iCyPhy

Concurrent Models of Computation

Edward A. Lee

Design, Modeling, and Architecture of Complex Industrial Systems (COMASIC)
Masters Program

Université Paris-Saclay

Saclay, France, Jan. 30, 2020

University of California, Berkeley

These slides

Last Week:

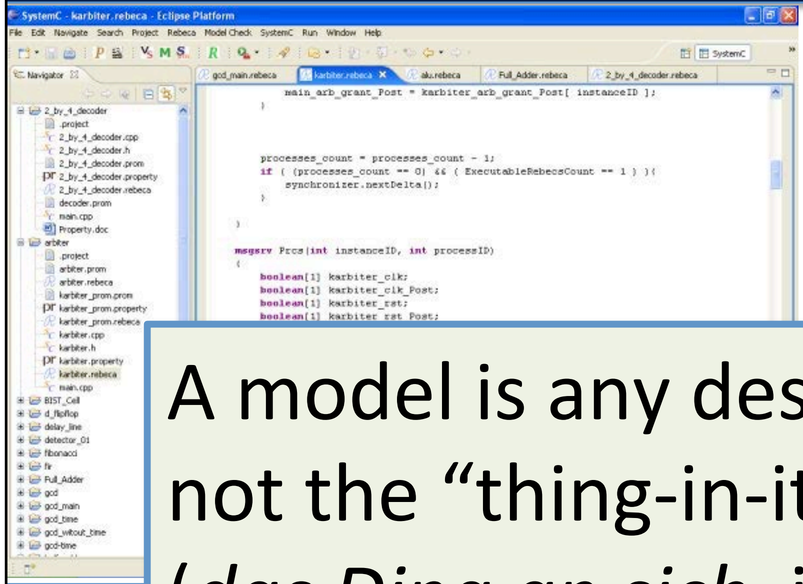
This Week:

http://ptolemy.org/~eal/presentations/Lee_ModelBasedDesignOfCPS_Saclay.pdf

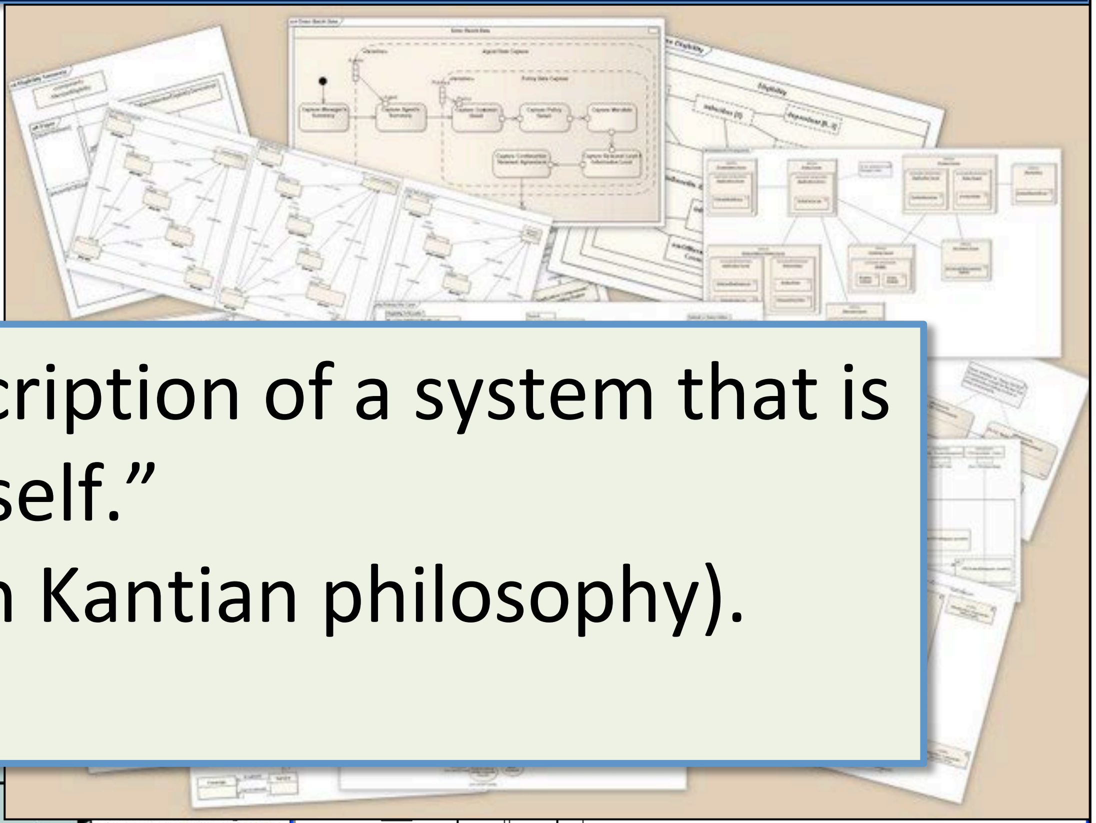
http://ptolemy.org/~eal/presentations/Lee_ConcurrentModelsOfComputation_Saclay.pdf

Outline

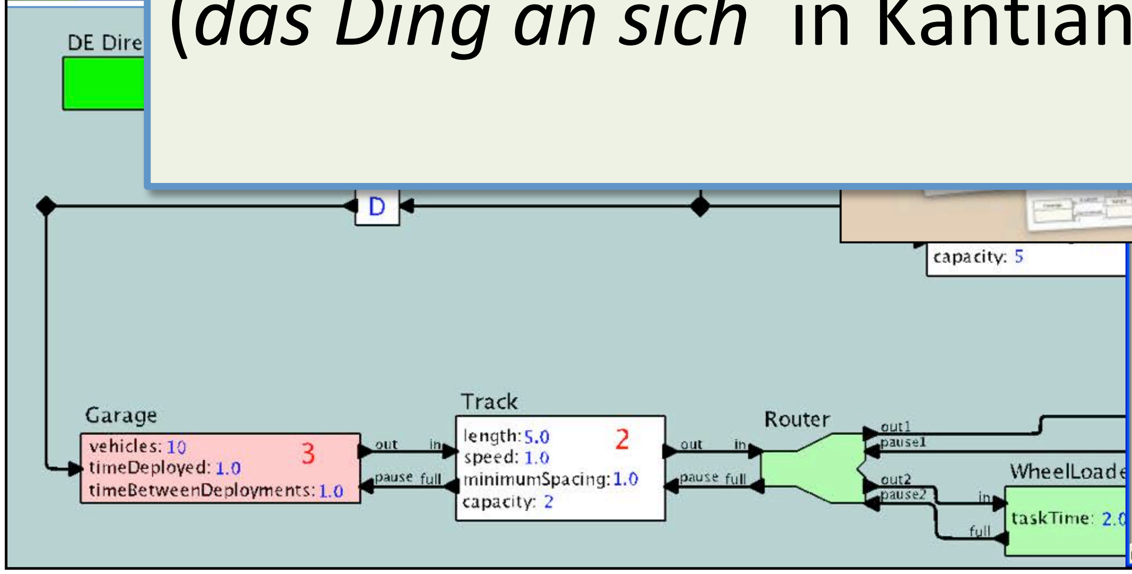
What is a Model?

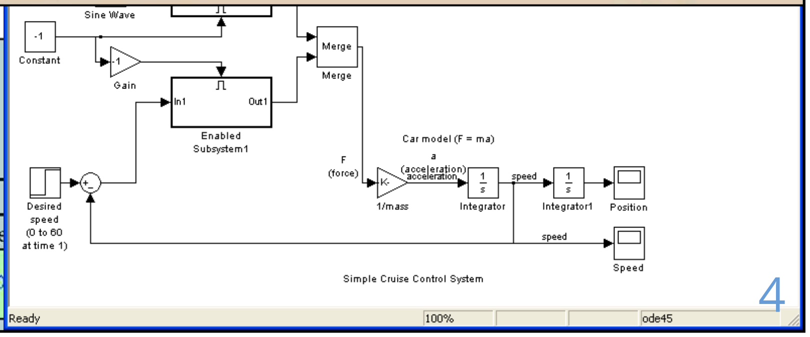


```
        main_arb_grant_Post = karbiter_arb_grant_Post[ instanceID ];
    }



    processes_count = processes_count - 1;
    if ( (processes_count == 0) && ( ExecutableRebecsCount == 1 ) ){
        synchronizer.nextDelta();
    }

}

msgsrv Prcs(int instanceID, int processID)
{
    boolean[1] karbiter_clk;
    boolean[1] karbiter_clk_Post;
    boolean[1] karbiter_rst;
    boolean[1] karbiter_rst_Post;
```



A model is any description of a system that is not the "thing-in-itself."
(*das Ding an sich* in Kantian philosophy).





Model of Computation

US NIST:

- A formal, abstract definition of a computer.

Wikipedia (on 1/18/20):

- a model which describes how an output of a mathematical function is computed given an input.

Example MoCs

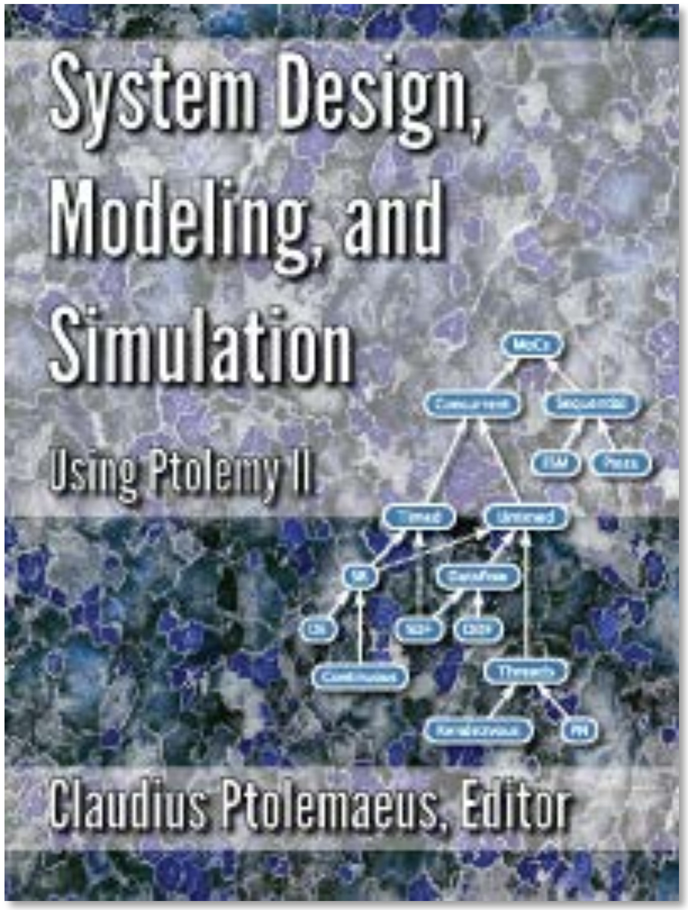
Sequential:
- Finite state machines
- Pushdown automata
- Turing machines

Functional:
- Lambda calculus
- Recursive functions
- Combinatory logic
- Rewriting systems

Concurrent:
- Cellular automata
- Kahn process networks
- Petri nets
- Dataflow
- Actors
- CSP (rendezvous)

Timed & Concurrent:
- Synchronous/Reactive
- Discrete events
- Continuous time

Concurrent MoCs


System Design, Modeling, and Simulation

Using Ptolemy II

Claudius Ptolemaeus, Editor

http://ptolemy.org/systems

“the rules that govern concurrent execution of the components and the communication between components”

Concurrency

From the Latin, concurrere, “run together”

Concurrency

Google:

- the fact of two or more events or circumstances happening or existing at the same time.

