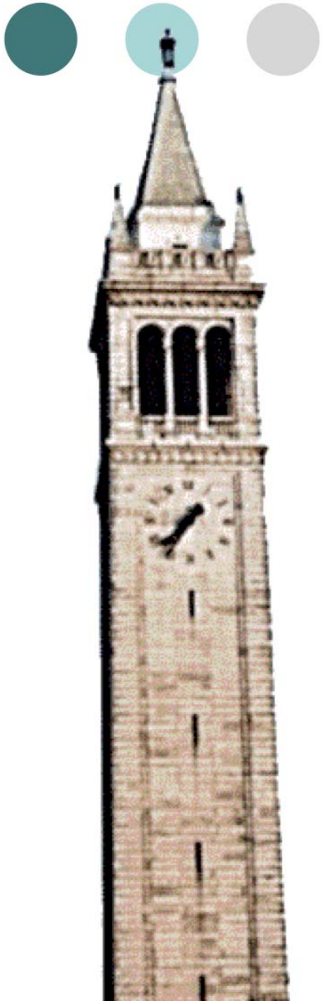


Beyond Embedded Systems:
Integrating Computation, Networking, and Physical Dynamics

Edward A. Lee

Robert S. Pepper Distinguished Professor
UC Berkeley

Invited Keynote Talk

ACM SIGPLAN/SIGBED 2009 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)

Dublin, Ireland

June 19-20, 2009

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 technologists and computer scientists.

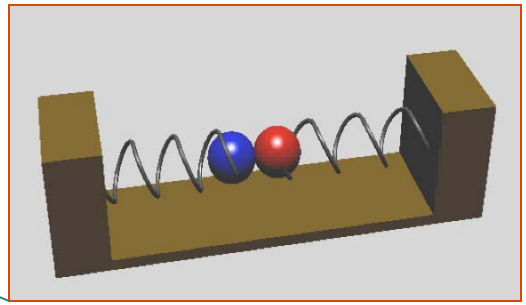
Board of Directors
- Edward A. Lee
- Alberto Sangiovanni-Vincentelli
- Shankar Sastry
- Claire Tomlin

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

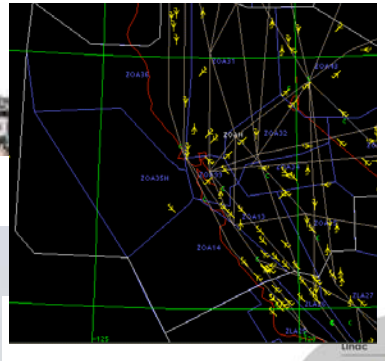
Applications
- Building systems
- Automotive
- Synthetic biology
- Medical systems
- Instrumentation
- Factory automation
- Avionics



Cyber-Physical Systems (CPS):

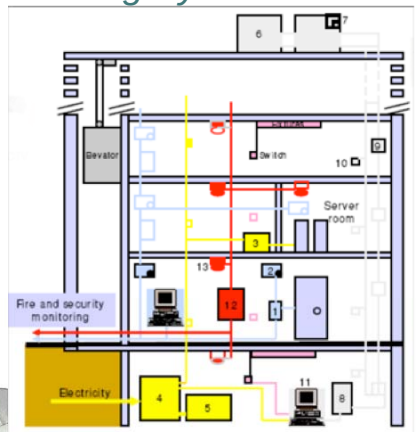
Orchestrating networked computational resources with physical systems

Avionics



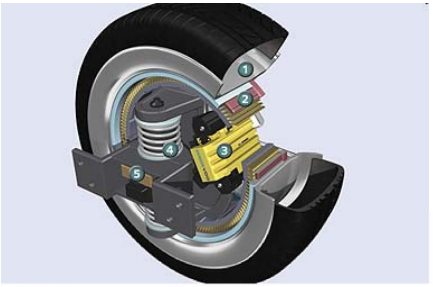
Transportation (Air traffic control at SFO)

Building Systems

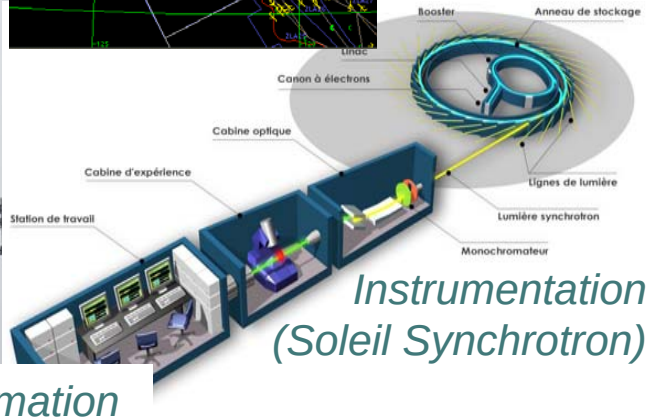


Telecommunications

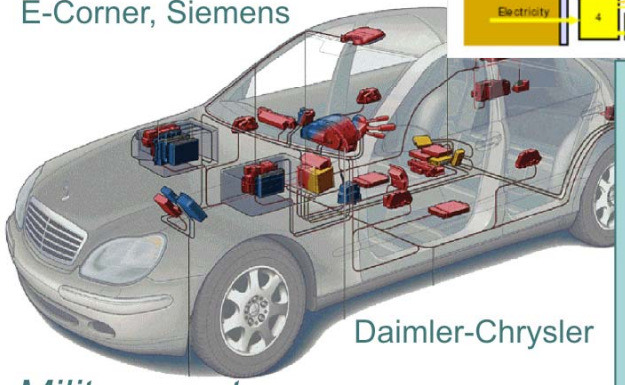
Automotive



E-Corner, Siemens

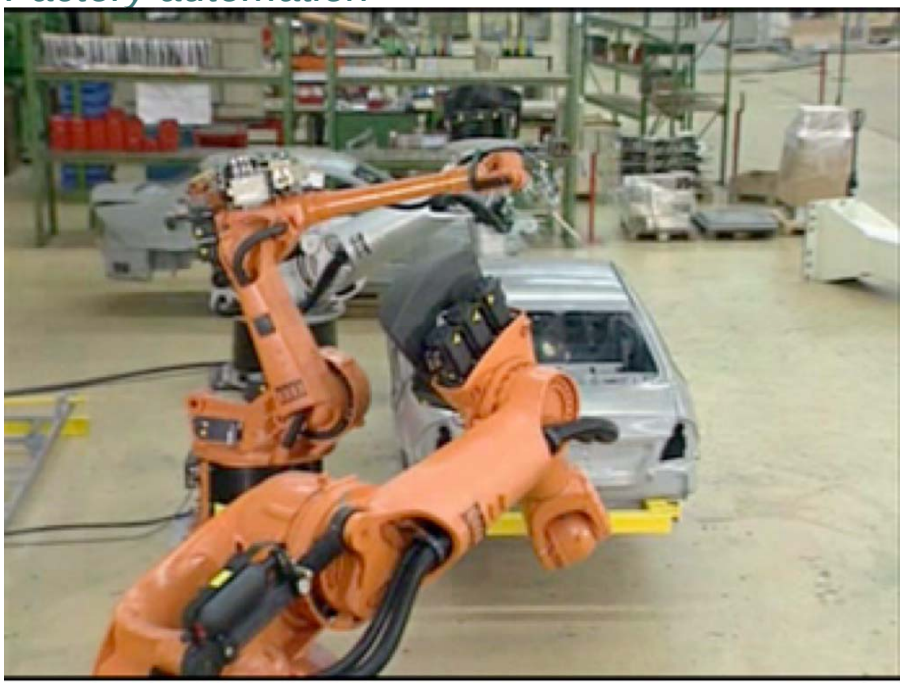


Instrumentation (Soleil Synchrotron)



Daimler-Chrysler

Factory automation

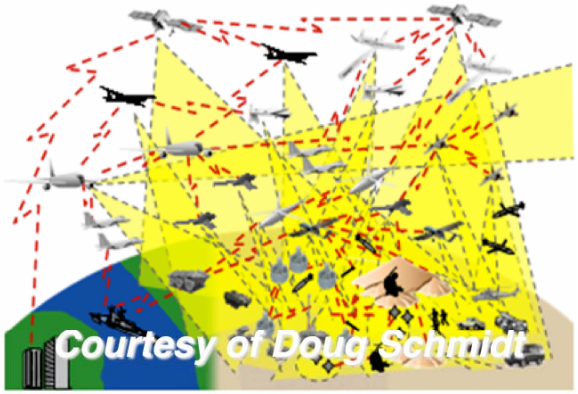


Courtesy of Kuka Robotics Corp.

