Temporal Semantics in Concurrent and Distributed Software

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

Workshop on
Strategic Directions in Software at Scale (S@S)

Berkeley, CA, August 18-19, 2010

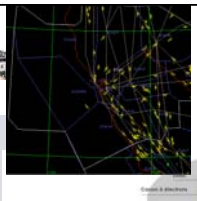
Abstract (from Workshop Goals)

Cyber-physical systems integrate computing and networking with physical processes. The temporal dynamics of software and networks becomes critical to predicting and controlling the interactions of system components. But software abstractions omit time. The theme of this discussion is to investigate the potential impact and technical implications of modifying these abstractions to embrace temporal dynamics.

2

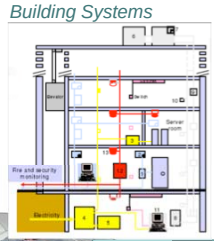
Cyber-Physical Systems (CPS):
Orchestrating networked computational resources with physical systems

Avionics



Transportation (Air traffic control at SFO)

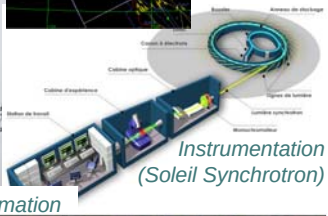
Building Systems



Telecommunications

Automotive

E-Corner, Siemens

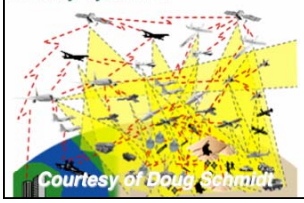


Instrumentation (Soleil Synchrotron)

Daimler-Chrysler

Power generation and distribution

Military systems:



Courtesy of Doug Schmidt

Courtesy of General Electric

Factory automation

Courtesy of Kuka Robotics Corp. 3

Where CPS Differs from
the traditional embedded software problem:

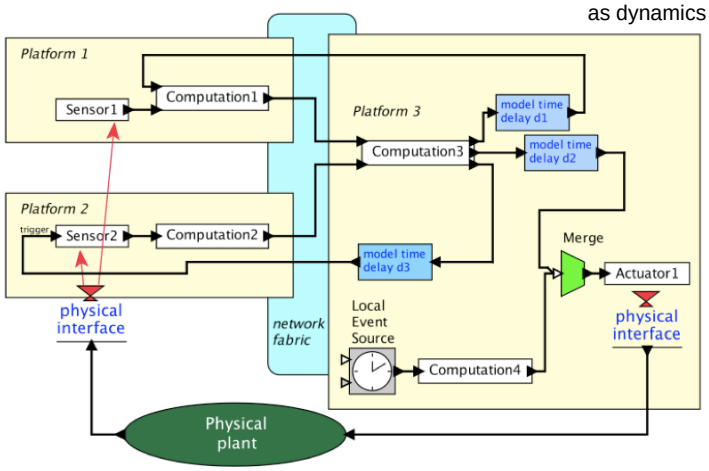
- *The traditional embedded software problem:*
  Embedded software is software on small computers. The technical problem is one of optimization (coping with limited resources).

- *The CPS problem:*
  Computation and networking integrated with physical processes. The technical problem is managing dynamics, time, and concurrency in networked computational + physical systems.

4

Approaching the CPS Challenge

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



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

5

A "Success" Story

The Boeing 777 was Boeing's first fly-by-wire aircraft. It is deployed, appears to be reliable, and is succeeding in the marketplace. Therefore, it must be a success. However…

6

A "Success" 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 behavior and require the software to be re-certified.

7

A "Success" Story

Apparently, the software does not specify the behavior that was certified.

Fly-by-wire is about controlling the dynamics of a physical system. It is not about the transformation of data, which is what Turing/Church "computation" does.

8

A Key Challenge: Timing is not Part of Software Semantics

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.

9

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)
- …

i.e., many of the innovations in CS over the last 40 years.

