems designers face versions of this problem. Aircraft manufacturers have to stockpile the electronic parts needed for the entire production line of an aircraft model to avoid having to recertify the software if the hardware changes. “Upgrading” a microprocessor in an engine control

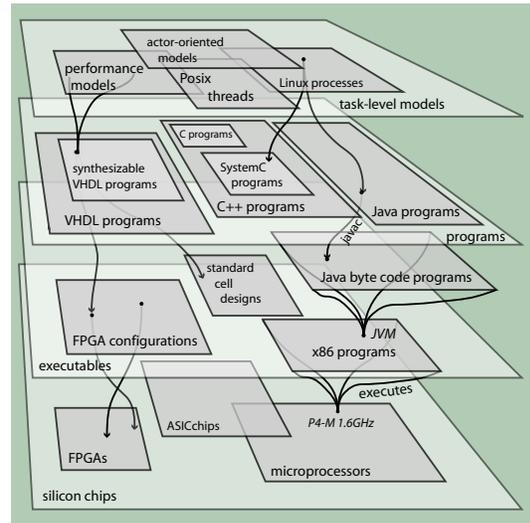


Figure 1: Abstraction layers in computing

unit for a car requires thorough re-testing of the system. Even “bug fixes” in the software or hardware can be extremely risky, since they can change timing behavior.

The design of an abstraction layer involves many choices, and computer scientists have chosen to hide timing properties from all higher abstractions. Wirth [46] says “It is prudent to extend the conceptual framework of sequential programming as little as possible and, in particular, to avoid the notion of execution time.” In an embedded system, however, computations interact directly with the physical world, where time cannot be abstracted away. Even general-purpose computing suffers from these choices. Since timing is neither specified in programs nor enforced by execution platforms, a program’s timing properties are not repeatable. Concurrent software often has timing-dependent behavior in which small changes in timing have big consequences.

Designers have traditionally covered these failures by finding worst case execution time (WCET) bounds and using real-time operating systems (RTOS’s) with predictable scheduling policies. But these require substantial margins for reliability, and ultimately reliability is (weakly) determined by bench testing of the complete implementation. Moreover, WCET has become an increasingly problematic fiction as processor architectures develop ever more elaborate techniques for dealing stochastically with deep pipelines, memory hierarchy, and parallelism. Modern processor architectures render WCET virtually unknowable; even simple problems demand heroic efforts. In practice, reliable WCET numbers come with many caveats that are increasingly rare in software. The processor ISA has failed to provide an adequate abstraction.

Timing behavior in RTOSs is coarse and becomes increasingly uncontrollable as the complexity of the system increases, e.g., by adding inter-process communication. Locks, priority inversion, interrupts and similar issues break the formalisms, forcing designers to rely on bench testing, which rarely identifies subtle timing bugs. Worse, these techniques produce brittle systems in which small changes can cause big failures. As a telling example, Patrick Lardieri of Lockheed Martin Advanced Technology Laboratories discussed some experiences with the JSF Program, saying¹ “Changing the instruction memory layout of the Flight Control Systems Control Law process to optimize Built in Test processing led to an unexpected performance change - [the] System went from meeting real-time requirements to missing

¹in a plenary talk at the National Workshop on High-Confidence Software Platforms for Cyber-Physical Systems (HCSP-CPS), Arlington, VA November 30–December 1, 2006.

most deadlines due to a change that was expected to have no impact on system performance.”

While there are no true guarantees in life, we should not blithely discard predictability that is achievable. Synchronous digital hardware—the technology on which computers are built—delivers astonishingly precise timing behavior with reliability that is unprecedented in any other human-engineered mechanism. Software abstractions, however, discard several orders of magnitude of precision. Compare the nanosecond-scale precision with which hardware can raise an interrupt request to the millisecond-level precision with which software threads respond. We don’t have to do it this way.

3 Background

Integration of physical processes and computing, of course, is not new. The term “embedded systems” has been used for some time to describe engineered systems that combine physical processes with computing. Successful applications include communication systems, aircraft control systems, automotive electronics, home appliances, weapons systems, games and toys, for example. However, most such embedded systems are closed “boxes” that do not expose the computing capability to the outside. The radical transformation that we envision comes from networking these devices. Such networking poses considerable technical challenges.

