Making Concurrency Mainstream

Edward A. Lee

Professor, Chair of EECS

UC Berkeley

Joint Invited Talk
CONCUR: Concurrency Theory &
FMICS: Formal Methods for Industrial Critical Systems

Bonn, Germany, August 27, 2006

Concurrency in Software Practice, As of 2006

- Threads
  - Shared memory, semaphores, mutexes, monitors…
- Message Passing
  - Synchronous, asynchronous, buffered, …

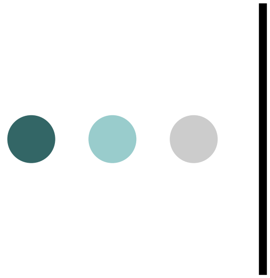
Everything else, regrettably, remains largely in the domain of research…

The Buzz

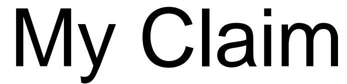
"Multicore architectures will (finally) bring parallel computing into the mainstream. To effectively exploit them, legions of programmers must emphasize concurrency."

The vendor push:

"Please train your computer science students to do extensive multithreaded programming."



Is this a good idea?



My Claim

Nontrivial software written with threads, semaphores, and mutexes are incomprehensible to humans and cannot and should not be trusted!

"The *Observer pattern* defines a one-to-many dependency between a subject object and any number of observer objects so that when the subject object changes state, all its observer objects are notified and updated automatically."

Design Patterns, Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides (Addison-Wesley Publishing Co., 1995. ISBN: 0201633612):

Observer Pattern in Java

```
public void addListener(listener) {…}

public void setValue(newValue) {
    myValue = newValue;

    for (int i = 0; i < myListeners.length; i++) {
        myListeners[i].valueChanged(newValue)
    }
}
```

Will this work in a multithreaded context?

Thanks to Mark S. Miller for the details of this example.

Observer Pattern With Mutual Exclusion (Mutexes)

```
public synchronized void addListener(listener) {…}

public synchronized void setValue(newValue) {
    myValue = newValue;

    for (int i = 0; i < myListeners.length; i++) {
        myListeners[i].valueChanged(newValue)
    }
}
```

Javasoft recommends against this.
What’s wrong with it?

Mutexes are Minefields

```
public synchronized void addListener(listener) {…}

public synchronized void setValue(newValue) {
    myValue = newValue;

    for (int i = 0; i < myListeners.length; i++) {
        myListeners[i].valueChanged(newValue)
    }
}
```

valueChanged() may attempt to acquire a lock on some other object and stall. If the holder of that lock calls addListener(), deadlock!

After years of use without problems, a Ptolemy Project code review found code that was not thread safe. It was fixed in this way. Three days later, a user in Germany reported a deadlock that had not shown up in the test suite.

Simple Observer Pattern Becomes Not So Simple

```
public synchronized void addListener(listener) {…}

public void setValue(newValue) {
    synchronized(this) {
        myValue = newValue;
        listeners = myListeners.clone();
    }

    for (int i = 0; i < listeners.length; i++) {
        listeners[i].valueChanged(newValue)
    }
}
```

while holding lock, make copy of listeners to avoid race conditions

notify each listener <u>outside</u> of synchronized block to avoid deadlock

This still isn’t right.
What’s wrong with it?

Simple Observer Pattern: How to Make It Right?

```
public synchronized void addListener(listener) {…}

public void setValue(newValue) {
    synchronized(this) {
        myValue = newValue;
        listeners = myListeners.clone();
    }

    for (int i = 0; i < listeners.length; i++) {
        listeners[i].valueChanged(newValue)
    }
}
```

Suppose two threads call setValue(). One of them will set the value last, leaving that value in the object, but listeners may be notified in the opposite order. The listeners may be alerted to the value changes in the wrong order!

```
/**
CrossRefList is a list that maintains pointers to other CrossRefLists.
…
@author Geroncio Galicia, Contributor: Edward A. Lee
@version $Id: CrossRefList.java,v 1.78 2004/04/29 14:50:00 eal Exp $
@since Ptolemy II 0.2
@Pt.ProposedRating Green (eal)
@Pt.AcceptedRating Green (bart)
*/
public final class CrossRefList implements Serializable  {
    …
    protected class CrossRef implements Serializable{
        …
        // NOTE: It is essential that this method not be
        // synchronized, since it is called by _farContainer(),
        // which is.  Having it synchronized can lead to
        // deadlock.  Fortunately, it is an atomic action,
        // so it need not be synchronized.
        private Object _nearContainer() {
            return _container;
        }

        private synchronized Object _farContainer() {
            if (_far != null) return _far._nearContainer();
            else return null;
        }
        …
    }
}
```

Code that had been in use for four years, central to Ptolemy II, with an extensive test suite with 100% code coverage, design reviewed to yellow, then code reviewed to green in 2000, causes a deadlock during a demo on April 26, 2004.

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.

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.

