Concurrency Demands New Foundations for Computing

Edward A. Lee

Robert S. Pepper Distinguished Professor

Chair of EECS

UC Berkeley

Invited Talk

ARTIST2 Workshop on
MoCC – Models of Computation and Communication

Zurich, Switzerland, November 16-17, 2006

A Look at “Computation”
Some Notation

- Natural numbers: $\mathbb{N} = \{0, 1, 2, \cdots\}$
- Sequences of bits (finite and infinite): B^{**}
- Functions on sequences of bits:

$$Q = (B^{**} \rightharpoonup B^{**})$$

- $Q = (B^{**} \rightharpoonup B^{**})$

A Look at "Computation"
Imperative Machines

Imperative machine = (A, c)

- Actions: $A \subset Q$
- Halt action: $h \in A, \forall\ b \in B^{**}, \quad h(b) = b$
- Control function: $c\colon B^{**} \rightarrow \mathbb{N}$

A Look at "Computation" Programs and Threads

- $Q = (B^{**} \rightharpoonup B^{**})$
- Actions: $A \subset Q$.
- Control: $c\colon B^{**} \rightarrow \mathbb{N}$

Sequential Program of length m:

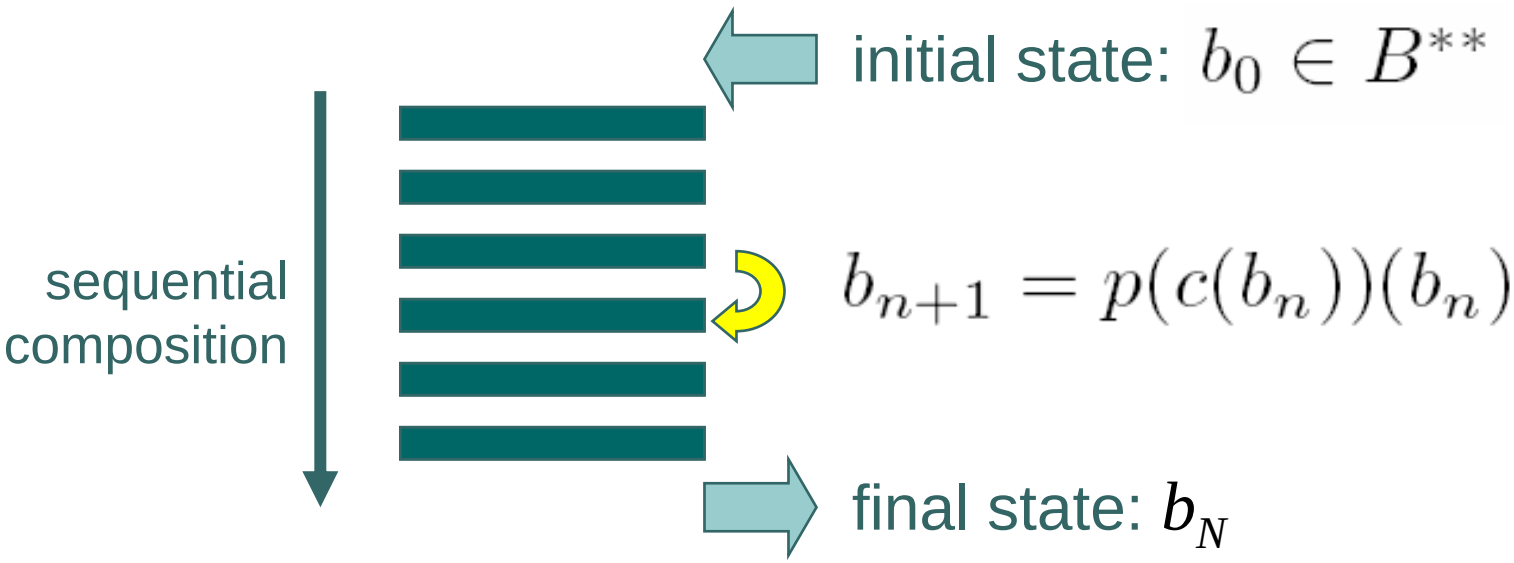
- $p\colon \mathbb{N} \rightarrow A$
- $\forall\ n \geq m, \quad p(n) = h$

Thread:

