Concurrent Semantics without the Notions of State or State Transitions

Edward A. Lee

Robert S. Pepper Distinguished Professor

Chair of EECS

UC Berkeley

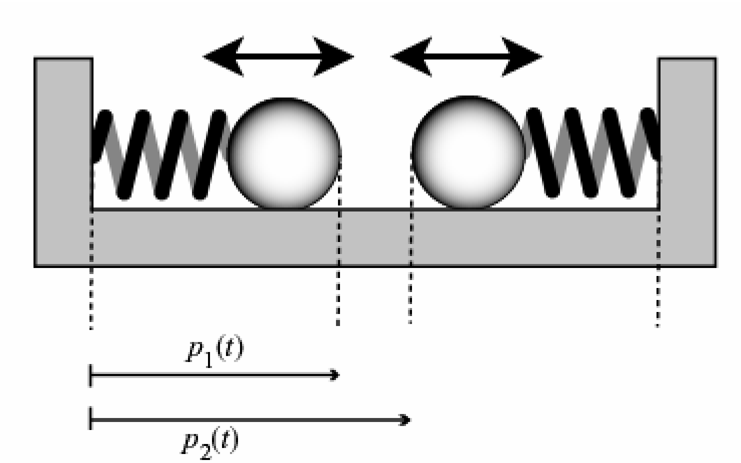
Invited talk

FORMATS 2006: 4-th International Conference on Formal Modelling and Analysis of Timed Systems, September 26, 2006

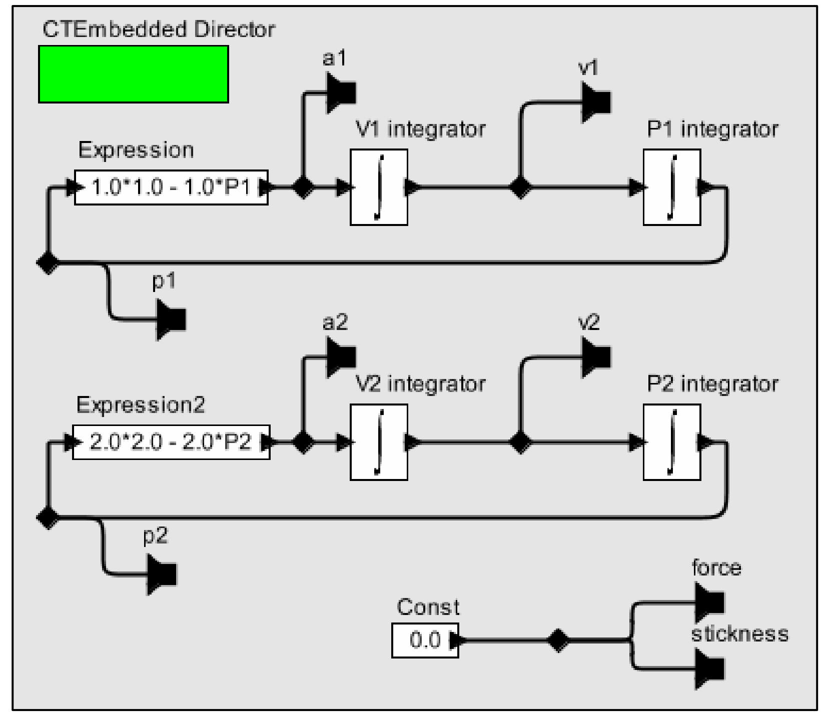
System State

- The notion of *system state* depends on an idealized abstraction, that an observer can “simultaneously” observe all parts of a system.
- This abstraction depends on the Newtonian-physical abstraction of time, which has a “global variable” *t* with the same value everywhere.

E.g. Differential Equation Models of Systems use the Newtonian-Physical Abstraction



$$\ddot{p_1}(t) = k_1(n_1 - p_1(t))/m_1$$
$$\ddot{p_2}(t) = k_2(n_2 - p_2(t))/m_2.$$



The variable *t* is shared by all components in the model.

