Embedded Software: Building the Foundations

Edward A. Lee

Professor, Chair of EE, and Associate Chair of EECS

CHESS: Center for Hybrid and Embedded Software Systems

UC Berkeley

BEARS Conference

Berkeley EECS Annual Research Symposium

February 10, 2005

Berkeley, CA

Abstract

Embedded software has traditionally been thought of as "software on small computers." In this traditional view, the principal problem is resource limitations (small memory, small data word sizes, and relatively slow clocks). Solutions emphasize efficiency; software is written at a very low level (in assembly code or C), operating systems with a rich suite of services are avoided, and specialized computer architectures such as programmable DSPs and network processors are developed to provide hardware support for common operations. These solutions have defined the practice of embedded software design and development for the last 25 years or so. However, thanks to the semiconductor industry's ability to follow Moore's law, the resource limitations of 25 years ago should have almost entirely evaporated today. Why then has embedded software design and development changed so little? It may be that extreme competitive pressure in products based on embedded software, such as consumer electronics, rewards only the most efficient solutions. This argument is questionable, however, since there are many examples where functionality has proven more important than efficiency. In this talk, we argue that resource limitations are not the only defining factor for embedded software, and may not even be the principal factor. Instead, the dominant factors are much higher reliability requirements than for desktop software, greater concurrency, and tighter timing requirements. These differences drive the technology towards different techniques than those that have been applied in conventional computer software. In this talk, we explore those techniques and map out a research agenda for embedded software.

Are Resource Limitations the Key Defining Factor for Embedded Software?

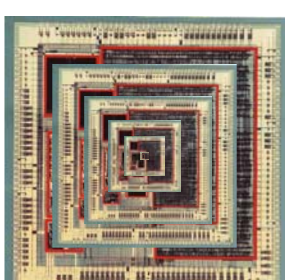
- small memory
- small data word sizes
- relatively slow clocks

To deal with these problems, emphasize efficiency:

- write software at a low level (in assembly code or C)
- avoid operating systems with a rich suite of services
- develop specialized computer architectures
  - programmable DSPs
  - network processors

This is how embedded SW has been designed for 25 years

Why hasn't Moore's law changed all this in 25 years?



Hints that Embedded SW Differs Fundamentally from General Purpose SW

- time matters
  - "as fast as possible" is not good enough
- concurrency is intrinsic
  - it's not an illusion
- object-oriented techniques are rarely used
  - classes and inheritance
  - dynamic binding
- processors avoid memory hierarchy
  - virtual memory
  - dynamically managed caches
- memory management is avoided
  - allocation/de-allocation
  - garbage collection

To be fair, there are some applications that use some of these techniques: e.g. Java in cell phones, but mainly providing the services akin to general purpose software.

Current fashion – Pay Attention to "Non-functional properties"

- Time
- Security
- Fault tolerance
- Power consumption
- Memory management

But the formulation of the question is very telling:

How is it that *when* a braking system applies the brakes is any less a *function* of the braking system than *how much* braking it applies?

What about "real time"?

Make it faster!

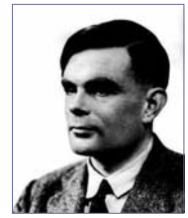
What if you need "absolutely positively on time" (APOT)?

Need to rethink everything: hardware architecture, software abstractions, etc.

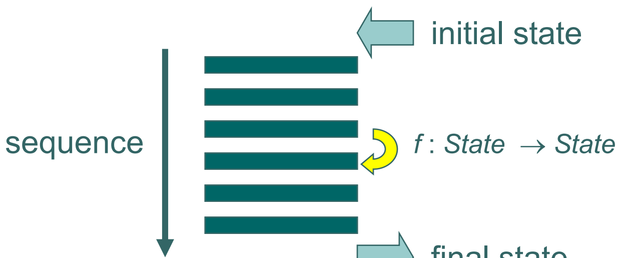
Real-Time Multitasking?

Prioritize and Pray!