Power generation and distribution

Courtesy of General Electric

Military systems:



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)*

Where CPS Differs from the traditional embedded systems problem:

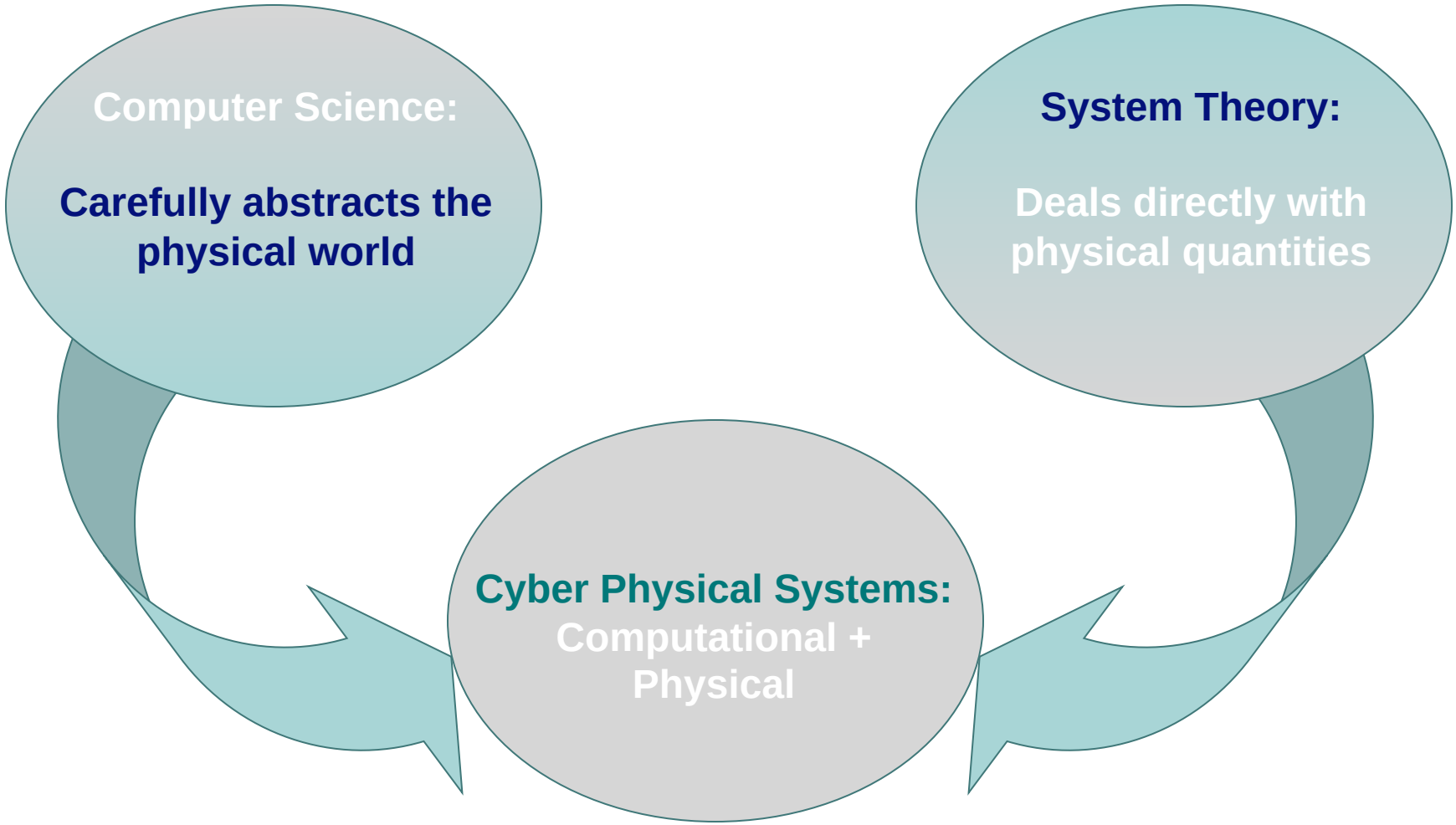
- *The traditional embedded systems problem:*

  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



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)
- …

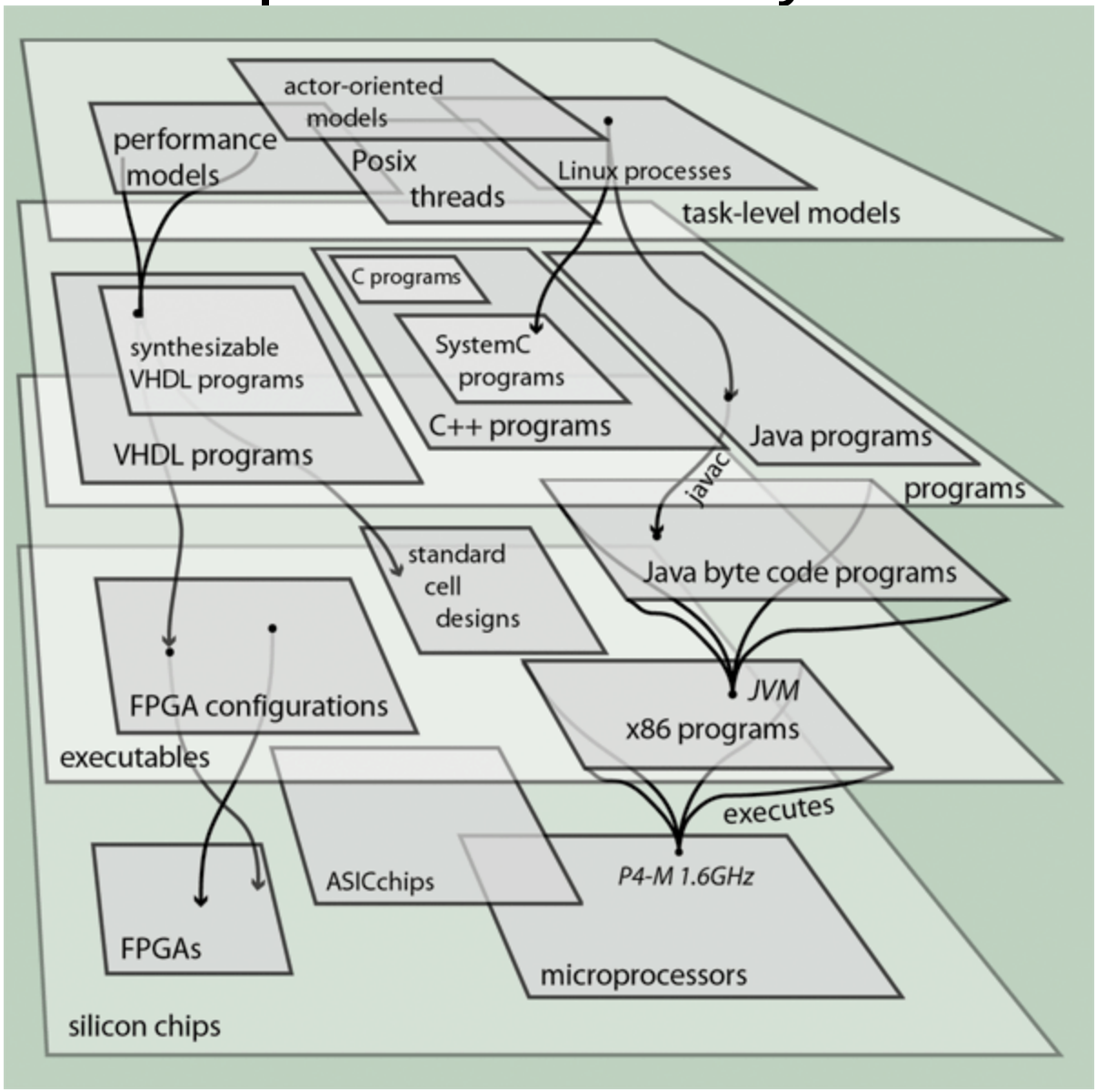
A Story

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

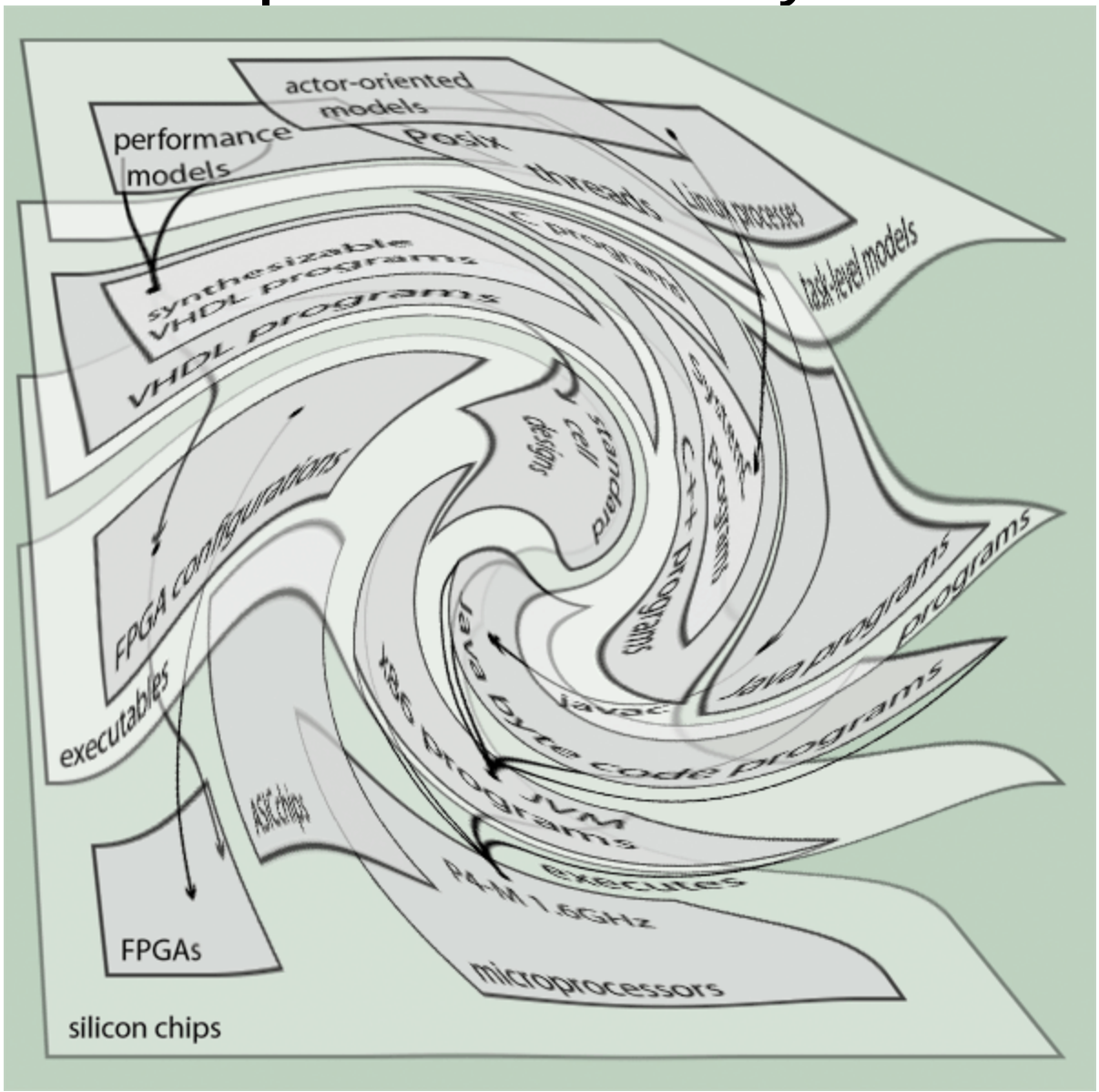
- **Stockpiling for a product run**
  - Some systems vendors have to purchase up front the entire expected part requirements for an entire product run.
- **Frozen designs**
  - Once certified, errors cannot be fixed and improvements cannot be made.
- **Product families**
  - Difficult to maintain and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