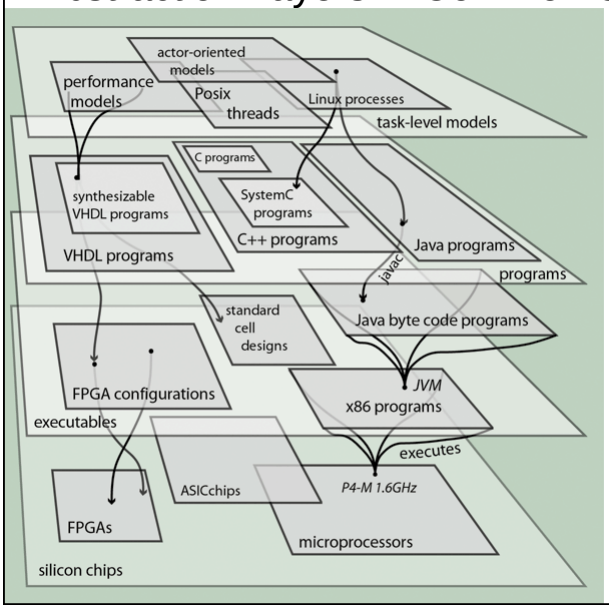
10

The software does not specify the behavior. 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.

11

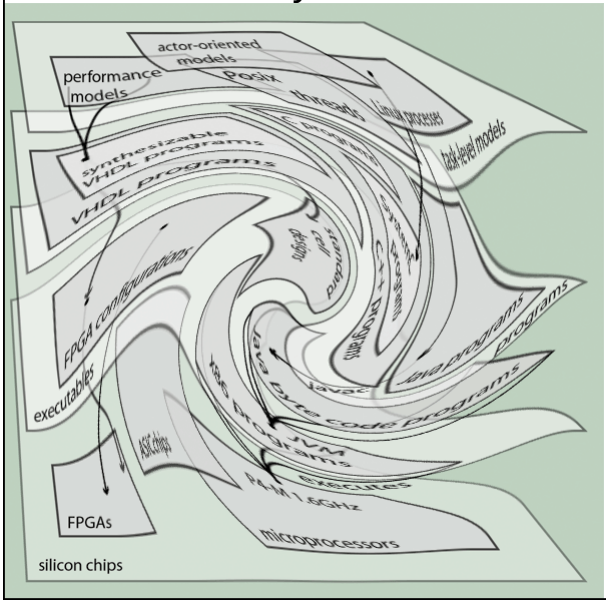
Abstraction Layers in Common Use



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

12

Abstraction Layers in Common Use

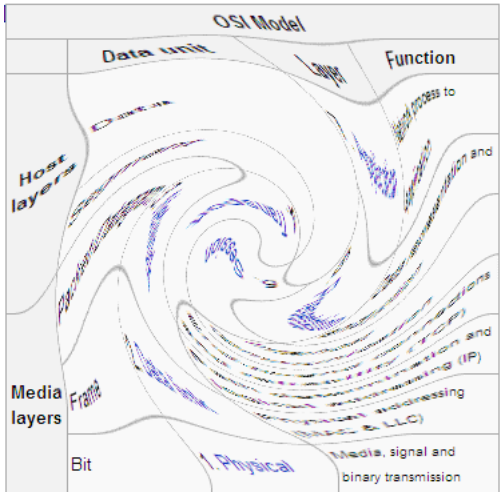


Every abstraction layer has failed in the fly-by-wire scenario.

The design *is* the implementation.

13

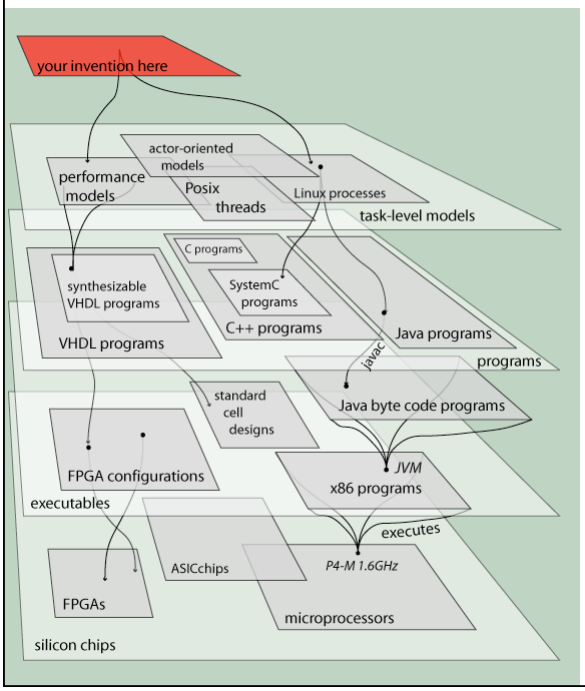
The Same Problem Arises in Networking



The point of these abstraction layers is to isolate a system designer from the details of the implementation below, and to provide an abstraction for other system designers to build on.

