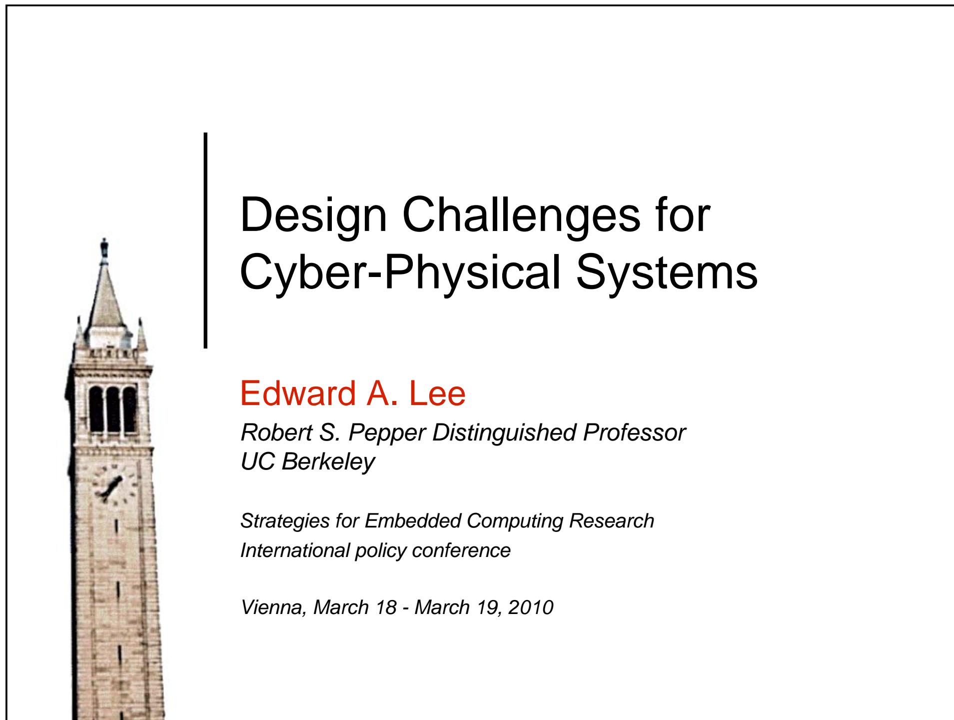


Abstract

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. These systems are multi-scale and heterogeneous, mixing wide ranges of technologies. One of the key challenges is that prevailing abstractions used in computing 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, whereas for physical systems, the passage of time is intrinsic in their dynamic behavior. This talk examines the obstacles in software and networking 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 modify key software technologies. These abstractions will have to embrace physical dynamics and computation in a unified way. This talk will discuss research challenges and potential solutions.

Cyber-Physical Systems (CPS):

Orchestrating networked computational resources with physical systems

Avionics

Transportation (Air traffic control at SFO)

Building Systems

Telecommunications

Automotive

E-Corner, Siemens

Instrumentation (Soleil Synchrotron)

Factory automation

Power generation and distribution

Daimler-Chrysler

Military systems:

Courtesy of Doug Schmidt

Courtesy of General Electric

Courtesy of Kuka Robotics Corp.