Abstraction Layers



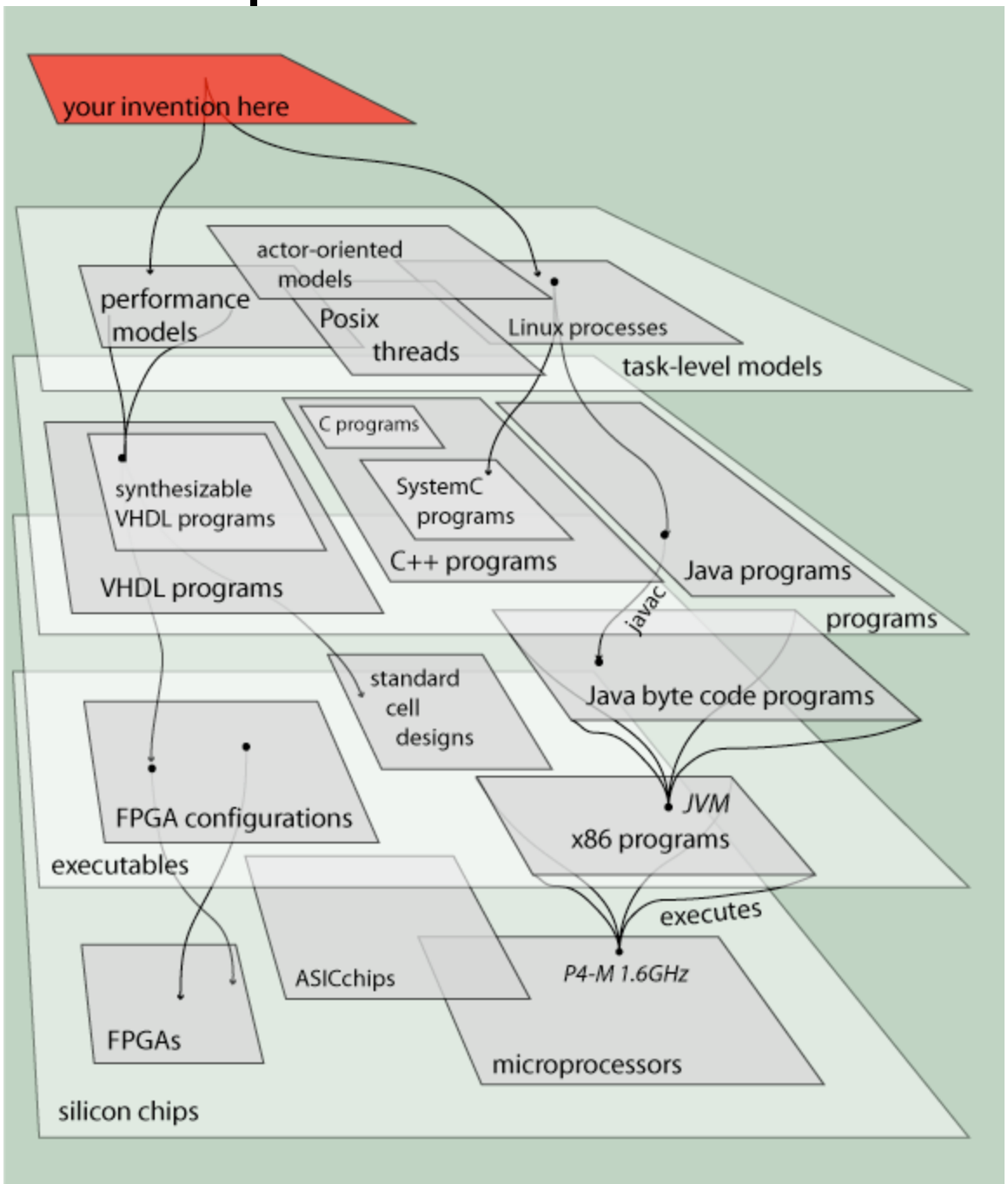
The purpose for an abstraction is to hide details of the implementation below and provide a platform for design from above.

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

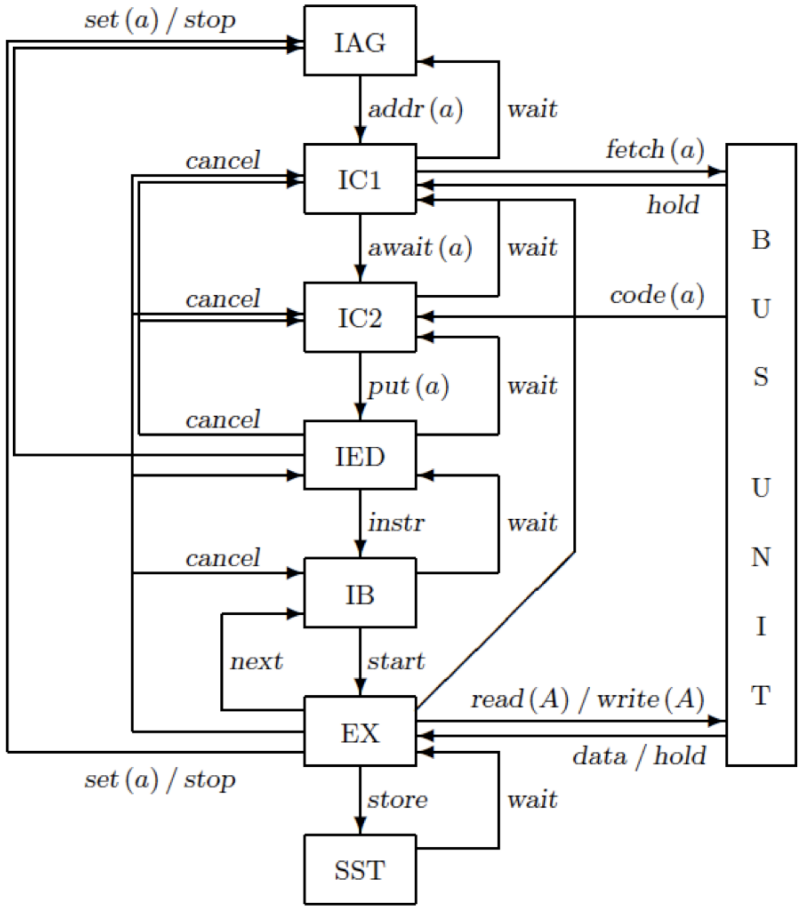
Example war story:

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

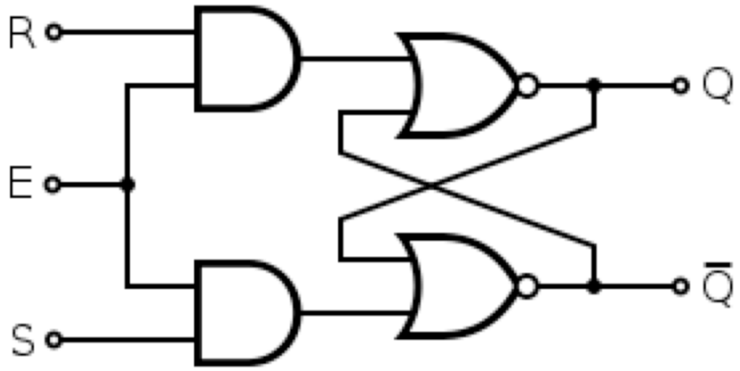
And the result is valid only for that exact Hardware and software!

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.

The Key Problem



Electronics technology delivers highly reliable and precise timing…

20.000 MHz (± 100 ppm)

… 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."*

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!