For example, prevailing practice in embedded software relies on bench testing for concurrency and timing properties. This has worked reasonably well, because programs are small, and because software gets encased in a box with no outside connectivity that can alter the behavior. However, the applications we envision demand that embedded systems be feature-rich and networked, so bench testing and encasing become inadequate. In a networked environment, it becomes impossible to test the software under all possible conditions. Moreover, general-purpose networking techniques themselves make program behavior much more unpredictable. A major technical challenge is to achieve predictable timing in the face of such openness.

Before DARPA began investing in embedded systems in the mid-1990s (principally through the MoBIES, SEC, and NEST programs), the research community devoted to this problem was small. Embedded systems were largely an industrial problem, one of using small computers to enhance the performance or functionality of a product. In this earlier context, embedded software differed from other software only in its resource limitations

(small memory, small data word sizes, and relatively slow clocks). In this view, the “embedded software problem” is an optimization problem. Solutions emphasize efficiency; engineers write software at a very low level (in assembly code or C), avoid operating systems with a rich suite of services, and use specialized computer architectures such as programmable DSPs and network processors that provide hardware support for common operations. These solutions have defined the practice of embedded software design and development for the last 30 years or so. In an analysis that remains as valid today as 19 years ago, Stankovic [41] laments the resulting misconceptions that real-time computing “is equivalent to fast computing” or “is performance engineering” (most embedded computing is real-time computing).

But the resource limitations of 30 years ago are surely not resource limitations today. Indeed, the technical challenges have centered more on predictability and robustness than on efficiency. Safety-critical embedded systems, such as avionics control systems for passenger aircraft, are forced into an extreme form of the “encased box” mentality. For example, in order to assure a 50 year production cycle for a fly-by-wire aircraft, an aircraft manufacturer is forced to purchase, all at once, a 50 year supply of the microprocessors that will run the embedded software. To ensure that validated real-time performance is maintained, these microprocessors must all be manufactured on the same production line from the same masks. The systems will be unable to benefit from the next 50 years of technology improvements without redoing the (extremely expensive) validation and certification of the software. Evidently, efficiency is nearly irrelevant compared to predictability, and predictability is difficult to achieve without freezing the design at the physical level. Clearly, something is wrong with the software abstractions being used.

A notable culprit is the lack of timing in computing abstractions. Indeed, this lack has been exploited heavily in such computer science disciplines as architecture, programming languages, operating systems, and networking. In architecture, for example, although synchronous digital logic delivers precise timing determinacy, advances have made it difficult or impossible to estimate or predict the execution time of software. Modern processor architectures use memory hierarchy (caches), dynamic dispatch, and speculative execution to improve average case performance of software, at the expense of predictability. These techniques make it nearly impossible to tell how long it will take to execute a particular piece of code.² To deal with these

²A glib response is that execution time in a Turing-complete language is undecidable anyway, so it’s not worth even trying to predict execution time. This is nonsense. No

architectural problems, embedded software designers may choose alternative processor architectures such as programmable DSPs not only for efficiency reasons, but also for predictability of timing.

Even less timing-sensitive applications have been affected. Anecdotal information from computer-based instrumentation, for example, indicates that the real-time performance delivered by today's PCs is about the same as was delivered by PCs in the mid-1980's. Twenty years of Moore's law have not improved things in this dimension. This is not entirely due to hardware architecture techniques, of course. Operating systems, programming languages, user interfaces, and networking technologies have become more elaborate. All have been built on an abstraction of software where time is irrelevant. No widely used programming language includes temporal properties in its semantics, and "correct" execution of a program has nothing to do with time. Benchmarks emphasize average-case performance, and timing predictability is irrelevant.

The prevailing view of real-time appears to have been established well before embedded computing was common [46]. "Computation" is accomplished by a terminating sequence of state transformations. This core abstraction underlies the design of nearly all computers, programming languages, and operating systems in use today. But unfortunately, this core abstraction may not fit CPS very well.

The most interesting and revolutionary cyber-physical systems will be networked. The most widely used networking techniques today introduce a great deal of timing variability and stochastic behavior. Today, embedded systems are often forced to use less widely accepted networking technologies (such as CAN busses in manufacturing systems and FlexRay in automotive applications), and typically must limit the geographic extent of these networks to a confined local area. What aspects of those networking technologies should or could be important in larger scale networks? Which are compatible with global networking techniques?