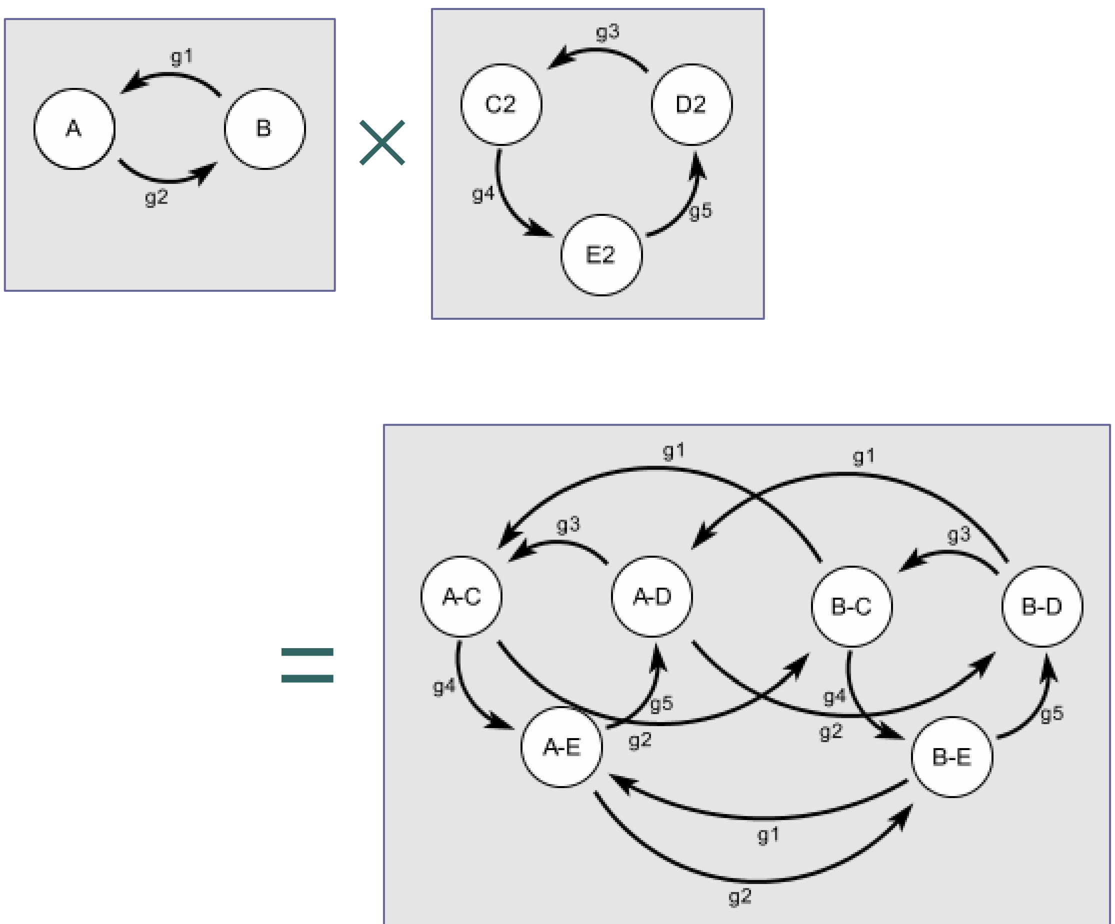
But Networked Software-Based Systems Don't Do This!

- No observer that can simultaneously observe all parts of a system.
- There is no common notion of time.

Should we patch the models?

- network time synchronization?
- nondeterministic interleavings?

Transition System Models of Software with Nondeterministic Interleavings



My claim: For most purposes, the resulting nondeterminism is not an interesting property of the system.

When it is an interesting property, then the system may be badly designed.

Is the Notion of System State a Good Foundation?

If poor foundations lead to problems (such as intractable models), is the right solution to patch the models, or to fix the foundations?

Need to question the foundations of computation for concurrency.

