Embedded Tutorial
CPS Foundations

Edward A. Lee

Robert S. Pepper Distinguished Professor
UC Berkeley

Special Session: Cyber-Physical Systems Demystified
Design Automation Conference (DAC 2010)

Annaheim, CA, Thursday, June 17, 2010

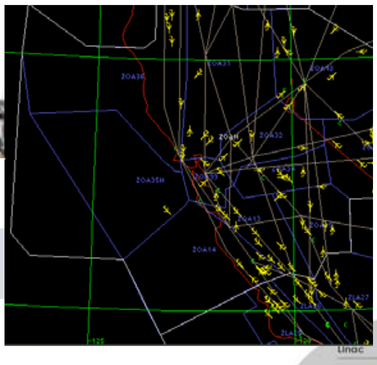
Abstract

This talk argues that cyber-physical systems present a substantial intellectual challenge that requires changes in both theories of computation and dynamical systems theory. The CPS problem is not the *union* of cyber and physical problems, but rather their *intersection*, and as such it demands models that embrace both. Two complementary approaches are identified: *cyberizing the physical* (CtP) means to endow physical subsystems with cyber-like abstractions and interfaces; and *physicalizing the cyber* (PtC) means to endow software and network components with abstractions and interfaces that represent their dynamics in time.

Cyber-Physical Systems (CPS):

Orchestrating networked computational resources with physical systems

Avionics

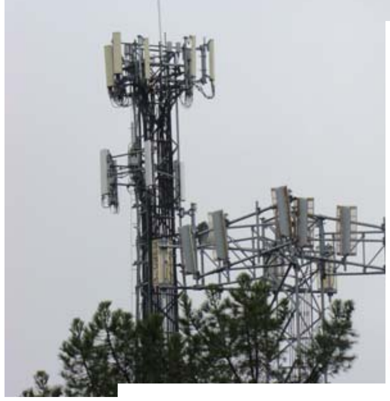


Transportation (Air traffic control at SFO)

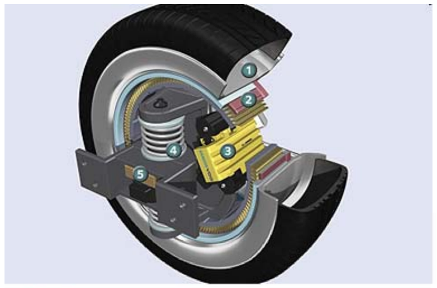
Building Systems

Elevator
Switch
Server room
Fire and security monitoring
Electricity

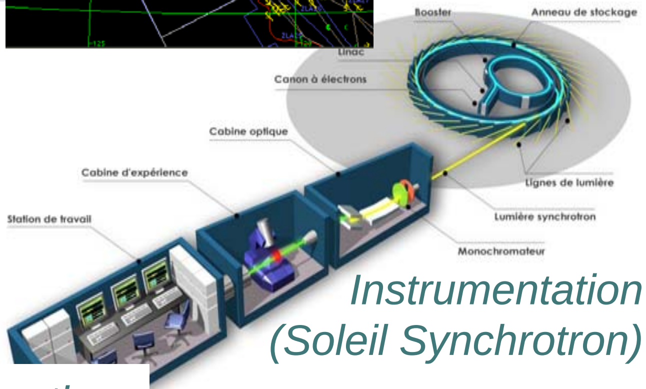
Telecommunications



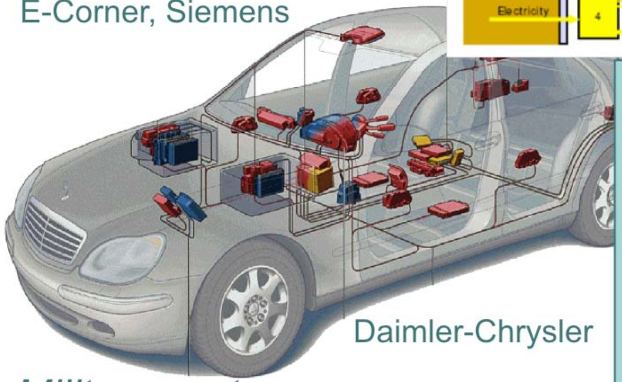
Automotive



E-Corner, Siemens



Instrumentation (Soleil Synchrotron)