To be specific, recent advances in time synchronization across networks promise networked platforms that share a common notion of time to a known precision [27]. How would that change how distributed cyber-physical applications are developed? What are the implications for security? Can we mitigate security risks created by the possibility of disrupting the shared notion of time? Can security techniques effectively exploit a shared notion

cyber-physical system that depends on timeliness can be deployed without timing assurances. If Turing completeness interferes with this, then Turing completeness must be sacrificed.

of time to improve robustness? In particular, although distributed denial of service attacks have proved surprisingly difficult to contend with in general purpose IT networks, could they be controlled in time synchronized networks?

Operating systems technology is also groaning under the weight of the requirements of embedded systems. RTOS's are still essentially best-effort technologies. To specify real-time properties of a program, the designer has to step outside the programming abstractions, making operating system calls to set priorities or to set up timer interrupts. Are RTOS's merely a temporary patch for inadequate computing foundations? What would replace them? Is the conceptual boundary between the operating system and the programming language (a boundary established in the 1960's) still the right one? It would be truly amazing if it were.

Cyber-physical systems by nature will be concurrent. Physical processes are intrinsically concurrent, and their coupling with computing requires, at a minimum, concurrent composition of the computing processes with the physical ones. Even today, embedded systems must react to multiple real-time streams of sensor stimuli and control multiple actuators concurrently. Regrettably, the mechanisms of interaction with sensor and actuator hardware, built for example on the concept of interrupts, are not well represented in programming languages. They have been deemed to be the domain of operating systems, not of software design. Instead, the concurrent interactions with hardware are exposed to programmers through the abstraction of threads.

Threads, however, are a notoriously problematic [31, 47]. This fact is often blamed on humans rather than on the abstraction. Sutter and Larus [42] observe that "humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code. Even careful people miss possible interleavings among even simple collections of partially ordered operations." The problem will get far worse with extensively networked cyber-physical systems.

Yet humans are actually quite adept at reasoning about concurrent systems. The physical world is highly concurrent, and our very survival depends on our ability to reason about concurrent physical dynamics. The problem is that we have chosen concurrent abstractions for software that do not even vaguely resemble the concurrency of the physical world. We have become so used to these computational abstractions that we have lost track of the fact that they are not immutable. Could it be that the difficulty of concurrent programming is a consequence of the abstractions, and that if we were willing to let go of those abstractions, then the problem would be fixable?

Embedded computing also exploits concurrency models other than threads. Programmable DSP architectures are often VLIW machines. Video signal processors often combine SIMD with VLIW and stream processing. Network processors provide explicit hardware support for streaming data. However, despite considerable innovative research, in practice, programming models for these domains remain primitive. Designers write low-level assembly code that exploits specific hardware features, and combine this assembly code with C code only where performance is not so critical.

For the next generation of cyber-physical systems, it is arguable that we must build concurrent models of computation that are far more deterministic, predictable, and understandable. Threads take the opposite approach. They make programs absurdly nondeterministic, and rely on programming style to constrain that nondeterminism to achieve deterministic aims. Can a more deterministic approach be reconciled with the intrinsic need for nondeterminism in many embedded applications? How should cyber-physical systems contend with the inherent unpredictability of the (networked) physical world?

It is essential that the US invest aggressively and immediately. As mentioned above, the DARPA projects of the late 1990s and early 2000s (MOBIES, SEC, and NEST), created a vibrant research community focused on embedded systems. However, DARPA has walked away from this problem area, and the research community is at risk of dissipating. Existing relatively small NSF programs (none of which is squarely centered on CPS) cannot pick up the slack alone.

Meanwhile, enormous competitive pressures are building. The European Union expects to spend more than 1 billion Euro in 2007, increasing to 1.5 billion Euro in 2010, on embedded systems research, with a specific aim towards fostering entrepreneurship and enhancing the competitive positioning of European companies (see the ARTEMIS project, Advanced Research and Technology for Embedded Intelligence and Systems, <http://www.artemis-office.org/>). Korea, Japan, Singapore, and China are also investing considerable amounts in embedded systems, shoring up the competitive positions of companies such as LG Electronics and Samsung. In Korea, for example, a large new government funded research lab called DGIST is under construction with embedded systems forming approximately one third of its mission (the other two thirds are in nanotechnologies and biosystems).

In the meantime, key industries in the US that stand to benefit most from CPS technology, such as the automotive and telecommunications industry, are barely surviving. They do not have the luxury of investing in long term research.