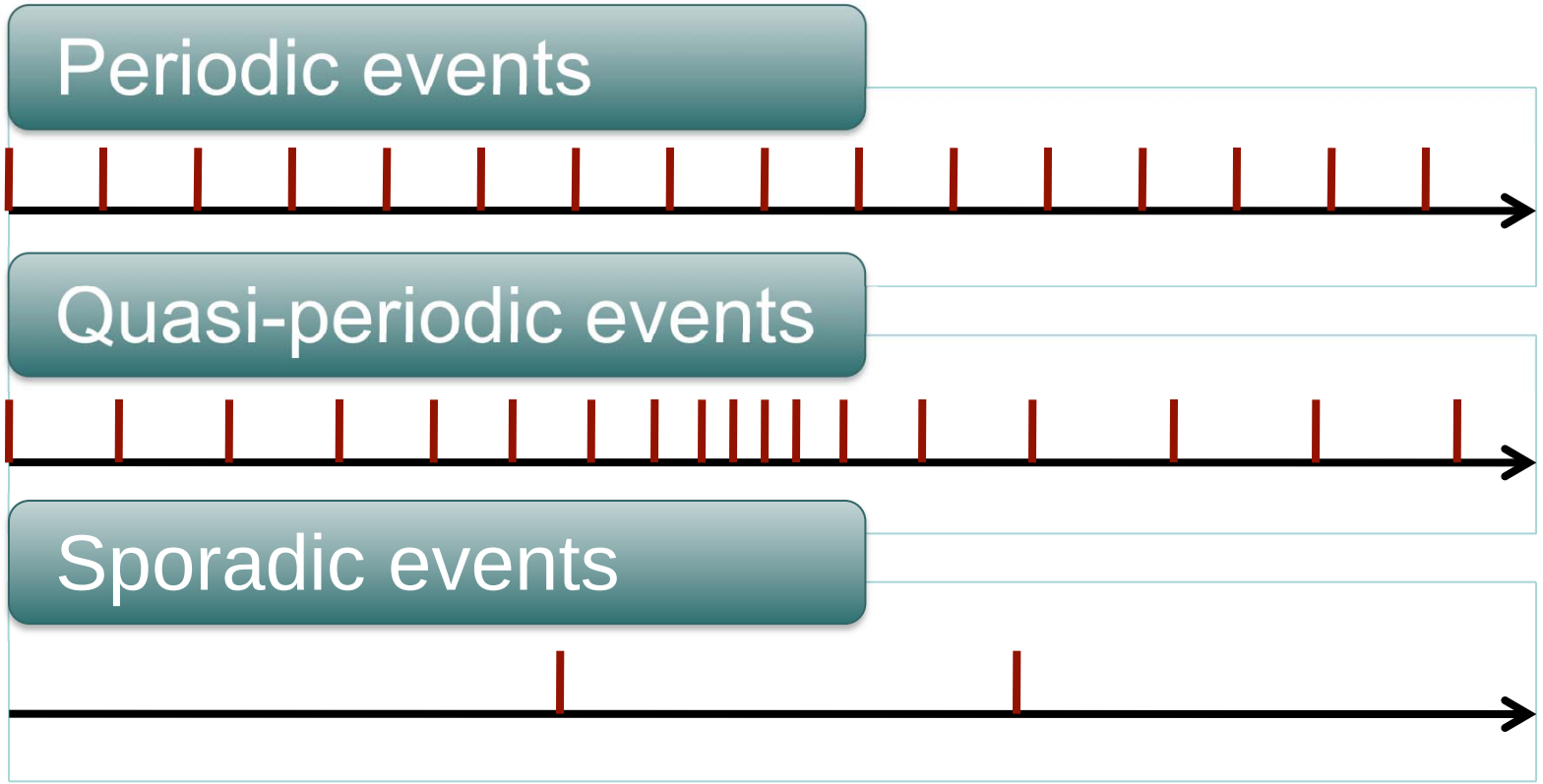
Threads are sequential processes that share memory. From the perspective of any thread, the entire state of the universe can change between any two atomic actions (itself an ill-defined concept).

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!

Consider an Automotive Example



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!

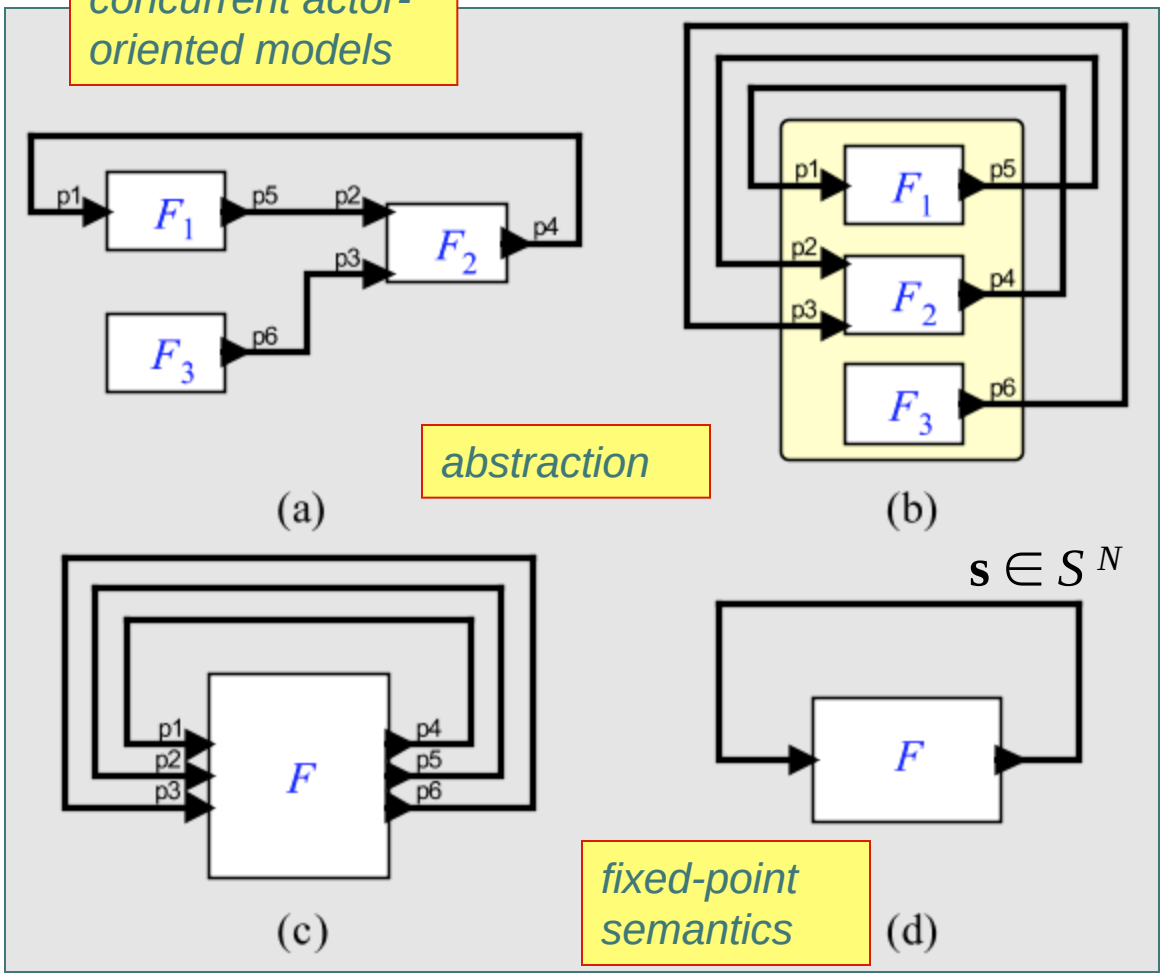
The Berkeley Solution

Time and concurrency in the core abstractions:

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

Foundations:
Timed-Computational Semantics.

concurrent actor-oriented models



super-dense time

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

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. ...

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?)

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.

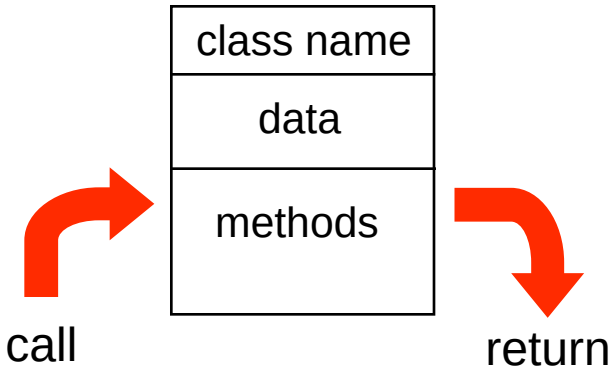
The Berkeley Solution

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:

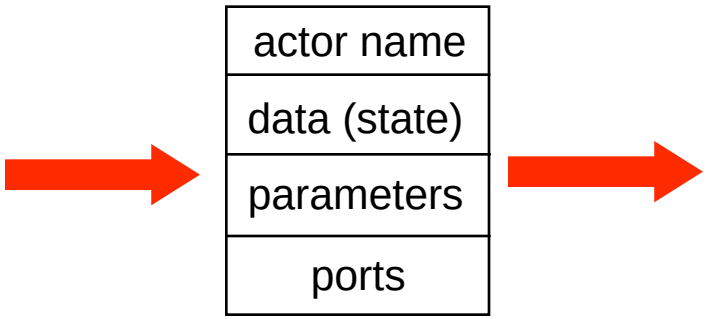


What flows through an object is sequential control

Things happen to objects

The alternative: Actor oriented:

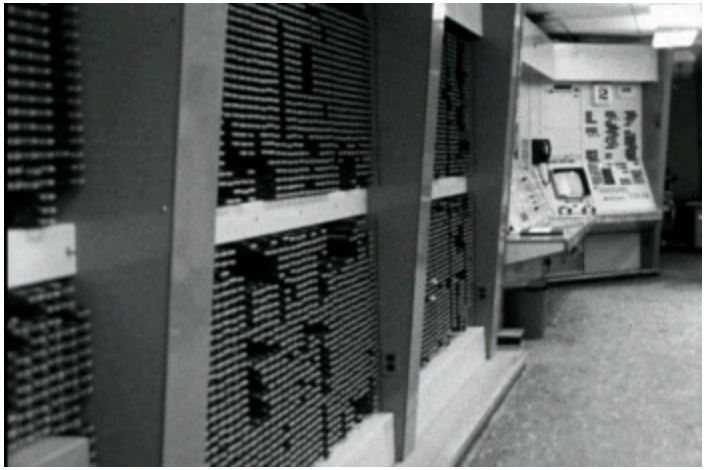
Actors make things happen



What flows through an object is evolving data

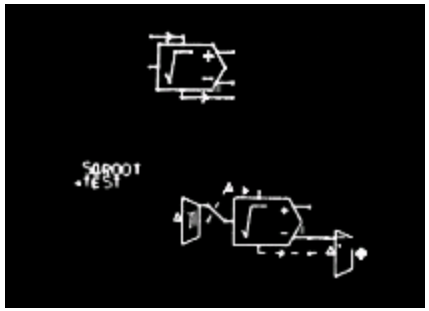
The First (?) Actor-Oriented Programming Language

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

- 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.