Dictionary.com:

- simultaneous occurrence; coincidence.

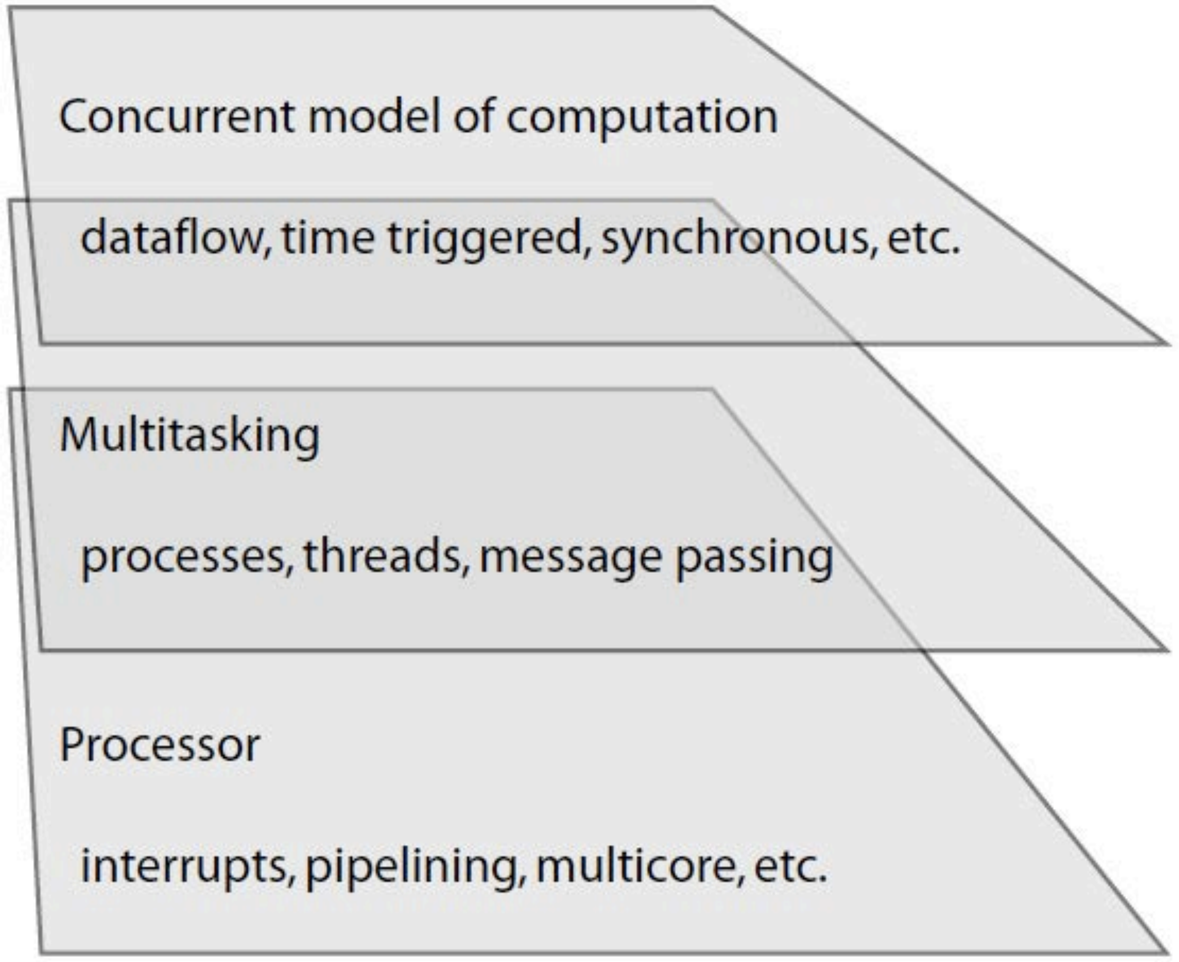
Webster:

- the simultaneous occurrence of events or circumstances

Points of Confusion

- The role of time
- Synchrony and asynchrony
- Concurrent vs. parallel
- Concurrent vs. nondeterministic

Layers of Abstraction for Concurrency in Programs


Concurrent model of computation
dataflow, time triggered, synchronous, etc.
Multitasking
processes, threads, message passing
Processor
interrupts, pipelining, multicore, etc.

Uses of Concurrency in Software

- Reacting to external events (interrupts)
- Exception handling (software interrupts)
- Creating the illusion of simultaneously running different programs (multitasking)
- Exploiting parallelism in the hardware (multicore, VLIW, server farms)
- Dealing with real-time constraints (preemption, deadlines, priorities)
- Distributed computation (networked)

Outline

- Definitions
- Threads
- Alternatives to Threads
- Actors
- Ports, Hierarchy, and Scheduling
- Deterministic Concurrency
- Time
- Discrete-Event Languages
- Reactors

Threads

Threads are sequential concurrent procedures that share memory.

They have been the most commonly used mechanism for building concurrent software, but this is changing, for good reasons.

Processes are collections of threads with their own memory. Communication between processes occurs via OS facilities (like pipes, sockets, or files).

Thread Mechanisms

- Without an OS, multithreading is achieved with interrupts. Timing is determined by external events.
- Generic OSs (Linux, Windows, OSX, …) provide thread libraries (e.g. pthreads) and provide no guarantees about when threads will execute.
- Real-time operating systems (RTOSs), like FreeRTOS, QNX, VxWorks, RTLinux, support a variety of ways of controlling when threads execute (priorities, preemption policies, deadlines, …).

Posix Threads (pthreads)

pthreads is an API (Application Program Interface) implemented by many operating systems, both real-time and not. It is a library of C procedures.

Standardized by the IEEE in 1988 to unify variants of Unix. Subsequently implemented in most other operating systems.

Some languages have threads built in, like Java, which uses pthreads under the hood.

Creating and Destroying Threads

```
#include <pthread.h>

void* threadFunction(void* arg) {
    ...
    return pointerToSomething or NULL;
}

int main(void) {
    pthread_t threadID;
    void* exitStatus;
    int value = something;
    pthread_create(&threadID, NULL, threadFunction, &value);
    ...
    pthread_join(threadID, &exitStatus);
    return 0;
}
```

Can pass in pointers to shared variables.

Can return pointer to something.
Do not return a pointer to an local variable!

Create a thread (may or may not start running!)

Becomes arg parameter to threadFunction.
Why is it OK that this is a local variable?

Return only after all threads have terminated.

What’s Wrong with This?

```
#include <pthread.h>
#include <stdio.h>
void *my_thread() {
 int ret = 42;
 return &ret;
}

int main() {
 pthread_t task_id;
 void *status;
 pthread_create(&task_id, NULL, my_thread, NULL);
 pthread_join(task_id, &status);
 printf("%d\n",*(int*)status); return 0;
}
```

Don’t return a pointer to a local variable, which is on the stack.

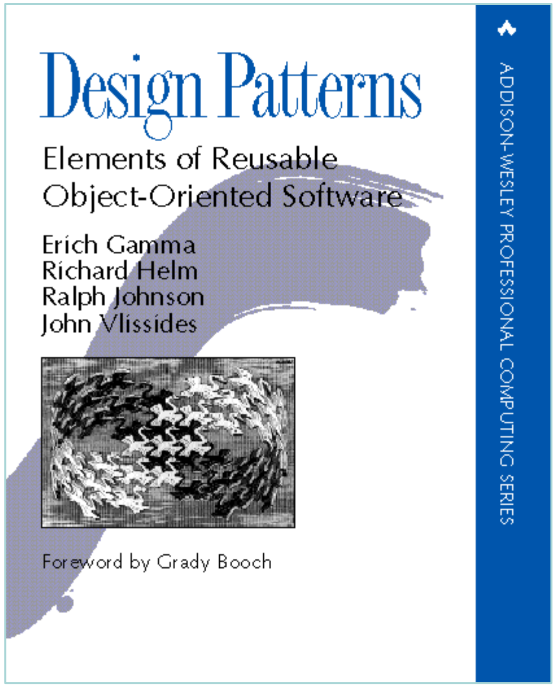
Notes

- Threads can (and often do) share variables
- Threads may or may not begin running immediately after being created.
- A thread may be suspended between any two *atomic* instructions (typically, assembly instructions, not C statements!) to execute another thread and/or interrupt service routine.
- Threads can often be given *priorities*, but these may not be respected by the thread scheduler.
- Threads may *block* on semaphores and mutexes.

A Scenario

Under Integrated Modular Avionics, software in the aircraft engine continually runs diagnostics and publishes diagnostic data on the local network.

Proper software engineering practice suggests using the observer pattern.



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch

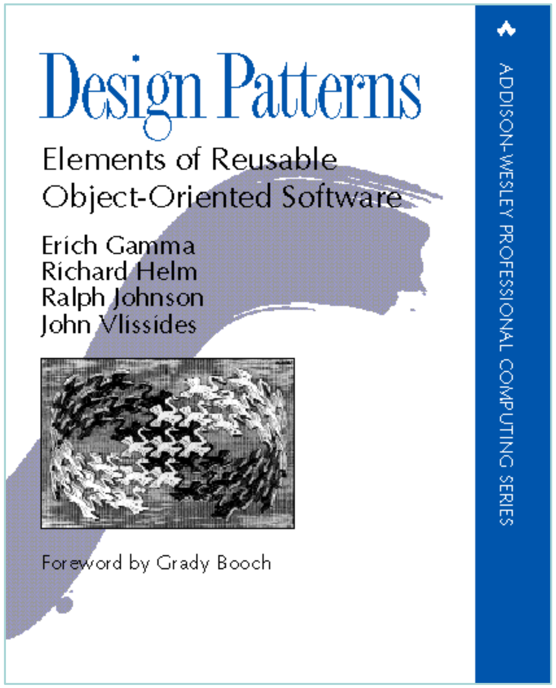
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

An observer process updates the cockpit display based on notifications from the engine diagnostics.