Lee, Berkeley 3

CPS Example – Printing Press

Bosch-Rexroth

- *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)*

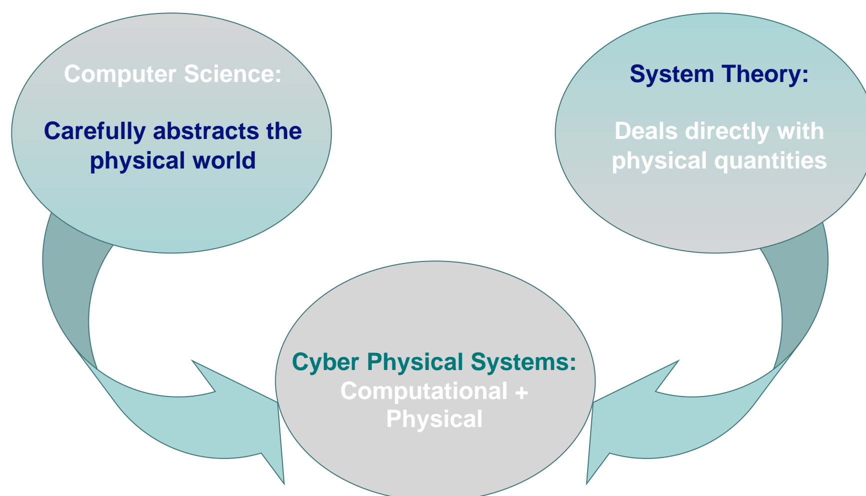
Lee, Berkeley 4

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.

Lee, Berkeley 5

CPS is Multidisciplinary



Lee, Berkeley 6

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.

Lee, Berkeley 7

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.

Lee, Berkeley 8

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)
- …

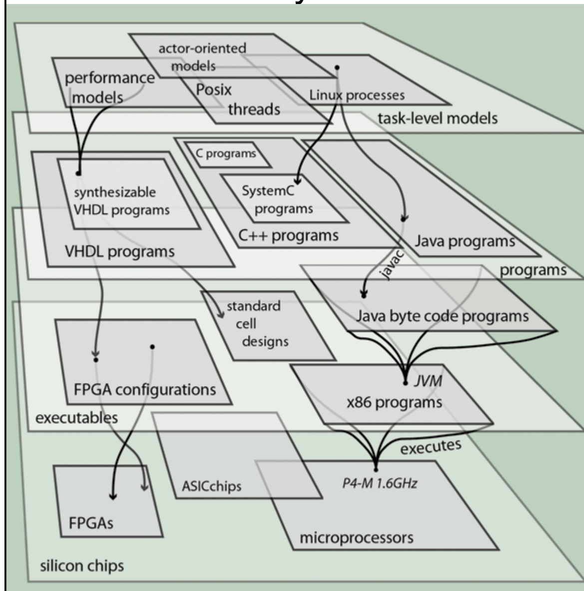
A Story

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

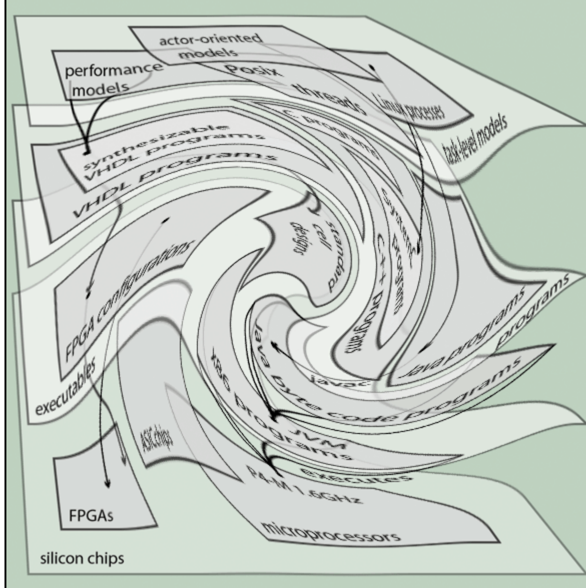
- **Stockpiling for a product run**
  - Some systems vendors have to purchase up front the entire expected part requirements for an entire product run.
- **Frozen designs**
  - Once certified, errors cannot be fixed and improvements cannot be made.
- **Product families**
  - Difficult to maintain and 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.