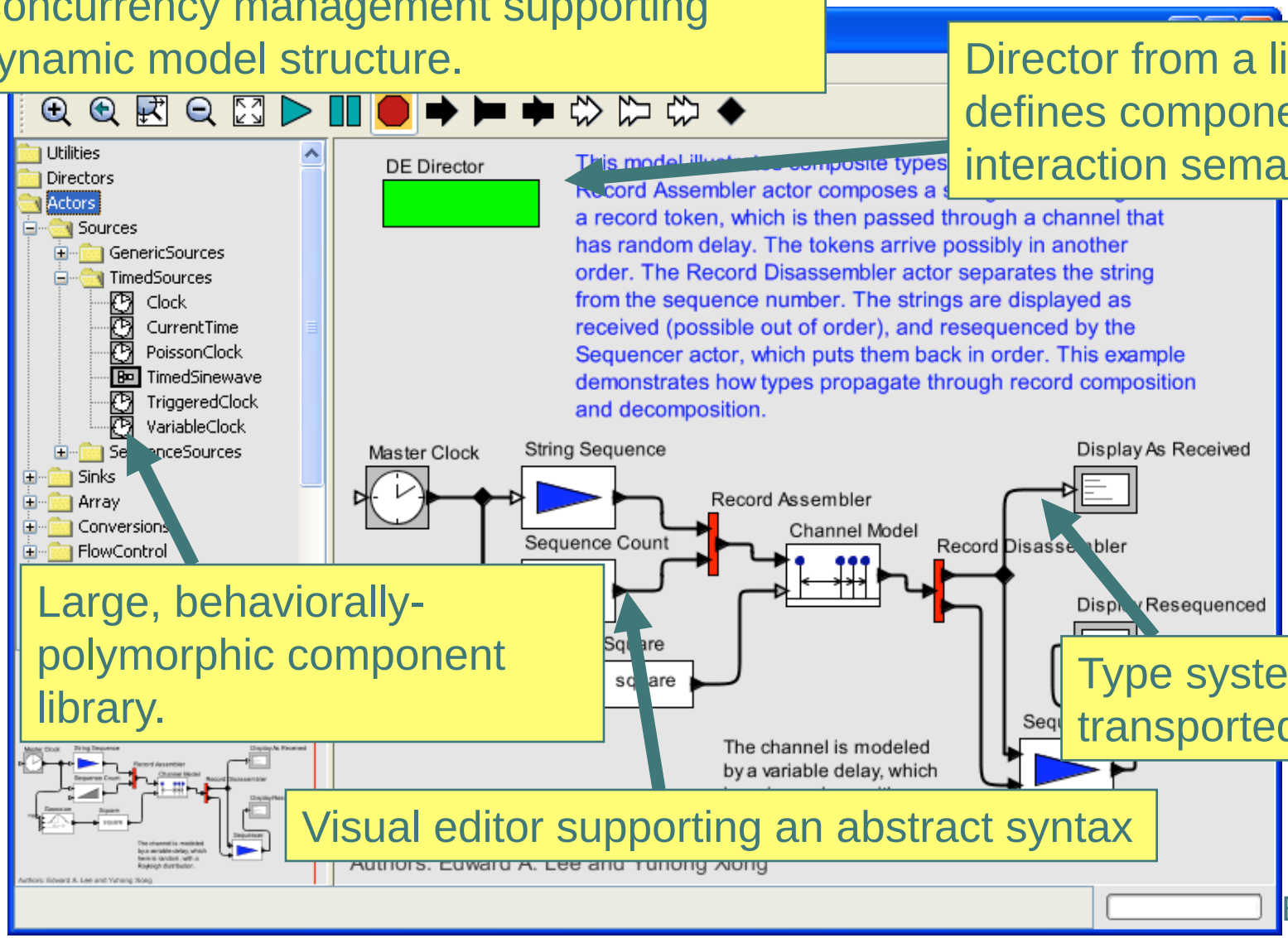
Give a *Component Technology* rather than New Languages

- 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

Concurrency management supporting dynamic model structure.

Director from a library defines component interaction semantics

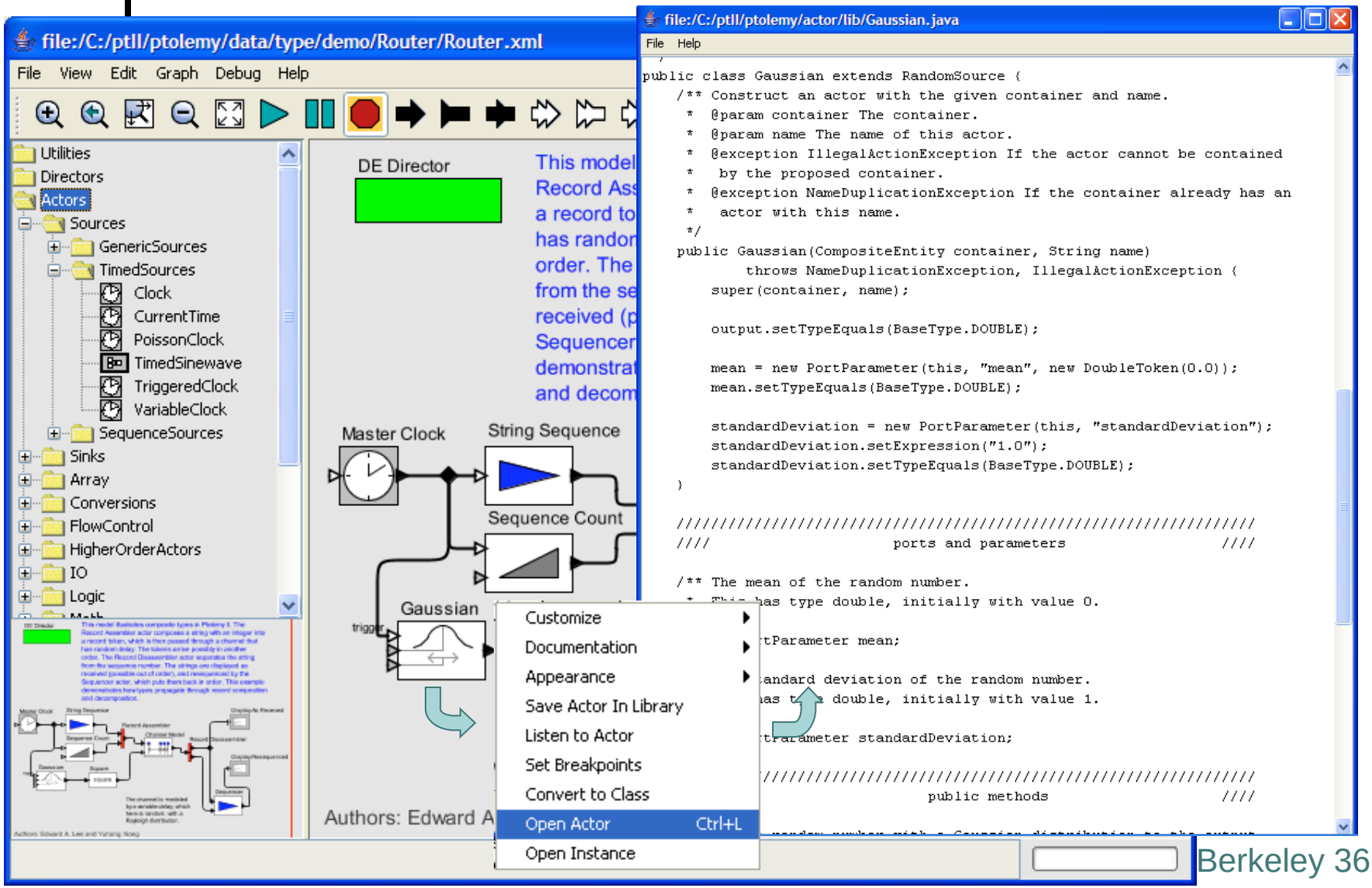


Large, behaviorally-polymorphic component library.

