Computing Needs Time

Edward A. Lee
Robert S. Pepper Distinguished Professor
UC Berkeley

Invited Talk:

Working day on time-oriented embedded systems
November 2, 2010, at ENSEEIHT
Toulouse, France
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 prevailing abstractions used in computing, however, do not mesh well with the physical world. Most critically, software systems speak about the passage of time only very indirectly and in non-compositional ways. This talk examines the obstacles in software technologies that are impeding progress, and in particular raises the question of whether today's computing and networking technologies provide an adequate foundation for CPS. It argues 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 software abstractions. These abstractions will have to embrace physical dynamics and computation in a unified way. This talk will discuss research challenges and potential solutions, with particular focus on two projects at Berkeley, PRET (which is developing computer architectures with temporal semantics) and PTIDES (which provides a programming model for distributed real-time systems).
Cyber-Physical Systems (CPS):
Orchestrating networked computational resources with physical systems

Automotive

Military systems:

E-Corner, Siemens

Daimler-Chrysler

Building Systems

Power generation and distribution

Transportation
(Air traffic control at SFO)

Avionics

Telecommunications

Instrumentation
(Soleil Synchrotron)

Factory automation

Courtesy of Kuka Robotics Corp.

Lee, Berkeley 3

Courtesy of General Electric

 Courtesy of Doug Schmidt

 Courtesy of General Electric

 Courtesy of Kuka Robotics Corp.
Where CPS Differs from the traditional embedded software problem:

- **The traditional embedded software problem:**
  Embedded software is software on small computers. The technical problem is one of optimization (coping with limited resources).

- **The CPS problem:**
  Computation and networking integrated with physical processes. The technical problem is managing dynamics, time, and concurrency in networked computational + physical systems.
Approaching the CPS Challenge

**Physicalizing the cyber (PtC):** to endow software and network components with abstractions and interfaces that represent their physical properties, such as dynamics in time.

**Cyberizing the Physical (CtP):** to endow physical subsystems with cyber-like abstractions and interfaces.
A “Success” Story

The Boeing 777 was Boeing’s first fly-by-wire aircraft. It is deployed, appears to be reliable, and is succeeding in the marketplace. Therefore, it must be a success. However...
A “Success” Story

In “fly by wire” aircraft, certification of the software is extremely expensive. Regrettably, it is not the software that is certified but the entire system. If a manufacturer expects to produce a plane for 50 years, it needs a 50-year stockpile of fly-by-wire components that are all made from the same mask set on the same production line. Even a slight change or “improvement” might affect behavior and require the software to be re-certified.
A “Success” Story

Apparently, the software does not specify the behavior that was certified.

Fly-by-wire is about controlling the dynamics of a physical system. It is not about the transformation of data, which is what Turing/Church “computation” does.
A Key Challenge:
Timing is not Part of Software Semantics

Correct execution of a program in C, C#, Java, Haskell, etc. has nothing to do with how long it takes to do anything. All our computation and networking abstractions are built on this premise.

Programmers have to step outside the programming abstractions to specify timing behavior.
Techniques that Exploit this Fact

- Programming languages
- Virtual memory
- Caches
- Dynamic dispatch
- Speculative execution
- Power management (voltage scaling)
- Memory management (garbage collection)
- Just-in-time (JIT) compilation
- Multitasking (threads and processes)
- Component technologies (OO design)
- Networking (TCP)
- ...

i.e., many of the innovations in CS over the last 40 years.
The software does not specify the behavior. Consequences:

- **Stockpiling for a product run**
  - Some systems vendors have to purchase up front the entire expected part requirements for an entire product run.

- **Frozen designs**
  - Once certified, errors cannot be fixed and improvements cannot be made.

- **Product families**
  - Difficult to maintain and evolve families of products together.
  - It is difficult to adapt existing designs because small changes have big consequences

- **Forced redesign**
  - A part becomes unavailable, forcing a redesign of the system.

- **Lock in**
  - Cannot take advantage of cheaper or better parts.

- **Risky in-field updates**
  - In the field updates can cause expensive failures.
Abstraction Layers in Common Use

The purpose of an abstraction is to hide details of the implementation below and provide a platform for design from above.
Abstraction Layers in Common Use

Every abstraction layer has failed in the fly-by-wire scenario.

The design *is* the implementation.
The Same Problem Arises in Networking

The point of these abstraction layers is to isolate a system designer from the details of the implementation below, and to provide an abstraction for other system designers to build on.

In today’s networks, timing is a property that emerges from the details of the implementation, and is not included in the abstractions. Timing is a performance metric, not a correctness criterion.
How about “raising the level of abstraction” to solve these problems?
But these higher abstractions rely on an increasingly problematic fiction: WCET

Example war story:

Analysis of:
- Motorola ColdFire
- Two coupled pipelines (7-stage)
- Shared instruction & data cache
- Artificial example from Airbus
- Twelve independent tasks
- Simple control structures
- Cache/Pipeline interaction leads to large integer linear programming problem

And the result is valid only for that exact Hardware and software!