- Initial state: $b_0 \in B^{**}$
- $\forall\ n \in \mathbb{N},\ b_{n+1} = p(c(b_n))(b_n)$

A Look at "Computation"
A Single Thread

- $Q = (B^{**} \rightharpoonup B^{**})$
- Actions: $A \subset Q$.
- Control: $c\colon B^{**} \rightarrow \mathbb{N}$
- Program: $p\colon \mathbb{N} \rightarrow A$



Computable Functions

- $Q = (B^{**} \rightharpoonup B^{**})$
- Actions: $A \subset Q$.
- Control: $c\colon B^{**} \to \mathbb{N}$
- Program: $p\colon \mathbb{N} \to A$
- Thread: $b_{n+1} = p(c(b_n))(b_n)$

A program

$$p\colon \mathbb{N} \to A$$

defines a (partial or total) function

$$P\colon B^{**} \rightharpoonup B^{**}$$

that is defined on all initial states

$$b_0 \in B^{**}$$

for which the program terminates.

Observations

- $Q = (B^{**} \rightharpoonup B^{**})$
- Actions: $A \subset Q$.
- Control: $c\colon B^{**} \to \mathbb{N}$
- Program: $p\colon \mathbb{N} \to A$
- Thread: $b_{n+1} = p(c(b_n))(b_n)$
- Program(2): $P\colon B^{**} \rightharpoonup B^{**}$

- The set of (finite) programs is countable.
- The set of functions Q is not countable.
- Many choices of $A \subset Q$ yield the same subset of Q that can be computed by terminating programs:
  - the “effectively computable” functions.
- Program composition by procedure call is function composition (neat and simple).

Program Composition by Interleaving Threads

- $Q = (B^{**} \rightharpoonup B^{**})$
- Actions: $A \subset Q$.
- Control: $c\colon B^{**} \to \mathbb{N}$
- Program: $p\colon \mathbb{N} \to A$
- Thread: $b_{n+1} = p(c(b_n))(b_n)$
- Program(2): $P\colon B^{**} \rightharpoonup B^{**}$

Multiple threads:

- $p_i\colon \mathbb{N} \to A, \quad i \in \{1, \cdots, M\}$
- $b_{n+1} = p_i(c(b_n))(b_n), \quad i \in \{1, \cdots, M\}$

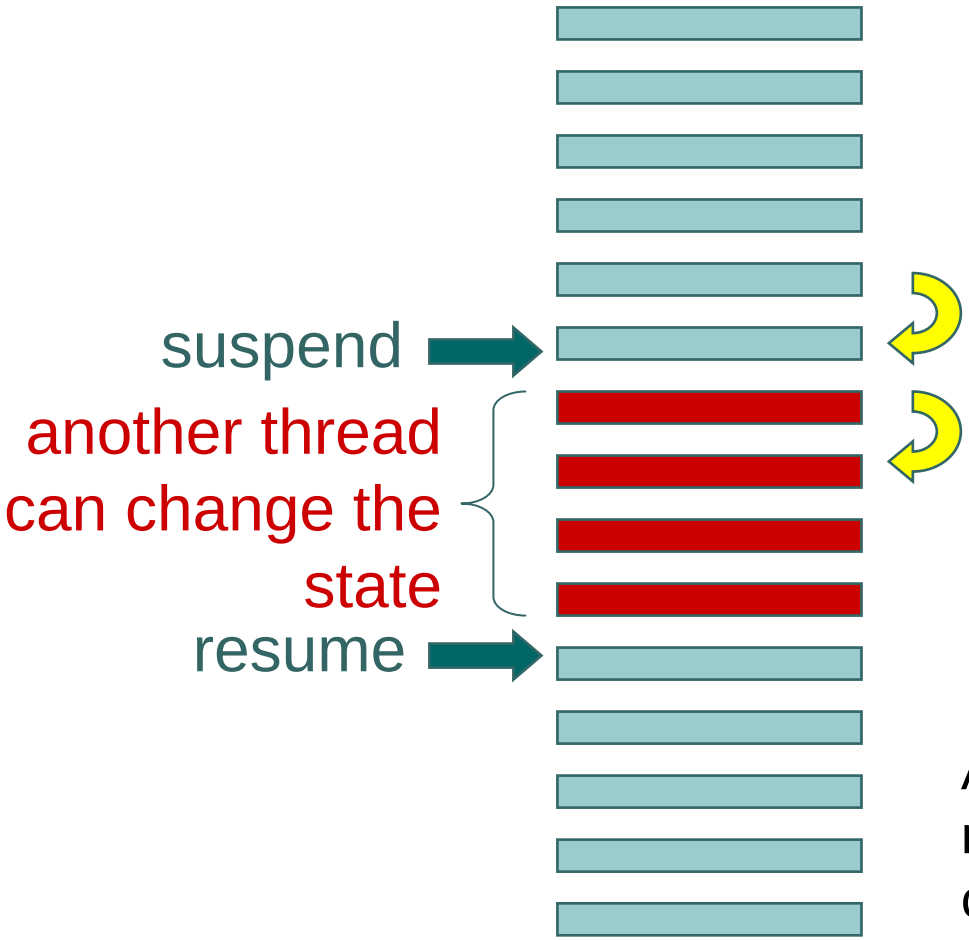
The essential and appealing properties of computation are lost:

- Programs are no longer functions
- Composition is no longer function composition.
- Very large numbers of behaviors may result.
- Can't tell when programs are equivalent.

Sadly, this is how most concurrent computation is done today.

Nondeterministic Interleaving

- $Q = (B^{**} \rightharpoonup B^{**})$
- Actions: $A \subset Q$.
- Control: $c\colon B^{**} \to \mathbb{N}$
- Program: $p\colon \mathbb{N} \to A$
- Thread: $b_{n+1} = p(c(b_n))(b_n)$
- Program(2): $P\colon B^{**} \rightharpoonup B^{**}$



$$b_n = p_1(c(b_{n-1}))(b_{n-1})$$

$$b_{n+1} = p_2(c(b_n))(b_n)$$

Apparently, programmers find this model appealing because nothing has changed in the *syntax* of programs.

To See That Current Practice is Bad, Consider a Simple Example

"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!

```
public synchronized void addChangeListener(ChangeListener listener) {
    NamedObj container = (NamedObj) getContainer();
    if (container != null) {
        container.addChangeListener(listener);
    } else {
        if (_changeListeners == null) {
            _changeListeners = new LinkedList();
            _changeListeners.add(0, listener);
        } else if (!_changeListeners.contains(listener)) {
            _changeListeners.add(0, listener);
        }
    }
}
```

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 outside 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!