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: $c \colon B^{**} \longrightarrow \mathbb{N}$

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

A Look at "Computation" Programs and Threads

Sequential Program of length m:

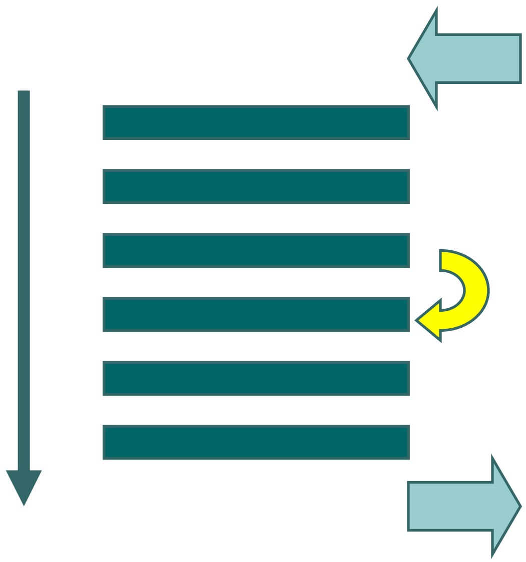
- $p\colon \mathbb{N} \to A$
- $\forall\, n \geq m, \quad p(n) = h$

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: $c \colon B^{**} \to \mathbb{N}$
- Program: $p \colon \mathbb{N} \to A$



initial state: $b_0 \in B^{**}$

sequential composition

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

final state: b_N

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:

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

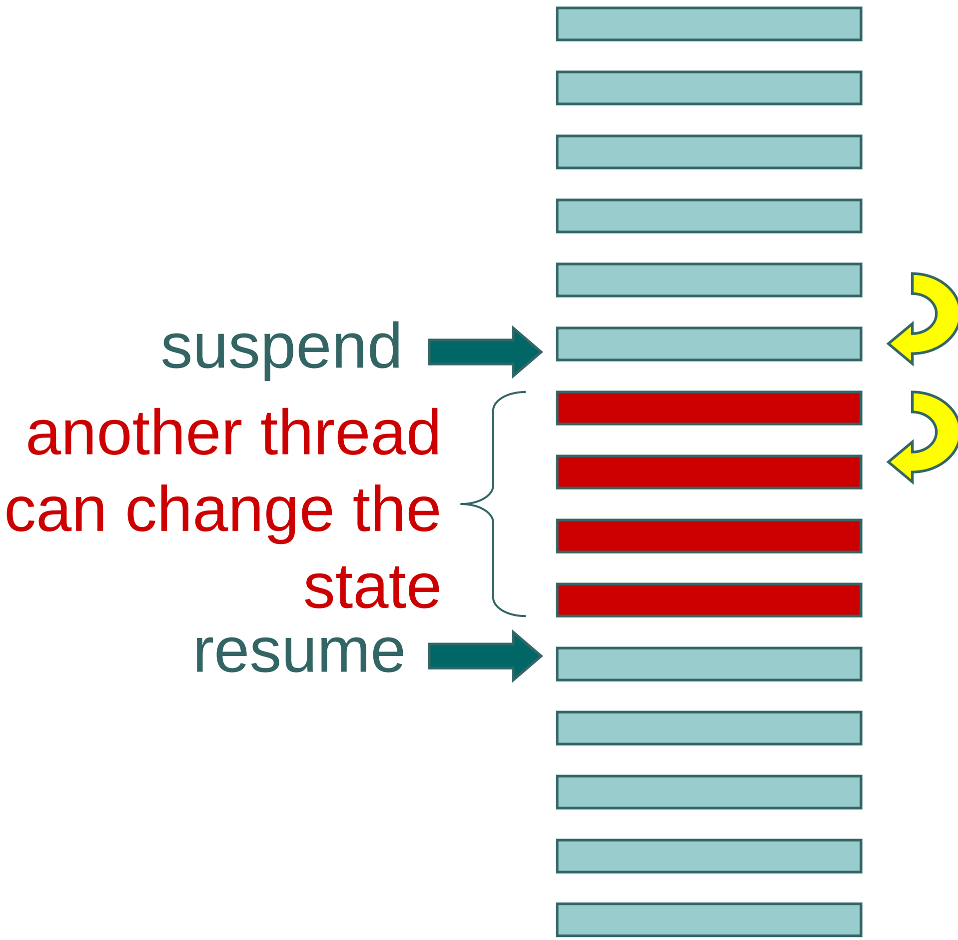
The essential and appealing properties of computation are lost:

- 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.