If concurrency were intrinsically hard, we would not function well in the physical world

It is not concurrency that is 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…

The Following are Only Partial Solutions

- Training programmers to use threads.
- Improve software engineering processes.
- Devote attention to “non-functional” properties.
- Use design patterns.

None of these deliver a rigorous, analyzable, and understandable model of concurrency.

We Can Incrementally Improve Threads

- Object Oriented programming
- Coding rules (Acquire locks in the same order…)
- Libraries (Stapl, Java 5.0, …)
- Patterns (MapReduce, …)
- Transactions (Databases, …)
- Formal verification (Blast, thread checkers, …)
- Enhanced languages (Split-C, Cilk, Guava, …)
- Enhanced mechanisms (Promises, futures, …)

But is it enough to refine a mechanism with flawed foundations?

Do Threads Have a Sound Foundation?

If the foundation is bad, then we either tolerate brittle designs that are difficult to make work, or we have to rebuild from the foundations.

Note that this whole enterprise is held up by threads

Succinct Problem Statement

Threads are wildly nondeterministic.

The programmer's job is to prune away the nondeterminism by imposing constraints on execution order (e.g., mutexes) and limiting shared data accesses (e.g., OO design).

Succinct Solution Statement

Instead of starting with a wildly nondeterministic mechanism and asking the programmer to rein in that nondeterminism, start with a deterministic mechanism and incrementally add nondeterminism where needed.

Under this principle, even the most effective of today's techniques (OO design, transactions, message passing, …) require fundamental rethinking.

Problems with the Foundations

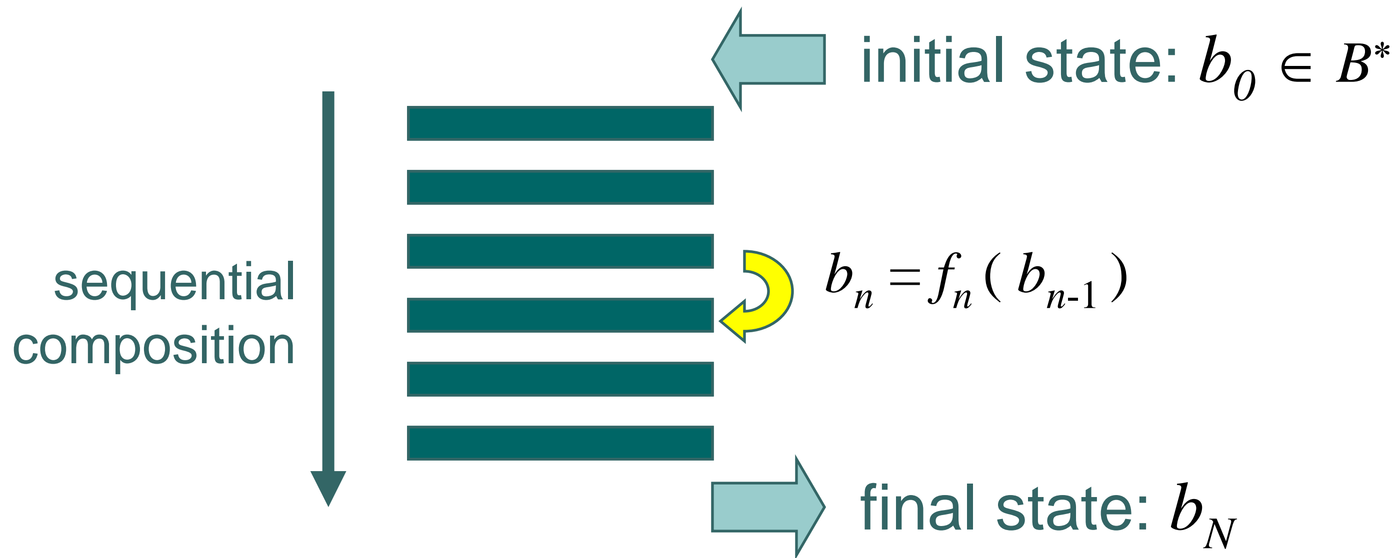
The 20-th century notion of "computation":

- Bits: $B = \{0, 1\}$
- Set of finite sequences of bits: B^*
- Computation: $f : B^* \to B^*$
- Composition of computations: $f \bullet f'$
- Programs specify compositions of computations