Typical thread programming problem

“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, 1995)


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Observer Pattern in C

```
// Value that when updated triggers notification
// of registered listeners.
int value;

// List of listeners. A linked list containing
// pointers to notify procedures.
typedef void* notifyProcedure(int);
struct element {…}
typedef struct element elementType;
elementType* head = 0;
elementType* tail = 0;

// Procedure to add a listener to the list.
void addListener(notifyProcedure listener) {…}

// Procedure to update the value
void update(int newValue) {…}

// Procedure to call when notifying
void print(int newValue) {…}
```

Observer Pattern in C

```
// Value that when updated triggers notification of registered listeners.
int value;

// List of listeners. A l
// pointers to notify pro
typedef void* notifyProce
struct element {…}
typedef struct element el
elementType* head = 0;
elementType* tail = 0;

// Procedure to add a lis
void addListener(notifyPr

// Procedure to update the value
void update(int newValue) {…}

// Procedure to call when notifying
void print(int newValue) {…}
```

```
typedef void* notifyProcedure(int);
struct element {
  notifyProcedure* listener;
  struct element* next;
};
typedef struct element elementType;
elementType* head = 0;
elementType* tail = 0;
```

Observer Pattern in C

```
// Value that
int value;

// List of lis
// pointers to
typedef void*
struct element
typedef struct
elementType* h
elementType* t

// Procedure t
void addListen

// Procedure t
void update(in

// Procedure t
void print(int newValue) {…}
```

```
// Procedure to add a listener to the list.
void addListener(notifyProcedure listener) {
  if (head == 0) {
    head = malloc(sizeof(elementType));
    head->listener = listener;
    head->next = 0;
    tail = head;
  } else {
    tail->next = malloc(sizeof(elementType));
    tail = tail->next;
    tail->listener = listener;
    tail->next = 0;
  }
}
```

Observer Pattern in C

```
// Value that when updated triggers notification of registered listeners.
int value;

// List of listeners. A linked list containing
// pointers to notify procedures.
typedef void* notifyProcedure(int);
struct element
typedef struct
elementType* h
elementType* t

// Procedure t
void addLister

// Procedure t
void update(ir

// Procedure t
void print(int
```

```
// Procedure to update the value
void update(int newValue) {
  value = newValue;
  // Notify listeners.
  elementType* element = head;
  while (element != 0) {
    (*(element->listener))(newValue);
    element = element->next;
  }
}
```

Observer Pattern in C

```
// Value that when updated triggers notification of registered listeners.
int value;

// List of listeners. A linked list containing
// pointers to notify procedures.
typedef void* notifyProcedure(int);
struct element {…}
typedef struct element elementType;
elementType* head = 0;
elementType* tail = 0;

// Procedure to add a listener to the lis
void addListener(notifyProcedure listener

// Procedure to update the value
void update(int newValue) {…}

// Procedure to call when notifying
void print(int newValue) {…}
```

Will this work in a multithreaded context?

Will there be unexpected/undesirable behaviors?

Observer Pattern in C: How to make this thread safe?

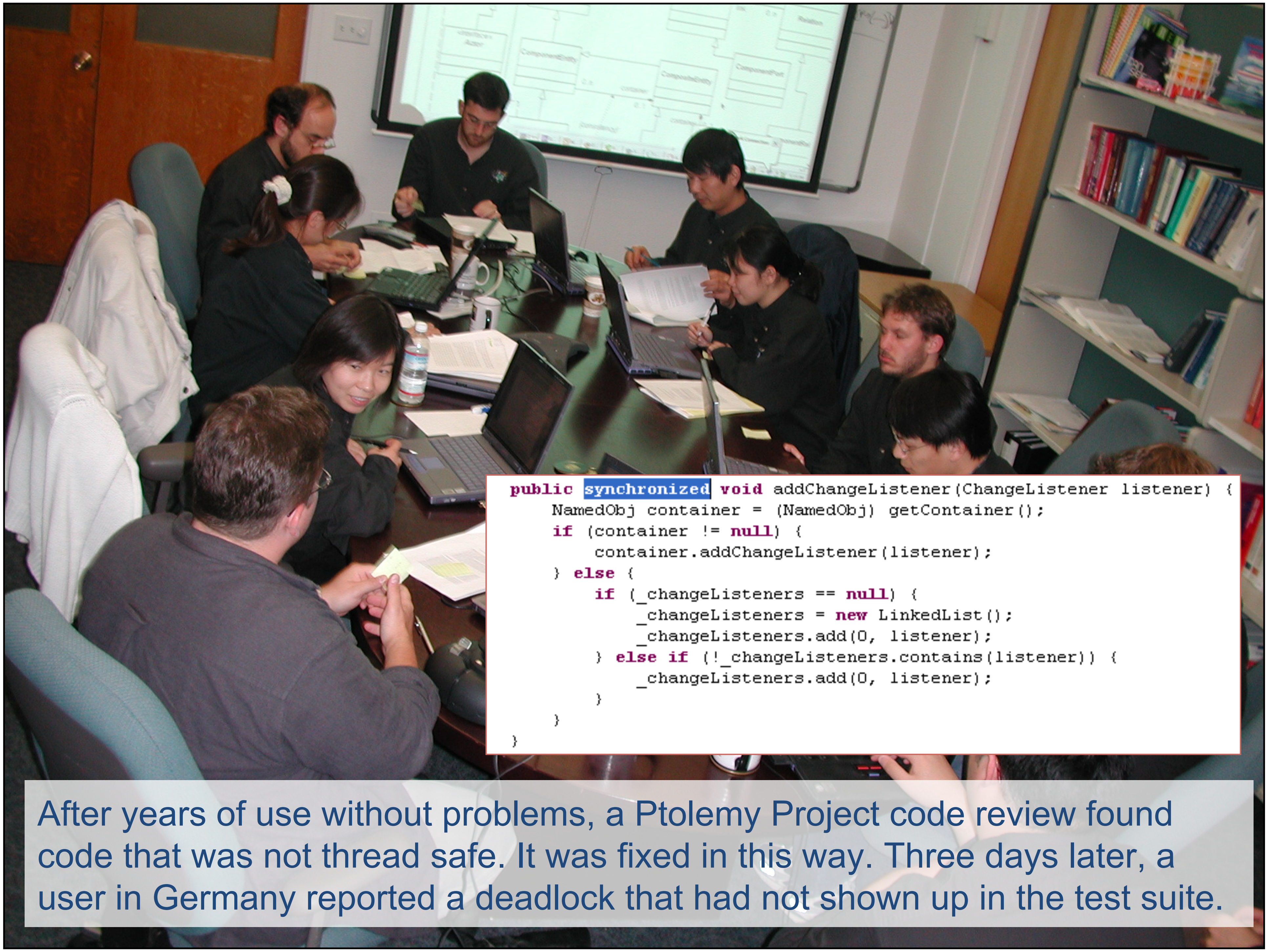
```
// Value that
int value;

// List of lis
// pointers to
typedef void*
struct element
typedef struct
elementType* h
elementType* t

// Procedure t
void addLister

// Procedure t
void update(ir

// Procedure t
void print(int newValue) {…}
```

```
// Procedure to add a listener to the list.
void addListener(notifyProcedure listener) {
  if (head == 0) {
    head = malloc(sizeof(elementType));
    head->listener = listener;
    head->next = 0;
    tail = head;
  } else {
    tail->next = malloc(sizeof(elementType));
    tail = tail->next;
    tail->listener = listener;
    tail->next = 0;
  }
}
```

Using Posix mutexes on the observer pattern in C

```
#include <pthread.h>
...
pthread_mutex_t lock;

void addListener(notify listener) {
  pthread_mutex_lock(&lock);
  ...
  pthread_mutex_unlock(&lock);
}

void update(int newValue) {
  pthread_mutex_lock(&lock);
  value = newValue;
  elementType* element = head;
  while (element != 0) {
    (*(element->listener))(newValue);
    element = element->next;
  }
  pthread_mutex_unlock(&lock);
}

int main(void) {
  pthread_mutex_init(&lock, NULL);
  ...
}
```

However, this carries a significant deadlock risk. The update procedure holds the lock while it calls the notify procedures. If any of those stalls trying to acquire another lock, and the thread holding that lock tries to acquire this lock, deadlock results.



```
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.

One possible "fix"

