Beyond Embedded Systems:
Integrating Computation, Networking, and Physical Dynamics

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

Invited Keynote Talk
Dublin, Ireland

June 19-20, 2009
Context of my work: Chess: Center for Hybrid and Embedded Software Systems

Board of Directors
- Edward A. Lee
- Alberto Sangiovanni-Vincentelli
- Shankar Sastry
- Claire Tomlin

Executive Director
- Christopher Brooks

Other key faculty at Berkeley
- Dave Auslander
- Ruzena Bajcsy
- Raz Bodik
- Karl Hedrick
- Kurt Keutzer
- George Necula
- Masayoshi Tomizuka
- Pravin Varaiya

This center, founded in 2002, blends systems theorists and application domain experts with software technologists and computer scientists.

Some Research Projects
- Precision-timed (PRET) machines
- Distributed real-time computing
- Systems of systems
- Theoretical foundations of CPS
- Hybrid systems
- Design technologies
- Verification
- Intelligent control
- Modeling and simulation

Applications
- Building systems
- Automotive
- Synthetic biology
- Medical systems
- Instrumentation
- Factory automation
- Avionics
Cyber-Physical Systems (CPS):
*Orchestrating networked computational resources with physical systems*

- **Automotive**
  - E-Corner, Siemens
  - Daimler-Chrysler

- **Military systems**
  - Courtesy of Doug Schmidt

- **Building Systems**
  - Courtesy of General Electric

- **Avionics**
  - Courtesy of Siemens

- **Telecommunications**
  - Courtesy of General Electric

- **Transportation**
  - (Air traffic control at SFO)

- **Instrumentation**
  - (Soleil Synchrotron)

- **Factory automation**
  - Courtesy of Kuka Robotics Corp.

- **Power generation and distribution**
  - Courtesy of General Electric
CPS Example – Printing Press

- **High-speed, high precision**
  - Speed: 1 inch/ms
  - Precision: 0.01 inch
  - -> Time accuracy: 10us

- **Open standards (Ethernet)**
  - Synchronous, Time-Triggered
  - IEEE 1588 time-sync protocol

- **Application aspects**
  - local (control)
  - distributed (coordination)
  - global (modes)
Where CPS Differs from the traditional embedded systems problem:

- **The traditional embedded systems 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.
CPS is Multidisciplinary

Computer Science:
Carefully abstracts the physical world

System Theory:
Deals directly with physical quantities

Cyber Physical Systems:
Computational + Physical
A Key Challenge

Models for the physical world and for computation diverge.

- physical: time continuum, ODEs, DAEs, PDEs, dynamics
- computational: a “procedural epistemology,” logic

There is a huge cultural gap.

Physical system models must be viewed as semantic frameworks, and theories of computation must be viewed as alternative ways of talking about dynamics.
First Challenge on the Cyber Side: Real-Time Software

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.

Timing of programs is not repeatable, except at very coarse granularity.

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)
- ...
A 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 timing and require the software to be re-certified.
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.
The purpose for an abstraction is to hide details of the implementation below and provide a platform for design from above.
Every abstraction layer has failed for real-time programs.

The design is the implementation.
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.
Second Challenge on the Cyber Side: *Concurrency*
(Needed for real time and multicore)

Threads dominate concurrent software.

- *Threads*: Sequential computation with shared memory.
- *Interrupts*: Threads started by the hardware.

Incomprehensible interactions between threads are the sources of many problems:

- Deadlock
- Priority inversion
- Scheduling anomalies
- Timing variability
- Nondeterminism
- Buffer overruns
- System crashes
My Claim

Nontrivial software written with threads is incomprehensible to humans, and it cannot deliver repeatable or predictable timing, except in trivial cases.
Perhaps Concurrency is Just Hard…

Sutter and Larus observe:

“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.”

Is Concurrency Hard?

It is not concurrency that is hard...
Threads are sequential processes that share memory. From the perspective of any thread, the entire state of the universe can change between any two atomic actions (itself an ill-defined concept).

*Imagine if the physical world did that…*
Concurrent programs using shared memory are incomprehensible because concurrency in the physical world does not work that way.

*We have no experience!*
Consider an Automotive Example

- Periodic events
- Quasi-periodic events
- Sporadic events

Consider handling this with timers, interrupts, threads, shared memory, priorities, and mutual exclusion. This is a nightmare!
The Current State of Affairs

We build embedded software on abstractions where time is irrelevant using concurrency models that are incomprehensible.

