Invited paper, International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC) May 6, 2008 Orlando, FL, USA

Cyber Physical Systems: Design Challenges

Edward A. Lee *
Center for Hybrid and Embedded Software Systems, EECS
University of California, Berkeley
Berkeley, CA 94720, USA
eal@eecs.berkeley.edu

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. Moreover, physical components are qualitatively different from object-oriented software components. Standard abstractions based on method calls and threads do not work. This paper examines the challenges in designing such systems, 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 time is inexorable and concurrency is intrinsic. Neither of these properties is present in today's computing and networking 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.

The economic impact of any of these applications would be huge. Computing and networking technologies today, however, may 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 pre-

*This work is supported by the National Science Foundation (CNS-0647591 and CNS-0720841).

dictable and reliable real-time performance difficult. Software component technologies, including object-oriented design and service-oriented architectures, are built on abstractions that match software better than physical systems. Many applications will not be achievable without substantial changes in the core abstractions.

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 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 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. 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 remaining gates 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 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 robustness. 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 harder to make wireless links predictable and reliable. So we compensate one level up, using robust coding and adaptive protocols.

The obvious question 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 with essentially 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 [19]. 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. Predictable and reliable software does not eliminate the need to design robust systems, but dramatically changes 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 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,

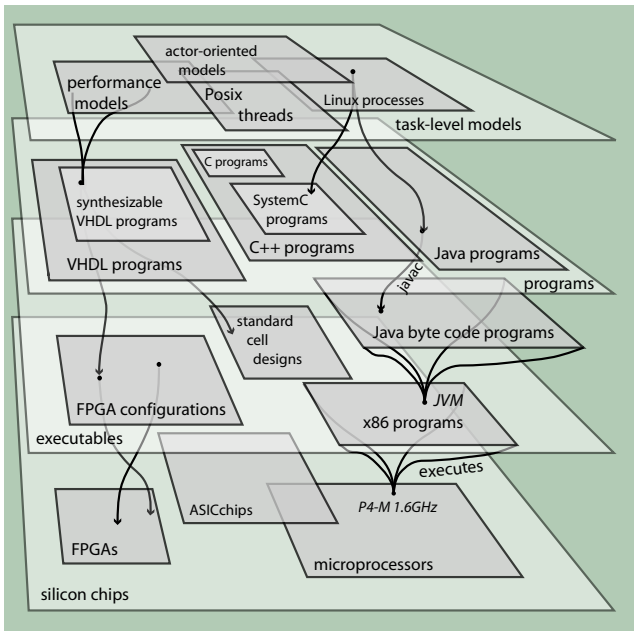

Figure 1. Abstraction layers in computing

nearly every abstraction has failed. The ISA, 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 network hides details of signaling from systems, but most standard networks provide no timing guarantees.

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 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 [31] 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. Moreover, WCET is 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.

While there are no true guarantees in life, we should not blithely discard predictability that is achievable. Synchronous digital hardware delivers astonishingly precise timing behavior, and software abstractions 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 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.

Historically, 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 [26] 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.

The lack of timing in computing abstractions has been exploited heavily in such computer science disciplines as architecture, programming languages, operating systems, and networking. Caches, dynamic dispatch, and speculative execution improve average case performance 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.[1] To deal with these architectural problems, designers may choose alternative processor architectures such as programmable DSPs not for efficiency, but for predictability.

Even less time-sensitive applications are affected. Anecdotal evidence from computer-based instrumentation, for example, indicates that real-time performance delivered by today's PCs is about the same as that 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, of course. Operating systems, programming languages, user interfaces, and networking have become more elaborate. All have been built on an abstraction of software where time is irrelevant.

The prevailing view of real-time appears to have been established well before embedded computing was common [31]. "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 often use specialized networking technologies (such as CAN busses in manufacturing systems and FlexRay in automotive applications). What aspects of those networking technologies should or could be important in larger scale networks? Which are compatible with global networking?