Daimler-Chrysler

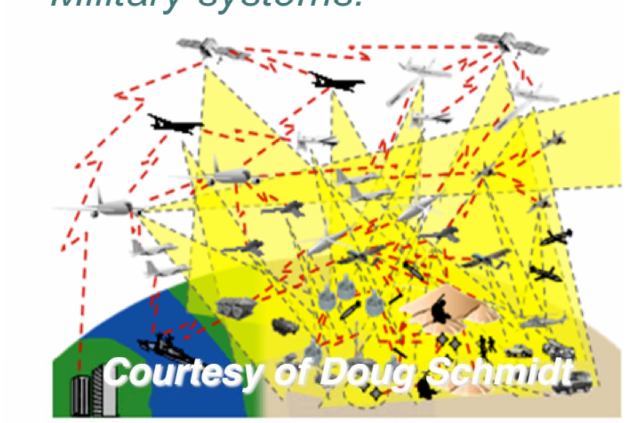
Power generation and distribution

Courtesy of General Electric

Factory automation

Courtesy of Kuka Robotics Corp.

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 software problem:

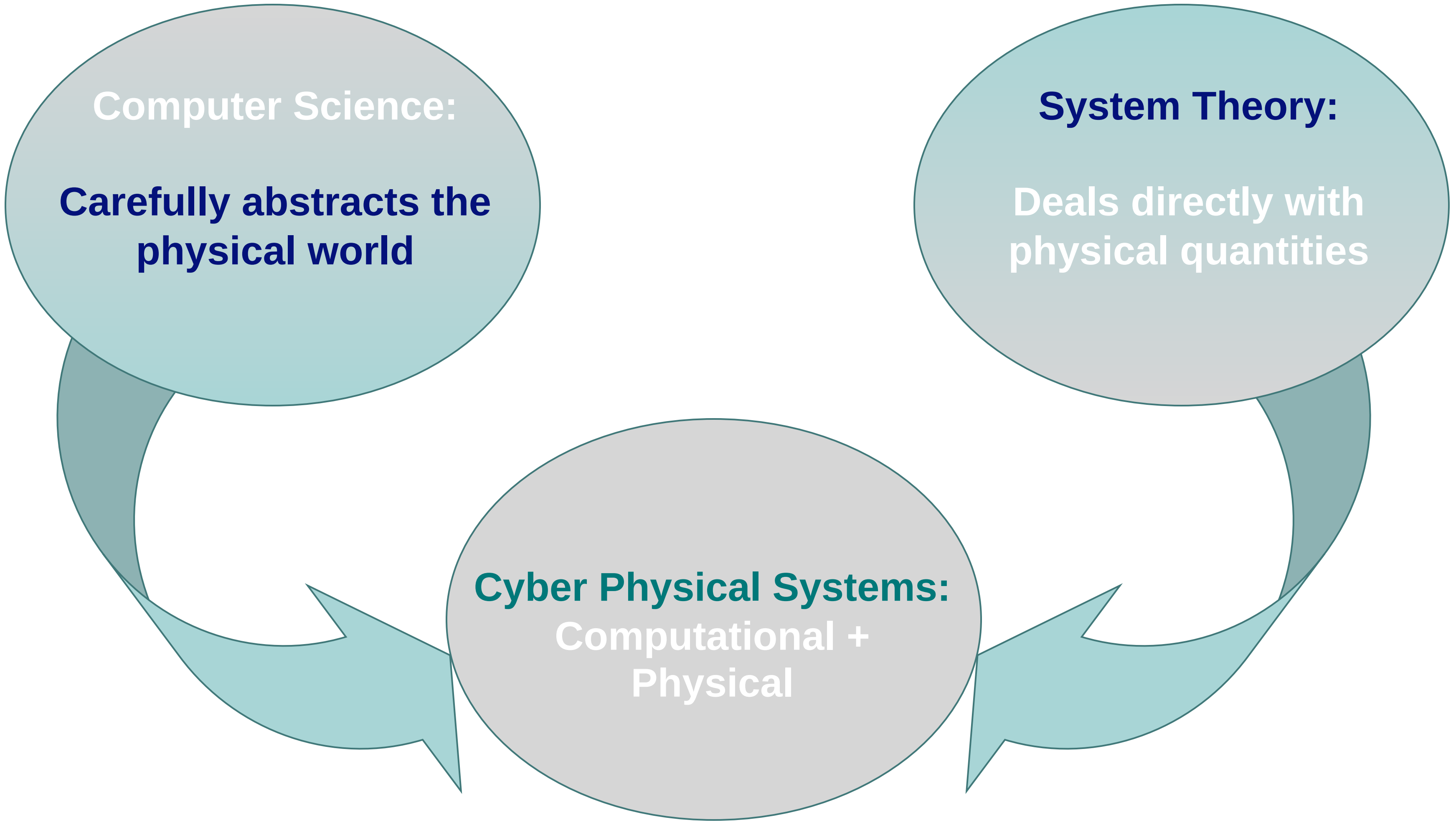
- *The traditional embedded software problem:*

  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.