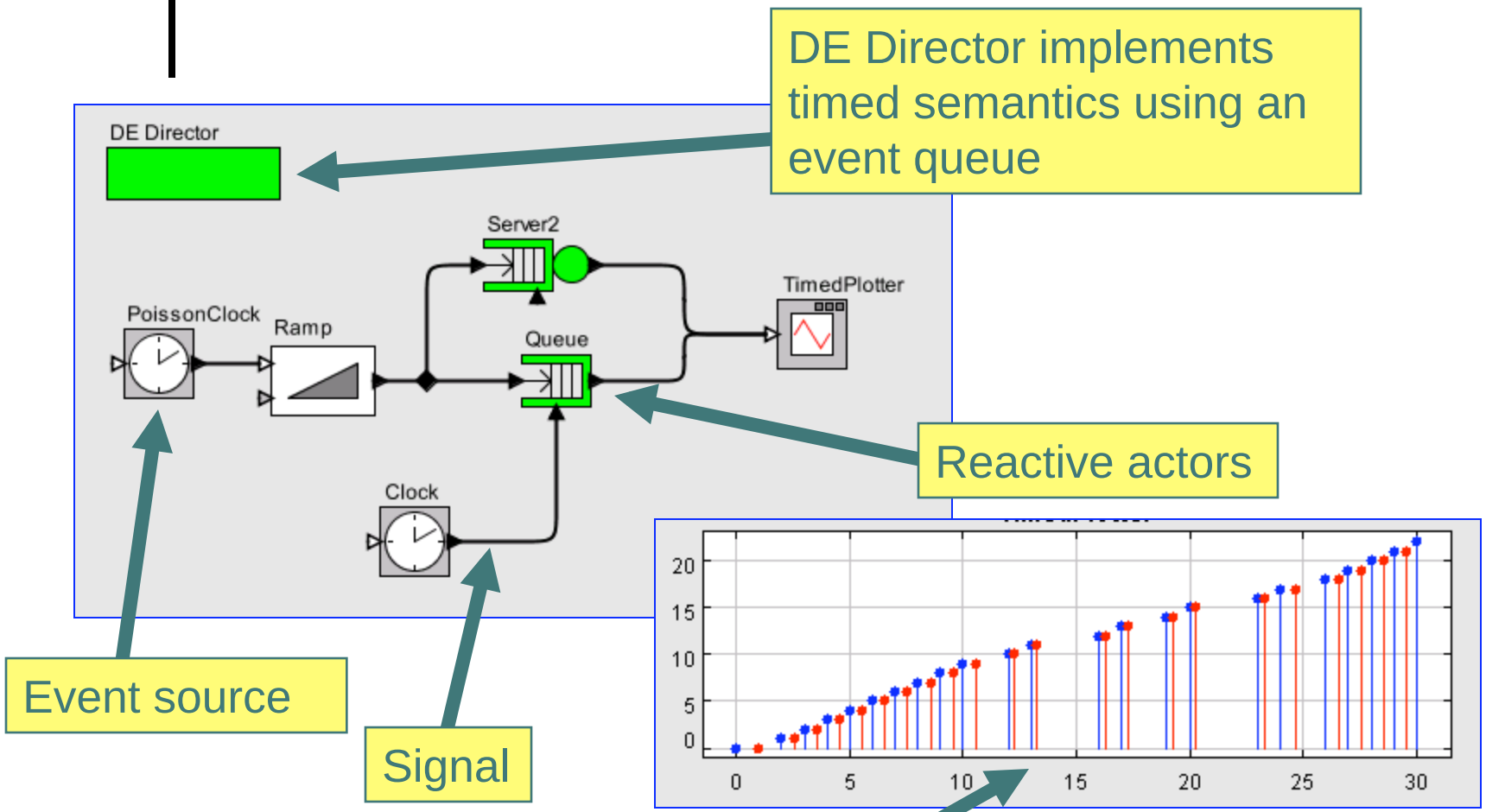
Type system for transported data

Visual editor supporting an abstract syntax

Approach: Concurrent Composition of Components designed with Conventional Languages



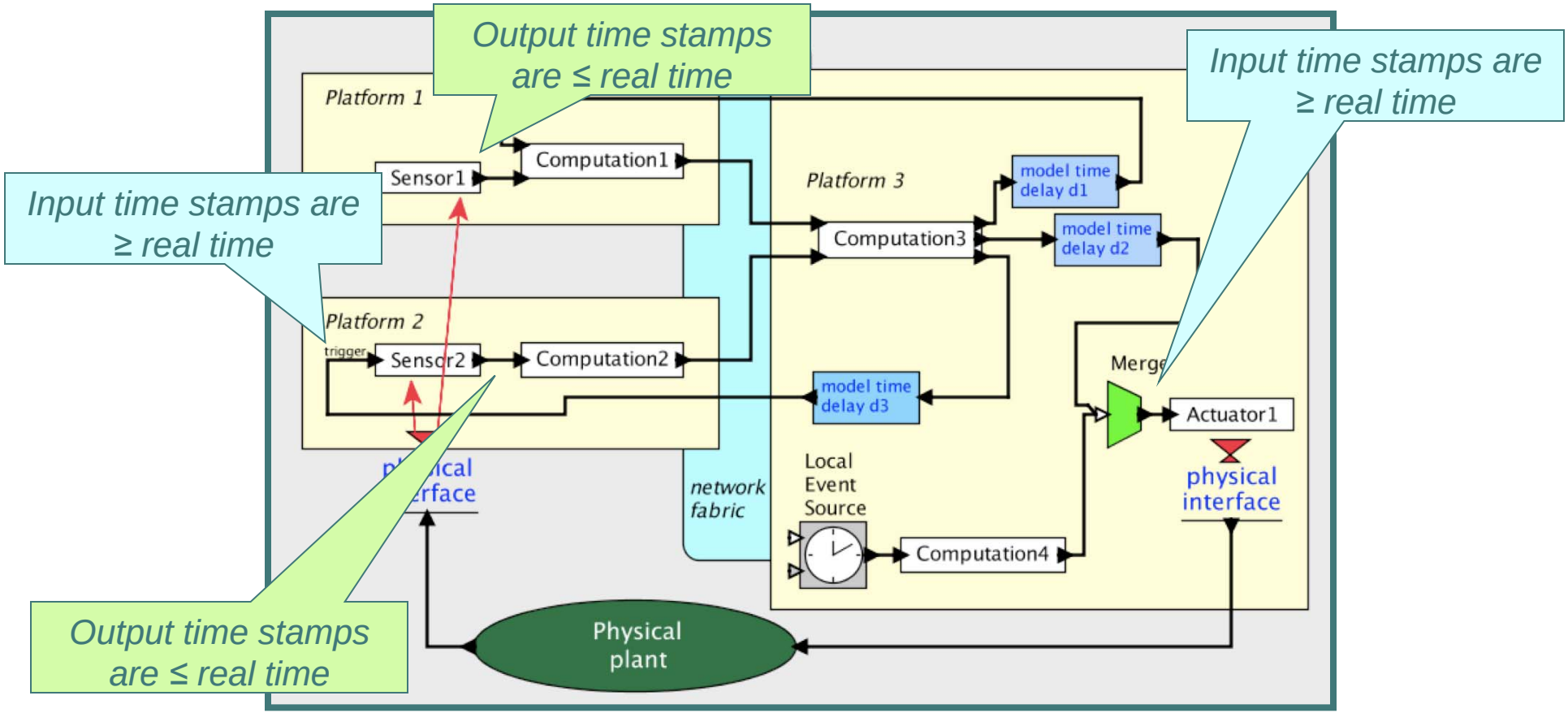
Example: Discrete Event Models



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

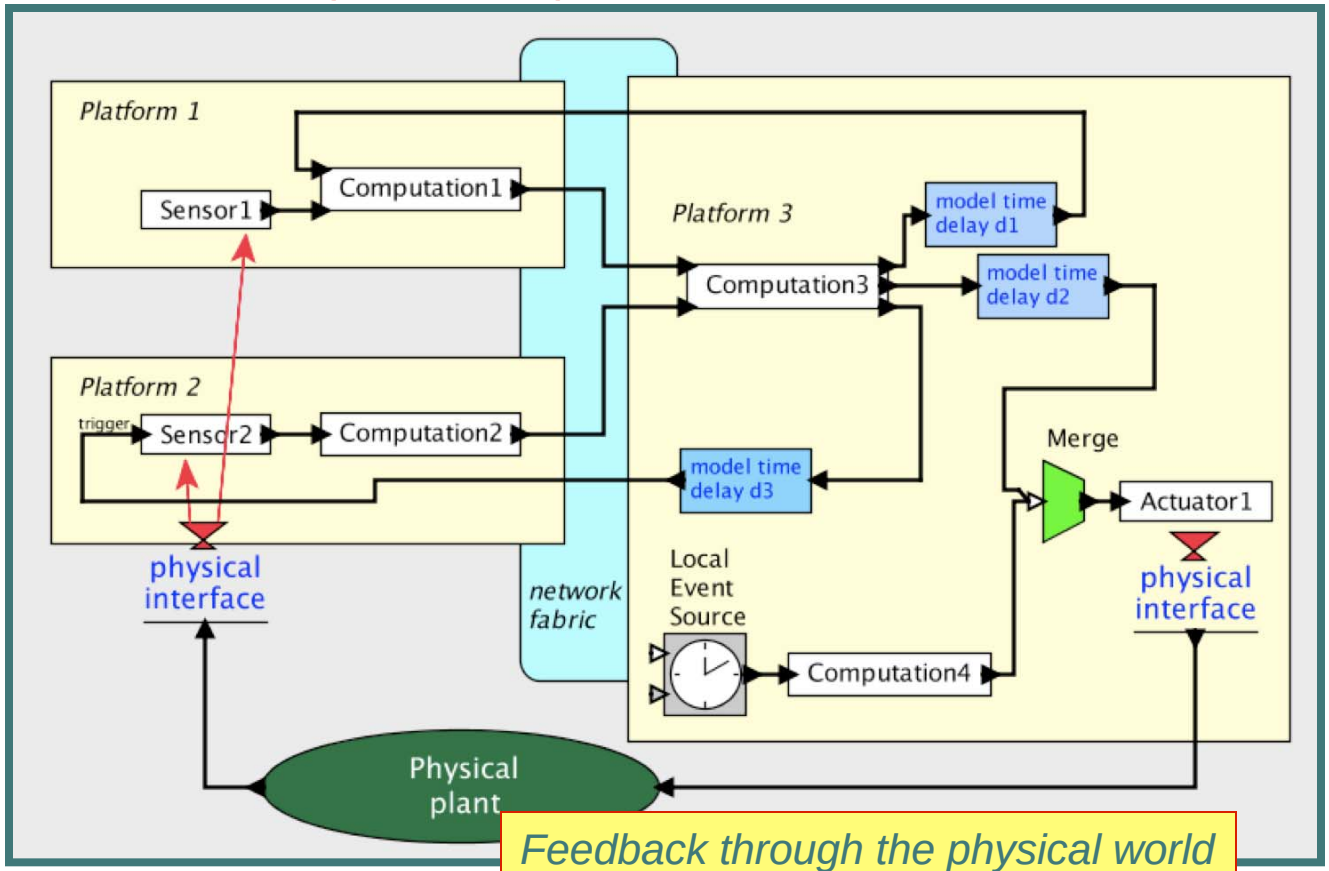
PTIDES: Programming Temporally Integrated Distributed Embedded Systems

