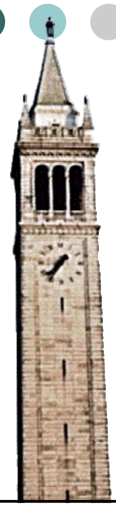
this talk is posted at http://chess.eecs.berkeley.edu/pubs/472.html



Component Architectures for Time-Sensitive Systems

Part 1

Edward A. Lee

Robert S. Pepper Distinguished Professor and

The Onassis Foundation Science Lecture Series
The 2008 Lectures in Computer Science
Embedded Networked Systems: Theory and Applications

With thanks to Chihhong Patrick Cheng, Thomas Huning Feng, Slobodan Matic, Hiren Patel, Eleftherios Matsikoudis, Yang Zhao, and Ye (Rachel) Zhou

Heraklion, Crete
July 24-28, 2008

Embedded Networked Systems

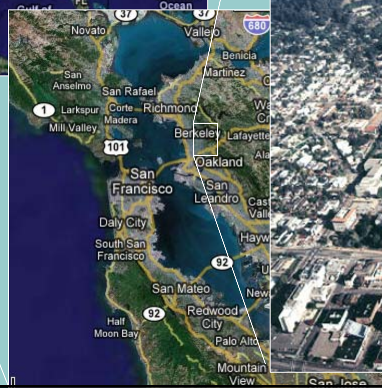
Embedded Systems are electronic components with software, that are specifically designed to provide services in various devices. The great majority (98%) of microprocessors are embedded, and are used in industrial sectors such as transport (avionics, space, automotive, trains), electrical and electronic appliances, process control, telecommunications, e-commerce, and e-health. The extensive and increasing use of embedded systems and their integration in everyday products marks a significant evolution in information science and technology.

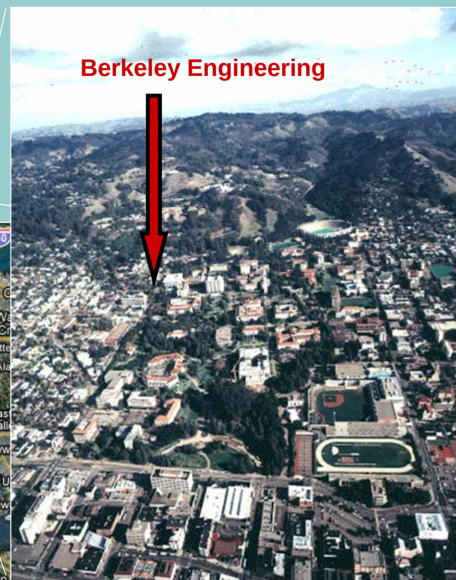
As opposed to other systems, embedded systems should meet requirements for autonomy and optimal use of their resources. This raises fundamental problems that call for enriching computer science with new concepts and paradigms, from control theory and electrical engineering.

The lectures will cover a range of topics spanning both theoretical and practical aspects of embedded systems design. This includes Component-based Design Techniques, Multi-core Architectures and Supercomputing, Wireless Networks, Formal Verification, Security and Timing Analysis.

From:
http://www.forth.gr/onassis/lectures/2008-07-21/

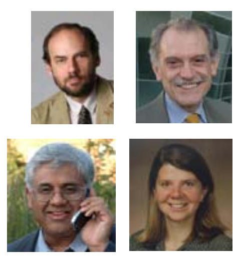
Where I am From: University of California at Berkeley





UC Berkeley has arguably the best public engineering school in the world.

Context of my work: Chess: Center for Hybrid and Embedded Software Systems



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

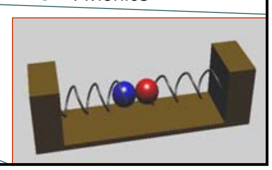
Board of Directors
- Edward A. Lee
- Alberto Sangiovanni-Vincentelli
- Shankar Sastry
- Claire Tomlin

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

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



Today

Morning:
- Why time sensitivity changes everything

Afternoon:
- What to do about it

Time-sensitive systems integrate physical processes, computation, and communication

Dec. 11, 2006: Dancers in Berkeley dancing in real time with dancers in Urbana-Champagne

- medical devices and systems
- assisted living and elder care
- energy conservation
- environmental control
- process control
- critical infrastructure (power, water)
- telepresence
- distributed physical games
- traffic control and safety
- financial networks
- advanced automotive systems,
- aviation systems
- distributed robotics
- military systems
- smart structures
- biosystems (morphogenesis,…)

Potential impact
- integrated medical systems
- safe/efficient transportation
- distributed micro power generation
- disaster recovery
- alternative energy
- social networking and games
- fair financial networks
- military dominance
- economic dominance
- energy efficient buildings
- pervasive adaptive communications
- distributed service delivery
- …

An Emerging Buzzword: Cyber-Physical Systems (CPS)

CPS: Orchestrating networked computational resources with physical processes.