4 Families of Solutions

In this section, I give a preliminary assessment of various possible solutions to the CPS problem. The conclusion is that none of these solutions is sufficiently complete to dodge the bullet. Reexamination of the core abstractions is going to be necessary. The consequences are enormous, requiring major rework in many branches of computer science.

4.1 Validation and Verification

In practice, a great deal of embedded software today is developed through a process of prototyping and testing on the bench. Since timing properties are not expressed in any programming language, they emerge from an implementation, and can be measured by examining traces of execution. Given that this practice has delivered many successful embedded systems, a reasonable approach is to improve the practice. For example, automated regression tests are difficult in this context, since hardware is very specific, and emulating the physical environment of the embedded software is challenging. Perhaps a reasonable improvement is to facilitate better testing. The Berkeley BEE2 project [12], for one, confronts this challenge by using a sophisticated FPGA-based system to emulate systems contexts for wireless components.

Another alternative is better simulation techniques. Joint simulation of hardware and software at adequate accuracy remains elusive, despite considerable investment. The Cadence product VCC, for example, aimed to support architectural evaluation of hardware/software designs, but failed in the marketplace.

One of the challenges is the high cost of cycle-accurate simulation of software executing on modern processors. As a data point, in a plenary talk at EMSOFT/CASES-ISSS/CODES (Embedded Week, 10/23/06, Seoul, Korea), Namsung Woo (Executive VP of Samsung) described Samsung's ViP, or Virtual Platform, which provides "function accurate and cycle-approximate" hardware and software co-simulation. Woo stated that it is simply too computationally expensive to provide cycle-accurate simulation. And even if the accuracy is sufficiently improved, simulations will still only represent specific software executing on specific hardware in a specific physical context. Changes to any of these variables (the software, hardware, or environment) invalidate the results of the simulation.

A third alternative is formal verification. This field has improved considerably in recent years, and has become far more effective for certain tasks.

Identifying potential deadlock conditions, for example, is easier than it used to be, despite fundamental challenges of decidability and computational complexity. However, properties that are not formally specified cannot be verified. Thus, for example, timing behavior of software, which is not expressed in the software, must be separately specified, and the connection between specifications and between specification and implementations becomes tenuous at best. This problem is even more fundamental than the widely cited skepticism about scalability and usability of formal verification techniques. These too remain considerable challenges. Formal verification is still carried out by verification experts, not by system designers. And despite considerable progress in automated abstraction (see for example [22]), scalability to realistic systems remains a major issue.

4.2 Certification

Another possible approach is to focus on certification, particularly composable certification, where certified components may be combined to yield certifiable systems [26]. The DO-178B Level A standard, for example, allows software to be used in critical avionics applications (see <http://www.rtca.org>). However, achieving compliance with this standard is challenging, and to the author's knowledge, few software design methodologies have any assurances of yielding compliant software. A singular success is SCADE [8] (Safety Critical Application Development Environment), a commercial product of Esterel Technologies, which builds on the synchronous language Lustre [18]. SCADE provides a graphical programming framework with the semantics of Lustre, one of the simplest of the so-called "synchronous languages" [7]. These languages have strong formal properties that yield quite effectively to formal verification techniques, but the simplicity of Lustre in large part accounts for SCADE being able to produce C or ADA code that is compliant with DO-178B Level A.

Although they are promising, synchronous languages are not (yet) widely used in embedded systems development. The jury is out. In the meantime, certification is an extremely expensive proposition, and what is certified is not software but systems. This results in inflexible designs, where nothing can change without redoing the certification process.

There are other promising efforts. For example, the Open License Society (<http://www.OpenLicenseSociety.org>) is after certifiable real-time operating systems. I believe that certification must be part of the solution, particular for safety critical software, but by itself, it cannot compensate for inadequacies in the technology.

4.3 Software Engineering

Software engineering is an art supported by the scientific method and technical tools. As with any art, it is better applied by people skilled in the art and facile with the tools. Understanding the art and improving the tools will almost certainly improve the product.

The first technique is better software engineering processes. These are, in fact, essential to get reliable CPS systems. However, they are not sufficient. An anecdote from the Ptolemy Project³ is telling (and alarming). In the early part of the year 2000, my group began developing the kernel of Ptolemy II [17], a modeling environment supporting concurrent models of computation. An early objective was to permit modification of concurrent programs via a graphical user interface while those concurrent programs were executing. The challenge was to ensure that no thread could ever see an inconsistent view of the program structure. The strategy was to use Java threads with monitors. This problem represents a common pattern in embedded systems, particularly ones that must be adaptive.