Given This, Transition Systems with Nondeterministic Interleavings Seem Like the Right Model for Concurrent Software!

- Threads are widely used in programming
- Threads are accurately modeled by nondeterministic interleaving of state transitions systems.
- Distributed software leverages the threads model.

But…

Threads are a bad idea because the nondeterminism is (mostly) spurious.

When it isn't, programs are incomprehensible and unreliable.

My Claim

Having an abstraction that matches current practice is good.

Having an abstraction that matches current practice that is bad is less clear…

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!

```java
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 <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!

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).

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

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

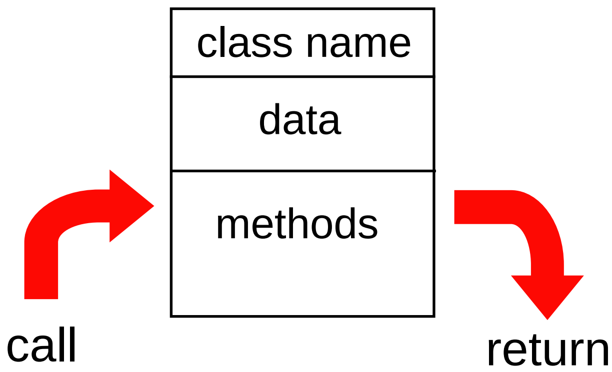
where $\mathcal{T}$ is a partially or totally ordered set.