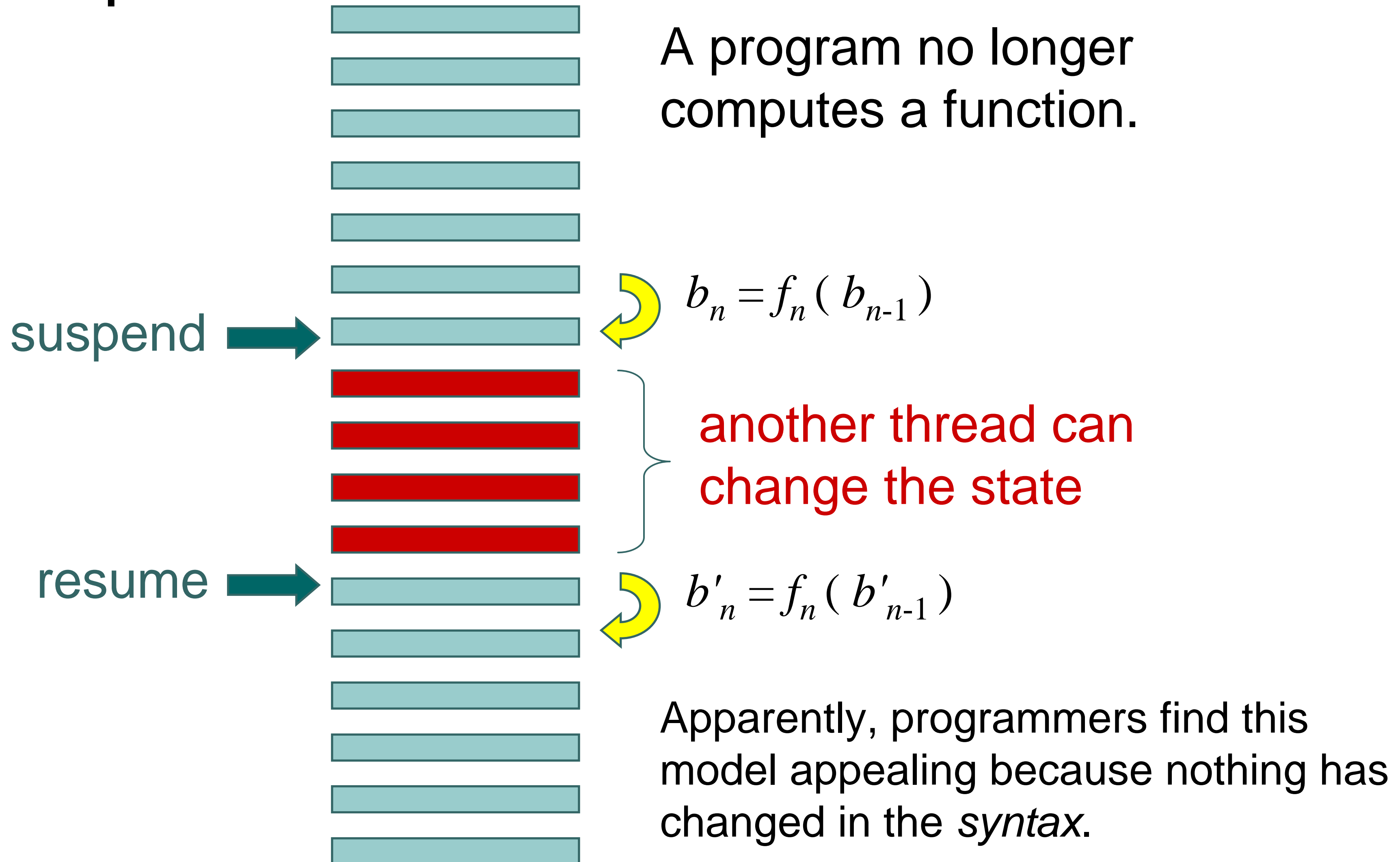
Threads augment this model to admit concurrency.

But this model does not admit concurrency gracefully.

Composition of Computations



When There are Threads, Everything Changes



Instead of a Program Specifying…

$$f : B^* \to B^*$$

… a Program Should Specify

$$f : (T \to B^*)^n \to (T \to B^*)^n$$

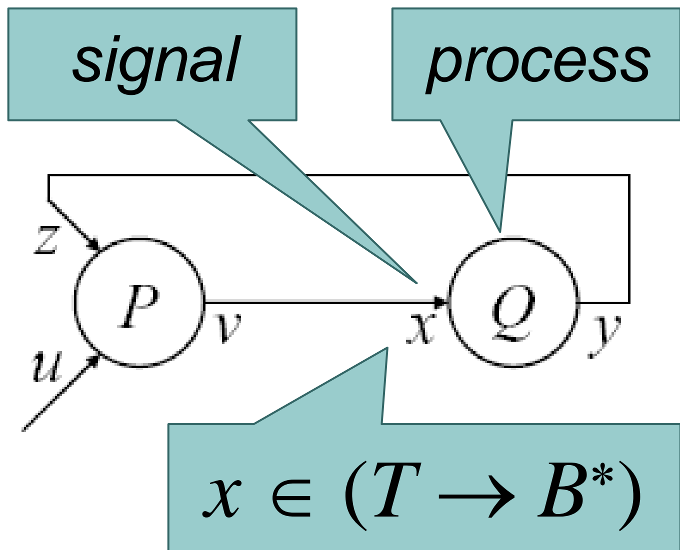
This is a function from the set of tuples of (possibly partial) functions from T to B^* into itself, for some partially ordered set of tags T.

Composition of concurrent components now becomes function composition.