A part of the Ptolemy Project experiment was to see whether effective software engineering practices could be developed for an academic research setting. We developed a process that included a code maturity rating system (with four levels, red, yellow, green, and blue), design reviews, code reviews, nightly builds, regression tests, and automated code coverage metrics [38]. Although admittedly naive compared to industrial best practices (this was, after all, an academic setting), the practices dramatically improved the quality of the software produced. But not enough to eliminate critical flaws. The portion of the kernel that ensured a consistent view of the program structure was written in early 2000, design reviewed to yellow, and code reviewed to green. The reviewers included concurrency experts, not just inexperienced graduate students. We wrote regression tests that achieved 100% code coverage. The nightly build and regression tests ran on a two processor SMP machine, which exhibited different thread behavior than the development machines, which all had a single processor. The Ptolemy II system itself began to be widely used, and every use of the system exercised this code. No problems were observed until the code deadlocked on April 26, 2004, four years later.

It is certainly true that our relatively rigorous software engineering practice identified and fixed many concurrency bugs. But the fact that a problem as serious as a deadlock that locked up the system could go undetected for four years despite this practice is alarming. How many more such problems

³See <http://ptolemy.eecs.berkeley.edu>

remain? How long do we need test before we can be sure to have discovered all such problems? Regrettably, I have to conclude that a rigorous process alone may never reveal all the problems in nontrivial code.

Of course, there are tantalizingly simple rules for avoiding deadlock. For example, always acquire locks in the same order [29]. However, this rule is very difficult to apply in practice because no method signature in any widely used programming language indicates what locks the method acquires. You need to examine the source code of all methods that you call, and all methods that those methods call, in order to confidently invoke a method. Even if we fix this language problem by making locks part of the method signature, this rule makes it extremely difficult to implement symmetric accesses (where interactions can originate from either end). And no such fix gets around the problem that reasoning about mutual exclusion locks is extremely difficult. If programmers cannot understand their code, then the code will not be reliable.

One might conclude that the problem is in the way Java realizes threads. Perhaps the `synchronized` keyword is not the best way of pruning the wild nondeterminism intrinsic in threads. Indeed, version 5.0 of Java, introduced in 2005, added a number of other mechanisms for synchronizing threads. These do in fact enrich the toolkit for the programmer to prune nondeterminacy. But the mechanisms (such as semaphores) still require considerable sophistication to use, and very likely will still result in incomprehensible programs with subtle lurking bugs. It is hard to imagine how a rigorous process will avoid these pitfalls.

Software engineering process improvements alone will not do the job. Another approach that can help is the use of vetted design patterns for concurrent computation, as in [29] and [39]. Indeed, these are an enormous help when the programmer's task identifiably matches one of the patterns. However, there are two difficulties. One is that implementation of the patterns, even with careful instructions, is still subtle and tricky. Programmers will make errors, and there are no scalable techniques for automatically checking compliance of implementations to patterns. More importantly, the patterns can be difficult to combine. Their properties are not typically composable, and hence nontrivial programs that require use of more than one pattern are unlikely to be understandable.

A very common use of patterns in concurrent computation is found in databases, particularly with the notion of transactions. Transactions support speculative unsynchronized computation on a copy of the data followed by a *commit* or *abort*. A commit occurs when it can be shown that no conflicts have occurred. Transactions can be supported on distributed hardware

(as is common for databases), or in software on shared-memory machines [40], or, most interestingly, in hardware on shared-memory machines [24]. In the latter case, the technique meshes well with cache consistency protocols that are required anyway on these machines. Transactions eliminate unintended deadlocks, but despite recent extensions for composability [19], remain a highly nondeterministic interaction mechanism. They are well-suited to intrinsically nondeterminate situations, where for example multiple actors compete nondeterministically for resources. But they are not well-suited for building determinate concurrent interactions.

A particularly interesting use of patterns is MapReduce, as reported by Dean and Ghemawat [15]. This pattern has been used for large scale distributed processing of huge data sets by Google. Whereas most patterns provide fine-grain shared data structures with synchronization, MapReduce provides a framework for the construction of large distributed programs. The pattern is inspired by the higher-order functions found in Lisp and other functional languages. The parameters to the pattern are pieces of functionality represented as code rather than pieces of data.