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]

A Consequence: Components Are No Longer Procedure-Based

The established: Object-oriented:

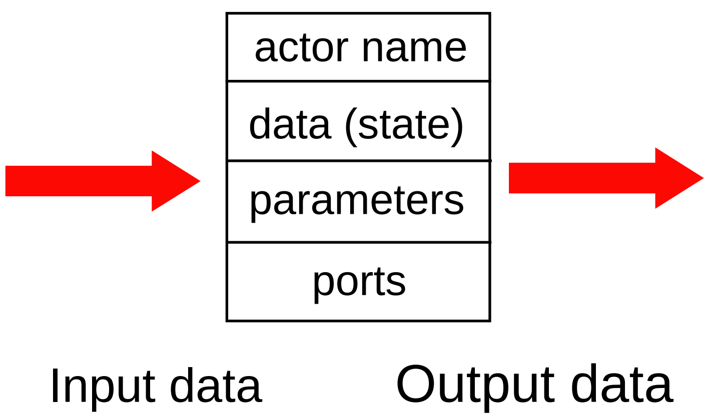


What flows through an object is sequential control

Things happen to objects

The alternative: “Actor oriented:”

Actors make things happen



What flows through an object is evolving data

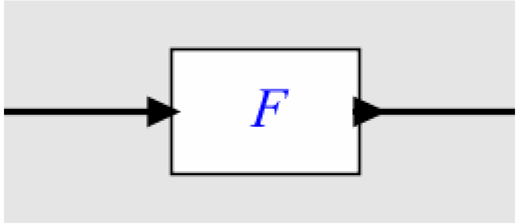
$$F : (\mathcal{T} \rightharpoonup B^{**}) \rightharpoonup (\mathcal{T} \rightharpoonup B^{**})$$

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"

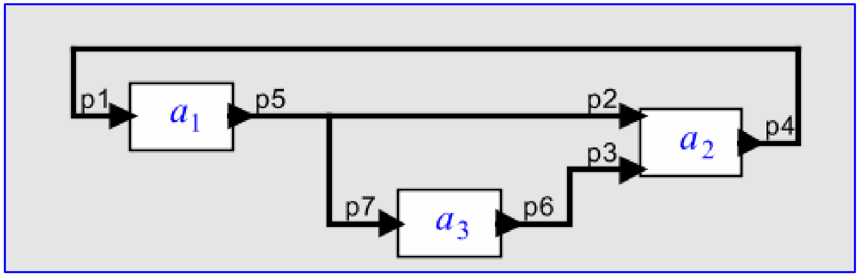
$$s_1 : \mathcal{T} \rightharpoonup B^{**}$$

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: $\sigma : P \to S$
- Actor with ports P_a is $a \subset (P_a \to S)$
- Connector between ports P_c is $c \subset (P_c \to S)$, where $\forall\ \sigma \in c,\ \exists s \in S$ such that $\forall\ p \in P_c,\ \sigma(p) = s$.



Note that nondeterministic actors are easily embraced by the model.

Principle: Put nondeterminism only where you need it!

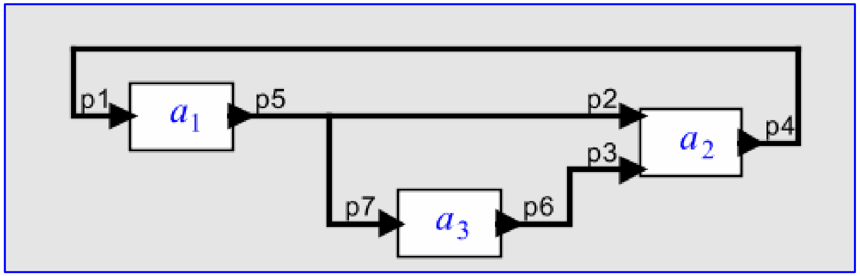