Standard Software Abstraction (20-th Century Computation)



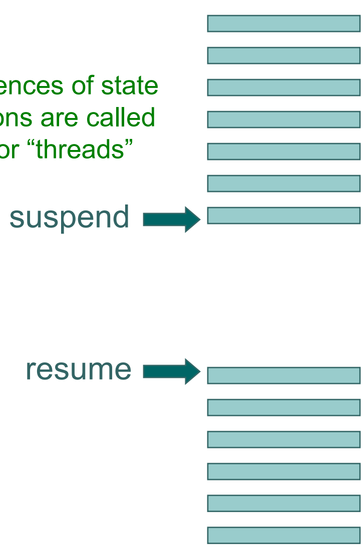
Alan Turing



- Time is irrelevant
- All actions are ordered

Standard Software Abstraction: Processes or Threads

Infinite sequences of state transformations are called "processes" or "threads"



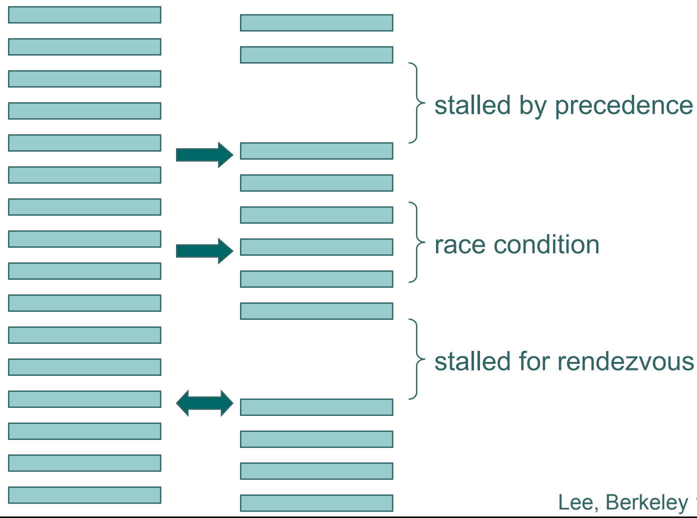
The operating system (typically) provides:

- suspend/resume
- mutual exclusion
- semaphores

Standard Software Abstraction: Concurrency via Interacting Threads

Potential for race conditions, deadlock, and livelock severely compromises software reliability.

These methods date back to the 1960's (Dijkstra).



A Stake in the Ground

Nontrivial concurrent programs based on threads, semaphores, and mutexes are incomprehensible to humans.

- No amount of process improvement is going to change this.
  - the human brain doesn't work this way.
- Formal methods may help
  - scalability?
  - understandability?
- Better concurrency abstractions will help more

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

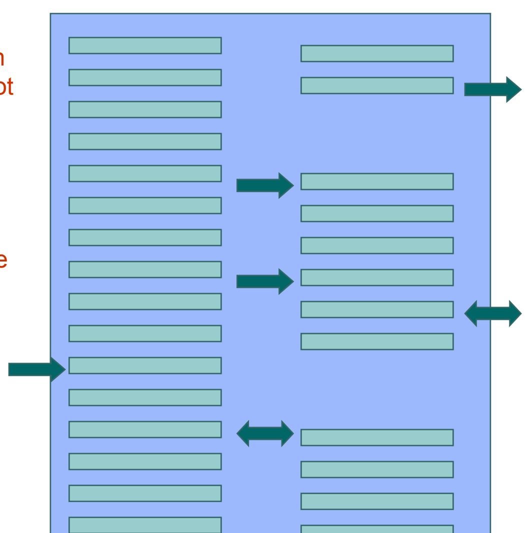
Image "borrowed" from an Iomega advertisement for Y2K software and disk drives, *Scientific American*, September 1999.

Diagnosis: Interacting Threads are Not Compositional

An aggregation of threads is not a thread.

What is it?

Many software failures are due to this ill-defined composition.



Better Concurrency Models

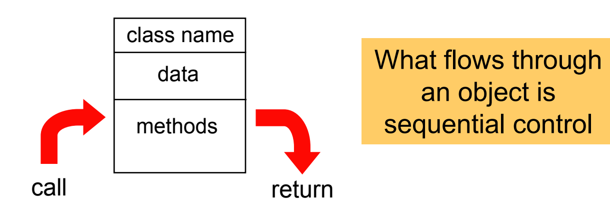
Better concurrency models exist.

We are building the foundations of a family of such models that we call *actor-oriented* models.