Lee, Berkeley 7

The CPS Vision

“The integration of physical systems and processes with networked computing has led to the emergence of a new generation of engineered systems: Cyber-Physical Systems (CPS). Such systems use computations and communication deeply embedded in and interacting with physical processes to add new capabilities to physical systems. These cyber-physical systems range from miniscule (pace makers) to large-scale (the national power-grid). Because computer-augmented devices are everywhere, they are a huge source of economic leverage.”

- *Charter for CPS Summit, St. Louis, April 25, 2008*

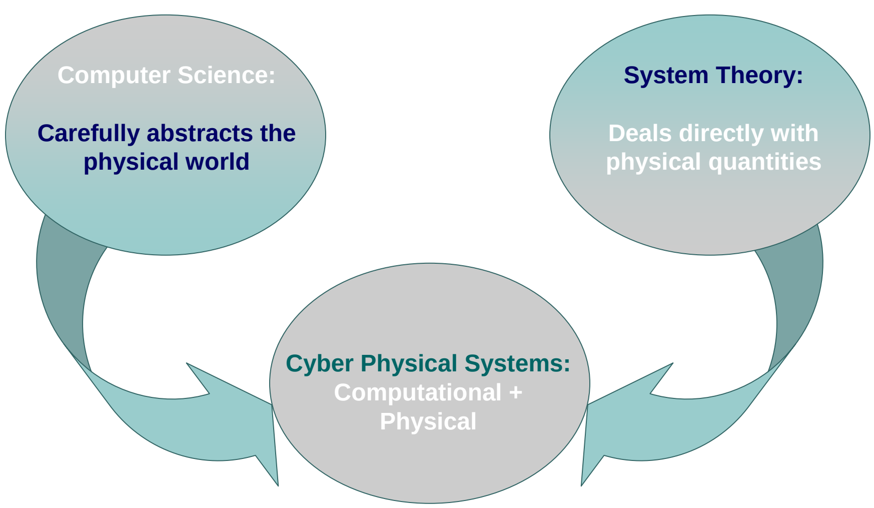
Lee, Berkeley 8

CPS Intellectual Challenge

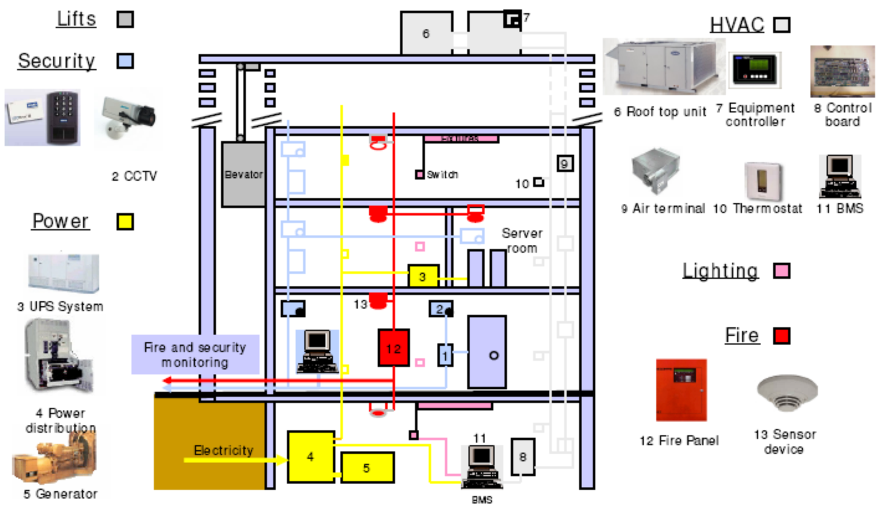
"…it is a profound revolution that turns entire industrial sectors into producers of cyber-physical systems. This is not about adding computing and communication equipment to conventional products where both sides maintain separate identities. This is about merging computing and networking with physical systems to create new revolutionary science, technical capabilities and products."

- Charter for CPS Summit, St. Louis, April 25, 2008

CPS is Multidisciplinary



CPS is Multidisciplinary



Lee, Berkeley 11

A Key Challenge

Models for the physical world and for computation diverge.

- physical: time continuum, ODEs, 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.