Composition of Components

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

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

where

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



Note that nondeterministic actors are easily embraced by the model.

Principle: Composition itself does not introduce nondeterminsm!

Structure of the Tag Set

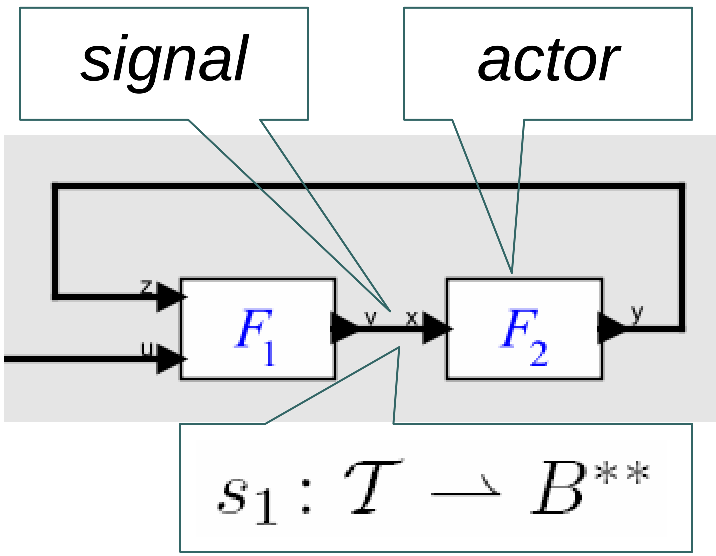
The algebraic properties of the tag set $\mathcal{T}$ are determined by the concurrency model, e.g.:

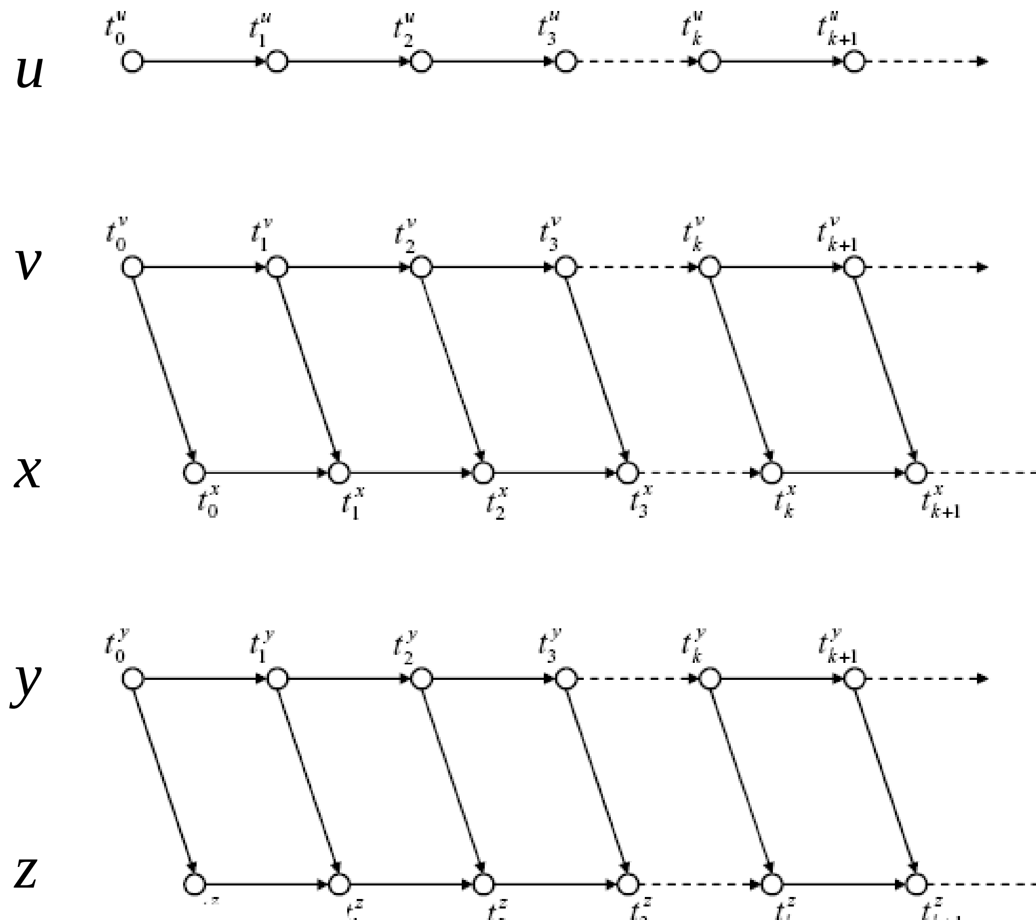
- Process Networks
- Synchronous/Reactive
- Time-Triggered
- Discrete Events
- Dataflow
- Rendezvous
- Continuous Time
- Hybrid Systems
- …

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

Ordering constraints on tags imposed by communication:

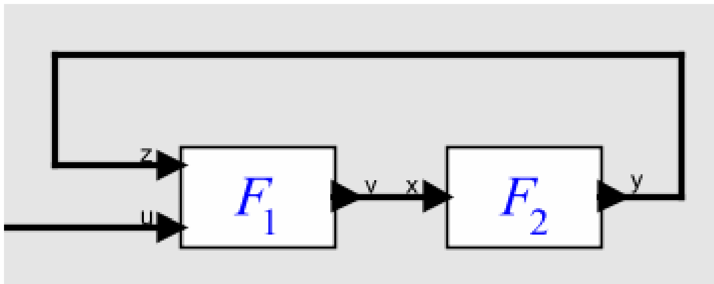




Each signal maps a totally ordered subset of $\mathcal{T}$ into values.

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

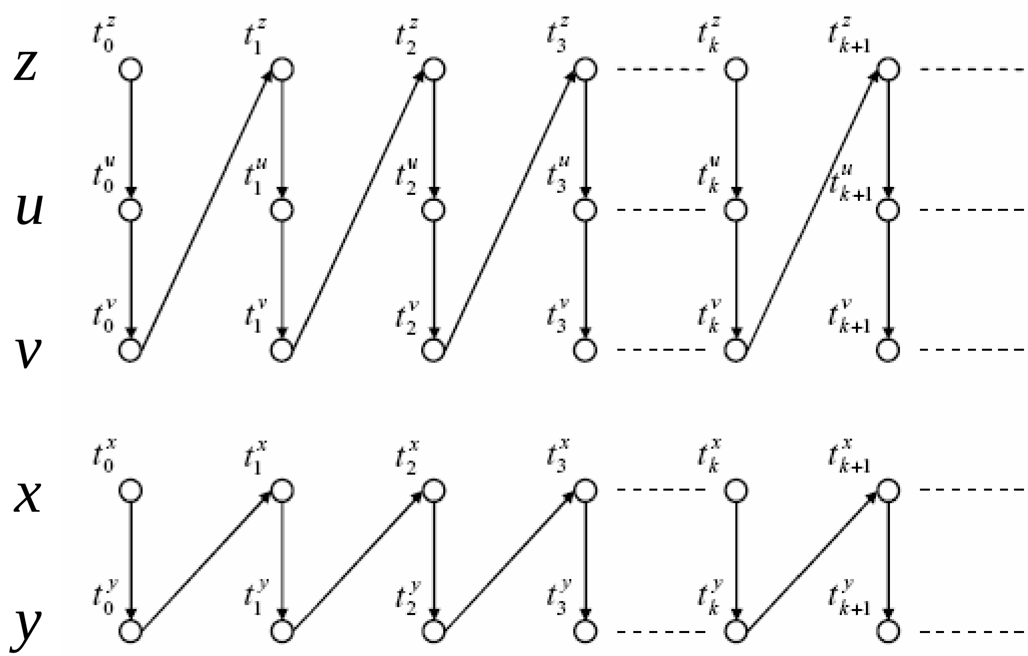
Example: Tag Set *T* for Kahn Process Networks