This is called the “tagged signal model”
[Lee & Sangiovanni-Vincentelli, 1998]

Example: Tag Set T for Kahn Process Networks

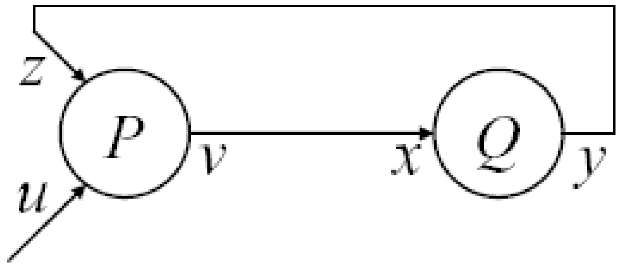
Ordering constraints on tags imposed by communication:



Each signal maps a totally ordered subset of T into values.

Example from Xiaojun Liu, Ph.D. Thesis, 2005.

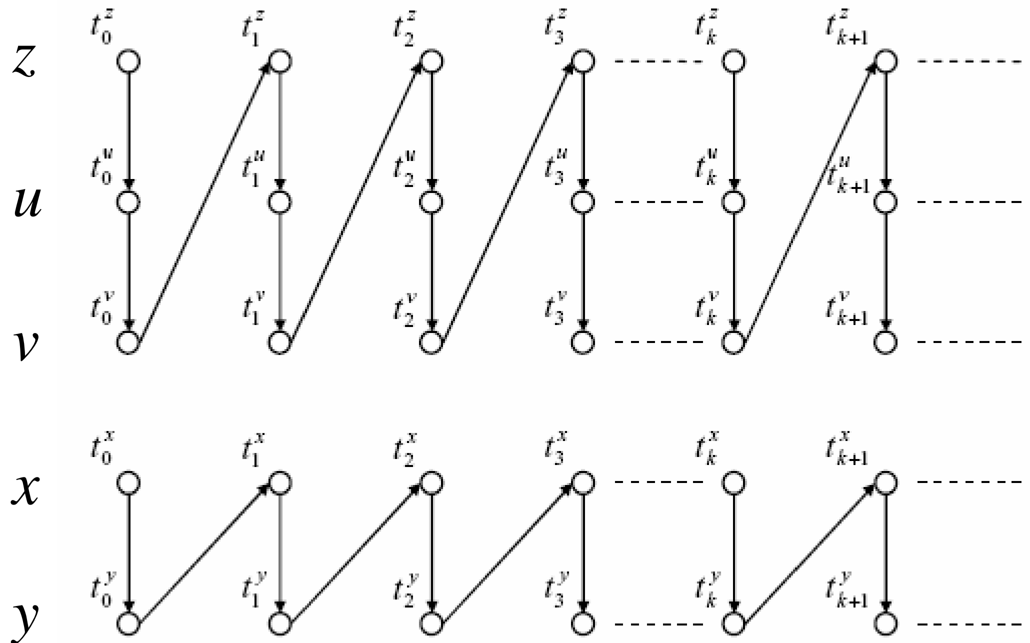
Example: Tag Set *T* for Kahn Process Networks




```
Process P(in z, u; out v)
{
  repeat {
    t1 = receive(z)
    t2 = receive(u)
    send(v, t1 + t2)
  }
}
```

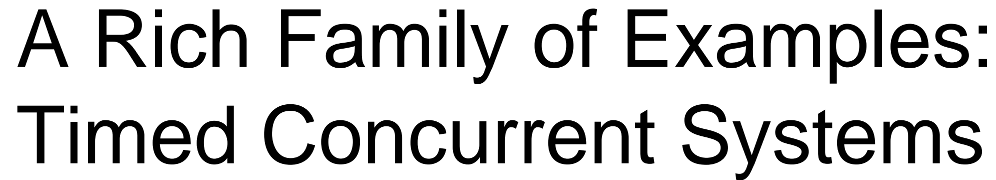
```
Process Q(in x; out y)
{
  repeat {
    t = receive(x)
    send(y, t)
  }
}
```

Ordering constraints on tags imposed by computation:



Composition of these constraints with the previous reveals deadlock.

Example from Xiaojun Liu, Ph.D. Thesis, 2005.



- Tag set is totally ordered.
  - Example: $T = \mathbb{R}_0 \times \mathbb{N}$, with lexicographic order ("super dense time").
- Used to model
  - hardware,
  - continuous dynamics,
  - hybrid systems,
  - embedded software
- Gives semantics to "cyber-physical systems".

See [Liu, Matsikoudis, Lee, CONCUR 2006].

The Catch…

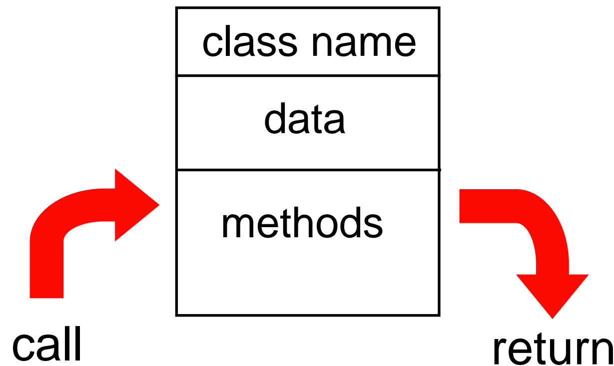
$$f : (T \to B^*)^n \to (T \to B^*)^n$$

- This is not what (mainstream) programming languages do.
- This is not what (mainstream) software component technologies do.