Lee, Berkeley 12

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)
- …

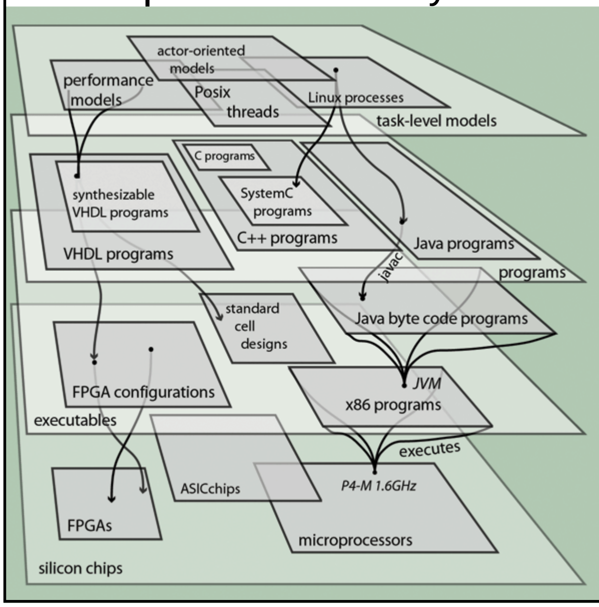
A Story

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.

Related Problems

- **Product families**
  - It is 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

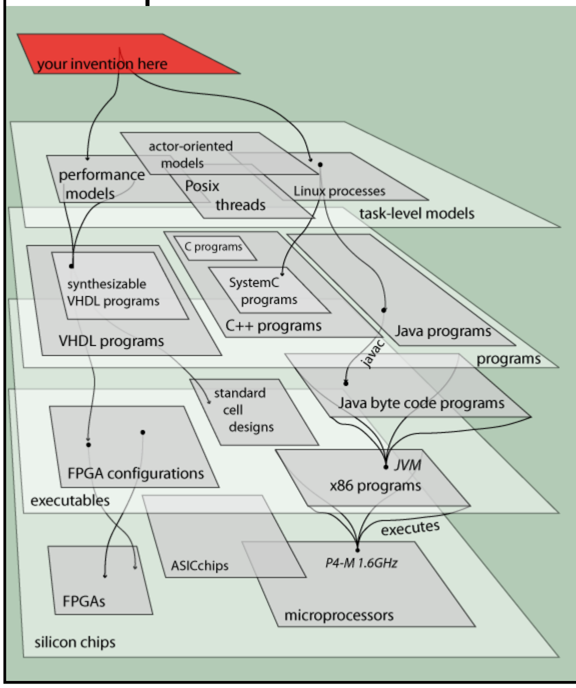


The purpose for an abstraction is to hide details of the implementation below and provide a platform for design from above.

Abstraction Layers

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

A war story:

Ferdinand et al. determine the WCET of astonishingly simple avionics code from Airbus running on a Motorola ColdFire 5307, a pipelined CPU with a unified code and data cache. Despite the software consisting of a fixed set of non-interacting tasks containing only simple control structures, their solution required detailed modeling of the seven-stage pipeline and its precise interaction with the cache, generating a large integer linear programming problem. The technique successfully computes WCET, but only with many caveats that are increasingly rare in software.

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

C. Ferdinand et al., "Reliable and precise WCET determination for a real-life processor." EMSOFT 2001.

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

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. It cannot deliver repeatable and predictable timing, except in trivial cases.

Consider a Simple Example

"The *Observer pattern* defines a one-to-many dependency between a subject object and any number of observer objects so that when the subject object changes state, all its observer objects are notified and updated automatically."

Design Patterns, Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides (Addison-Wesley Publishing Co., 1995. ISBN: 0201633612):

Observer Pattern in Java

```
public void addListener(listener) {…}

public void setValue(newValue) {
    myValue = newValue;

    for (int i = 0; i < myListeners.length; i++) {
        myListeners[i].valueChanged(newValue)
    }
}
```

Will this work in a multithreaded context?

Thanks to Mark S. Miller for the details of this example.

Observer Pattern With Mutual Exclusion (Mutexes)

```
public synchronized void addListener(listener) {…}

public synchronized void setValue(newValue) {
    myValue = newValue;

    for (int i = 0; i < myListeners.length; i++) {
        myListeners[i].valueChanged(newValue)
    }
}
```

Javasoft recommends against this. What's wrong with it?