To be specific, recent advances in time synchronization across networks promise networked platforms that share a common notion of time to a known precision [16]. 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 of time to improve robustness?

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.

[1] A 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 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.

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 [19, 32]. This fact is often blamed on humans rather than on the abstraction. Sutter and Larus [27] 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 quite adept at reasoning about concurrent systems. The physical world is 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 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 are willing to let go of those abstractions, then the problem would be fixable?

4 Solutions

These problems are not entirely new; many creative researchers have made contributions. Advances in formal verification, emulation and simulation techniques, certification methods, software engineering processes, design patterns, and software component technologies all help. We would be lost without these improvements. But I 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.

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 [10] and Cilk [8] 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 [5] 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 [29]. 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.

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 [12]. 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 performance; 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 [15]. Scratchpad memories can be used in place of caches [3]. Deep interleaved pipelines can be efficient and deliver predictable timing [20]. Memory management pause times can be bounded [4]. Programming languages can be extended with timed semantics [13]. Appropriately chosen concurrency models can be tamed with static analysis [6]. Software components can be made intrinsically concurrent and timed [21]. Networks can provide high-precision time synchronization [16]. Schedulability analysis can provide admission control, delivering run-time adaptability without timing imprecision [7].

Complementing bottom-up approaches are top-down solutions that center on the concept of model-based design [28]. 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 [33]), 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 [23]. The concept of "logical execution time" [13] 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 [22]). 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" [11] is extremely promising. These theories can reflect causality properties [34], which abstract temporal behavior, real-time resource usage [30], timing constraints [14], protocols [17], depletable resources [9], and many others [1].

A particularly attractive approach that may allow for leveraging the considerable investment in software technology is to develop coordination languages [24], which introduce new semantics at the component interaction level rather than at the programming language level. Manifold [25] and Reo [2] are two examples, as are a number of other "actor oriented" approaches [18].

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] F. Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[3] 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.

[4] 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*, pages 466–478, Catania, Sicily, November 2003.

[5] 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.

[6] G. Berry. The effectiveness of synchronous languages for the development of safety-critical systems. White paper, Esterel Technologies, 2003.

[7] E. Bini and G. C. Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers*, 53(11):1462–1473, 2004.

[8] 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, August 1995.

[9] 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, October 13-15, 2003 2003. Springer.

[10] 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, November 1993. ACM Press.

[11] 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, October, 2001 2001. Springer-Verlag.

[12] S. A. Edwards and E. A. Lee. The case for the precision timed (PRET) machine. In *Design Automation Conference (DAC)*, San Diego, CA, June 4-8 2007.

[13] 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.

[14] 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.

[15] 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, August 2006. Springer.

[16] S. Johannessen. Time synchronization in a local area network. *IEEE Control Systems Magazine*, pages 61–69, 2004.

[17] 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, 14-16 May 2003 2003. IEEE Computer Society.

[18] 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, September 24 2003.

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

[20] E. A. Lee and D. G. Messerschmitt. Pipeline interleaved programmable dsps: Architecture. *IEEE Trans. on Acoustics, Speech, and Signal Processing*, ASSP-35(9), 1987.

[21] 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.

[22] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[23] 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.

[24] 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.

[25] 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.

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

[27] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

[28] J. Sztipanovits and G. Karsai. Model-integrated computing. *IEEE Computer*, page 110112, 1997.

[29] O. Tardieu and S. A. Edwards. SHIM: Scheduling-independent threads and exceptions in SHIM. In *EMSOFT*, Seoul, Korea, October 22-24 2006. ACM Press.

[30] L. Thiele, E. Wandeler, and N. Stoimenov. Real-time interfaces for composing real-time systems. In *EMSOFT*, Seoul, Korea, October 23-25 2006. ACM Press.

[31] N. Wirth. Toward a discipline of real-time programming. *Communications of the ACM*, 20(8):577–583, 1977.

[32] 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, June 9-14 2003.

[33] 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, April 3-6 2007. IEEE.

[34] 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, October 22-25 2006. ACM.