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 (***cyberizing the physical***), and theories of computation must be viewed as alternative ways of talking about dynamics (***physicalizing the cyber***)

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.

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)
- …

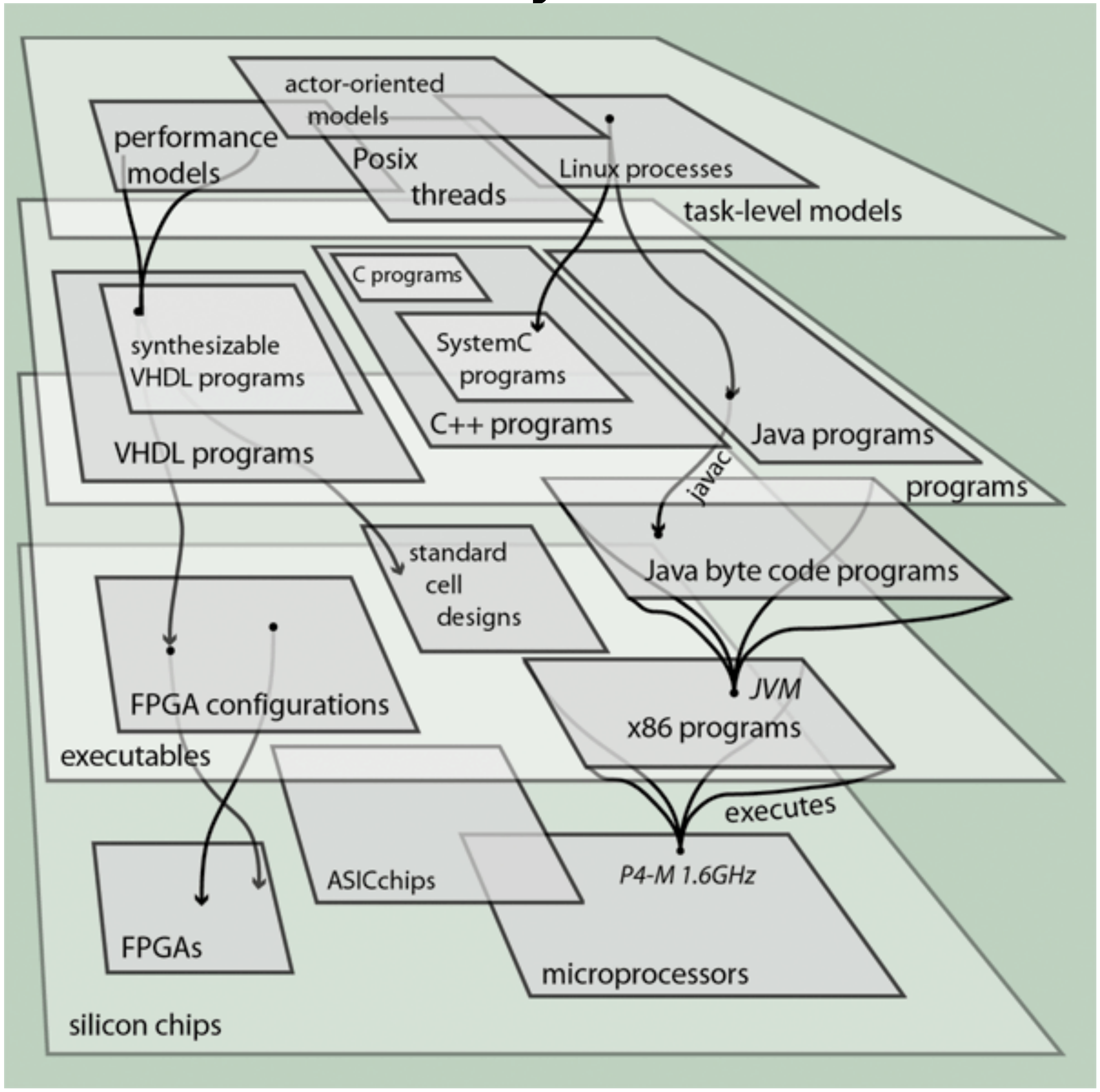
A Story

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