If the simplest design patterns yield such problems, what about non-trivial designs?

```
/**
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.

My Claim

Nontrivial concurrent software written with threads is incomprehensible to humans and cannot be trusted!

Maybe better abstractions would lead to better practice…

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).

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.

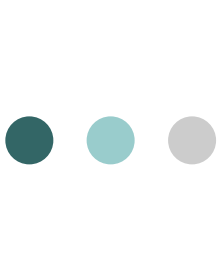
If concurrency were intrinsically hard, we would not function well in the physical world

It is not concurrency that is hard…

…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…



Yet threads are the basis for all widely used concurrency models, as well as the basis for I/O interactions and network interactions in modern computers.

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.

The question is how to do this and still get concurrency.

We Need to Replace the Core Notion of "Computation"

Instead of

$$P : B^{**} \rightharpoonup B^{**}$$

we need

$$F : (\mathcal{T} \rightharpoonup B^{**}) \rightharpoonup (\mathcal{T} \rightharpoonup B^{**})$$

where $\mathcal{T}$ is a partially or totally ordered set.

We have called this the "tagged signal model" [Lee & Sangiovanni-Vincentelli, 1998]. Related models:

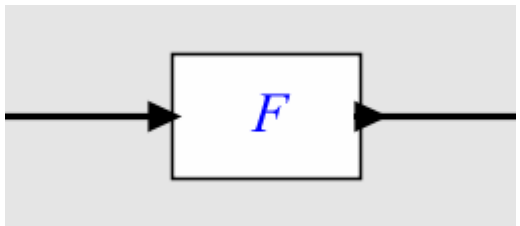
- Interaction Categories [Abramsky, 1995]
- Interaction Semantics [Talcott, 1996]
- Abstract Behavioral Types [Arbab, 2005]

Actors and Signals

If computation is

$$F : (\mathcal{T} \rightharpoonup B^{**}) \rightharpoonup (\mathcal{T} \rightharpoonup B^{**})$$

then a program is an "actor:"



Given an input "signal"

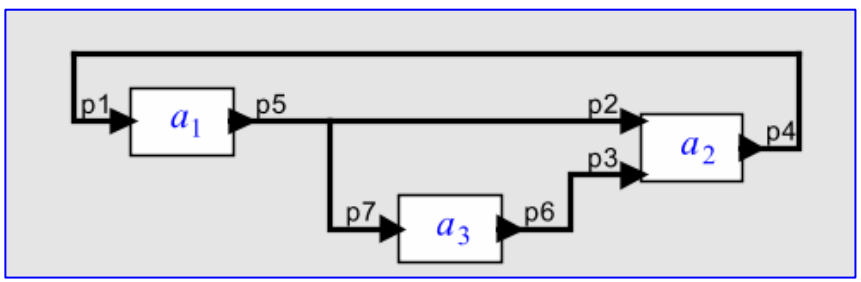
$$s_1 : \mathcal{T} \rightharpoonup B^{**}$$

it produces an output "signal"

$$s_2 : \mathcal{T} \rightharpoonup B^{**}$$

A General Formulation

- Signals: $S = (\mathcal{T} \rightharpoonup B^{**})$
- Ports: $P = \{p_1, p_2, p_3, p_4\}$
- Behavior: $\sigma : P \rightarrow S$
- Actor with ports P_a is $a \subset (P_a \rightarrow S)$

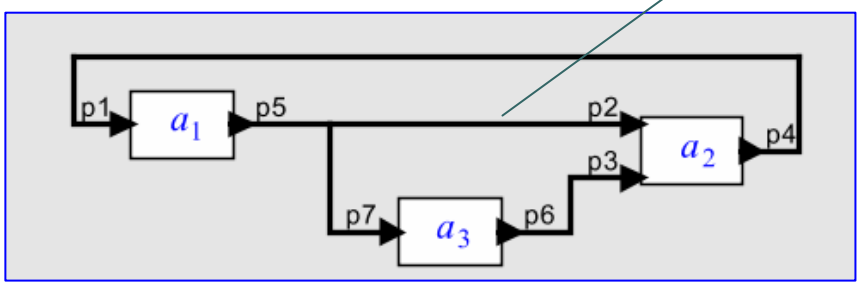


Note that nondeterministic actors are easily embraced by the model.

Principle: Put nondeterminism only where you need it!

Connectors are Actors Too

Identity Connector between ports P_c is $c \subset (P_c \rightarrow S)$, where $\forall\ \sigma \in c,\ \exists s \in S$ such that $\forall\ p \in P_c,\ \sigma(p) = s.$



Connector with three ports

Composition of Components