```
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.

By “Timed Systems” We Mean 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…

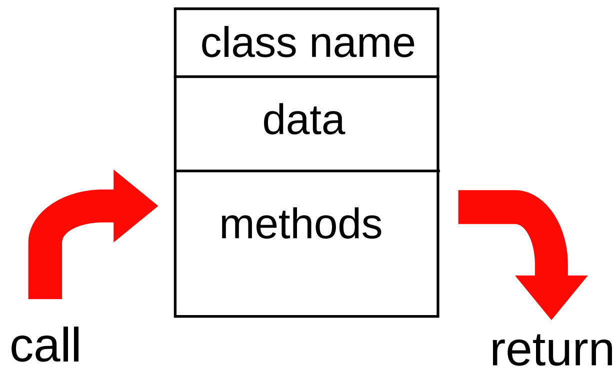
$$F : (\mathcal{T} \rightharpoonup B^{**}) \rightharpoonup (\mathcal{T} \rightharpoonup B^{**})$$

- 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:

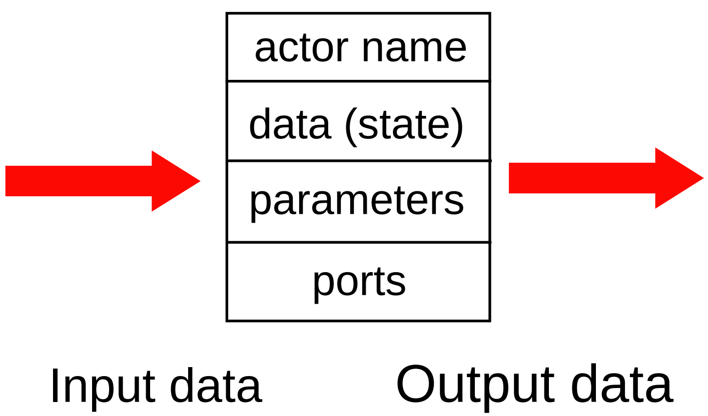


What flows through an object is sequential control

Things happen to objects

The alternative: “Actor oriented:”

Actors make things happen



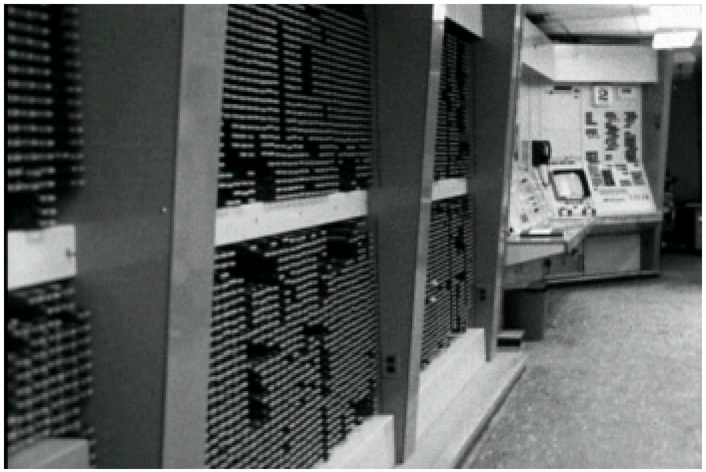
What flows through an object is evolving data

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

The First (?) Actor-Oriented Programming Language

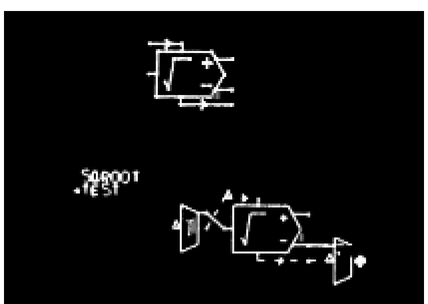
The On-Line Graphical Specification of Computer Procedures

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).