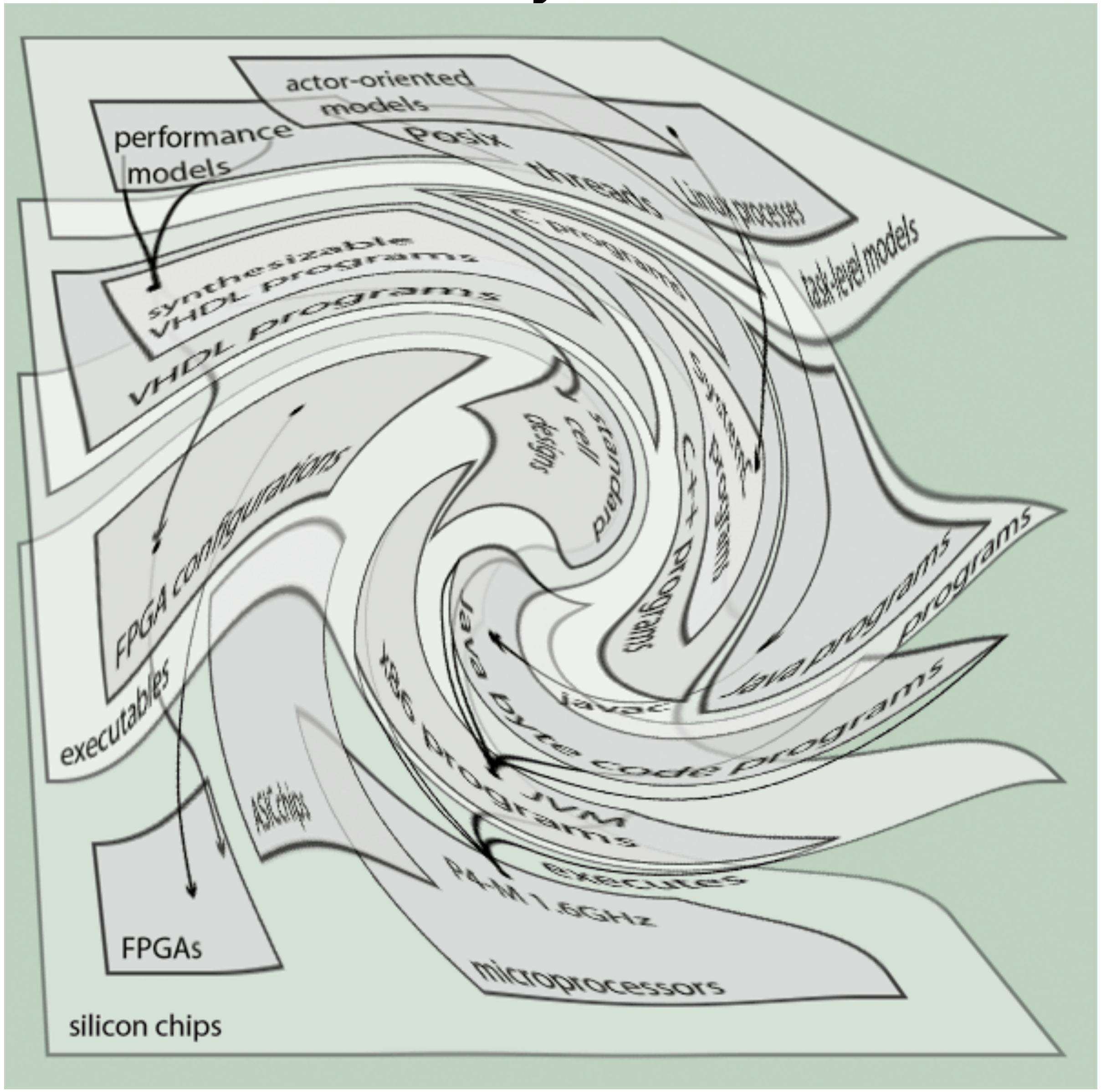
- **Stockpiling for a product run**
  - Some systems vendors have to purchase up front the entire expected part requirements for an entire product run.
- **Frozen designs**
  - Once certified, errors cannot be fixed and improvements cannot be made.
- **Product families**
  - Difficult to maintain and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