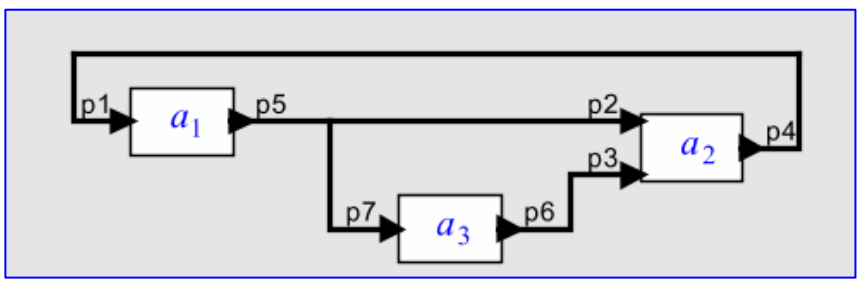
Given two actors a with ports P_a and b with ports P_b, the composition is an actor

$$a \wedge b \subset ((P_a \cup P_b) \rightarrow S)$$

where

$$a \wedge b = \{\sigma \mid \sigma \downarrow_{P_a} \in a \text{ and } \sigma \downarrow_{P_b} \in b\}$$

This notation from [Benveniste, Carloni, Caspi, Sangiovanni-Vincentelli, EMSOFT '03]



Note that nondeterministic actors are easily embraced by the model.

Principle: Composition itself does not introduce nondeterminsm!

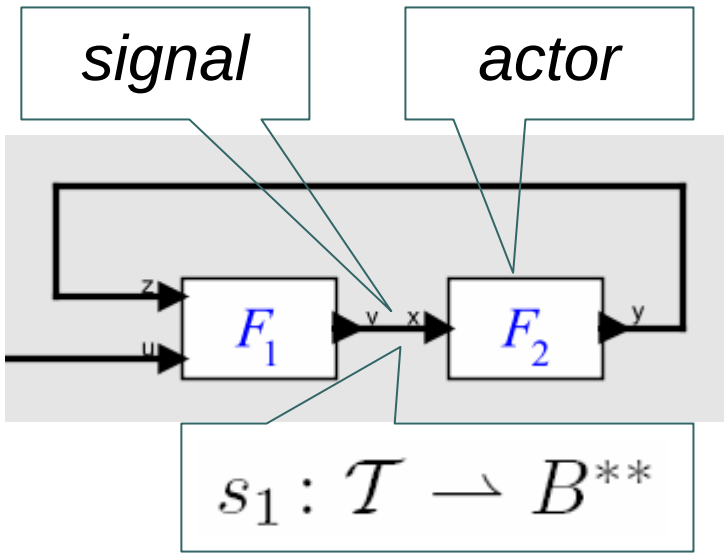
Structure of the Tag Set

The algebraic properties of the tag set $\mathcal{T}$ are determined by the concurrency model, e.g.:

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

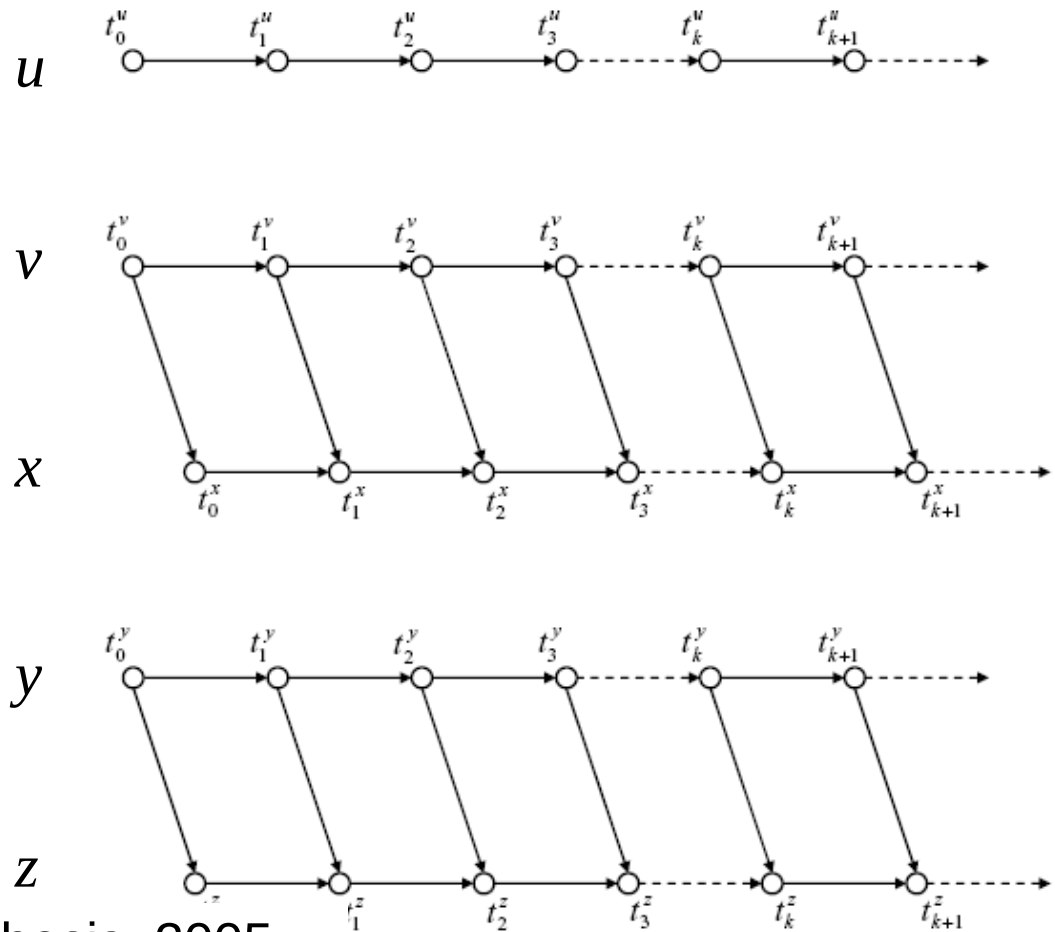
Associated with these may be a richer model of the connectors between actors.

Example of a Partially Ordered Tag Set *T* for Kahn Process Networks



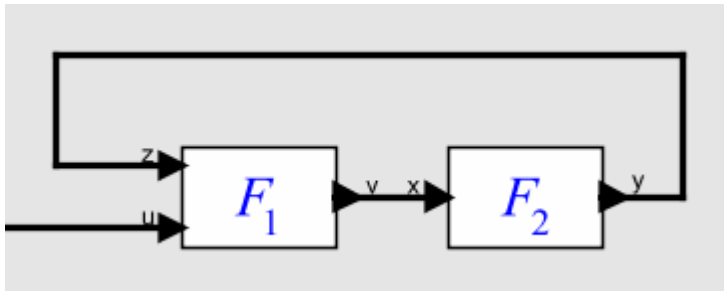
Each signal maps a totally ordered subset of $\mathcal{T}$ into values.

Ordering constraints on tags imposed by communication:



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

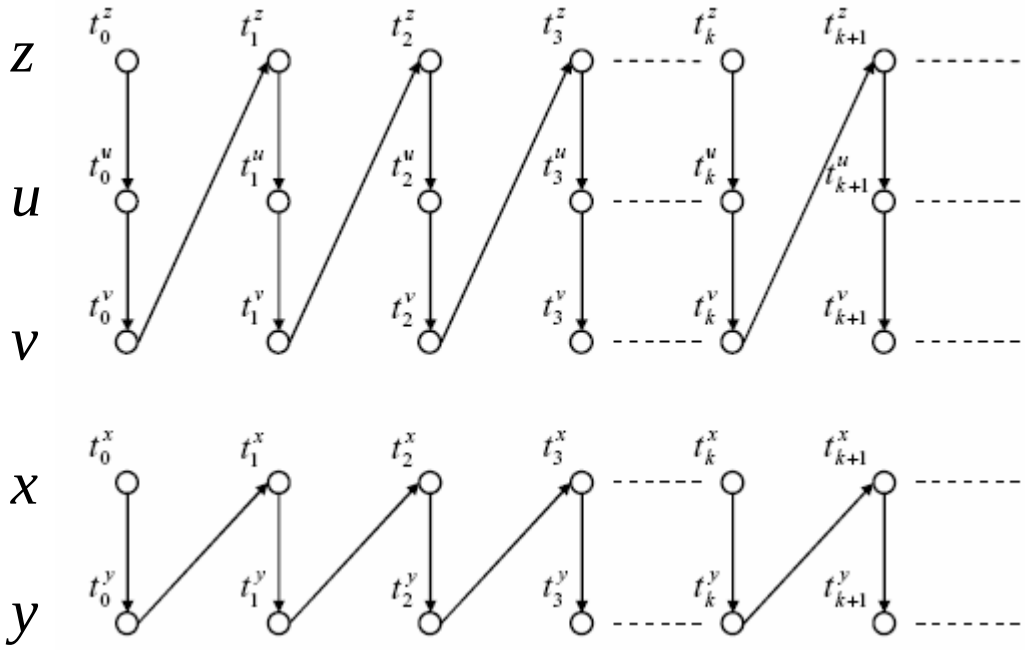
Example: Tag Set *T* for Kahn Process Networks