Let’s tackle the second problem first…

Object Oriented vs. Actor Oriented

The established: Object-oriented:

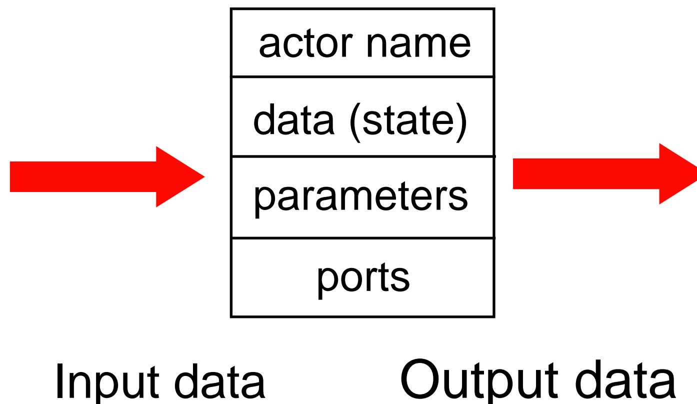


What flows through an object is sequential control

Things happen to objects

The alternative: Actor oriented:

Actors make things happen

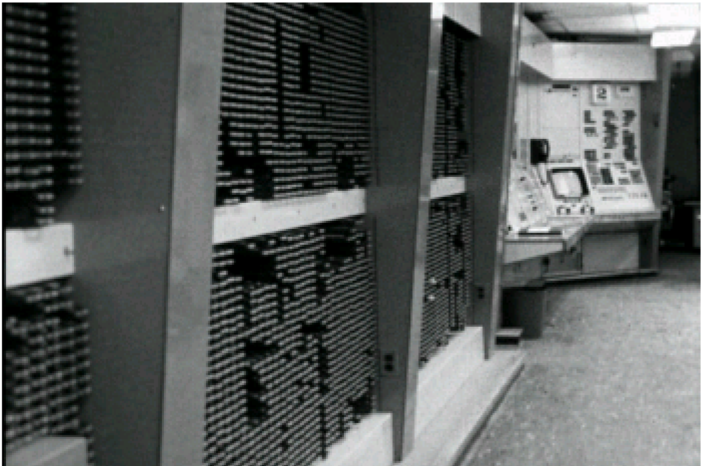


What flows through an object is evolving data

The First (?) Actor-Oriented Programming Language

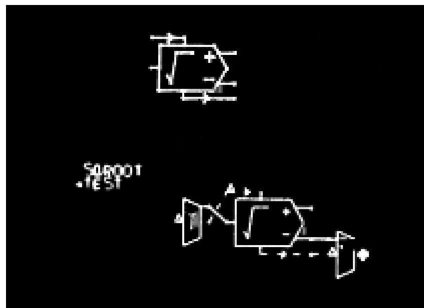
The On-Line Graphical Specification of Computer Procedures

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 "Languages"

- CORBA event service (distributed push-pull)
- ROOM and UML-2 (dataflow, Rational, IBM)
- VHDL, Verilog (discrete events, Cadence, Synopsys, ...)
- LabVIEW (structured dataflow, National Instruments)
- Modelica (continuous-time, constraint-based, Linkoping)
- OPNET (discrete events, Opnet Technologies)
- SDL (process networks)
- Occam (rendezvous)
- Simulink (Continuous-time, The MathWorks)
- SPW (synchronous dataflow, Cadence, CoWare)
- …

Many of these are domain specific.

Many of these have visual syntaxes.

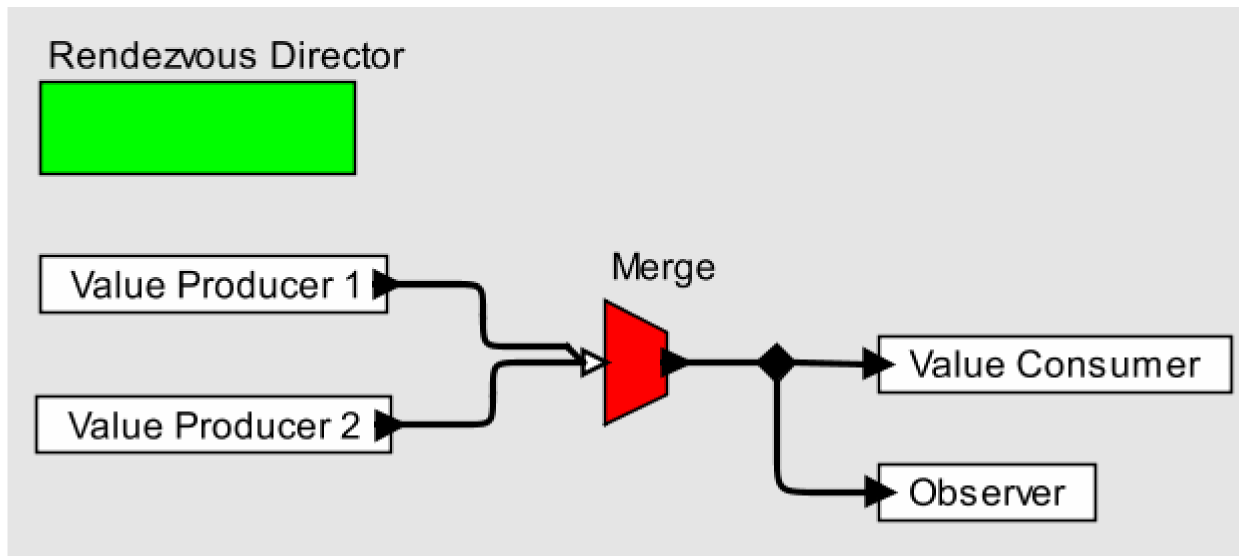
The semantics of these differ considerably, but all can be modeled as