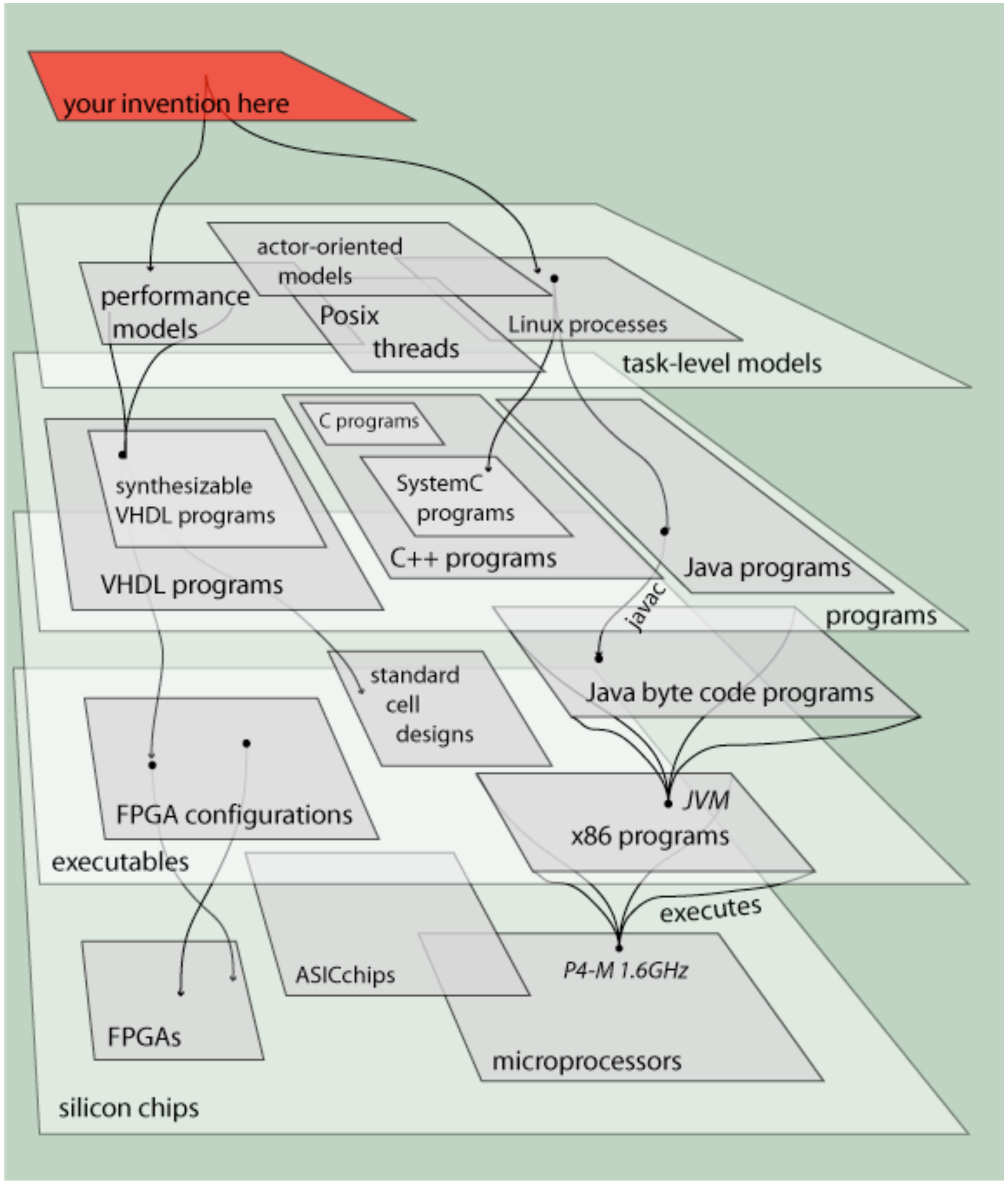
The purpose for an abstraction is to hide details of the implementation below and provide a platform for design from above.

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?