Mutexes are Minefields

```
public synchronized void addListener(listener) {…}

public synchronized void setValue(newValue) {
    myValue = newValue;

    for (int i = 0; i < myListeners.length; i++) {
        myListeners[i].valueChanged(newValue)
    }
}
```

valueChanged() may attempt to acquire a lock on some other object and stall. If the holder of that lock calls addListener(), deadlock!

Lee, Berkeley 27

```
public synchronized void addChangeListener(ChangeListener listener) {
    NamedObj container = (NamedObj) getContainer();
    if (container != null) {
        container.addChangeListener(listener);
    } else {
        if (_changeListeners == null) {
            _changeListeners = new LinkedList();
            _changeListeners.add(0, listener);
        } else if (!_changeListeners.contains(listener)) {
            _changeListeners.add(0, listener);
        }
    }
}
```

After years of use without problems, a Ptolemy Project code review found code that was not thread safe. It was fixed in this way. Three days later, a user in Germany reported a deadlock that had not shown up in the test suite.

Simple Observer Pattern Becomes Not So Simple

```
public synchronized void addListener(listener) {…}

public void setValue(newValue) {
    synchronized(this) {
        myValue = newValue;
        listeners = myListeners.clone();
    }

    for (int i = 0; i < listeners.length; i++) {
        listeners[i].valueChanged(newValue)
    }
}
```

while holding lock, make copy of listeners to avoid race conditions

notify each listener outside of synchronized block to avoid deadlock

This still isn't right.
What's wrong with it?

Simple Observer Pattern: How to Make It Right?

```
public synchronized void addListener(listener) {…}

public void setValue(newValue) {
    synchronized(this) {
        myValue = newValue;
        listeners = myListeners.clone();
    }

    for (int i = 0; i < listeners.length; i++) {
        listeners[i].valueChanged(newValue)
    }
}
```

Suppose two threads call setValue(). One of them will set the value last, leaving that value in the object, but listeners may be notified in the opposite order. The listeners may be alerted to the value changes in the wrong order!

If the simplest design patterns yield such problems, what about non-trivial designs?

```
/**
CrossRefList is a list that maintains pointers to other CrossRefLists.
…
@author Geroncio Galicia, Contributor: Edward A. Lee
@version $Id: CrossRefList.java,v 1.78 2004/04/29 14:50:00 eal Exp $
@since Ptolemy II 0.2
@Pt.ProposedRating Green (eal)
@Pt.AcceptedRating Green (bart)
*/
public final class CrossRefList implements Serializable  {
    …
    protected class CrossRef implements Serializable{
        …
        // NOTE: It is essential that this method not be
        // synchronized, since it is called by _farContainer(),
        // which is.  Having it synchronized can lead to
        // deadlock.  Fortunately, it is an atomic action,
        // so it need not be synchronized.
        private Object _nearContainer() {
            return _container;
        }

        private synchronized Object _farContainer() {
            if (_far != null) return _far._nearContainer();
            else return null;
        }
        …
    }
}
```

Code that had been in use for four years, central to Ptolemy II, with an extensive test suite with 100% code coverage, design reviewed to yellow, then code reviewed to green in 2000, causes a deadlock during a demo on April 26, 2004.

Lee, Berkeley 31

What it Feels Like to Use the *synchronized* Keyword in Java

Image "borrowed" from an Iomega advertisement for Y2K software and disk drives, *Scientific American*, September 1999.

Lee, Berkeley 32

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."*

H. Sutter and J. Larus. Software and the concurrency revolution. ACM Queue, 3(7), 2005.

Is Concurrency Hard?

It is not concurrency that is hard…

…It is Threads that are 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…

Succinct Problem Statement

Threads are wildly nondeterministic.

The programmer's job is to prune away the nondeterminism by imposing constraints on execution order (e.g., mutexes) and limiting shared data accesses (e.g., OO design).

We Can Incrementally Improve Threads

- Object Oriented programming
- Coding rules (Acquire locks in the same order…)
- Libraries (Stapl, Java 5.0, …)
- Patterns (MapReduce, …)
- Transactions (Databases, …)
- Formal verification (Blast, thread checkers, …)
- Enhanced languages (Split-C, Cilk, Guava, …)
- Enhanced mechanisms (Promises, futures, …)

But is it enough to refine a mechanism with flawed foundations?

The Result: Brittle Designs

Small changes have big consequences…

Patrick Lardieri, *Lockheed Martin ATL*, about a vehicle management system in the JSF program:

> "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 - System went from meeting real-time requirements to missing most deadlines due to a change that was expected to have no impact on system performance."

National Workshop on High-Confidence Software Platforms for Cyber-Physical Systems (HCSP-CPS) Arlington, VA November 30 –December 1, 2006

The Current State of Affairs

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

Just think what we could do with the right abstractions!