Patterns may be encapsulated into libraries by experts, as has been done with MapReduce, the concurrent data structures in Java 5.0, and STAPL in C++ [2]. This greatly improves the reliability of implementations, but requires some programmer discipline to constrain all concurrent interactions to occur via these libraries. Folding the capabilities of these libraries into languages where syntax and semantics enforce these constraints may eventually lead to much more easily constructed concurrent programs. But no libraries have yet emerged that address timing of software.

Higher-order patterns such as MapReduce offer some particularly interesting challenges and opportunities for language designers. These patterns function at the level of *coordination languages* [36] rather than more traditional *programming languages*. New coordination languages that are compatible with established programming languages (such as Java and C++) are much more likely to gain acceptance than new programming languages that replace the established languages.

4.4 New Technologies

Although the improvements described above can help, we believe that to realize its full potential, CPS systems will require fundamentally new technologies. It is possible that these will emerge as incremental improvements on existing technologies, but given the lack of timing in the core abstractions of computing, this seems improbable. Any complete solution will need to

fix this lack.

Nonetheless, incremental improvements can have a considerable impact. For example, concurrent programming can be done in much better ways than threads. For example, Split-C [13] and Cilk [10] are C-like languages supporting multithreading with constructs that are easier to understand and control than raw threads. A related approach combines language extensions with constraints that limit expressiveness of established languages in order to get more consistent and predictable behavior. For example, the Guava language [6] constrains Java so that unsynchronized objects cannot be accessed from multiple threads. It further makes explicit the distinction between locks that ensure the integrity of read data (read locks) and locks that enable safe modification of the data (write locks). SHIM also provides more controllable thread interactions [44]. These language changes prune away considerable nondeterminacy without sacrificing much performance, but they still have deadlock risk, and again, none of them confronts the lack of temporal semantics.

Another approach that puts more emphasis on the avoidance of deadlock is *promises*, as realized for example by Mark Miller in the E programming language⁴. These are also called futures, and are originally attributable to Baker and Hewitt [20]. Here, instead of blocking to access shared data, programs proceed with a proxy of the data that they expect to eventually get, using the proxy as if it were the data itself. Although a fascinating approach for general purpose computing, this approach does not seem well suited for CPS, since if anything it exacerbates timing unpredictability.

As stated above, I believe that the best approach has to be predictable where it is technically feasible. Predictable concurrent computation is possible, but it requires approaching the problem differently. Instead of starting with a highly nondeterministic mechanism like threads, and relying on the programmer to prune that nondeterminacy, we should start with deterministic, composable mechanisms, and introduce nondeterminism only where needed.

One approach that is very much a bottom-up approach is to modify computer architectures to deliver precision timing [16]. This can allow for deterministic orchestration of concurrent actions. But it leaves open the question of how the software will be designed, given that programming languages and methodologies have so thoroughly banished time from the domain of discourse.

Achieving timing precision is easy if we are willing to forgo perfor-

⁴See <http://www.erights.org/>

mance; the engineering challenge is to deliver both precision and performance. While we cannot abandon structures such as caches and pipelines and 40 years of progress in programming languages, compilers, operating systems, and networking, many will have to be re-thought. Fortunately, throughout the abstraction stack, there is much work on which to build. ISAs can be extended with instructions that deliver precise timing with low overhead [25]. Scratchpad memories can be used in place of caches [4]. Deep interleaved pipelines can be efficient and deliver predictable timing [32]. Memory management pause times can be bounded [5]. Programming languages can be extended with timed semantics [21]. Appropriately chosen concurrency models can be tamed with static analysis [8]. Software components can be made intrinsically concurrent and timed [33]. Networks can provide high-precision time synchronization [27]. Schedulability analysis can provide admission control, delivering run-time adaptability without timing imprecision [9].

Complementing bottom-up approaches are top-down solutions that center on the concept of model-based design [43]. In this approach, “programs” are replaced by “models” that represent system behaviors of interest. Software is synthesized from the models. This approach opens a rich semantic space that can easily embrace temporal dynamics (see for example [48]), including even the continuous temporal dynamics of the physical world.

But many challenges and opportunities remain in developing this relatively immature technology. Naive abstractions of time, such as the discrete-time models commonly used to analyze control and signal processing systems, do not reflect the true behavior of software and networks [35]. The concept of “logical execution time” [21] offers a more promising abstraction, but ultimately still relies on being able to get worst-case execution times for software components. This top-down solution depends on a corresponding bottom-up solution.