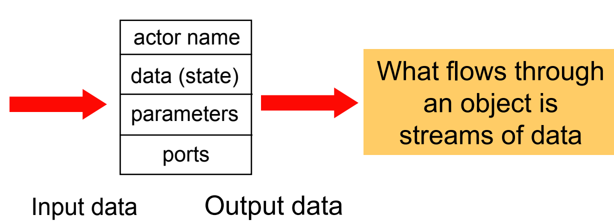
- Semantics of distributed discrete-event systems
- Process networks & algebras
- Hybrid systems
- Models and meta models for model-integrated computing

What is an *Actor-Oriented* MoC?

Traditional component interactions:



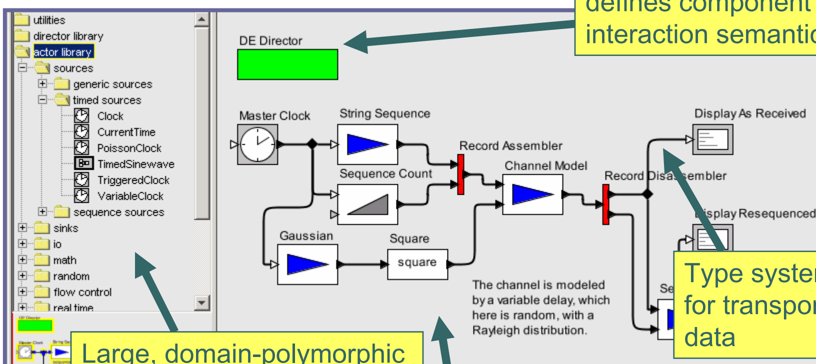
Actor oriented:



Ptolemy II: A Laboratory for Experimenting with Actor-Oriented Models of Computation

Basic Ptolemy II infrastructure:

Director from a library defines component interaction semantics



Type system for transported data

Large, domain-polymorphic component library.

Visual editor supporting an abstract syntax

Models of Computation Implemented in Ptolemy II

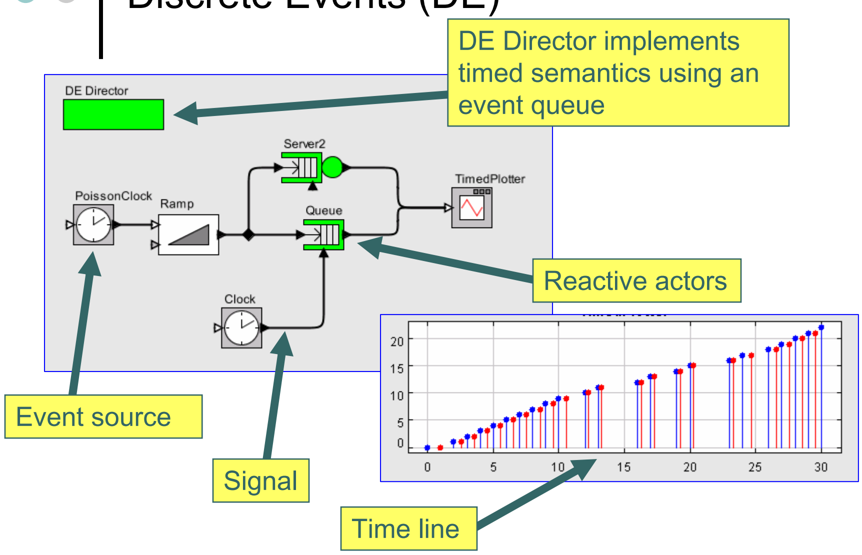
- CI – Push/pull component interaction
- Click – Push/pull with method invocation
- CSP – concurrent threads with rendezvous
- CT – continuous-time modeling
- DE – discrete-event systems
- DDE – distributed discrete events
- DDF – Dynamic dataflow
- DPN – distributed process networks
- DT – discrete time (cycle driven)
- FSM – finite state machines
- Giotto – synchronous periodic
- GR – 2-D and 3-D graphics
- PN – process networks
- SDF – synchronous dataflow
- SR – synchronous/reactive
- TM – timed multitasking

Most of these are actor oriented.