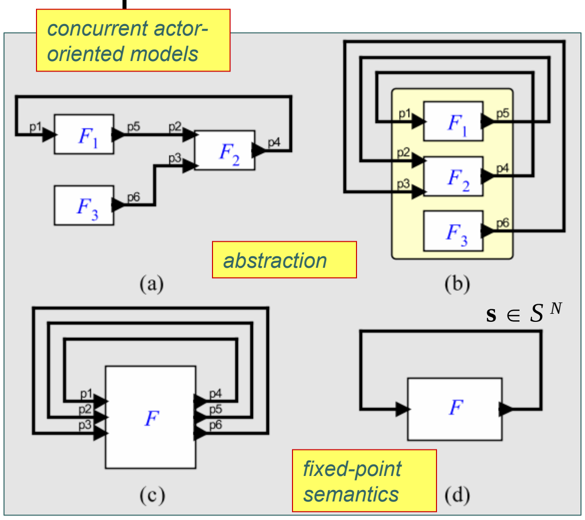
The Solution Space

Reintroduce time into the core abstractions:

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

Foundations: Timed-Computational Semantics.

super-dense time



- Signal: $s \colon \mathbb{R}_+ \times \mathbb{N} \rightharpoonup V_\varepsilon$
- Set of signals: S
- Tuples of signals: $\mathbf{s} \in S^N$
- Actor: $F \colon S^N \to S^M$

A unique least fixed point, $\mathbf{s} \in S^N$ such that $F(\mathbf{s}) = \mathbf{s}$, exists and be constructively found if S^N is a CPO and F is (Scott) continuous.

Causal systems operating on signals are usually naturally (Scott) continuous.

Some Reading on Foundations

Ph.D. Theses:

[1] Haiyang Zheng, "Operational Semantics of Hybrid Systems," May 18, 2007.

[2] Ye Zhou, "Interface Theories for Causality Analysis in Actor Networks," May 15, 2007.

[3] Xiaojun Liu, "Semantic Foundation of the Tagged Signal Model," December 20, 2005.

Papers:

[1] Lee and Matsikoudis, "The Semantics of Dataflow with Firing," in *From Semantics to Computer Science: Essays in memory of Gilles Kahn*, Cambridge 2008.

[2] Zhou and Lee. "Causality Interfaces for Actor Networks," ACM Trans. on Embedded Computing Systems, April 2008.

[3] Lee, " Application of Partial Orders to Timed Concurrent Systems," article in *Partial order techniques for the analysis and synthesis of hybrid and embedded systems*, in CDC 07.

[4] Liu and Lee, "CPO Semantics of Timed Interactive Actor Networks," Technical Report No. UCB/EECS-2007-131, November 5, 2007 (under review).

[5] Lee and Zheng, "Leveraging Synchronous Language Principles for Heterogeneous Modeling and Design of Embedded Systems," EMSOFT '07.

[6] Liu, Matsikoudis, and Lee. "Modeling Timed Concurrent Systems," CONCUR '06.

[7] Cataldo, Lee, Liu, Matsikoudis and Zheng "A Constructive Fixed-Point Theorem and the Feedback Semantics of Timed Systems," WODES'06

etc. ...

Our Solution

Reintroduce time into 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?)

See S. Edwards and E. A. Lee, "**The Case for the Precision Timed (PRET) Machine**," in the *Wild and Crazy Ideas* Track of the *Design Automation Conference* (DAC), June 2007.

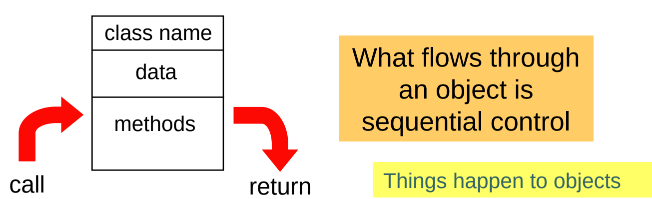
Our Solution

Reintroduce time into the core abstractions:

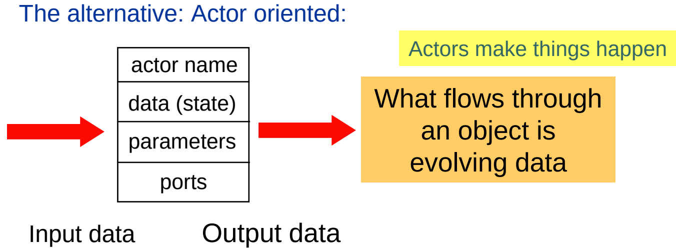
- *Foundations:* Timed computational semantics.
- *Bottom up*: Make timing repeatable.
- *Top down*: Timed, concurrent components.
- *Holistic*: Model engineering.