In today's networks, timing is a property that emerges from the details of the implementation, and is not included in the abstractions. *Timing is a performance metric, not a correctness criterion.*

14



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

15

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

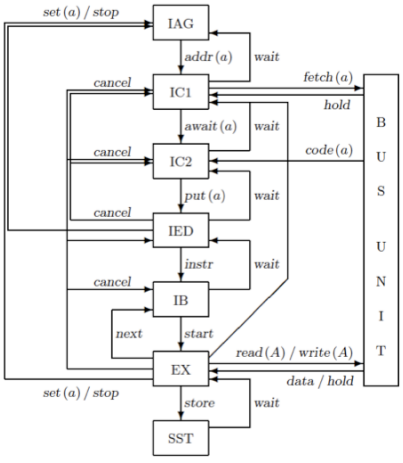
Example war story:

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!

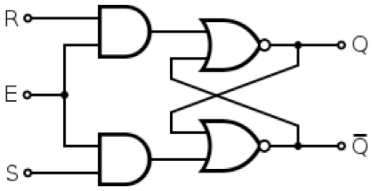
Fundamentally, the ISA of the processor has failed to provide an adequate abstraction.



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

16

The Key Problem



Electronics technology delivers highly reliable and precise timing…

20.000 MHz (± 100 ppm)

… and the overlaying software abstractions discard it.

17

What to do about it?

A Bottom Up Approach:
Make Timing a Semantic Property of Computers

Precision-Timed (PRET) Machines

Just as we expect reliable logic operations, we should expect repeatable timing.

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.

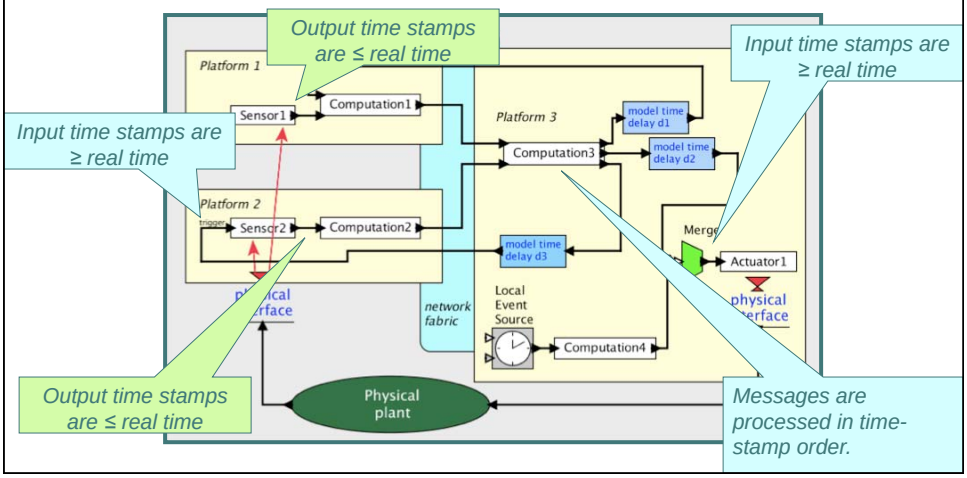
18

What to do about it?

A Top Down Approach:

Make Timing a Semantic Property of Software Components

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



PTIDES Relies on Network Time Synchronization with an Error that can be Bounded

Press Release October 1, 2007

National Semiconductor
The Sight & Sound of Information

NEWS RELEASE

For More Information Contact

Media Contact
Naomi Mitchell
National Semiconductor
(408) 721-2142
naomi.mitchell@nsc.com

Reader Information
Design Support Group
(800) 272-9959
www.national.com

Industry's First Ethernet Transceiver with IEEE 1588 PTP Hardware Support from National Semiconductor Delivers Outstanding Clock Accuracy

Using DP83640, Designers May Choose Any Microcontroller, FPGA or ASIC to Achieve 8- Nanosecond Precision with Maximum System Flexibility

This may become routine!