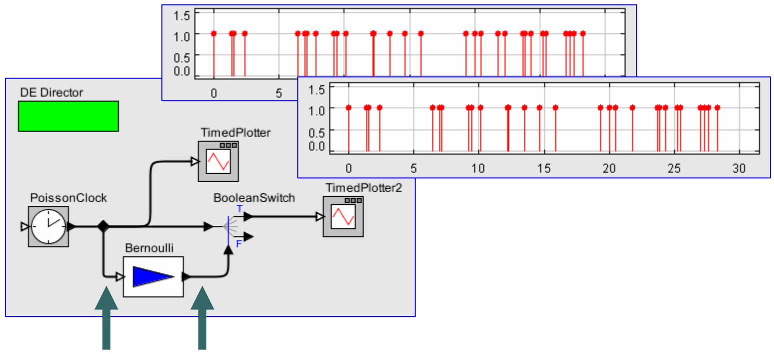
Models of Computation Implemented in Ptolemy II

- CI – Push/pull component interaction
- Click – Push/pull with method invocation
- CSP – concurrent threads with rendezvous
- CT – continuous-time modeling
- DE – discrete-event systems
- DDE – distributed discrete events
- DDF – Dynamic dataflow
- DPN – distributed process networks
- DT – discrete time (cycle driven)
- FSM – finite state machines
- Giotto – synchronous periodic
- GR – 2-D and 3-D graphics
- PN – process networks
- SDF – synchronous dataflow
- SR – synchronous/reactive
- TM – timed multitasking

Example Model of Computation: Discrete Events (DE)

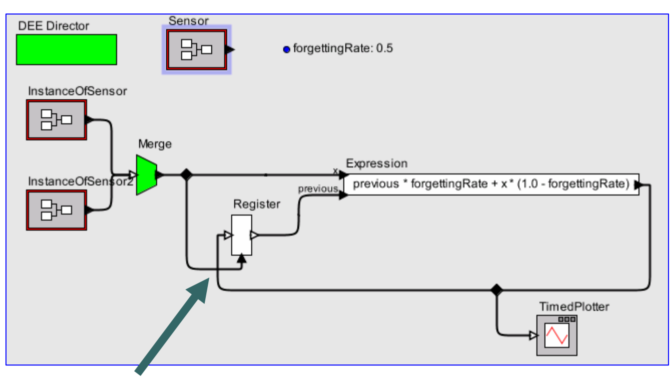


Semantics Clears Up Subtleties: E.g. Simultaneous Events



By default, an actor produces events with the same time as the input event. But in this example, we expect (and need) for the BooleanSwitch to "see" the output of the Bernoulli in the same "firing" where it sees the event from the PoissonClock. Events with identical time stamps are also ordered, and reactions to such events follow data precedence order.

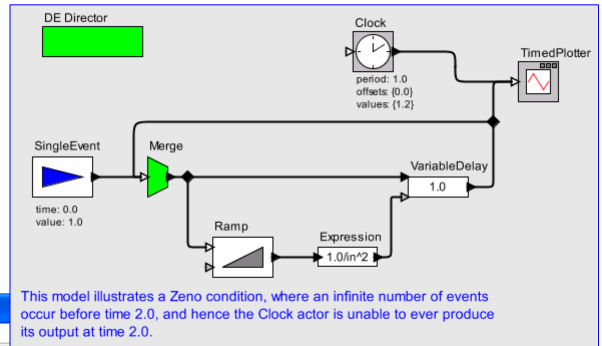
Semantics Clears Up Subtleties: E.g. Feedback

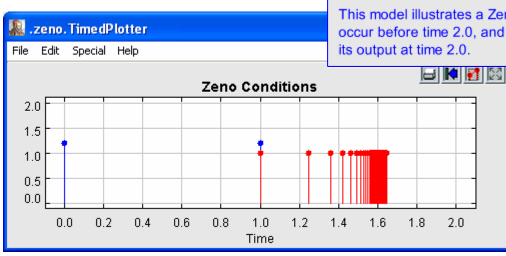


Data precedence analysis has to take into account the non-strictness of this actor (that an output can be produced despite the lack of an input).

Semantics Clears Up Subtleties: E.g. Zeno Systems

DE systems may have an infinite number of events in a finite amount of time. Carefully constructed semantics gives these systems meaning.