$$f : (T \to B^*)^n \to (T \to B^*)^n$$

with appropriate choices of the set T.

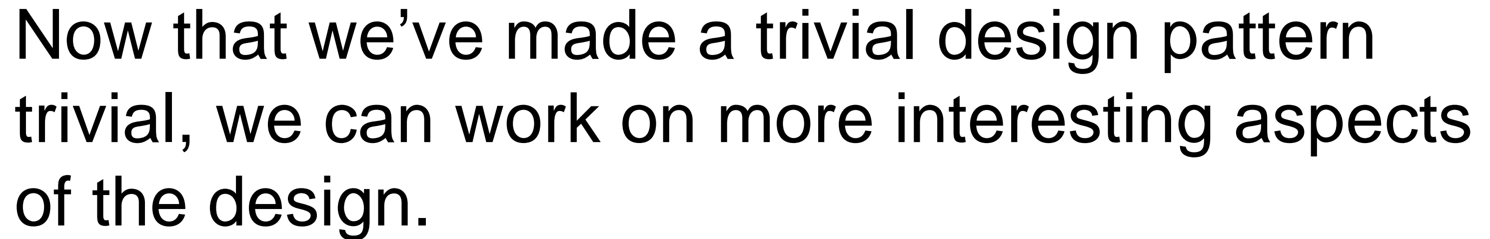
"The *Observer pattern* defines a one-to-many dependency between a subject object and any number of observer objects so that when the subject object changes state, all its observer objects are notified and updated automatically."

Observer Pattern using an Actor-Oriented Language with Rendezvous Semantics



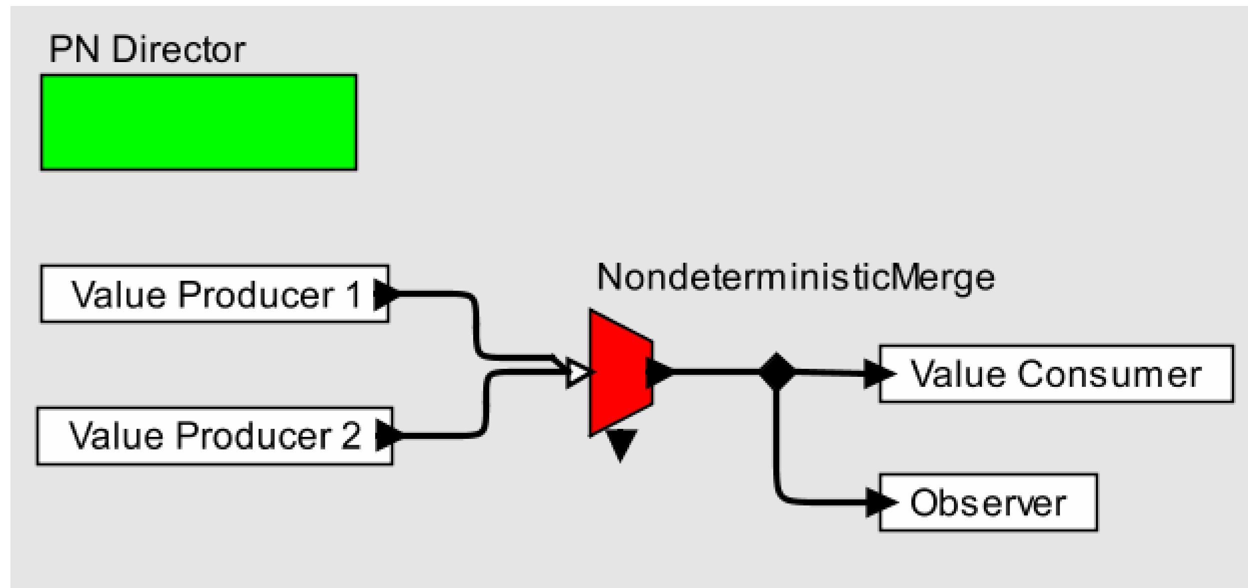
Each actor is a process, communication is via rendezvous, and the Merge explicitly represents nondeterministic multi-way rendezvous.