Some of the most intriguing aspects of model-based design center on explorations of rich possibilities for interface specifications and composition. Reflecting behavioral properties in interfaces, of course, has also proved useful in general-purpose computing (see for example [34]). But where we are concerned with properties that have not traditionally been expressed at all in computing, the ability to develop and compose specialized “interface theories” [14] is extremely promising. These theories can reflect causality properties [49], which abstract temporal behavior, real-time resource usage [45], timing constraints [23], protocols [28], depletable resources [11], and many others [1].

A particularly attractive approach that may allow for leveraging the

considerable investment in software technology is to develop coordination languages [36], which introduce new semantics at the component interaction level rather than at the programming language level. Manifold [37] and Reo [3] are two examples, as are a number of other “actor oriented” approaches [30].

5 Conclusion

To fully realize the potential of CPS, the core abstractions of computing need to be rethought. Incremental improvements will, of course, continue to help. But effective orchestration of software and physical processes requires semantic models that reflect properties of interest in both.

References

- [1] L. d. Alfaro and T. A. Henzinger. Interface-based design. In M. Broy, J. Gruenbauer, D. Harel, and C. Hoare, editors, *Engineering Theories of Software-intensive Systems*, volume NATO Science Series: Mathematics, Physics, and Chemistry, Vol. 195, pages 83–104. Springer, 2005.
- [2] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An adaptive, generic parallel C++ library. In *Wkshp. on Lang. and Comp. for Par. Comp. (LCPC)*, pages 193–208, Cumberland Falls, Kentucky, 2001.
- [3] F. Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.
- [4] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *Trans. on Embedded Computing Sys.*, 1(1):6–26, 2002.
- [5] D. F. Bacon, P. Cheng, and V. Rajan. The Metronome: A simpler approach to garbage collection in real-time systems. In *Workshop on Java Technologies for Real-Time and Embedded Systems*, Catania, Sicily, 2003.
- [6] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: a dialect of Java without data races. In *ACM SIGPLAN conference on Object-oriented*

programming, systems, languages, and applications, volume 35 of *ACM SIGPLAN Notices*, pages 382–400, 2000.

- [7] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.
- [8] G. Berry. The effectiveness of synchronous languages for the development of safety-critical systems. White paper, Esterel Technologies, 2003.
- [9] E. Bini and G. C. Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers*, 53(11):1462–1473, 2004.
- [10] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, ACM SIGPLAN Notices, pages 207 – 216, Santa Barbara, California, 1995.
- [11] A. Chakrabarti, L. de Alfaro, and T. A. Henzinger. Resource interfaces. In R. Alur and I. Lee, editors, *EMSOFT*, volume LNCS 2855, pages 117–133, Philadelphia, PA, 2003. Springer.
- [12] C. Chang, J. Wawrzynek, and R. W. Brodersen. Bee2: A high-end reconfigurable computing system. *IEEE Design and Test of Computers*, 22(2):114–125, 2005.
- [13] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. v. Eicken, and K. Yelick. Parallel programming in Split-C. In *ACM/IEEE Conference on Supercomputing*, pages 262 – 273, Portland, OR, 1993. ACM Press.
- [14] L. deAlfaro and T. A. Henzinger. Interface theories for component-based design. In *First International Workshop on Embedded Software (EMSOFT)*, volume LNCS 2211, pages 148–165, Lake Tahoe, CA, 2001. Springer-Verlag.
- [15] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Sixth Symposium on Operating System Design and Implementation (OSDI)*, page 137150, San Francisco, CA, 2004. USENIX Association.

- [16] S. A. Edwards and E. A. Lee. The case for the precision timed (pret) machine. In *Design Automation Conference (DAC)*, San Diego, CA, 2007.
- [17] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuen-dorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(2):127–144, 2003.
- [18] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1319, 1991.
- [19] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *ACM Conference on Principles and Practice of Parallel Programming (PPoPP)*, pages 48–60, Chicago, IL, 2005. ACM Press.
- [20] J. Henry G. Baker and C. Hewitt. The incremental garbage collection of processes. In *Proceedings of the Symposium on AI and Programming Languages*, volume 12 of *ACM SIGPLAN Notices*, pages 55–59, 1977.
- [21] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *EMSOFT 2001*, volume LNCS 2211, Tahoe City, CA, 2001. Springer-Verlag.
- [22] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *15th International Conference on Computer-Aided Verification (CAV)*, volume 2725 of *Lecture Notes in Computer Science*, pages 262–274. Springer-Verlag, 2003.
- [23] T. A. Henzinger and S. Matic. An interface algebra for real-time components. In *12th Annual Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE Computer Society Press, 2006.
- [24] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, San Diego, California, United States, 1993. ACM Press.
- [25] N. J. H. Ip and S. A. Edwards. A processor extension for cycle-accurate real-time software. In *IFIP International Conference on Embedded and Ubiquitous Computing (EUC)*, volume LNCS 4096, pages 449–458, Seoul, Korea, 2006. Springer.