```
#include <pthread.h>
...
pthread_mutex_t lock;

void addListener(notify listener) {
  pthread_mutex_lock(&lock);
  ...
  pthread_mutex_unlock(&lock);
}

void update(int newValue) {
  pthread_mutex_lock(&lock);
  value = newValue;
  ... copy the list of listeners ...
  pthread_mutex_unlock(&lock);
  elementType* element = headCopy;
  while (element != 0) {
    (*(element->listener))(newValue);
    element = element->next;
  }
}

int main(void) {
  pthread_mutex_init(&lock, NULL);
  ...
}
```

What is wrong with this?

Notice that if multiple threads call update(), the updates will occur in some order. But there is no assurance that the listeners will be notified in the same order. Listeners may be mislead about the "final" value.

This is a very simple, commonly used design pattern. 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…

What it Feels Like to Use Mutexes

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

Claim

Nontrivial software written with threads, semaphores, and mutexes is incomprehensible to humans.

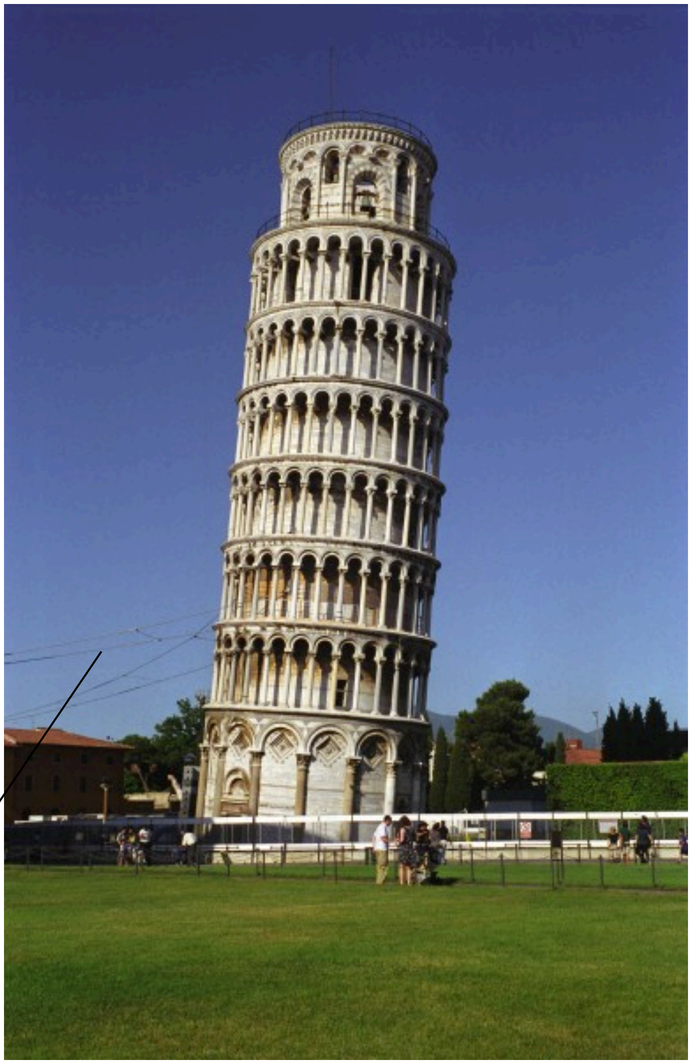
→ *Need better ways to program concurrent systems*

→ *Better tools to analyze and reason about concurrency (e.g. model checking)*

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 thing is held up by threads



Problems with the Foundations

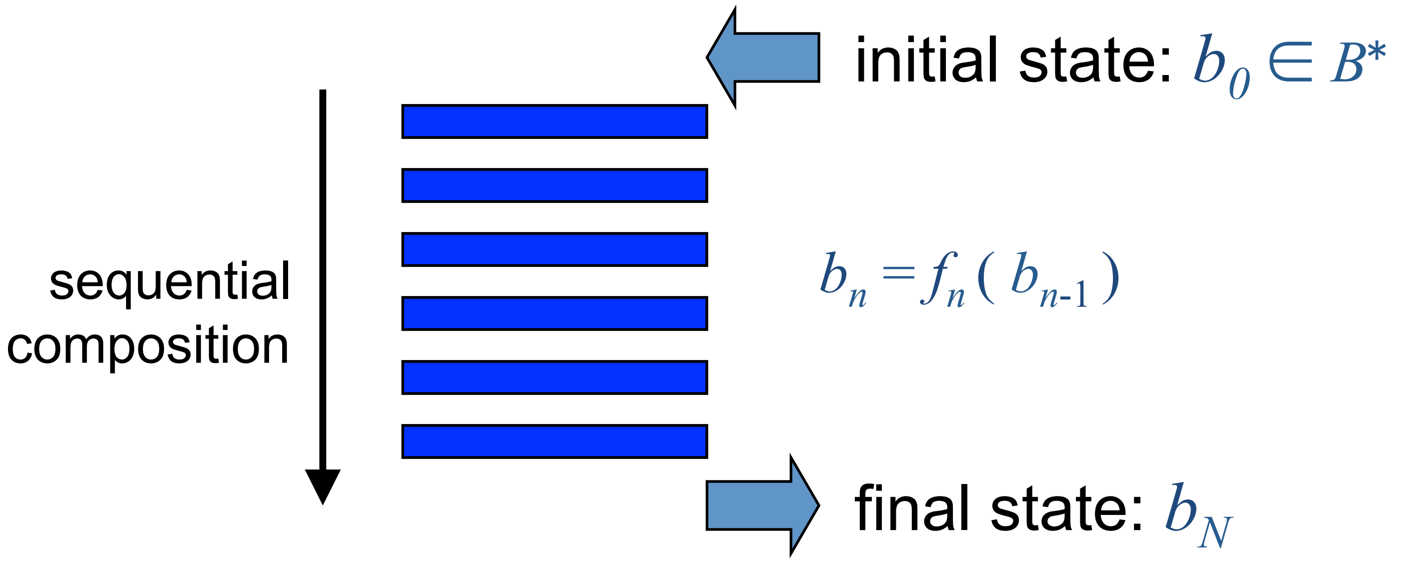
A model of computation:

- 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.

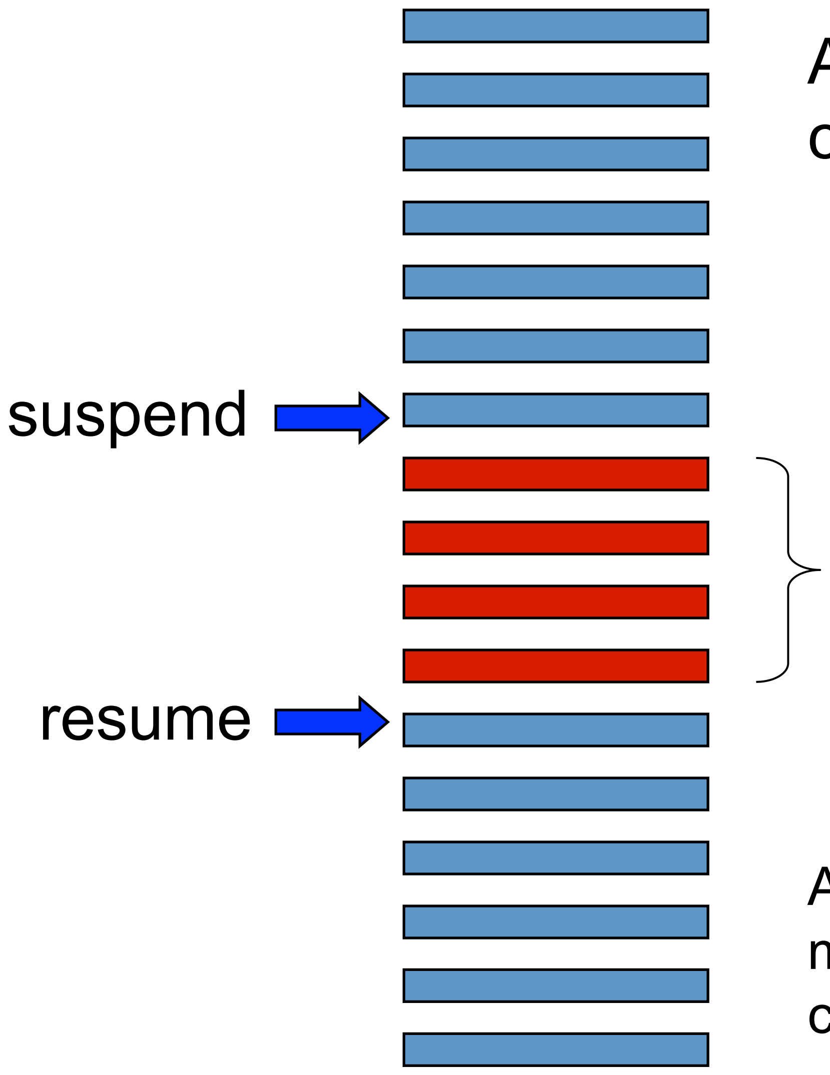
But this model does not admit concurrency gracefully.

Basic Sequential Computation



Formally, composition of computations is function composition.

When There are Threads, Everything Changes



A program no longer computes a function.

$$b_n = f_n(b_{n-1})$$

another thread can change the state

$$b'_n = f_n(b'_{n-1})$$

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

Succinct Problem Statement

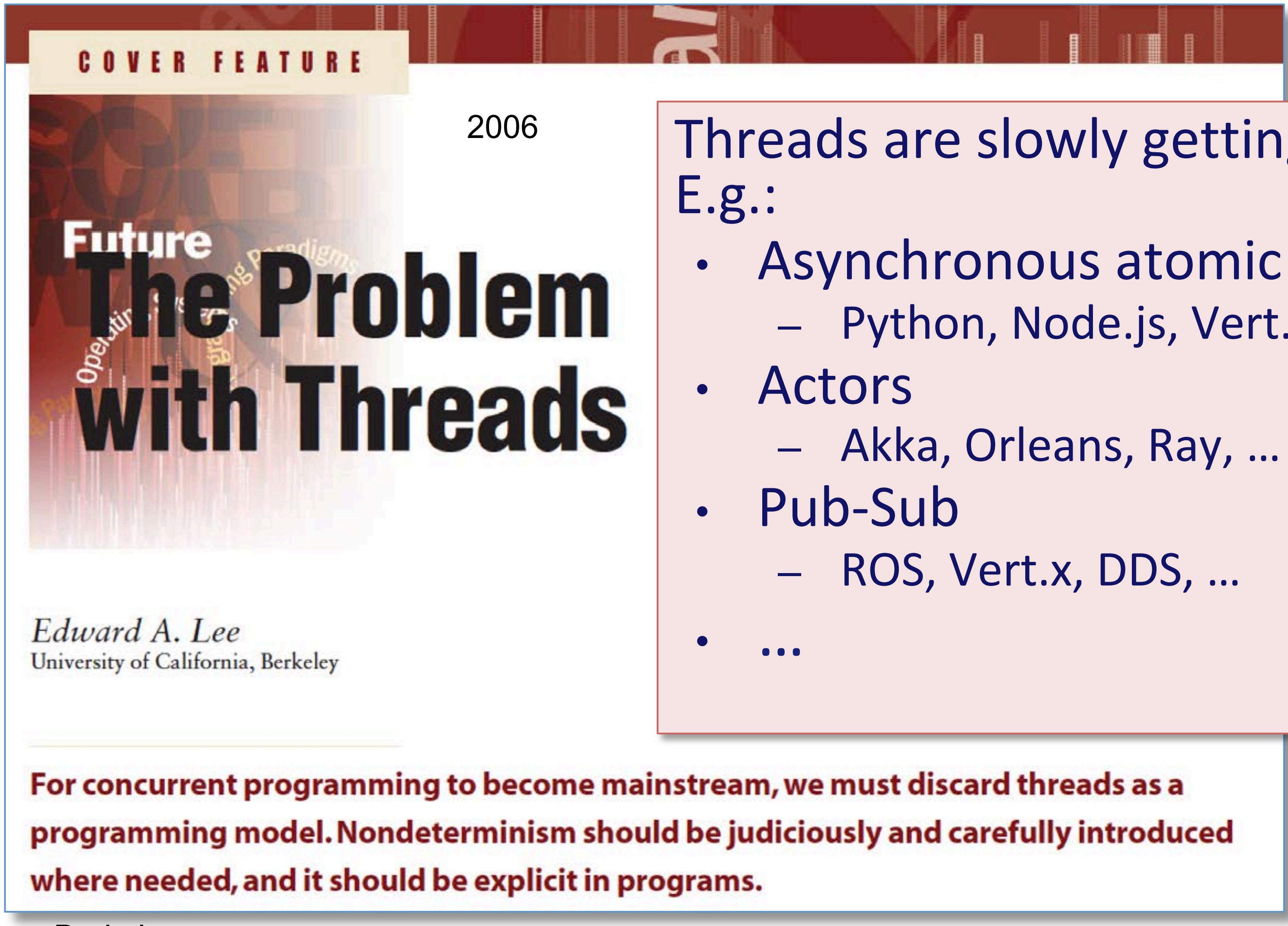
Threads are wildly nondeterministic.

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).

Incremental Improvements to Threads

- Object Oriented programming
- Coding rules (Acquire locks in the same order…)
- Libraries (Stapl, Java concurrent collections, …)
- Message passing (Actors, ...)
- Publish and subscribe (ROS, MQTT, DDS, ...)
- Transactions (Databases, …)
- Patterns (MapReduce, …)
- Formal verification (Model checking, …)
- Enhanced languages (Split-C, Cilk, Guava, …)
- Enhanced mechanisms (Promises, futures, asynchronous atomic callbacks …)

Threads: An Unnecessary Source of Nondeterminism in Software



COVER FEATURE

2006

Future
Operating
Paradigms

The Problem with Threads

Edward A. Lee
University of California, Berkeley

For concurrent programming to become mainstream, we must discard threads as a programming model. Nondeterminism should be judiciously and carefully introduced where needed, and it should be explicit in programs.

Threads are slowly getting replaced. E.g.:

- Asynchronous atomic callbacks
  - Python, Node.js, Vert.x, …
- Actors
  - Akka, Orleans, Ray, …
- Pub-Sub
  - ROS, Vert.x, DDS, …
- …

Message-passing programs *may* be better