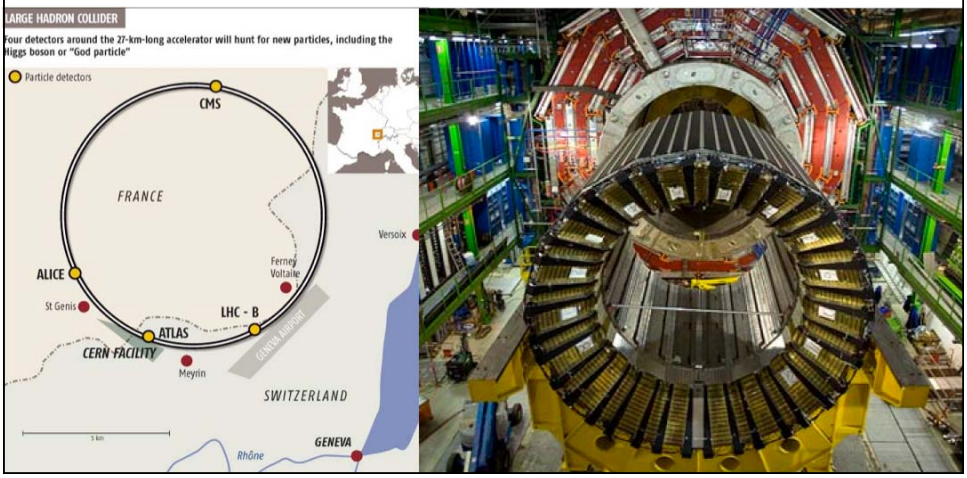
With this PHY, clocks on a LAN agree on the current time of day to within 8ns, far more precise than older techniques like NTP.

A question we are addressing at Berkeley: How does this change how we develop distributed CPS software?

20

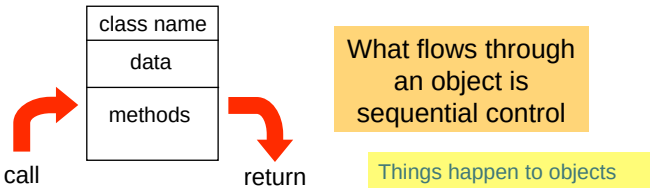
An Extreme Example: The Large Hadron Collider

The WhiteRabbit project at CERN is synchronizing the clocks of computers 10 km apart to within about 80 psec using a combination of IEEE 1588 PTP and synchronous ethernet.



More Generally than PTIDES: *Rethinking Software Components to Admit Time*. Object Oriented vs. Actor Oriented

The established: Object-oriented:

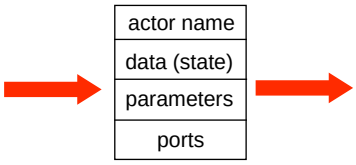


What flows through an object is sequential control

Things happen to objects

The alternative: Actor oriented:

Actors make things happen



What flows through an object is evolving data

Input data Output data

22

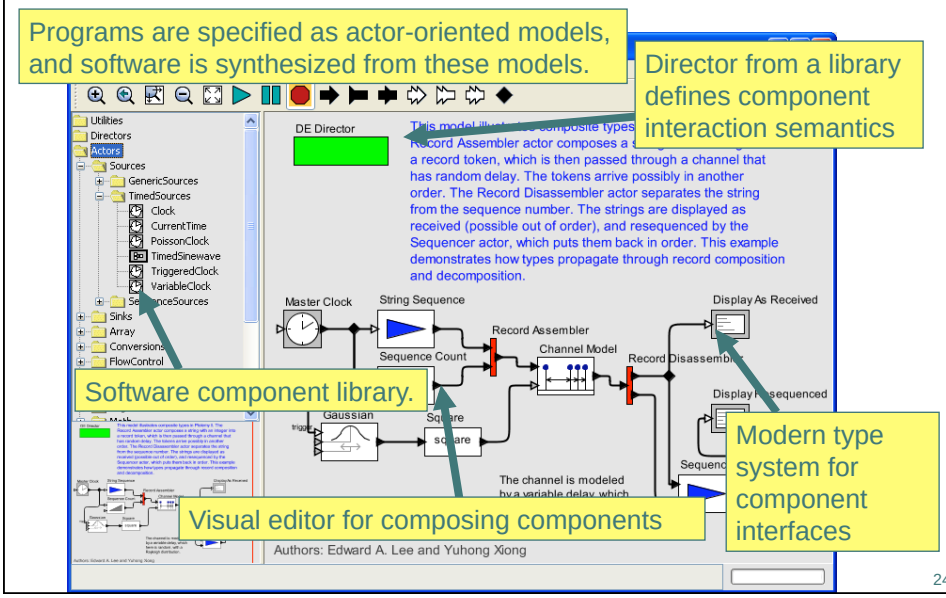
Examples of Actor-Oriented Systems

- 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.

23

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



24

Questions

- What proportion of software problems arise from uncontrolled or unexpected timing of interaction between software components?
- If computation and networking speeds continue to improve, can we just circumvent the timing problem by over provisioning?
- Are there intermediate solutions that do not require redoing or modifying so much of what computer science has done for the last 40 years?

25