Object Oriented vs. Actor Oriented

The established: Object-oriented:



The alternative: Actor oriented:



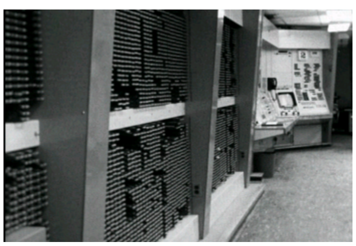
New Component Technology is more Palatable than New Languages

- 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

The First (?) Actor-Oriented Programming Language

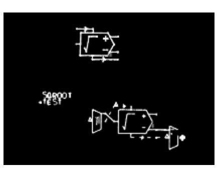
The On-Line Graphical Specification of Computer Procedures

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



MIT Lincoln Labs TX-2 Computer

Bert Sutherland with a light pen



Partially constructed actor-oriented model with a class definition (top) and instance (below).

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).

Examples of Actor-Oriented Systems

- 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, ...)
- LabVIEW (structured dataflow, National Instruments)
- Modelica (continuous-time, constraint-based, Linkoping)
- OPNET (discrete events, Opnet Technologies)
- SDL (process networks)
- Occam (rendezvous)
- Simulink (Continuous-time, The MathWorks)
- SPW (synchronous dataflow, Cadence, CoWare)
- …

Most of these are domain specific.

Many of these have visual syntaxes.

The semantics of these differ considerably, with significantly different approaches to concurrency.

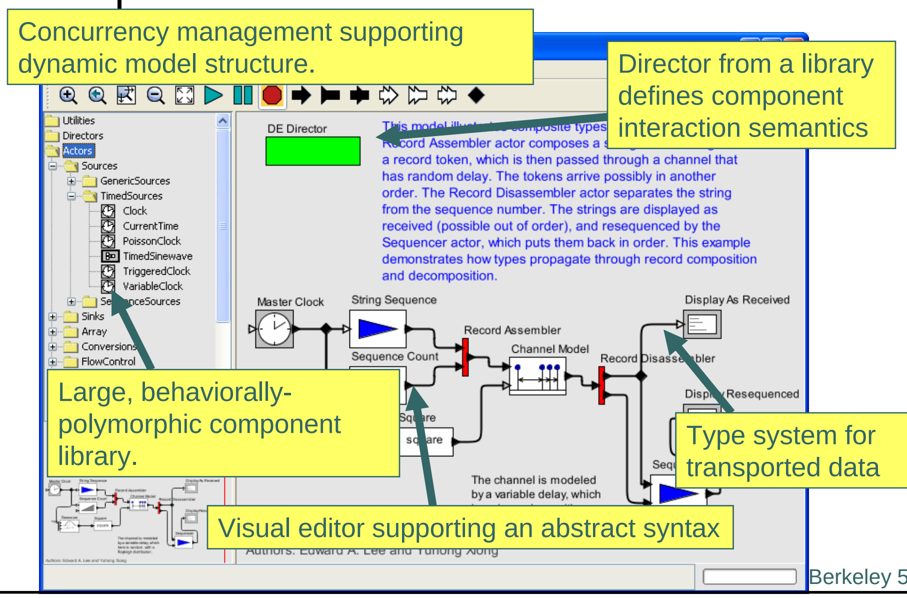
Challenges

The technology is immature:

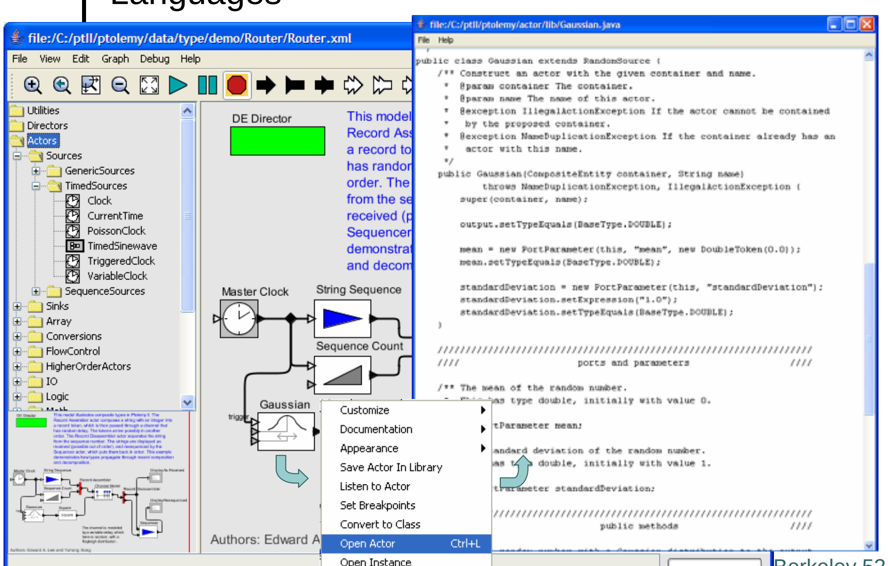
- Commercial actor-oriented systems are domain-specific
- Development tools are limited
- Little language support in C++, C#, Java
- Modularity mechanisms are underdeveloped
- Type systems are primitive
- Compilers (called “code generators”) are underdeveloped
- Formal methods are underdeveloped
- Libraries are underdeveloped