```
void* producer(void* arg) {
    int i;
    for (i = 0; i < 10; i++) {
        send(i);
    }
    return NULL;
}
void* consumer(void* arg) {
    while(1) {
        printf("received %d\n", get());
    }
    return NULL;
}
int main(void) {
    pthread_t threadID1, threadID2;
    void* exitStatus;
    pthread_create(&threadID1, NULL, producer, NULL);
    pthread_create(&threadID2, NULL, consumer, NULL);
    pthread_join(threadID1, &exitStatus);
    pthread_join(threadID2, &exitStatus);
    return 0;
}
```

But there is still risk of deadlock and unexpected nondeterminism!

Recall Challenge Problem

A software component on a microprocessor in an aircraft door provides two network services:

1. "open"
2. "disarm"

Assume state is closed and armed.

What should it do when it receives a request "open"?

Image by Christopher Doyle from Horley, United Kingdom - A321 Exit Door, CC BY-SA 2.0

Recall Challenge Problem

A software component on a microprocessor in an aircraft door provides two network services:

1. “open”
2. “disarm”

Assume state is closed and armed.

What should it do when it receives a request “open”?

Image from *The Telegraph*, Sept. 9, 2015

Outline

- Definitions
- Threads
- Alternatives to Threads
- Actors
- Ports, Hierarchy, and Scheduling
- Deterministic Concurrency
- Time
- Discrete-Event Languages
- Reactors

Asynchronous Atomic Callbacks

- Main event loop.
- Event handlers (“callbacks”) run to completion atomically.

Augment with worker threads that communicate with:

- Immutable data
- Publish-and-subscribe busses

Asynchronous Atomic Callbacks: Periodic Actions

```
var x = 0;
function increment() {
  x = x + 1;
}
function decrement() {
  x = x - 2;
}
function observe() {
  console.log(x);
}
setInterval(increment, 1000);
setInterval(decrement, 2000);
setInterval(observe, 4000);
```

- Shared variable x
- Timed actions on x

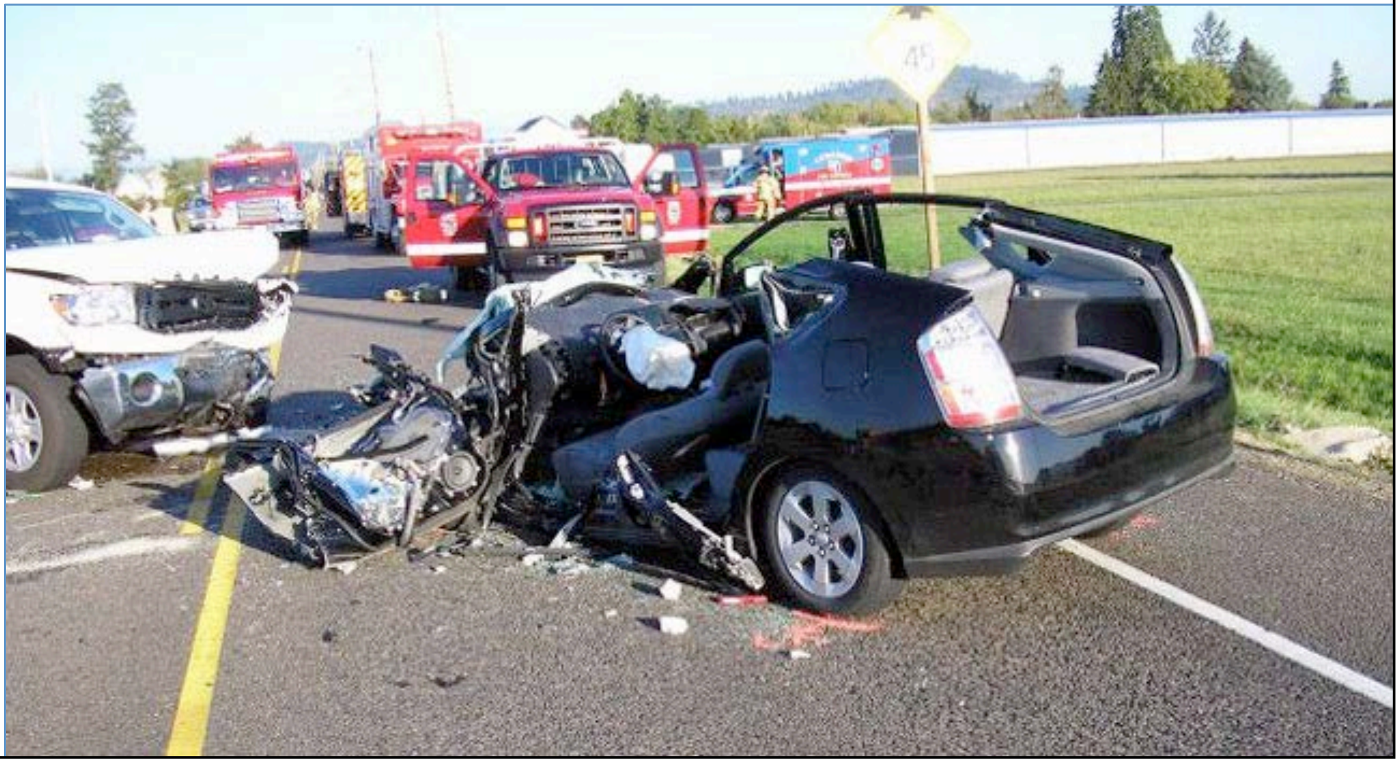
- +1 every second
- −2 every two seconds
- Observe every 4 seconds

On Node.js v5.3.0, MacOS Sierra:

```
0, 0, 0, 0, 0, −1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, …
```

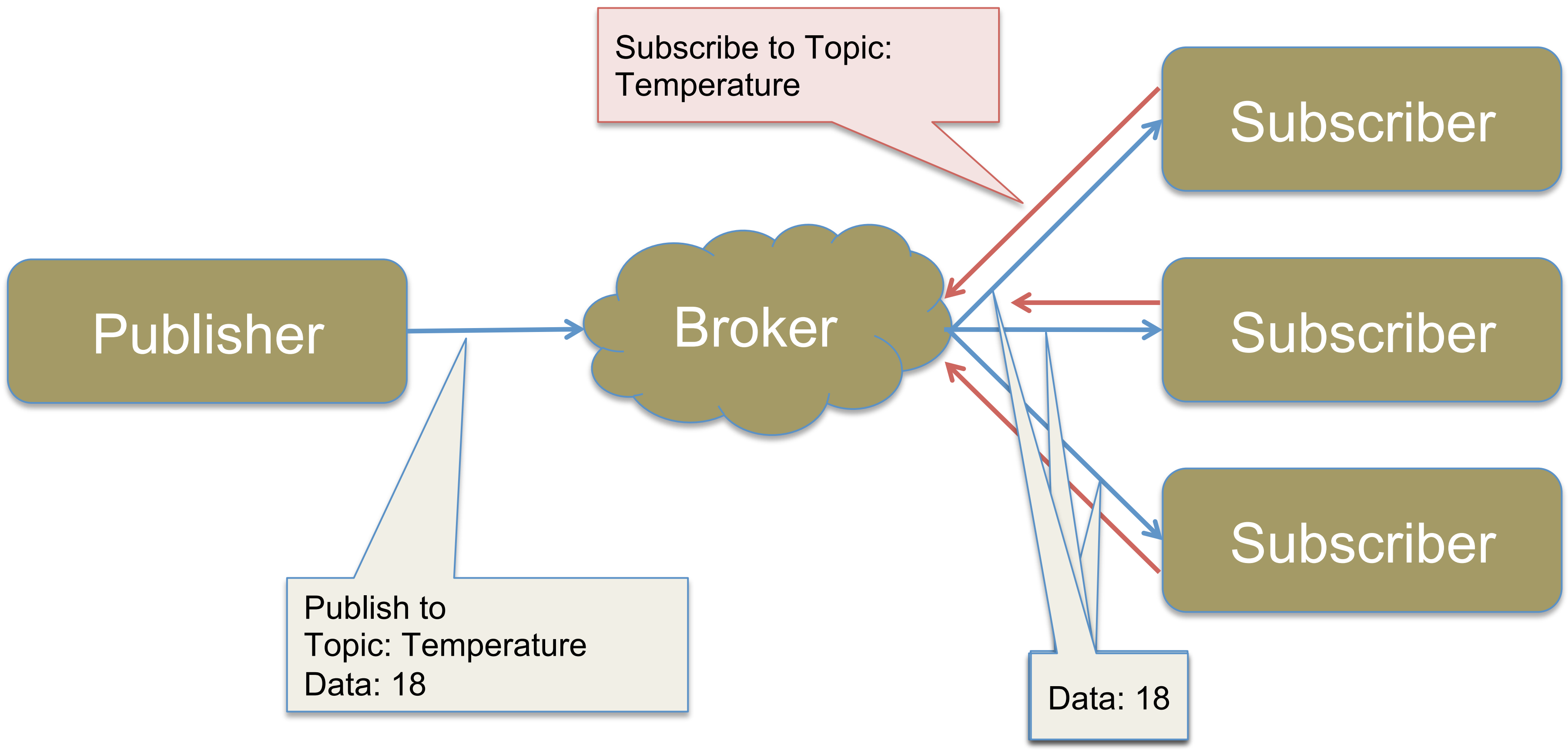
“Toyota” Style of Design

NASA's Toyota Study Released by Dept. of Transportation released in 2011 found that Toyota software was “untestable.”



Possible victim of unintended acceleration.

Publish and Subscribe (Pub Sub)


Subscribe to Topic: Temperature
Subscriber
Publisher
Broker
Subscriber
Subscriber
Publish to Topic: Temperature Data: 18
Data: 18

ROS DDS XMPP MQTT.ORG

Etc.

Recall Challenge Problem

A software component on a microprocessor in an aircraft door provides two network services:

1. "open"
2. "disarm"

Assume state is closed and armed.

What should it do when it receives a request "open"?

Image by Christopher Doyle from Horley, United Kingdom - A321 Exit Door, CC BY-SA 2.0

Recall Challenge Problem

A software component on a microprocessor in an aircraft door provides two network services:

1. "open"
2. "disarm"

Assume state is closed and armed.

What should it do when it receives a request "open"?

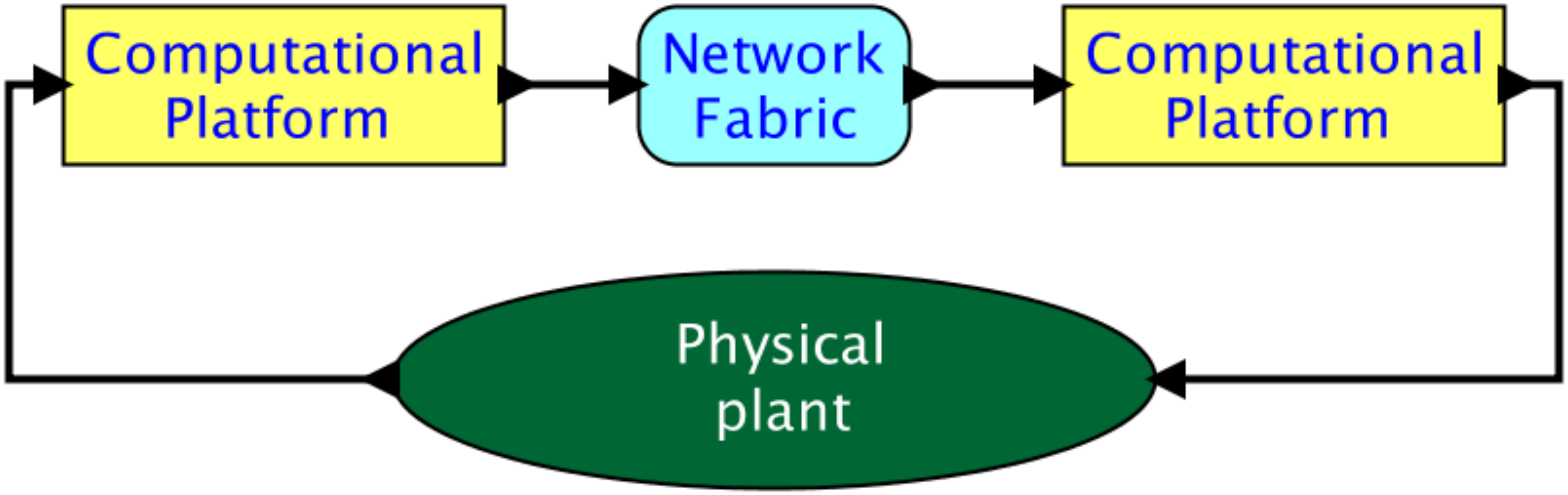
Image from *The Telegraph*, Sept. 9, 2015

Another Answer to Threads: Actors

Actors are concurrent objects that communicate by sending each other messages.

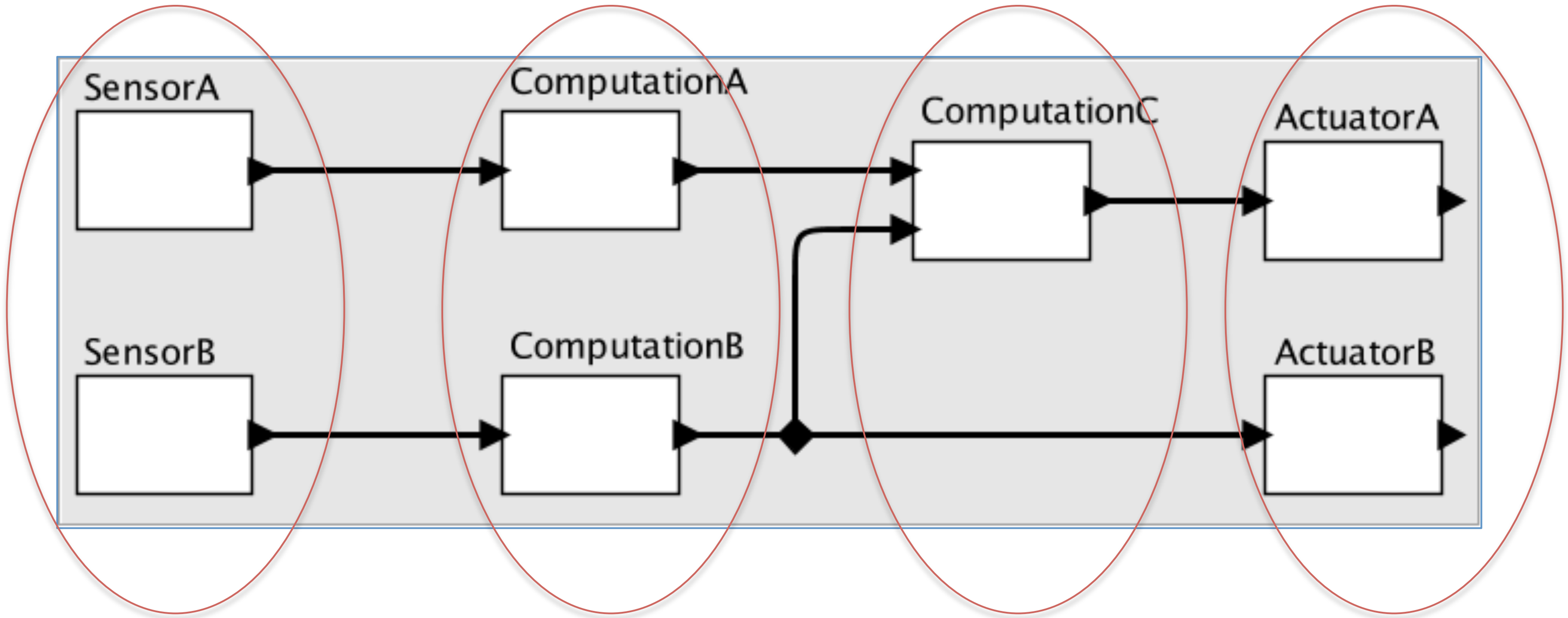
- Erlang [Armstrong, et al. 1996]
- Rebeca [Sirjani and Jaghoori, 2011]
- Akka [Roestenburg, et al. 2017]
- Ray [Moritz, et al. 2018]
- …

Cyber Physical Systems Demand More



Predictability requires determinacy and depends on timing, including execution times and network delays.

Motivation: Some Questions of Interest



What combinations of periodic, sporadic, arrival curve behaviors are manageable?

How do execution times affect feasibility? How can we know execution times?

How do we get repeatable and testable behavior even when communication is across networks?

How do we specify, ensure, and enforce deadlines?

Outline

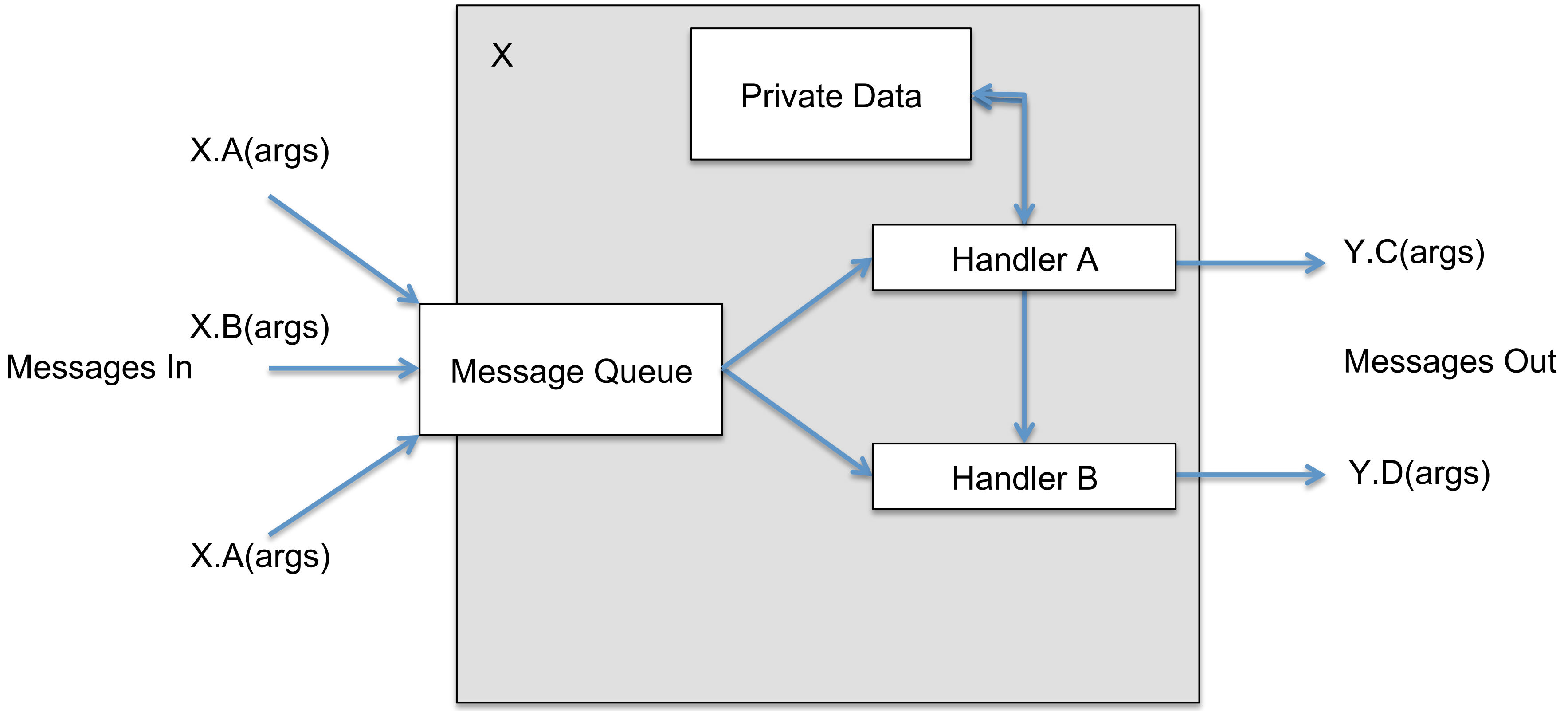
- Definitions
- Threads
- Alternatives to Threads
- Actors
- Ports, Hierarchy, and Scheduling
- Deterministic Concurrency
- Time
- Discrete-Event Languages
- Reactors

Actors, Loosely

Actors are concurrent objects that communicate by sending each other messages.

Hewitt/Agha Actors

Data + Message Handlers



[Hewitt, 1977] [Agha, 1986, 1990, 1997]

Example

An actor with simple operations on its state:

```
Actor Foo {
    int state = 1;
    handler double(){
        state *= 2;
    }
    handler increment(arg){
        state += arg;
        print state;
    }
}
```

Example

An actor that uses actor Foo:

```
Actor Bar {
   handler main(){
      Foo x = new Foo();
      x.double();
      x.increment(1);
   }
}
```

Semantics is “send and forget.”

Composition

```
Actor Bar {
   handler main(){
      Foo x = new Foo();
      x.double();
      x.increment(1);
   }
}
```

```
Actor Foo {
     int state = 1;
     handler double(){
          state *= 2;
     }
     handler increment(arg){
          state += arg;
          print state;
     }
}
```

What is printed?

Pass-Through Actor

Baz: Given an actor of type Foo, send it “double”:

```
Actor Baz {
   handler pass(Foo x){
      x.double();
   }
}
```

New Composition

```
Actor Bar {
    handler main(){
        Foo x = new Foo();
        Baz z = new Baz();
        z.pass(x);
        x.increment(1);
    }
}
```

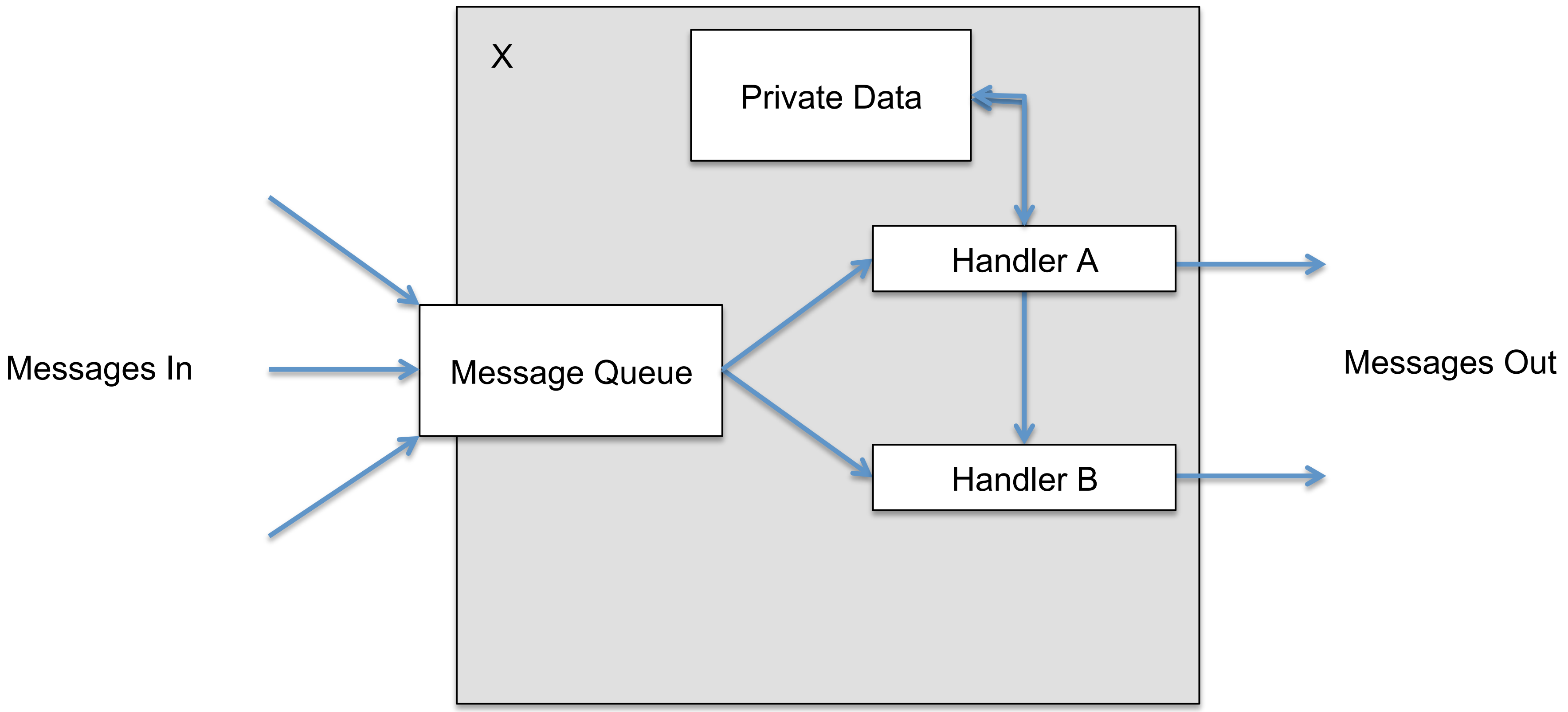
```
Actor Baz {
    handler pass(Foo x){
        x.double();
    }
}
```

What is printed?

```
Actor Foo {
        int state = 1;
        handler double(){
                state *= 2;
        }
        handler increment(arg){
                state += arg;
                print state;
        }
}
```

Hewitt/Agha Actors are Not Predictable

Messages are handled in nondeterministic order.



Aside: Innovation in Ray

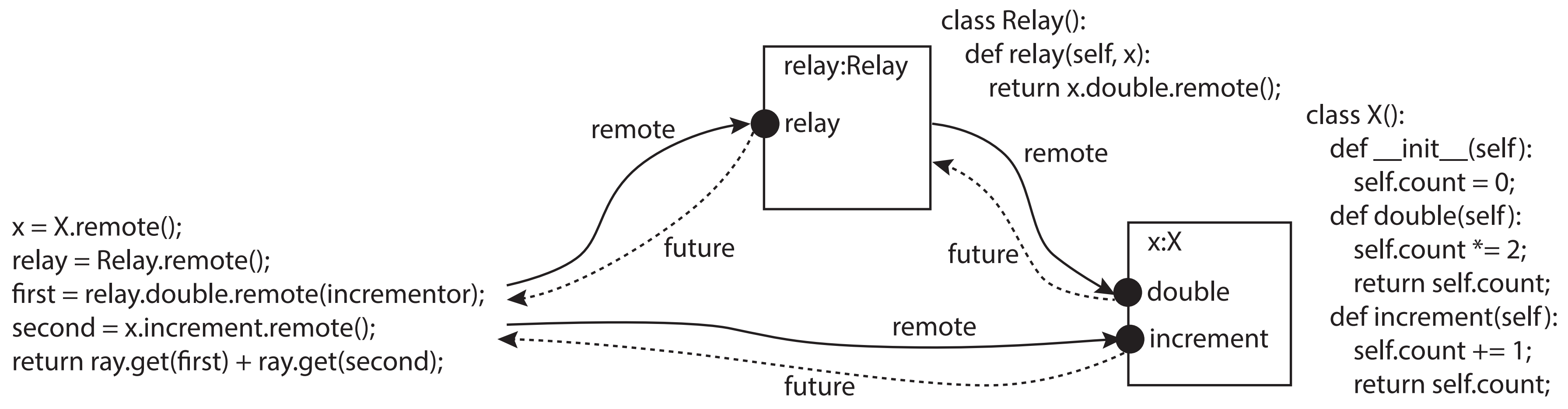
[Moritz, et al. 2017]

Messages can return “futures”:

```
Actor Bar {
   handler main(){
      Foo x = new Foo();
      Future a = x.double();
      Future b = x.increment(1);
      print a.get() + b.get();
   }
}
```

Semantics is still “send and forget,” but later remember.

Unexpected Nondeterminism Example from Ray



```
x = X.remote();
relay = Relay.remote();
first = relay.double.remote(incrementor);
second = x.increment.remote();
return ray.get(first) + ray.get(second);
```

```
class Relay():
  def relay(self, x):
    return x.double.remote();
```

```
class X():
  def __init__(self):
    self.count = 0;
  def double(self):
    self.count *= 2;
    return self.count;
  def increment(self):
    self.count += 1;
    return self.count;
```

The Relay actor is the actor version of a "no op," but it makes the program nondeterministic.

[Moritz, et al., "Ray: A Distributed Framework for Emerging AI Applications" arXiv, 2018]

One Solution: Analyze and Use Dependencies

```
Actor Bar {
   handler main(){
      Foo x = new Foo();
      Baz z = new Baz();
      z.pass(x);
      x.increment(1);
   }
}
```

```
Actor Baz {
   handler pass(Foo x){
      x.double();
   }
}
```

But how? Where is the dependence graph?