Just think what we could do with the right abstractions!
The Berkeley Solution

Time and concurrency in the core abstractions:

- **Foundations**: Timed computational semantics.
- **Bottom up**: Make timing repeatable.
- **Top down**: Timed, concurrent components.
- **Holistic**: Model engineering.
Causal systems operating on signals are usually naturally (Scott) continuous.
Some Reading on Foundations

Ph.D. Theses:


Papers:


etc. ...
The Berkeley Solution

Time and concurrency in the core abstractions:

- **Foundations**: Timed computational semantics.
- **Bottom up**: Make timing repeatable.
- **Top down**: Timed, concurrent components.
- **Holistic**: Model engineering.
Bottom Up: Make Timing Repeatable

**Precision-Timed (PRET) Machines**

*Make temporal behavior as important as logical function.*

Timing precision with performance: Challenges:

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

The Berkeley Solution

Time and concurrency in the core abstractions:

- **Foundations**: Timed computational semantics.
- **Bottom up**: Make timing repeatable.
- **Top down**: Timed, concurrent components.
- **Holistic**: Model engineering.
Object Oriented vs. Actor Oriented

The established: Object-oriented:

- **class name**
- data
- methods

What flows through an object is sequential control

Things happen to objects

The alternative: Actor oriented:

- **actor name**
- data (state)
- parameters
- ports

What flows through an object is evolving data

Actors make things happen
The First (?) Actor-Oriented Programming Language

*The On-Line Graphical Specification of Computer Procedures*

W. R. Sutherland, Ph.D. Thesis, MIT, 1966

MIT Lincoln Labs TX-2 Computer

Bert Sutherland with a light pen

Bert Sutherland used the first acknowledged object-oriented framework (Sketchpad, created by his brother, Ivan Sutherland) to create the first actor-oriented programming language (which had a visual syntax).

Partially constructed actor-oriented model with a class definition (top) and instance (below).
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.
Give a Component Technology rather than New Languages

- It leverages:
  - Language familiarity
  - Component libraries
  - Legacy subsystems
  - Design tools
  - The simplicity of sequential reasoning
- It allows for innovation in
  - Distributed time-sensitive system design
  - Hybrid systems design
  - Service-oriented architectures
- Software is intrinsically concurrent
  - Better use of multicore machines
  - Better use of networked systems
  - Better potential for robust design
Ptolemy II: Our Laboratory for Experiments with Actor-Oriented Design

Concurrent management supporting dynamic model structure.

Director from a library defines component interaction semantics.

Large, behaviorally-polymorphic component library.

Visual editor supporting an abstract syntax.

Type system for transported data.

This model illustrates composite types. The Record Assembler actor composes a record token, which is then passed through a channel that has random delay. The tokens arrive possibly in another order. The Record Disassembler actor separates the string from the sequence number. The strings are displayed as received (possible out of order), and resequenced by the Sequencer actor, which puts them back in order. This example demonstrates how types propagate through record composition and decomposition.
Approach: Concurrent Composition of Components designed with Conventional Languages
Example: Discrete Event Models

DE Director implements timed semantics using an event queue.

Components send time-stamped events to other components, and components react in chronological order.
PTIDES: Programming Temporally Integrated Distributed Embedded Systems

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

... and being explicit about time delays means that we can analyze control system dynamics...

Feedback through the physical world
Experimental Setup

- **Ptides Model**
- **Code Generator**
- **Software Component Library**
- **Mixed Simulator**
- **Ptides OS**
- **HW Platform**
- **HW in the Loop Simulator**
- **Plant Model**
- **Network Model**

**Ptolemy II Ptides domain**

**Ptolemy II Discrete-event, Continuous, and Wireless domains**

- **Analysis**
  - Schedulability Analysis
  - Causality Analysis
  - Program Analysis

**HW Platform**

- Luminary Micro 8962

**HW in the Loop**

- Ptolemy II Ptides domain
- Ptolemy II Discrete-event, Continuous, and Wireless domains

**Lee, Berkeley 40**
The Berkeley Solution

Time and concurrency in the core abstractions:

- **Foundations**: Timed computational semantics.
- **Bottom up**: Make timing repeatable.
- **Top down**: Timed, concurrent components.
- **Holistic**: Model engineering.
Model Engineering Topics

- **Model transformation**
  - Model optimization
  - Scalable model construction (big models from small descriptions)
  - Product families (multiple products from one model)
  - Design refactoring
  - Workflow automation

- **Model ontologies**
  - Property annotations and inference
  - Sound foundation based on type theories
  - Scalable to large models

- **Multimodeling**
  - Hierarchical multimodeling
  - Multi-view modeling
  - Meta modeling
Model Ontologies: Built on Hindley-Milner Type Theories

- A lattice is a partially ordered set (poset) where every subset has a least upper bound (LUB) and a greatest lower bound (GLB).

- Modern type systems (including the Ptolemy II type system, created by Yuhong Xiong) are based on efficient algorithms for solving inequality constraints on lattices.
Property Lattices capture domain-specific semantic information

- Components in a model (e.g. parameters, ports) can have properties drawn from a lattice.
- Components in a model (e.g. actors) can impose constraints on property relationships.
- The type system infrastructure can infer properties and detect errors.
Property Systems

Input constraint and one constraint on a constant leads to inference of semantic information throughout the model.

Car simulator. This model takes as input a desired speed and implements a simple proportional controller with the specified time constant to achieve that speed. It outputs the acceleration, speed, and position of the car.
Beyond Embedded to Cyber-Physical Systems

*The Berkeley Approach*

- **Foundations**
  - *Concurrency and time*

- **Bottom up**
  - *Make behaviors predictable and repeatable*

- **Top down**
  - *Actor component architectures*

- **Holistic**
  - *Model engineering*
The Ptolemy Pteam