We are addressing these problems.

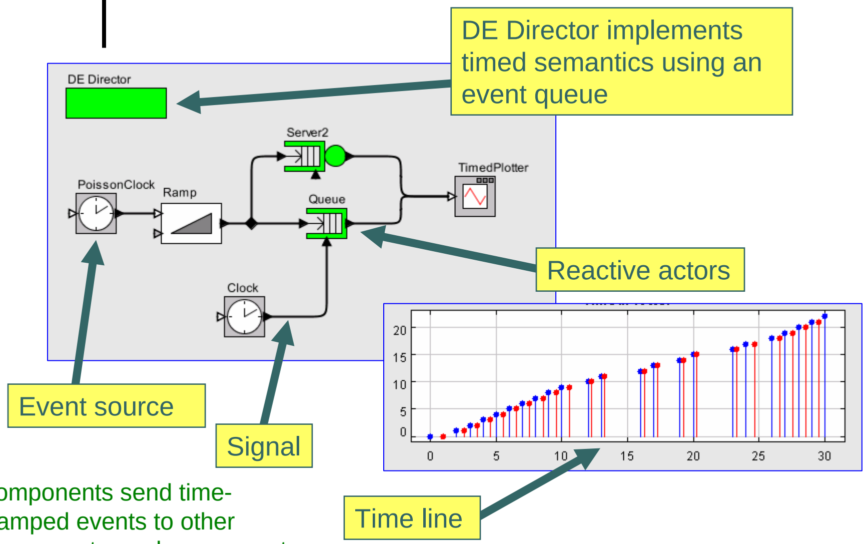
Ptolemy II: Our Laboratory for Experiments with Actor-Oriented Design



Approach: Concurrent Composition of Components designed with Conventional Languages



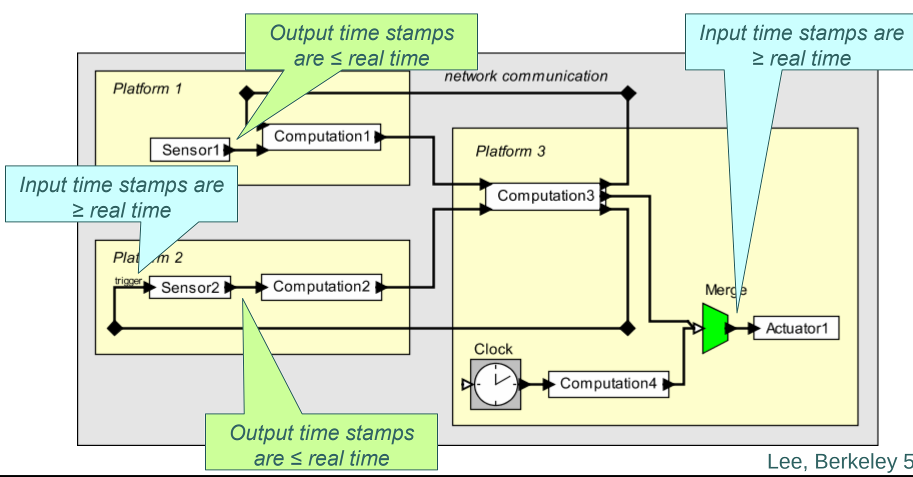
Example: Discrete Event Models



Components send time-stamped events to other components, and components react in chronological order.

PTIDES: Programming Temporally Integrated Distributed Embedded Systems

Distributed execution under DE semantics, with "model time" and "real time" bound at sensors and actuators.