Abstraction Layers in Common Use



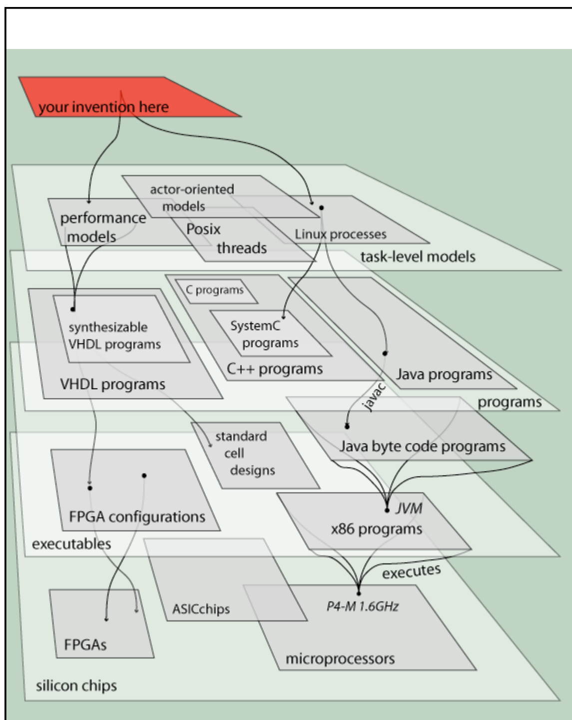
The purpose for an abstraction is to hide details of the implementation below and provide a platform for design from above.

Abstraction Layers in Common Use



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

The design *is* the implementation.



How about “raising the level of abstraction” to solve these problems?

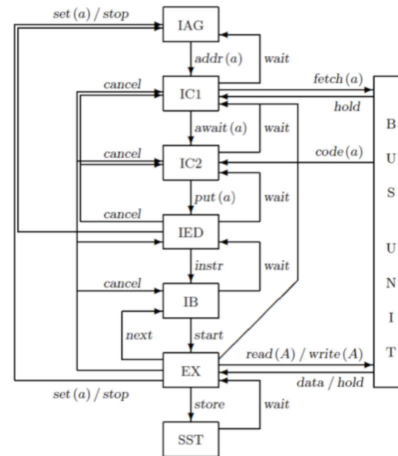
But these higher abstractions rely on an increasingly problematic fiction: WCET

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!

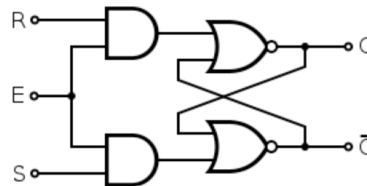
Fundamentally, the ISA of the processor has failed to provide an adequate abstraction.



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

Lee, Berkeley 15

The Key Problem



Electronics technology delivers highly reliable and precise timing…

20.000 MHz (± 100 ppm)

… and the overlaying software abstractions discard it.

Lee, Berkeley 16

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

Even distributed software commonly goes to considerable lengths to emulate this rather poor abstraction using middleware that supports RPC, proxies, and data replication.

Lee, Berkeley 17

My Claim

Nontrivial software written with threads is incomprehensible to humans, and it cannot deliver repeatable or predictable behavior, except in trivial cases.

Lee, Berkeley 18

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…

Concurrent programs using shared memory are incomprehensible because concurrency in the physical world does not work that way.

We have no experience!

Concurrent Programs with Threads and Interrupts are *Brittle*

Small changes can have big consequences.

Consider a multithreaded program on multicore:

Theorem (Richard Graham, 1976): If a task set with fixed priorities, execution times, and precedence constraints is optimally scheduled on a fixed number of processors, then increasing the number of processors, reducing execution times, or weakening precedence constraints can increase the schedule length.

Lee, Berkeley 23

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!