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)
- 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

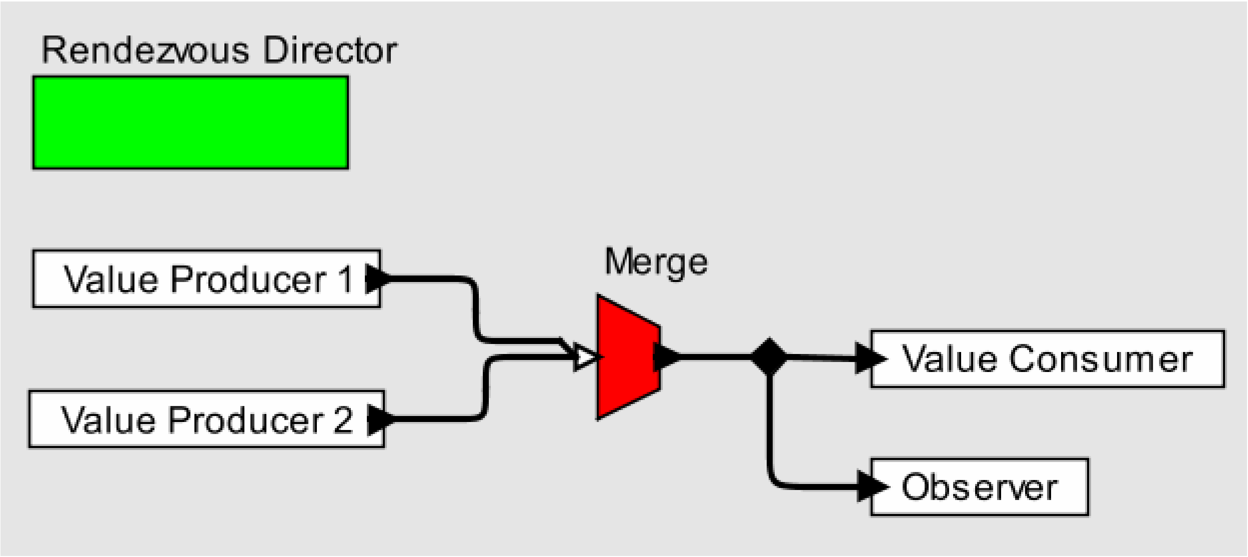
$$F : (\mathcal{T} \rightharpoonup B^{**}) \rightharpoonup (\mathcal{T} \rightharpoonup B^{**})$$

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*

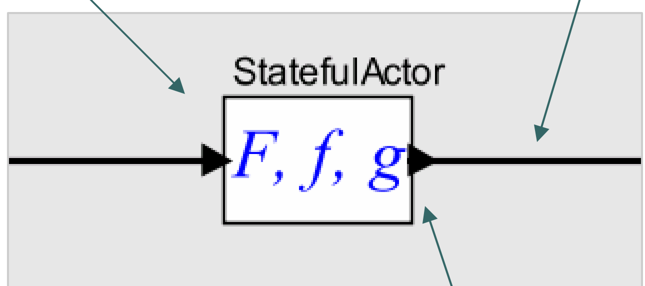
Reconciling Imperative and Actor Semantics: Stateful Actor Abstract Semantics

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).

$$F : S_1 \to S_2$$

$s_1 \in S_1$



$s_2 \in S_2$

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

$$g : S_1 \times \Sigma \to \Sigma$$

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.

The Solution

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

Semantics of these coordination languages should not and need not resort to state transitions.

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

A Challenge for the FORMATS Community

To develop notions of operational semantics, equivalence, and verification that do not rely on transition systems, simulation/bisimulation, and model checking, all of which presuppose a strong notion of system state.

Weak versions of these are available…

Can they be made strong enough?