This is realized here in a *coordination language* with a *visual syntax*.



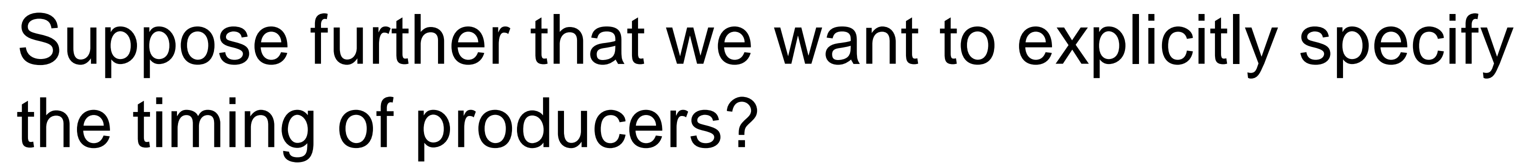
E.g., suppose we don’t care how long notification of the observer is deferred, as long as the observer is notified of all changes in the right order?

Observer Pattern using an Actor-Oriented Language with Kahn Semantics (Extended with Nondeterministic Merge)

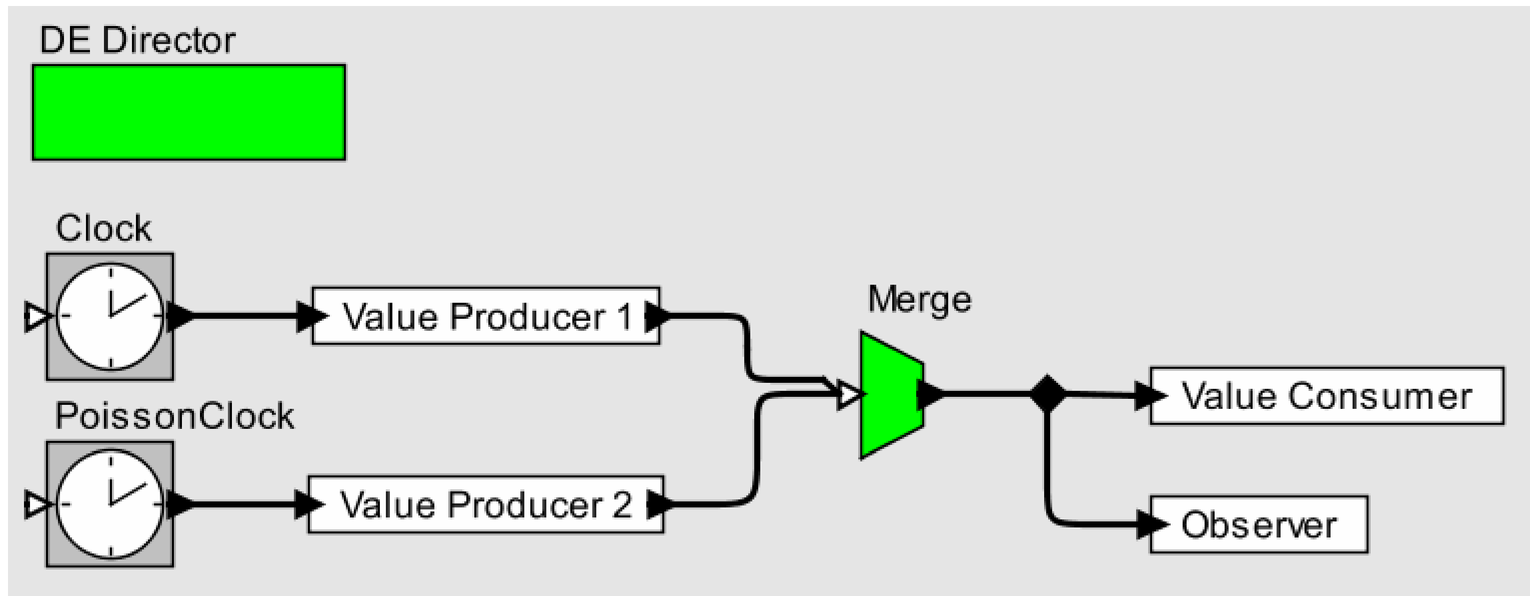


Each actor is a process, communication is via streams, and the NondeterministicMerge explicitly merges streams nondeterministically.

Again a *coordination language* with a *visual syntax*.


Suppose further that we want to explicitly specify the timing of producers?