Lee, Berkeley 24

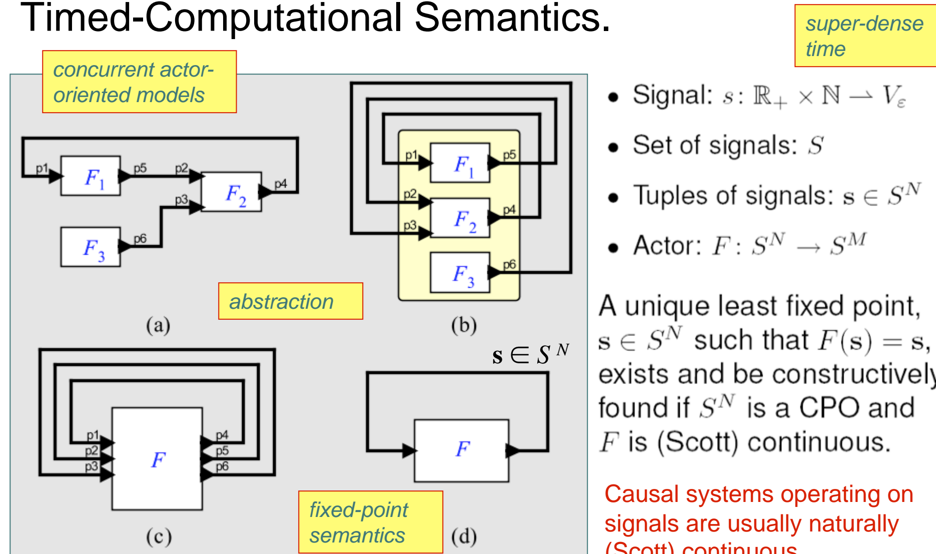
The Berkeley Approach

Time and concurrency in the core abstractions:

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

Lee, Berkeley 25

Foundations: Timed-Computational Semantics.

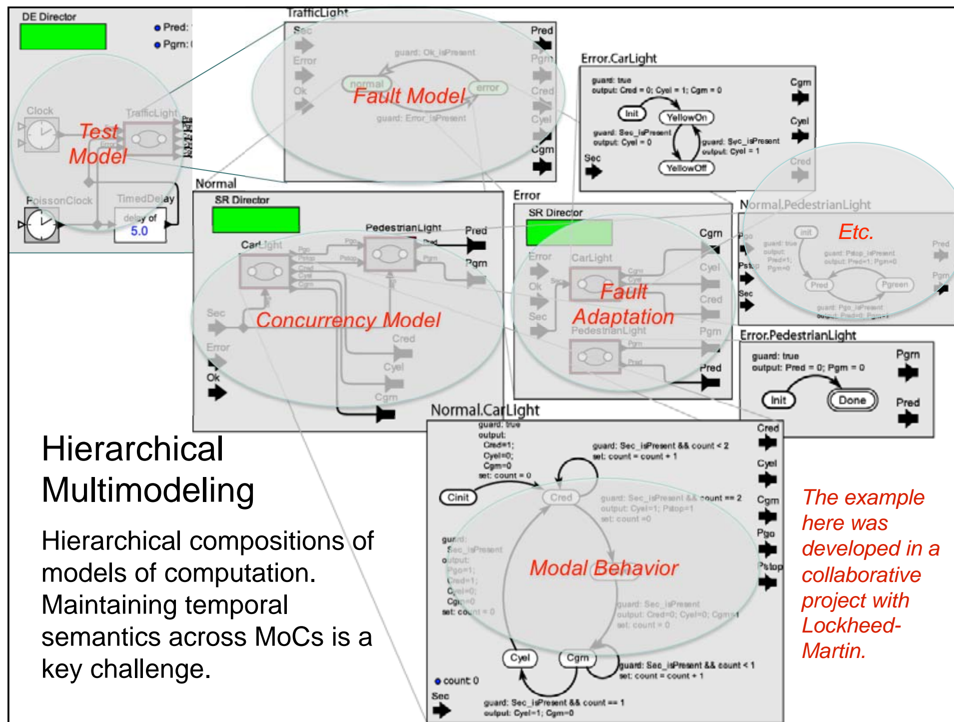


- Signal: $s : \mathbb{R}_+ \times \mathbb{N} \rightharpoonup V_\varepsilon$
- Set of signals: S
- Tuples of signals: $\mathbf{s} \in S^N$
- Actor: $F : S^N \to S^M$

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.

Lee, Berkeley 26



The Berkeley Approach

Time and concurrency in the core abstractions:

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

Lee, Berkeley 28

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.