```
Actor F1(in z, u; out v)
{
  repeat {
    t1 = receive(z)
    t2 = receive(u)
    send(v, t1 + t2)
  }
}
```

```
Actor F2(in x; out y)
{
  repeat {
    t = receive(x)
    send(v, 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.

More Examples: Timed Systems
(those with Totally Ordered Tag Sets)

- 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…

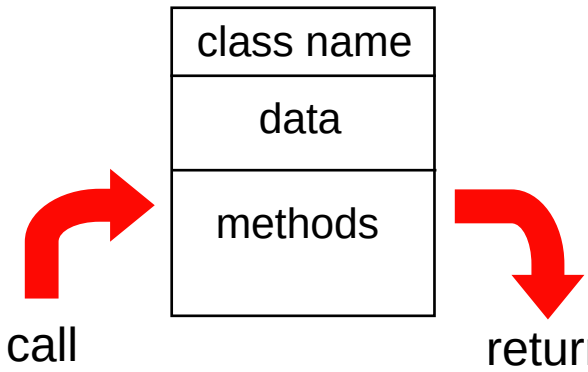
$$F : (\mathcal{T} \rightharpoonup B^{**}) \rightharpoonup (\mathcal{T} \rightharpoonup B^{**})$$

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

The second problem is easier to solve…

Actor-Oriented Design

The established: Object-oriented:

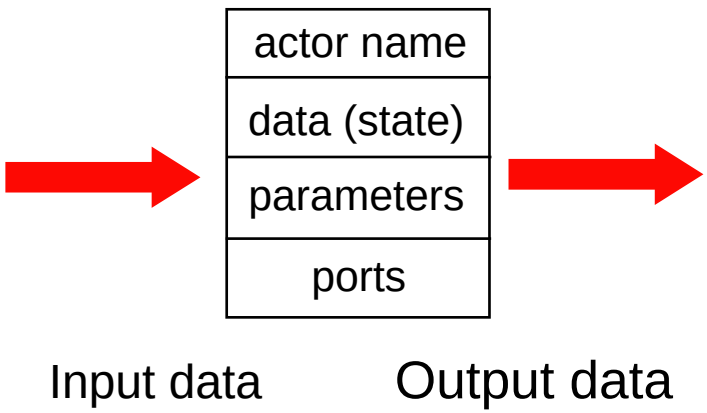


What flows through an object is sequential control

Things happen to objects

The alternative: “Actor oriented:”

Actors make things happen

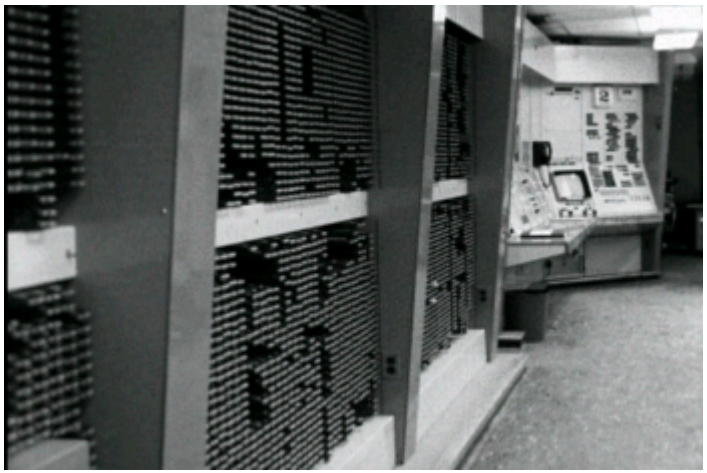


What flows through an object is evolving data

$$F : (\mathcal{T} \rightharpoonup B^{**}) \rightharpoonup (\mathcal{T} \rightharpoonup B^{**})$$

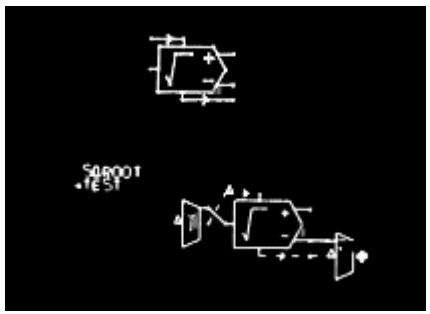
The First (?) Actor-Oriented Programming Language

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).