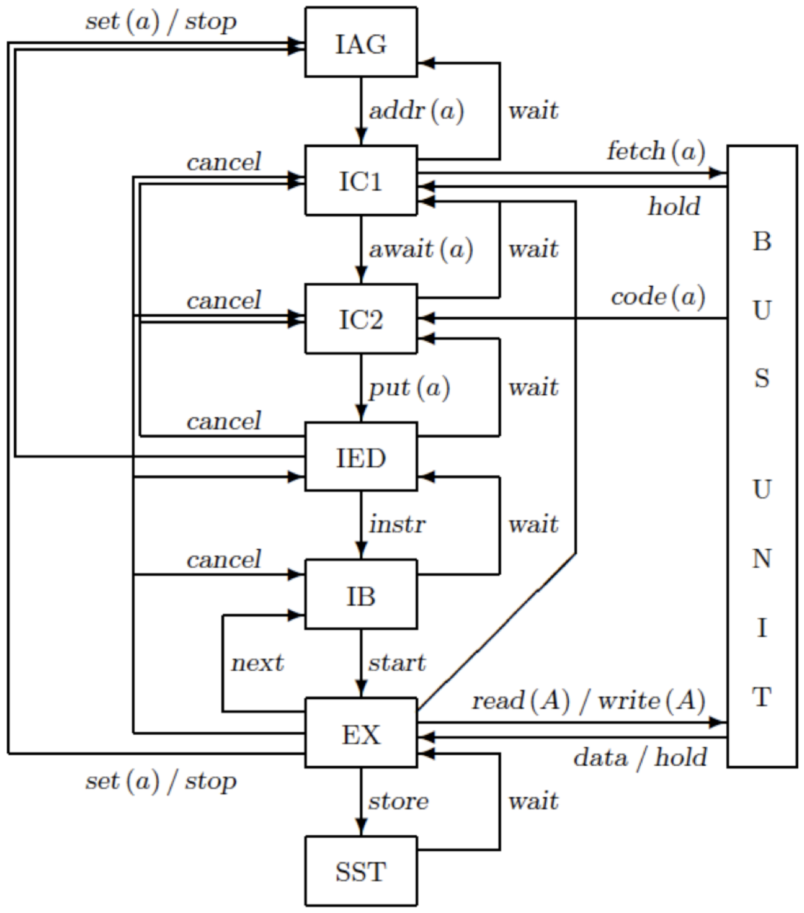
But these higher abstractions rely on an increasingly problematic fiction: WCET

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

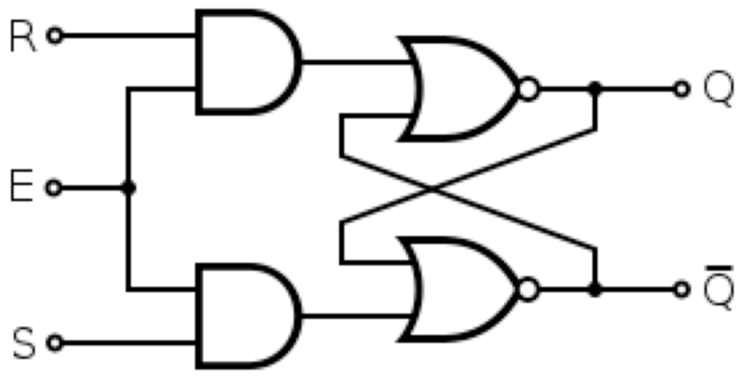
And the result is valid only for that exact Hardware and software!

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.

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

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

My Claim

Nontrivial software written with threads is incomprehensible to humans, and it cannot deliver repeatable or predictable behavior, 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!

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.