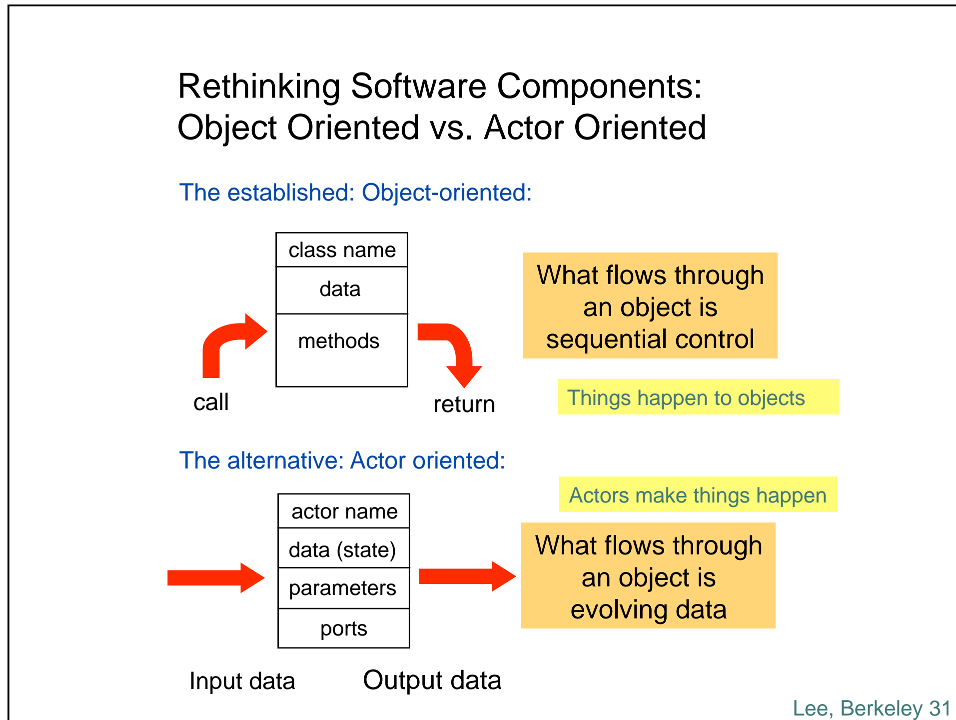
Lee, Berkeley 29

The Berkeley Approach

Time and concurrency in the core abstractions:

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

Lee, Berkeley 30



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.

Lee, Berkeley 32

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

Concurrency management supporting dynamic model structure.

Director from a library defines component interaction semantics

Large, behaviorally-polymorphic component library.

Type system for transported data

Visual editor supporting an abstract syntax

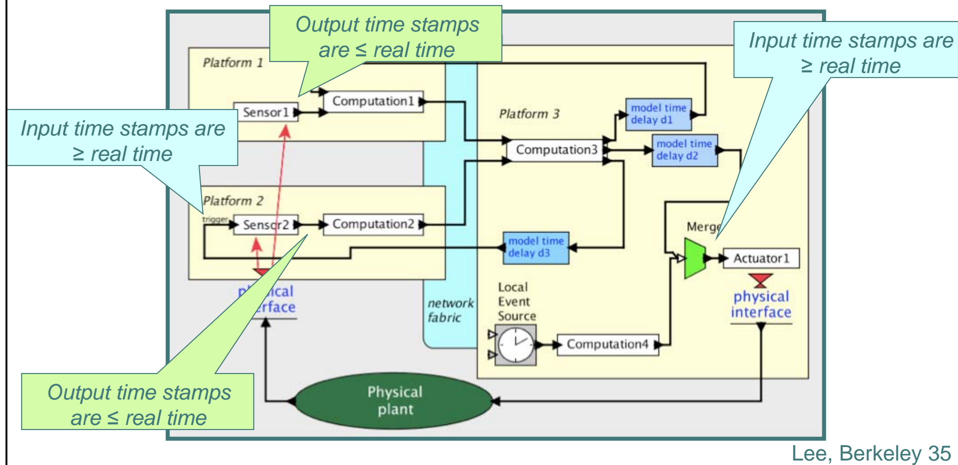
Berkeley 33

Approach: Concurrent Composition of *Components* designed with Conventional Languages

Berkeley 34

A Key Concern: Timing Properties in the Interface of Software Components

One approach is a model of computation that we call PTIDES, which combines discrete events with a binding to real time.