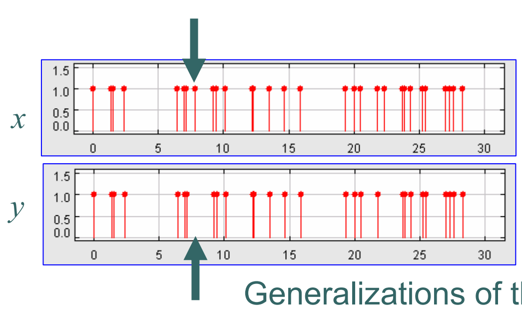


A Rough Sense of Discrete-Event Semantics: Metric Space Formulation

Cantor metric:

$$d(x, y) = 1/2^{\tau}$$

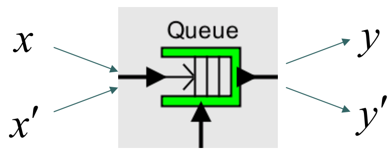
where τ is the earliest time where x and y differ.



The set of signals with this metric form a complete metric space.

Generalizations of this metric handle multiple events at the same time.

Causality in DE



Causal:

$$d(y, y') \le d(x, x')$$

Strictly causal:

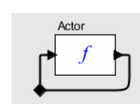
$$d(y, y') < d(x, x')$$

Delta causal:

$$\exists \delta < 1,$$
$$d(y, y') \le \delta\, d(x, x')$$

A delta-causal component is a “contraction map.”

Fixed Point Theorem (Banach Fixed Point Theorem)



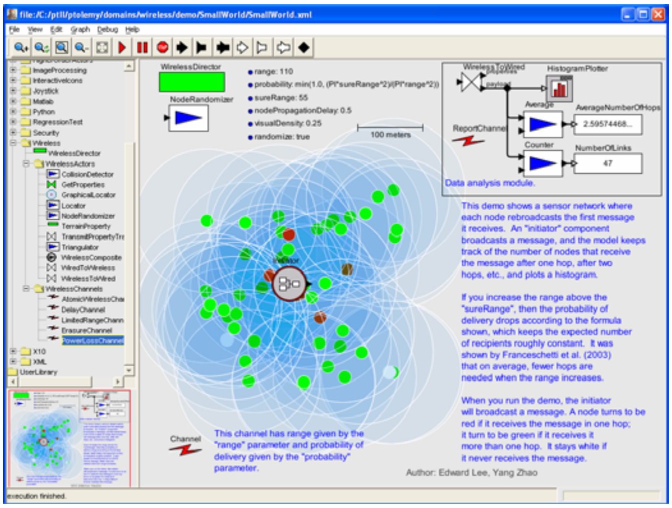
Let $(S^n = [T \to V]^n, d)$ be a *complete* metric space and $f: S^n \to S^n$ be a delta causal function. Then f has a unique fixed point, and for any point $s \in S^n$, the following sequence converges to that fixed point:

$$s_1 = s,\ s_2 = f(s_1),\ s_3 = f(s_2),\ \ldots$$

This means no Zeno!

Current work: Other formulations (using generalized ultrametric spaces, ordinals, and posets) give meaning to a broader class of systems.

Application of DE Modeling Wireless Sensor Nets in VisualSense



VisualSense extends the Ptolemy II discrete-event domain with communication between actors representing sensor nodes being mediated by a *channel*, which is another actor.

The example at the left shows a grid of nodes that relay messages from an *initiator* (center) via a channel that models a low (but non-zero) probability of long range links being viable.

Example of Current Research Challenges

Use distributed discrete-event systems as a timed model of computation for embedded software in unreliable, sporadically connected networks, such as wireless sensor networks.

The most interesting possibilities are based on distributed consensus algorithms (as in Croquet, Reed, Lamport).

Research challenges include:

- Defining the semantics
- Combining the semantics heterogeneously with others. E.g.:
  - Signal processing for channel modeling
  - TinyOS for node functionality
- Creating efficient runtime environments
- Building the design environment

Conclusion

- Threads are a poor concurrent MoC
- There are many better concurrent MoCs
- The ones we know are the tip of the iceberg
- Ptolemy II is a lab for experimenting with them
- This is a rich research area.

http://ptolemy.eecs.berkeley.edu