Your Speaker in 1966

Examples of Actor-Oriented Coordination 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)
- Ptolemy (various, Berkeley)
- 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

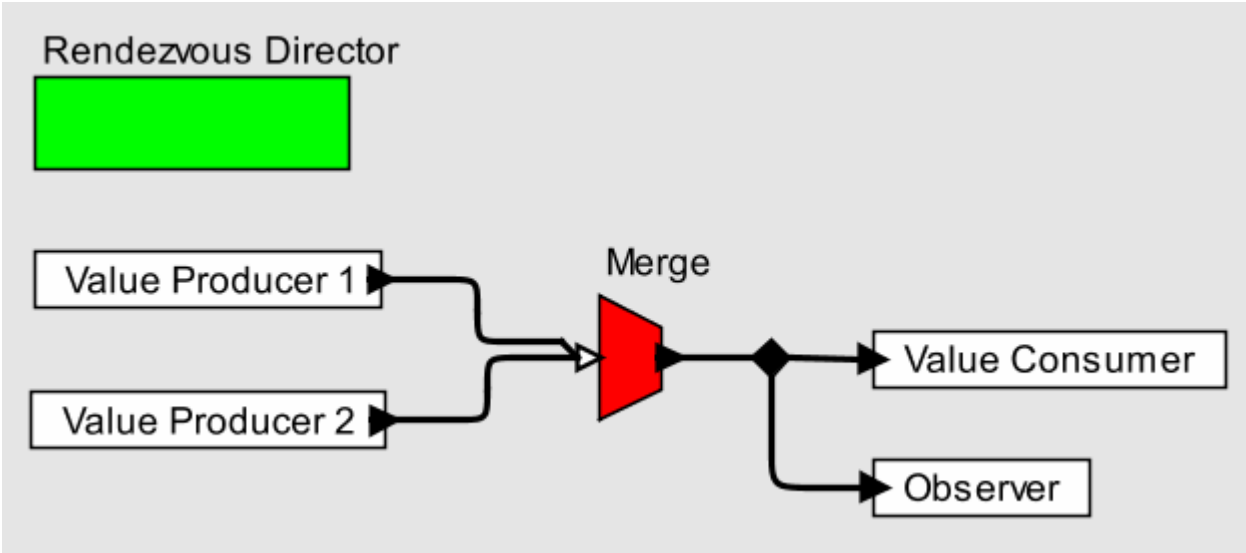
$$F : (\mathcal{T} \rightharpoonup B^{**}) \rightharpoonup (\mathcal{T} \rightharpoonup B^{**})$$

with appropriate choices of the set T.

Recall the Observer Pattern

"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*.

Recall The Catch …

$$F : (\mathcal{T} \rightharpoonup B^{**}) \rightharpoonup (\mathcal{T} \rightharpoonup B^{**})$$

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

Programming Languages

Imperative reasoning is simple and useful

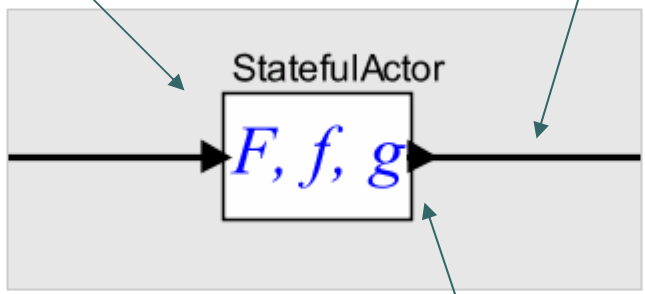
Keep it!

Reconciling Imperative and Actor Semantics: Stateful Actor Abstract Semantics

An actor is a function from input signals to output signals. That function is defined in terms of two functions.

Signals are monoids (can be incrementally constructed) (e.g. streams, discrete-event signals).

$$F : S_1 \rightarrow S_2$$



$s_1 \in S_1$ $s_2 \in S_2$

$$f : S_1 \times \Sigma \rightarrow S_2$$

$$g : S_1 \times \Sigma \rightarrow \Sigma$$

state space

A port is either an input or an output.

The function f gives outputs in terms of inputs and the current state. The function g updates the state.

But for Timed MoCC's, we Have a Problem

- Timing in imperative languages is unpredictable!
- The fix for this runs deep:
  - Need new architectures:
    - Replace cache memories with scratchpads
    - Replace dynamic dispatch with pipeline interleaving
  - Need decidable subsets of standard languages
  - Need precise and tight WCET bounds.
  - Need new OS, networking, …

Summary

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

Semantics of these coordination languages is what MoCC is about.

See the Ptolemy Project for explorations of several such (domain-specific) languages: http://ptolemy.org