Fundamentally, the ISA of the processor has failed to provide an adequate abstraction.

The Key Problem

Electronics technology delivers highly reliable and precise timing…

…and the overlaying software abstractions discard it.

```plaintext
// Perform the convolution.
for (int i=0; i<10; i++) {
    x[i] = a[i]*b[j-i];
    // Notify listeners.
    notify(x[i]);
}
```
The Standard Practice

Today: WCET Analysis

Today, we augment the model with minute details of the realization.

(not just application logic, but ISA, how the ISA is realized, what memory technology is used, how much memory of each kind, what I/O hardware is used, exact timing of inputs, etc.)

We can do better!
Projects at Berkeley

Time and concurrency in the core abstractions:

- **Foundations**: Timed computational semantics.
- **Bottom up**: Make timing repeatable.
- **Top down**: Timed, concurrent components.
- **Holistic**: Model engineering.
A Bottom Up Approach: Make Timing a Semantic Property of Computers

Precision-Timed (PRET) Machines

Just as we expect reliable logic operations, we should expect repeatable timing.

Timing precision with performance: Challenges:

- ISAs with timing (deadline instructions?)
- Memory hierarchy (scratchpads?)
- Deep pipelines (interleaving?)
- Predictable memory management (Metronome?)
- Languages with timing (discrete events? Giotto?)
- Predictable concurrency (synchronous languages?)
- Composable timed components (actor-oriented?)
- Multicore PRET (conflict free networks on chip?)
- Precision networks (TTA? Time synchronization?)

Extending an ISA with Timing Instructions

[V1] Best effort:

```plaintext
set_time r1, 1s
// Code block
delay_until r1
```

[V2] Late miss detection

```plaintext
set_time r1, 1s
// Code block
branch_expired r1, <target>
delay_until r1
```

[V3] Immediate miss detection

```plaintext
set_time r1, 1s
exception_on_expire r1, 1
// Code block
deactivate_exception 1
delay_until r1
```

[V4] Exact execution:

```plaintext
set_time r1, 1s
// Code block
MTFD r1
```
Other variations

[V2]-[V4] could all have a variant that does not control the *minimum* execution time of the block of code, but only controls the *maximum*.

[V2] Late miss detection

```plaintext
set_time r1, 1s
// Code block
branch_expired r1, <target>
delay_until r1
```

[V3] Immediate miss detection

```plaintext
set_time r1, 1s
exception_on_expire r1, 1
// Code block
deactivate_exception 1
delay_until r1
```
Exporting the Timed Semantics to a Low-Level Language (like C)

Example:

```c
tryin (500ms) {
    // Code block
} catch {
    panic();
}
```

This realizes variant 3, “immediate miss detection,” using `setjmp` and `longjmp` to handle timing exceptions. `setjmp()` returns 0 when directly invoked, and returns non zero when invoked by `longjmp()`.

If the code block takes longer than 500ms to run, then exception 0 will be thrown, and the handler code will run `longjmp`, which will return control flow to `setjmp`, but returning non zero. The else statement will then be run, causing the panic procedure to be called.

```c
jmp_buf buf;
if ( !setjmp(buf) ){
    set_time r1, 500ms
    exception_on_expire r1, 0
    // Code block
    deactivate_exception 0
} else {
    panic();
}
```

This pseudocode is neither C-level nor assembly, but is meant to explain an assembly-level implementation.
Summary of ISA extensions

- **[V1]** Execute a block of code taking at least a specified \textit{time} [Ip & Edwards, 2006]

- **[V2]** Do [V1], and then conditionally branch if the specified \textit{time} was exceeded.

- **[V3]** Do [V1], but if the specified \textit{time} is exceeded during execution of the block, branch immediately to an exception handler.

- **[V4]** Execute a block of code taking exactly the specified \textit{time}. MTFD

**Variants:**
- For V2 – V4, may not impose minimum execution time.
- \textit{Time} may be literal (seconds) or abstract (cycles).
Projects at Berkeley

Time and concurrency in the core abstractions:

- **Foundations**: Timed computational semantics.
- **Bottom up**: Make timing repeatable.
- **Top down**: Timed, concurrent components.
- **Holistic**: Model engineering.
A Top Down Approach: Make Timing a Semantic Property of Software Components

PTIDES: Distributed execution under discrete-event semantics, with “model time” and “real time” bound at sensors and actuators.