- [26] D. Jackson, M. Thomas, L. I. Millett, and C. on Certifiably Dependable Software Systems. Software for dependable systems: Sufficient evidence? software for dependable systems: Sufficient evidence? daniel jackson, martyn thomas, and lynette i. millett, editors may 9, 2007. Technical report, National Academies Press, May 9 2007.
- [27] S. Johannessen. Time synchronization in a local area network. *IEEE Control Systems Magazine*, pages 61–69, 2004.
- [28] H. Kopetz and N. Suri. Compositional design of RT systems: A conceptual basis for specification of linking interfaces. In *6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2003)*, pages 51–60, Hakodate, Hokkaido, Japan, 2003. IEEE Computer Society.
- [29] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, Reading MA, 1997.
- [30] E. A. Lee. Model-driven development - from object-oriented design to actor-oriented design. In *Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation (a.k.a. The Monterey Workshop)*, Chicago, 2003.
- [31] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.
- [32] E. A. Lee and D. G. Messerschmitt. Pipeline interleaved programmable dsps: Architecture. *IEEE Trans. on Acoustics, Speech, and Signal Processing*, ASSP-35(9), 1987.
- [33] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12(3):231–260, 2003.
- [34] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.
- [35] T. Nghiem, G. J. Pappas, A. Girard, and R. Alur. Time-triggered implementations of dynamic controllers. In *EMSOFT*, pages 2–11, Seoul, Korea, 2006. ACM Press.
- [36] G. Papadopoulos and F. Arbab. Coordination models and languages. In M. Zelkowitz, editor, *Advances in Computers - The Engineering of Large Systems*, volume 46, pages 329–400. Academic Press, 1998.

- [37] G. A. Papadopoulos, A. Stavrou, and O. Papapetrou. An implementation framework for software architectures based on the coordination paradigm. *Science of Computer Programming*, 60(1):27–67, 2006.
- [38] H. J. Reekie, S. Neuendorffer, C. Hylands, and E. A. Lee. Software practice in the Ptolemy project. Technical Report Series GSRC-TR-1999-01, Gigascale Semiconductor Research Center, University of California, Berkeley, April 1999.
- [39] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects*. Wiley, 2000.
- [40] N. Shavit and D. Touitou. Software transactional memory. In *ACM symposium on Principles of Distributed Computing*, pages 204–213, Ottawa, Ontario, Canada, 1995. ACM Press.
- [41] J. A. Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems. *Computer*, 21(10):10–19, 1988.
- [42] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.
- [43] J. Sztipanovits and G. Karsai. Model-integrated computing. *IEEE Computer*, page 110112, 1997.
- [44] O. Tardieu and S. A. Edwards. SHIM: Scheduling-independent threads and exceptions in SHIM. In *EMSOFT*, Seoul, Korea, 2006. ACM Press.
- [45] L. Thiele, E. Wandeler, and N. Stoimenov. Real-time interfaces for composing real-time systems. In *EMSOFT*, Seoul, Korea, 2006. ACM Press.
- [46] N. Wirth. Toward a discipline of real-time programming. *Communications of the ACM*, 20(8):577–583, 1977.
- [47] N. Zeldovich, A. Yip, F. Dabek, R. T. Morris, D. Mazieres, and F. Kaashoek. Multiprocessor support for event-driven programs. In *USENIX Annual Technical Conference*, San Antonio, Texas, USA, 2003.
- [48] Y. Zhao, E. A. Lee, and J. Liu. A programming model for time-synchronized distributed real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Bellevue, WA, USA, 2007. IEEE.

- [49] Y. Zhou and E. A. Lee. A causality interface for deadlock analysis in dataflow. In *ACM & IEEE Conference on Embedded Software (EMSOFT)*, Seoul, South Korea, 2006. ACM.