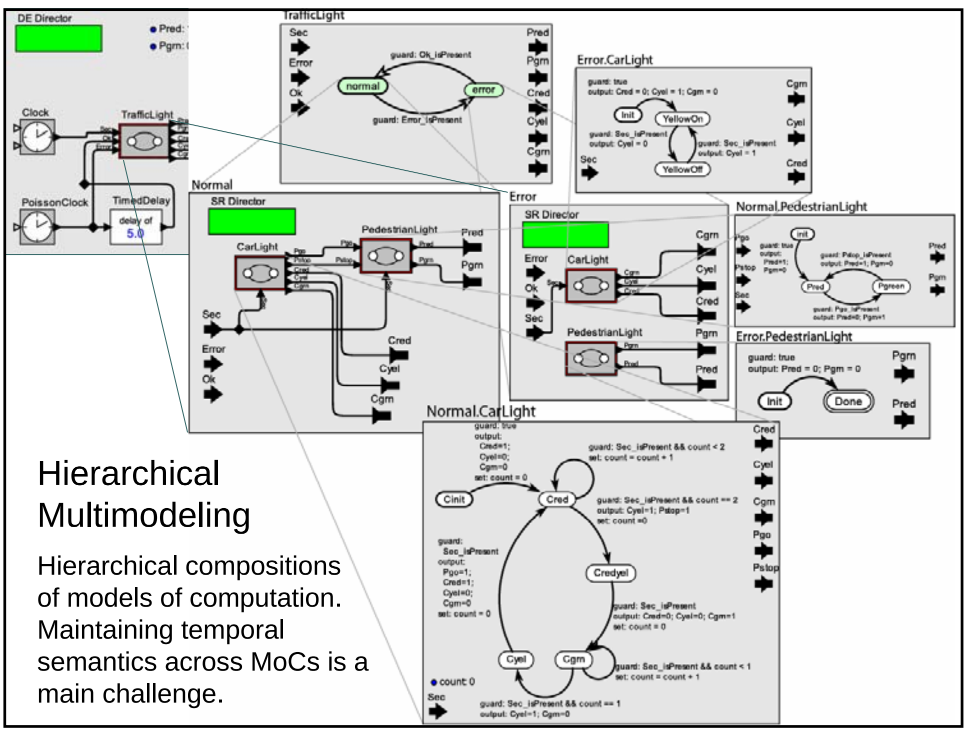


Our Solution

Reintroduce time into the core abstractions:

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

Lee, Berkeley 55



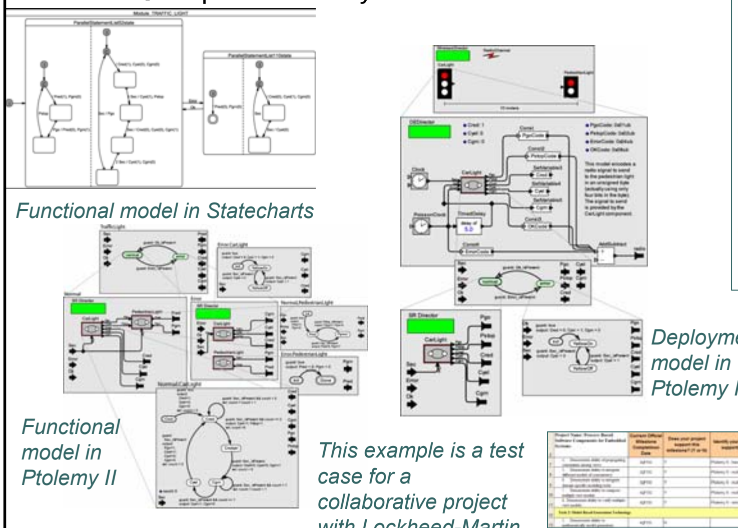
Hierarchical Multimodeling

Hierarchical compositions of models of computation. Maintaining temporal semantics across MoCs is a main challenge.

Multi-View Modeling:

Distinct and separate models of the same system are constructed to model different aspects of the system.

Verification model in SMV



Lee, Berkeley 57

Model Engineering Projects

- Data ontologies
- Property annotations
- Model transformations
- Higher-order actors
- Workflow management

Lee, Berkeley 58

Making Time Essential in Computation

Reintroduce time into the core abstractions:

- *Foundations:* Timed computational semantics.
  - *Abstract semantics on super-dense time*
- *Bottom up*: Make timing repeatable.
  - Precision-timed (PRET) machines
- *Top down*: Timed, concurrent components.
  - Distributed real-time discrete-events (PTIDES)
- *Holistic*: Model engineering.
  - Mulimodeling, ontologies, property system, …