Lee, Berkeley 35

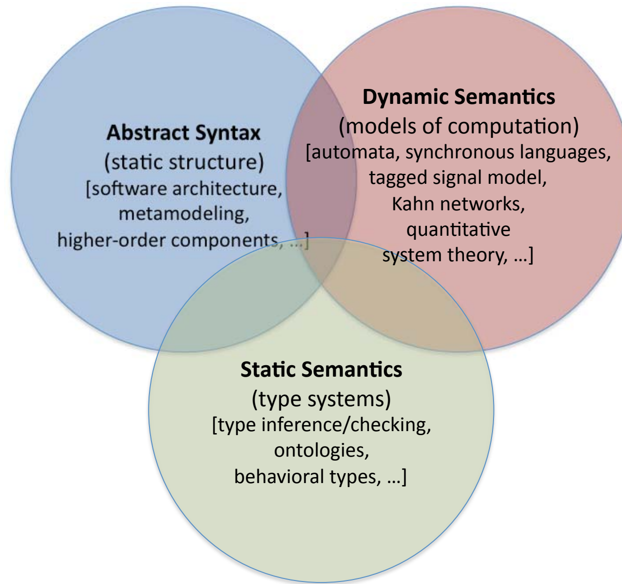
The Berkeley Approach

Time and concurrency in the core abstractions:

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

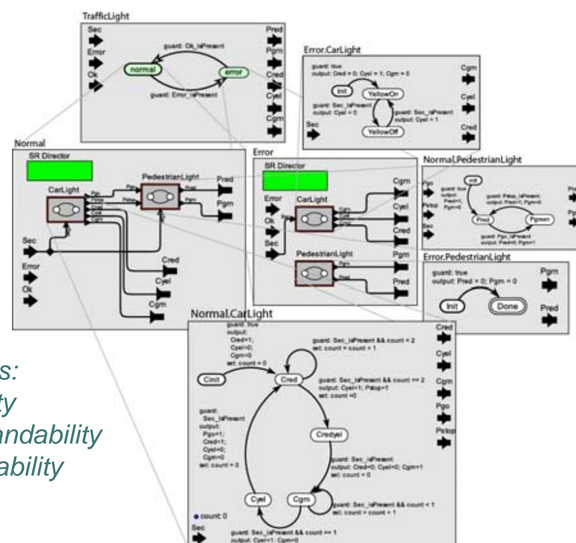
Lee, Berkeley 36

A Taxonomy of Modeling Issues



Lee, Berkeley 37

Abstract Syntax
Communicating Hierarchical Components



Challenges:
- *Scalability*
- *Understandability*
- *Composability*

Lee, Berkeley 38

Static Semantics
Correctly Composing Models

Challenges:
- *Scalability*
- *Understandability*
- *Composability*
- *Consistency*

Simple example of a type lattice

Our approach:

Leverage/generalize type theories:
- Foundations: Fixed-point theorems for monotonic functions on mathematical lattices.
- Modern type systems are based on efficient algorithms for solving inequality constraints on lattices.
- Such lattices, however, can represent much more than data types.

Lee, Berkeley 39

Dynamic Semantics
Correctly Composing Models

Challenges:
- *Scalability*
- *Understandability*
- *Composability*
- *Consistency*
- *Verification*

Behavioral types can represent dynamic properties of components of a system *within a type-theoretic framework that enables compatibility checking. We will:*
- *Identify properties of interfaces that enable composition and show how compositional interfaces can be used in hierarchical heterogeneous specifications.*
- *Build prototype software that composes interfaces.*
- *Refine algorithms for composition of interfaces and identify performance bottlenecks*

Our approach:
Behavioral Types

Lee, Berkeley 40

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*