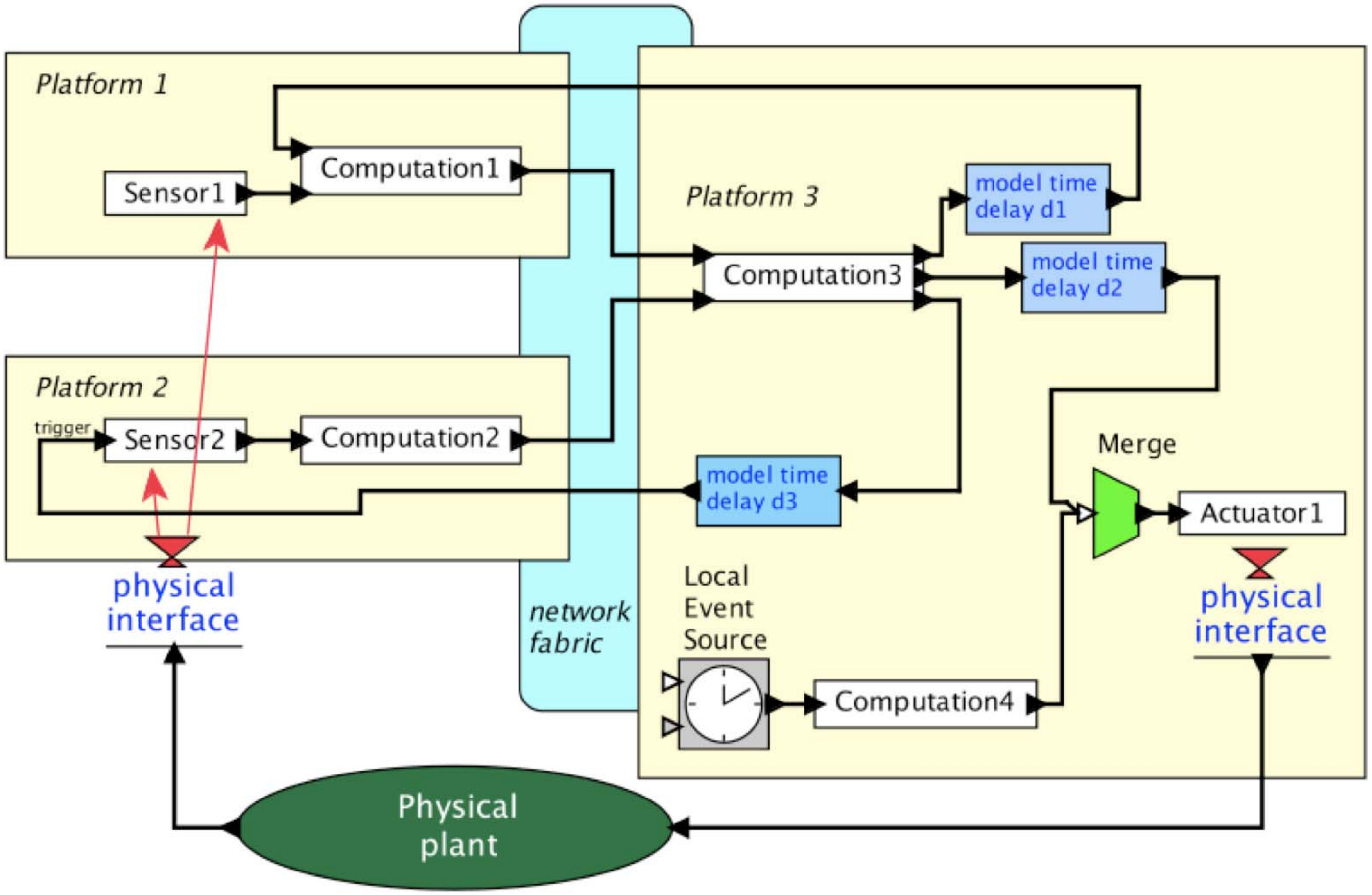
The Current State of Affairs

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!

Approaching the CPS Challenge

Physicalizing the cyber (PtC): to endow software and network components with abstractions and interfaces that represent their dynamics in time.



Cyberizing the Physical (CtP): to endow physical subsystems with cyber-like abstractions and interfaces

Projects at Berkeley focused on Physicalizing the Cyber

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.

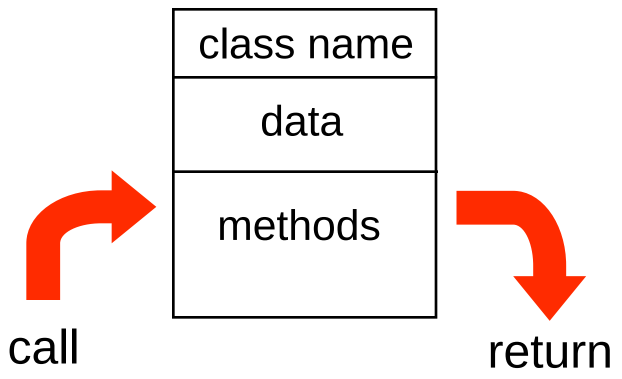
Projects at Berkeley focused on Physicalizing the Cyber

Time and concurrency in the core abstractions:

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

Rethinking Software Components: Object Oriented vs. Actor Oriented

The established: Object-oriented:

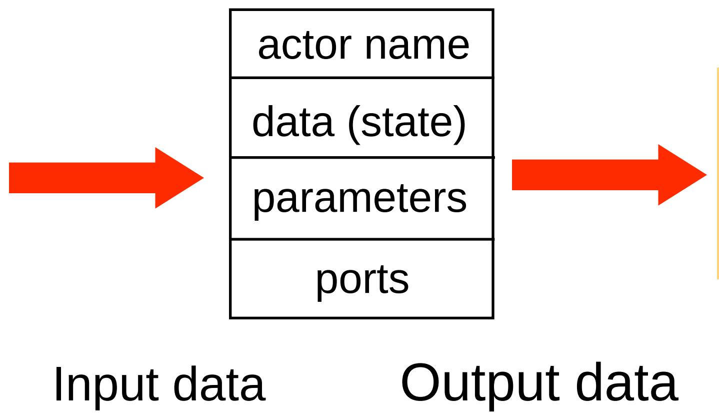


What flows through an object is sequential control

Things happen to objects

The alternative: Actor oriented:

Actors make things happen



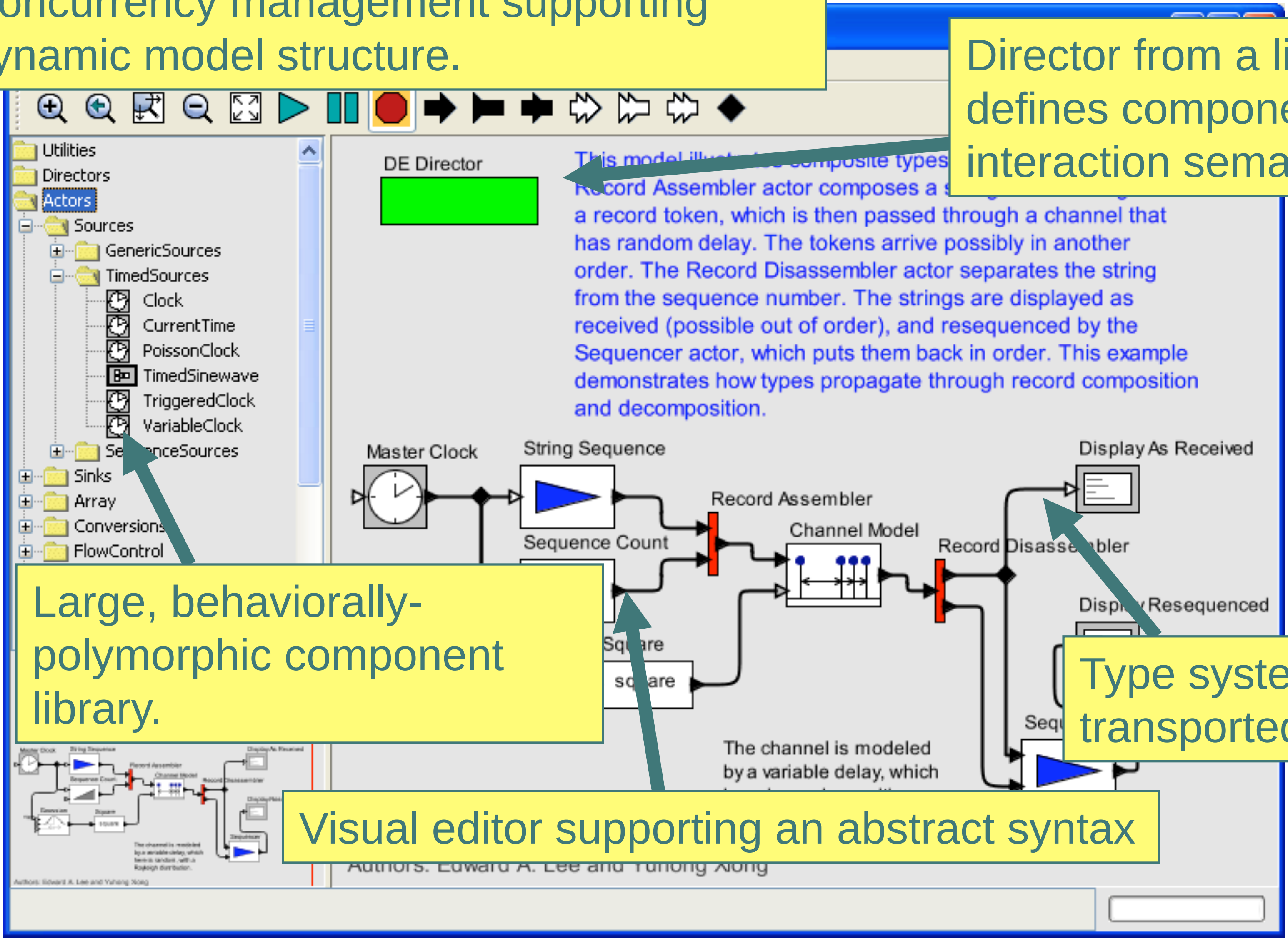
What flows through an object is evolving data

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


This model illustrates composite types. The Record Assembler actor composes a ... 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.

The channel is modeled by a variable delay, which ...

Authors: Edward A. Lee and Yuhong Xiong

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

Conclusion

The core intellectual challenge in CPS is in developing abstractions that embrace computation, networking, and physical dynamics in a coherent way.