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

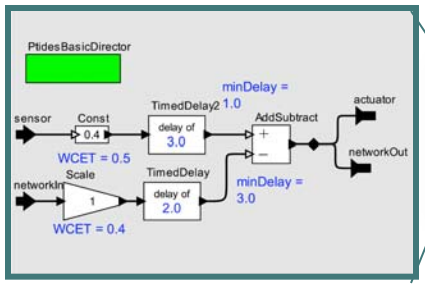


PTIDES: Programming Temporally Integrated Distributed Embedded Systems

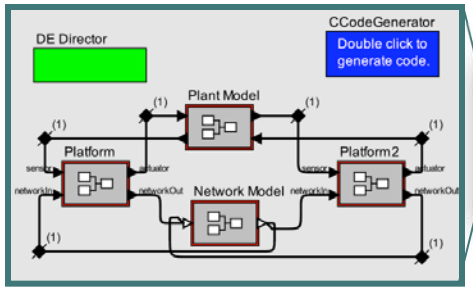
… and being explicit about time delays means that we can analyze control system dynamics…



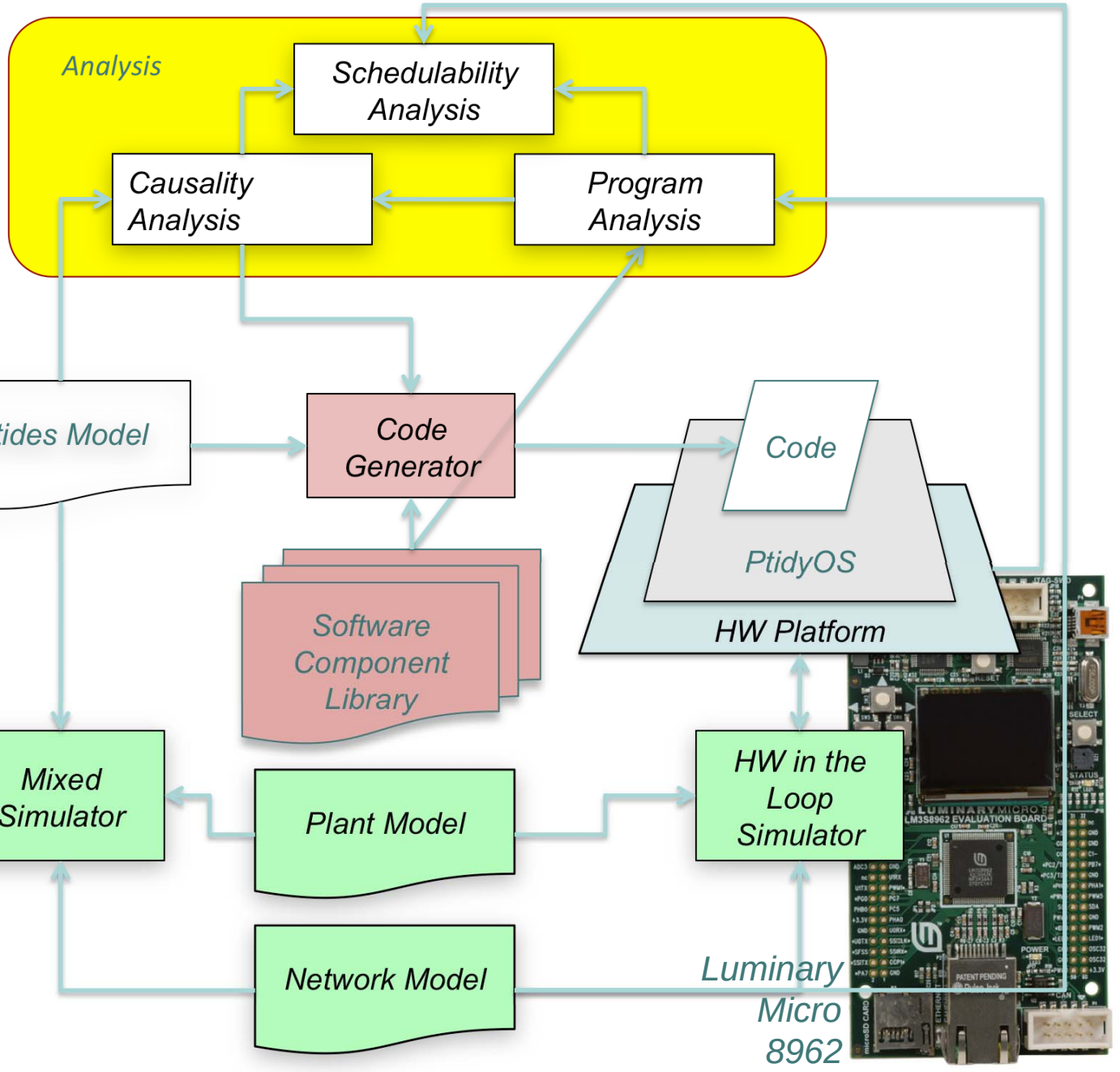
Experimental Setup



Ptolemy II Ptides domain



Ptolemy II Discrete-event, Continuous, and Wireless domains



The Berkeley Solution

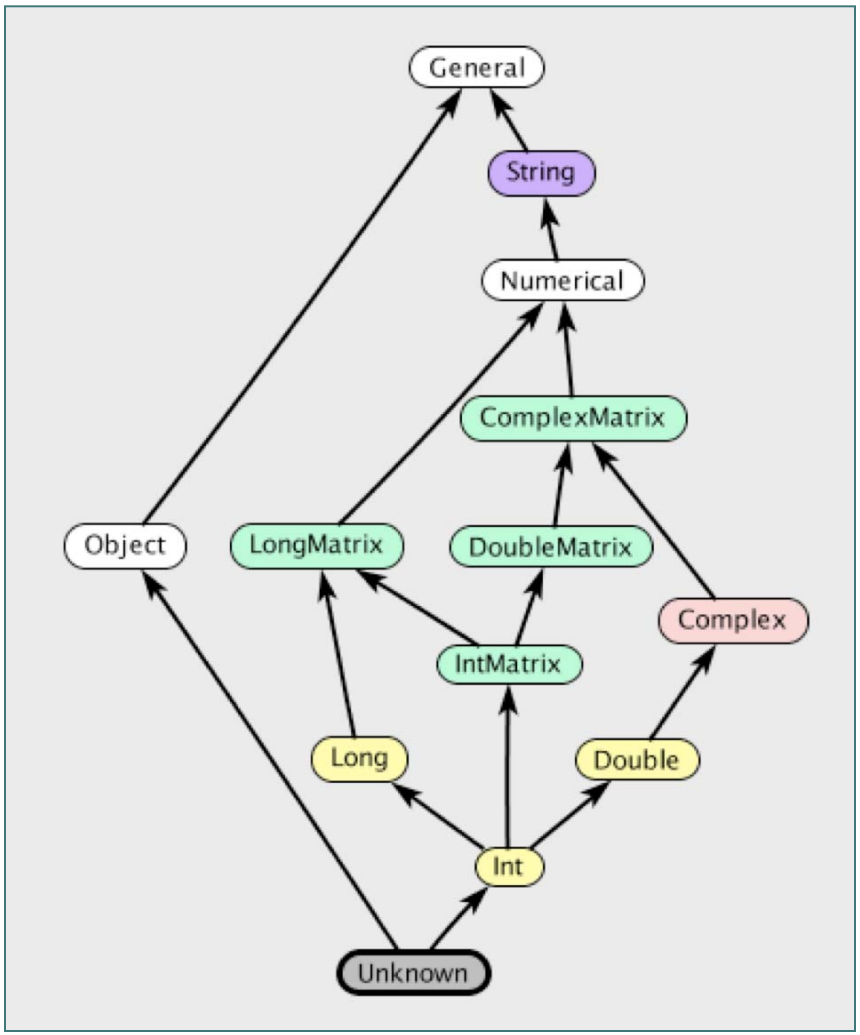
Time and concurrency in the core abstractions:

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

Model Engineering Topics

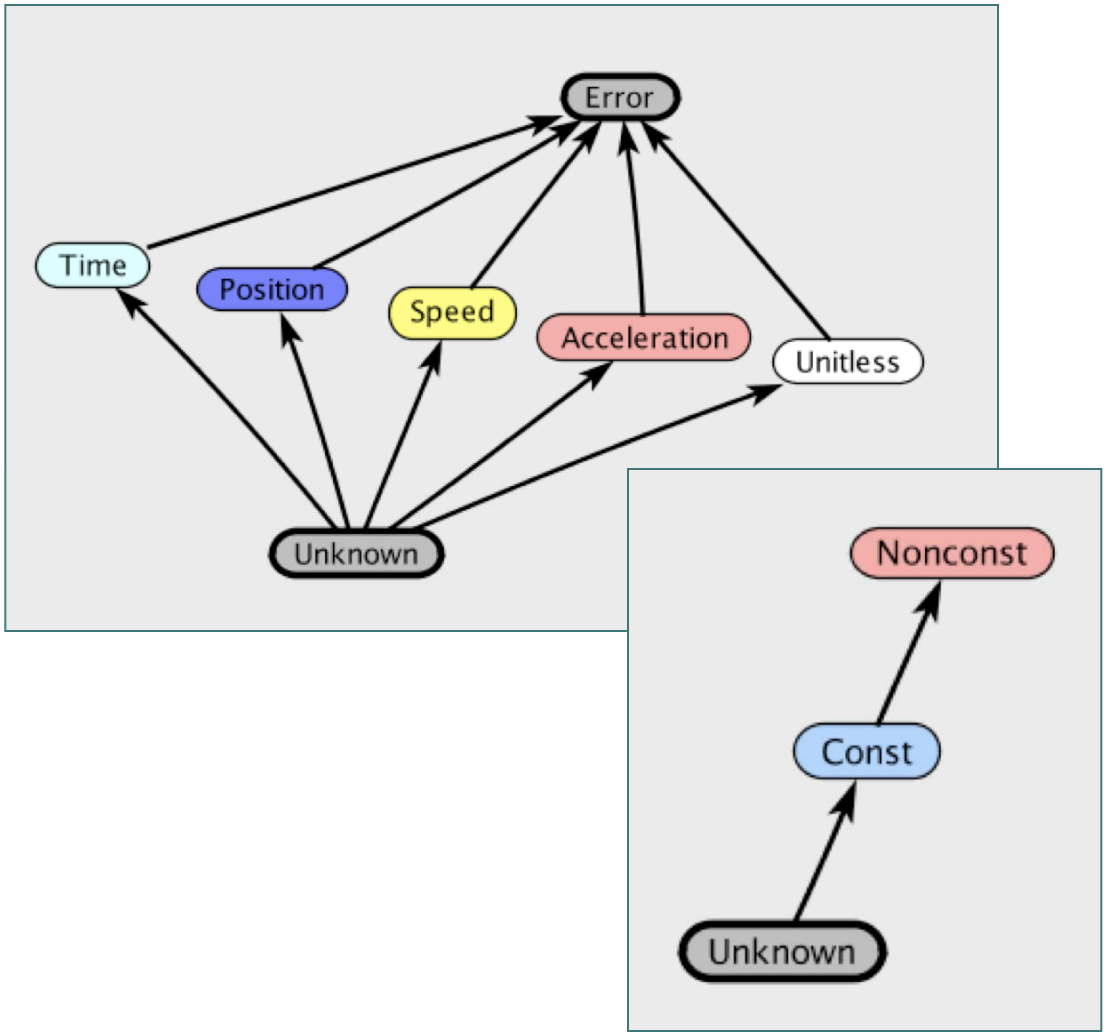
- Model transformation
  - Model optimization
  - Scalable model construction (big models from small descriptions)
  - Product families (multiple products from one model)
  - Design refactoring
  - Workflow automation
- Model ontologies
  - Property annotations and inference
  - Sound foundation based on type theories
  - Scalable to large models
- Multimodeling
  - Hierarchical multimodeling
  - Multi-view modeling
  - Meta modeling

Model Ontologies:
Built on Hindley-Milner Type Theories



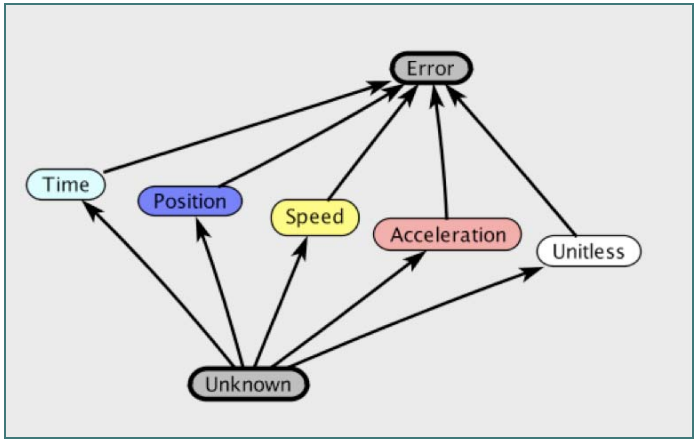
- A *lattice* is a partially ordered set (poset) where every subset has a least upper bound (LUB) and a greatest lower bound (GLB).
- Modern type systems (including the Ptolemy II type system, created by Yuhong Xiong) are based on efficient algorithms for solving inequality constraints on lattices.

Property Lattices capture domain-specific semantic information

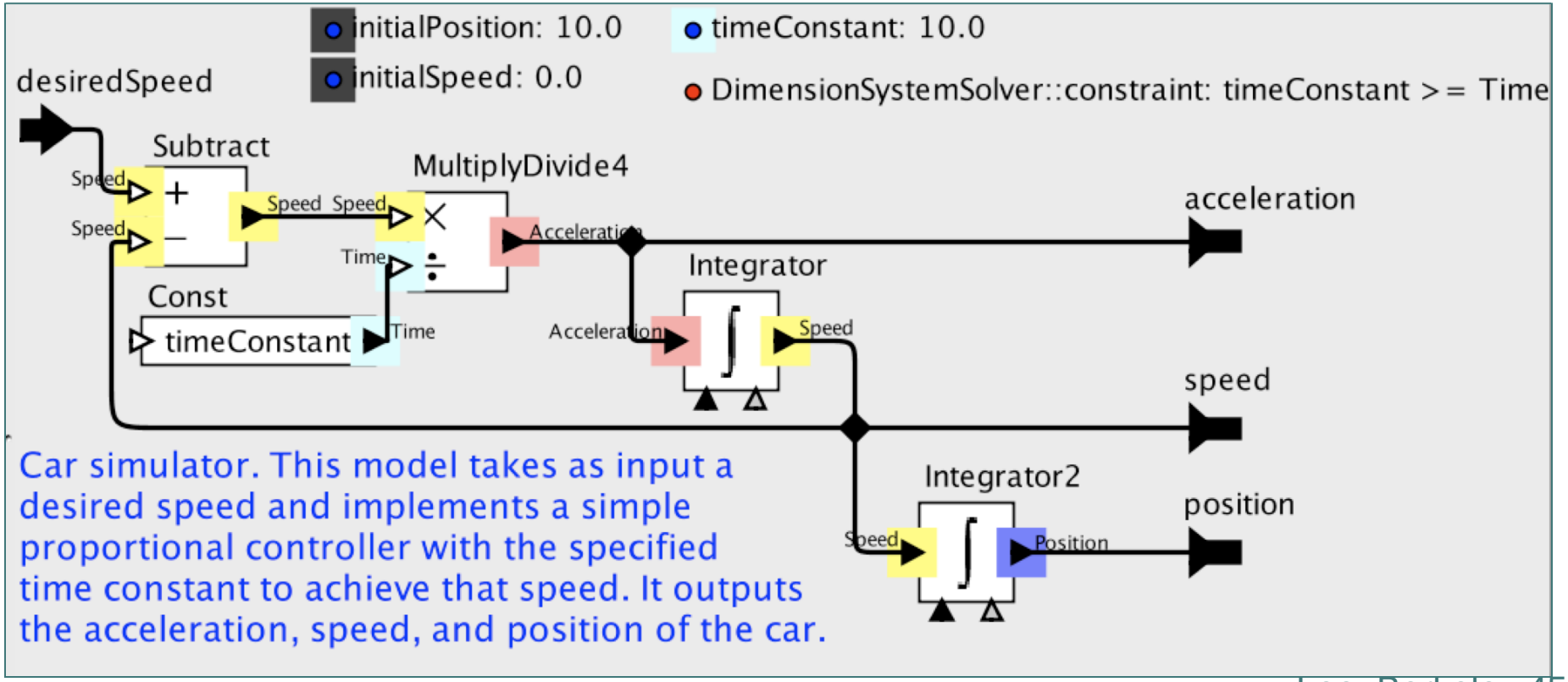


- Components in a model (e.g. parameters, ports) can have properties drawn from a lattice.
- Components in a model (e.g. actors) can impose constraints on property relationships.
- The type system infrastructure can infer properties and detect errors.

Property Systems



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



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