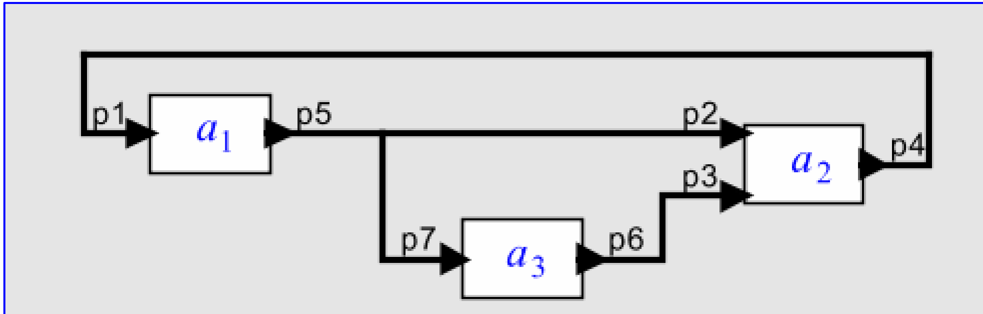
Observer Pattern using an Actor-Oriented Language with Discrete Event Semantics



Messages have a (semantic) time, and actors react to messages chronologically. Merge now becomes deterministic.

Again a *coordination language* with a *visual syntax*.

Composition Semantics



Each actor is a function:

$$f : (T \to B^*)^n \to (T \to B^*)^n$$

Composition in three forms:

- Cascade connections
- Parallel connections
- Feedback connections

The nontrivial part of this is feedback, but this community knows how to handle that.

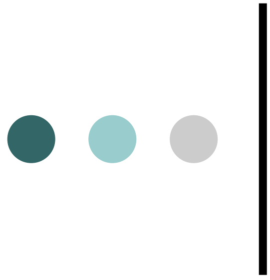
The model of computation determines the set T:

- *Process Networks*
- *Synchronous/Reactive*
- *Time-Triggered*
- *Discrete Events*
- *Dataflow*
- *Rendezvous*
- *Continuous Time*
- *…*

Recall The Catch …

$$f : (T \to B^*)^n \to (T \to B^*)^n$$

- This is not what (mainstream) programming languages do.
  - *What to do here?*
- This is not what (mainstream) software component technologies do.
  - *Actor-oriented components*



Nothing!

(Almost)

Do Not Ignore the Challenges

- Computation is deeply rooted in the sequential paradigm.
  - Threads appear to adhere to this paradigm, but throw out its essential attractiveness.
- Programmers are reluctant to accept new syntax.
  - Regrettably, syntax has a bigger effect on acceptance than semantics, as witnessed by the wide adoption of threads.
- Only general purpose languages get attention.
  - A common litmus test: must be able to write the compiler for the language in the language.

Opportunities

- New syntaxes can be accepted when their purpose is orthogonal to that of established languages.
  - Witness UML, a family of languages for describing object-oriented design, complementing C++ and Java.
- Coordination languages can provide capabilities orthogonal to those of established languages.
  - The syntax can be noticeably distinct (as in the diagrams shown before).

Actor-oriented design can be accomplished through *coordination languages* that complement rather than replace existing languages.

Actor-oriented component architectures implemented in *coordination languages* that complement rather than replace existing languages.

With good design of these coordination languages, this will deliver understandable concurrency.

See the Ptolemy Project for explorations of several such languages: http://ptolemy.org

Conclusion

- Transformation of bits is the foundation of computation.
- Threads are the foundation of concurrent programming practice.

- The foundations are flawed.
  - Threads discard the most essential features of computation.
  - Threads are incomprehensible to humans.
  - They appeal because they make (almost) no changes to syntax.

- Concurrent computation needs a new foundation.
  - Actor oriented component models.
  - Coordination languages with actor semantics.
  - Visual syntaxes to seduce users.
  - These have a chance of acceptance!

References

1. The Ptolemy Project: http://ptolemy.org
2. X. Liu, E. Matsikoudis, and E. A. Lee, "Modeling Timed Concurrent Systems," in *CONCUR 2006 - Concurrency Theory*, Bonn, Germany, (LNCS 4137, Springer), August 27-30, 2006.
3. E. A. Lee, "The Problem with Threads," *Computer*, vol. 39, pp. 33-42, 2006.
4. X. Liu and E. A. Lee, "CPO Semantics of Timed Interactive Actor Networks," UC Berkeley, Berkeley, CA, Technical Report EECS-2006-67, May 18 2006.
5. A. Cataldo, E. A. Lee, X. Liu, E. Matsikoudis, and H. Zheng, "A Constructive Fixed-Point Theorem and the Feedback Semantics of Timed Systems," in Workshop on Discrete Event Systems (WODES), Ann Arbor, Michigan, July 10-12, 2006.
6. X. Liu, "Semantic Foundation of the Tagged Signal Model," EECS Department, University of California, Berkeley, CA, PhD Thesis December 20 2005.
7. E. A. Lee, "Concurrent Models of Computation for Embedded Software: Lecture Notes for EECS 290N," EECS Department, University of California, Berkeley, CA, Technical Report UCB/ERL M05/2, January 4 2005.
8. E. A. Lee and A. Sangiovanni-Vincentelli, "A Framework for Comparing Models of Computation," IEEE Transactions on Computer-Aided Design of Circuits and Systems, vol. 17, pp. 1217-1229, 1998.