Input time stamps are ≥ real time

Output time stamps are ≤ real time

Messages are processed in time-stamp order.
First Test Case

- **Tunneling Ball Device**
  - sense ball
  - track disk
  - adjust trajectory

This device was designed by Jeff Jensen, now at National Instruments.
Tunneling Ball Device in Action
Tunneling Ball Device

Mixed event sequences

Periodic Events

Quasi Periodic Events

Sporadic Events

[Diagram of Tunneling Ball Device]
Second Test Case: Distributed Synchrophasor Measurement & Control

Power swing and Unstability detection

Synchrophasor data: $\Theta_1, V_1, I_1, \Theta_{v11}$
Power factor: $\cos(\Theta_{v11})$

Synchrophasor data: $\Theta_2, V_2, I_2, \Theta_{v12}$
Power factor: $\cos(\Theta_{v12})$

Thanks to Vaselin Skendzic, Schweitzer Engineering
Experiment Diagram

Grid emulator built with National Instruments PXI

Primary Measurement Unit (PMU) built with Renesas demo boards with DP83640

Ethernet bridge or 1588 boundary/transparent clock

Synchrophasor Vector Processing unit (SVP) built with Renesas demo board with DP83640

Thanks to Vaselin Skendzic, Schweitzer Engineering
Distributed PTIDES Relies on Network Time Synchronization with Bounded Error

This may become routine!
With this PHY, clocks on a LAN agree on the current time of day to within 8ns, far more precise than older techniques like NTP.

A question we are addressing at Berkeley: How does this change how we develop distributed CPS software?
An Extreme Example: The Large Hadron Collider

The WhiteRabbit project at CERN is synchronizing the clocks of computers 10 km apart to within about 80 psec using a combination of IEEE 1588 PTP and synchronous ethernet.
More Generally than PTIDES: *Rethinking Software Components to Admit Time.*
Object Oriented vs. Actor Oriented

The established: Object-oriented:

<table>
<thead>
<tr>
<th>class name</th>
<th>data</th>
<th>methods</th>
</tr>
</thead>
</table>

What flows through an object is sequential control

Things happen to objects

The alternative: Actor oriented:

<table>
<thead>
<tr>
<th>actor name</th>
<th>data (state)</th>
<th>parameters</th>
<th>ports</th>
</tr>
</thead>
</table>

What flows through an object is evolving data

Actors make things happen

Input data  Output data
Examples of Actor-Oriented Systems

- UML 2 and SysML (activity diagrams)
- ASCET (time periods, interrupts, priorities, preemption, shared variables)
- Autosar (software components w/ sender/receiver interfaces)
- Simulink (continuous time, The MathWorks)
- LabVIEW (structured dataflow, National Instruments)
- SCADE (synchronous, based on Lustre and Esterel)
- CORBA event service (distributed push-pull)
- ROOM and UML-2 (dataflow, Rational, IBM)
- VHDL, Verilog (discrete events, Cadence, Synopsys, ...)
- Modelica (continuous time, constraint-based, Linkoping)
- OPNET (discrete events, Opnet Technologies)
- SDL (process networks)
- Occam (rendezvous)
- SPW (synchronous dataflow, Cadence, CoWare)
- ...

The semantics of these differ considerably in their approaches to concurrency and time. Some are loose (ambiguous) and some rigorous. Some are strongly actor-oriented, while some retain much of the flavor (and flaws) of threads.
Ptolemy II: Our Laboratory for Experiments with Actor-Oriented Design

Programs are specified as actor-oriented models, and software is synthesized from these models.

Director from a library defines component interaction semantics

Software component library.

Modern type system for component interfaces

Visual editor for composing components

Authors: Edward A. Lee and Yuhong Xiong
The pThales Domain: Multi-Dimensional Dataflow

A Berkeley-Thales Collaboration

Application parameters
- lambda: .03
- TpulsE: 5E-5
- SubarraySpacing: .09
- nb_pxls: 32
- nb_rng: 1E3/rg_size / rg_min
- rg_size: 30
- rg_min: 50
- Beamwidth: 15
- org_rg_corr: 3
- nb_ant: 8
- nb_beams: 7
- K_chirp: 2
- jam_CIR: 30
- jam_power: 2000
- Targ1_CIR: 4
- Targ1_V: 100
- Targ1_Dist: 3000
- Targ1_RCS: 2
- jam_freq_MHz: 1
- nb_rng_corr: 10
- noise_power: 2
- Igh_chirp: 16

Created by Remi Barrere & Eric Lenormand, Thales
Conclusions

Today, timing is a property only of realizations of computational systems.

Tomorrow, timing will be a semantic property of computational models.

_Raffaello Sanzio da Urbino – The Athens School_
The Ptolemy Pteam

John Eidson
Edward Lee
Jia Zou
Isaac Liu
Christopher Brooks
Ben Lickly
Thomas Mandl
Jeff Jensen
Shanna Shaye Forbes
Hiroshi Subramo
Yasemin Demir
Patricia Delorier
Hugo Andrade
Slobodan Matic
Eleftherios Matsikoudis
Stefan Resmerita
Edward Lee
Lee, Berkeley 40
This book strives to identify and introduce the durable intellectual ideas of embedded systems as a technology and as a subject of study. The emphasis is on modeling, design, and analysis of cyber-physical systems, which integrate computing, networking, and physical processes.