```
Actor Foo {
      int state = 1;
      handler double(){
            state *= 2;
      }
      handler increment(arg){
            state += arg;
            print state;
      }
}
```

One Solution: Analyze and Use Dependencies

```
Actor Bar {
   handler main(){
      Foo x = new Foo();
      Baz z = new Baz();
      z.pass(x);
      x.increment(1);
   }
}
```

```
Actor Baz {
   handler pass(Foo x){
      if (something) {
         x.double();
      }
   }
}
```

And what if the dependence graph is data dependent?

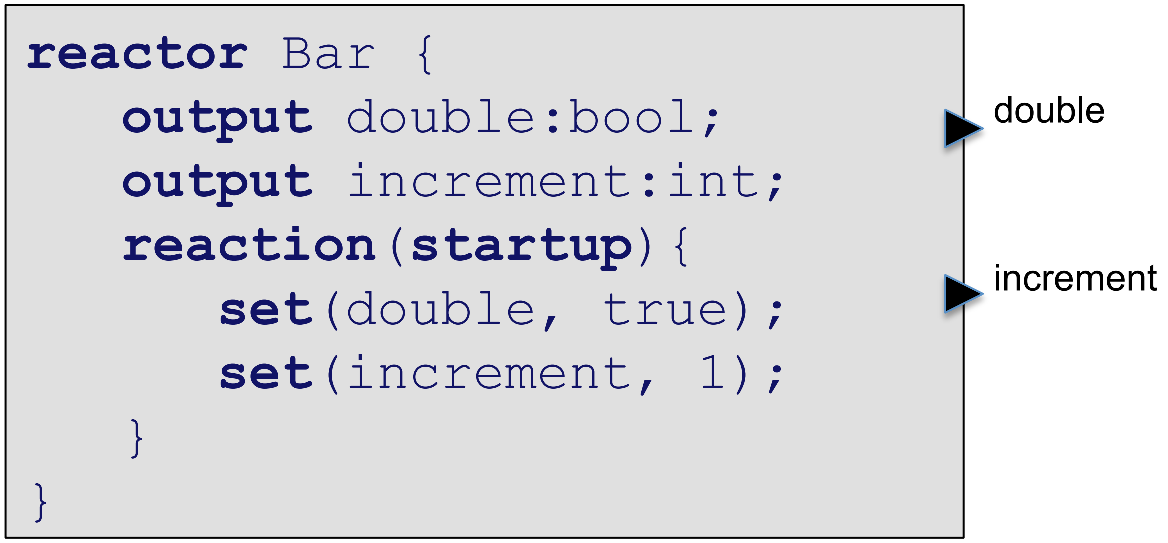
```
Actor Foo {
      int state = 1;
      handler double(){
            state *= 2;
      }
      handler increment(arg){
            state += arg;
            print state;
      }
}
```

Outline

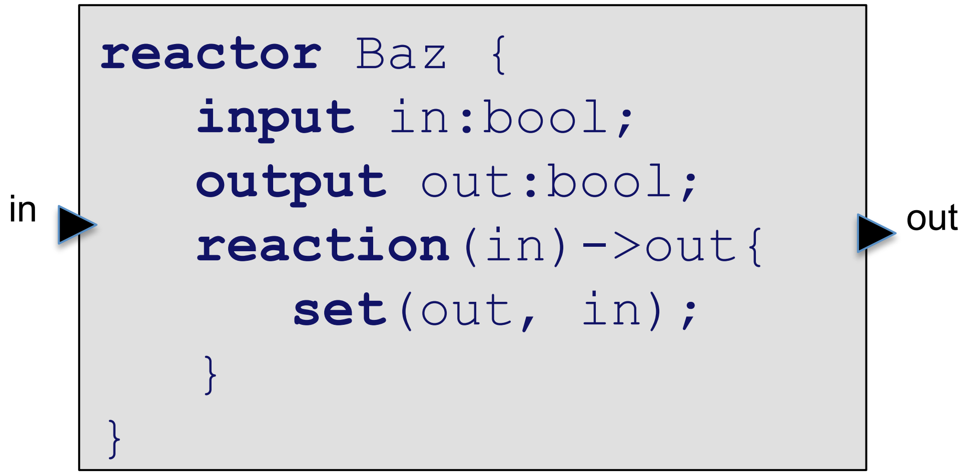
- Definitions
- Threads
- Alternatives to Threads
- Actors
- Ports, Hierarchy, and Scheduling
- Deterministic Concurrency
- Time
- Discrete-Event Languages
- Reactors

Part 1 of our Solution: Ports

Instead of referring to other actors, an actor refers to its own ports.



```
reactor Bar {
    output double:bool;
    output increment:int;
    reaction(startup){
        set(double, true);
        set(increment, 1);
    }
}
```

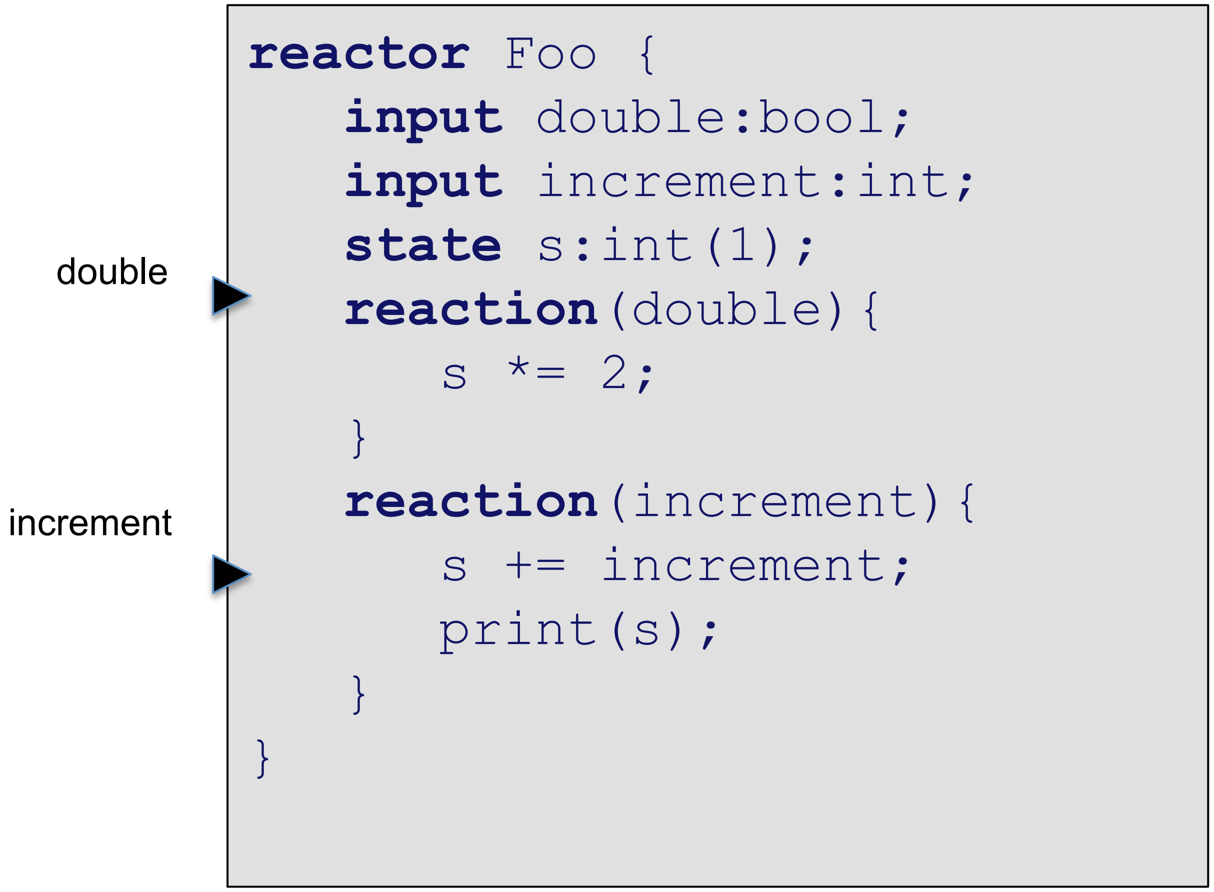


```
reactor Baz {
    input in:bool;
    output out:bool;
    reaction(in)->out{
        set(out, in);
    }
}
```

[Ptolemeus, 2014]

Part 1 of our Solution: Ports

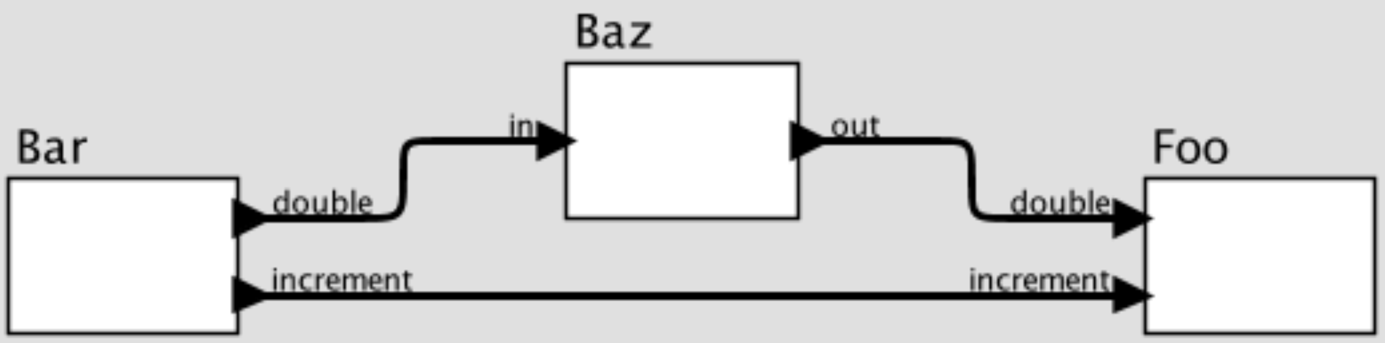
Input ports do not look much different from ordinary message handlers.



```
reactor Foo {
    input double:bool;
    input increment:int;
    state s:int(1);
    reaction(double){
        s *= 2;
    }
    reaction(increment){
        s += increment;
        print(s);
    }
}
```

Part 2 of our Solution: Hierarchy

```
main reactor Top {
    x = new Foo();
    y = new Bar();
    z = new Baz();
    y.double -> z.in;
    y.increment -> x.increment;
    z.out -> x.double;
}
```

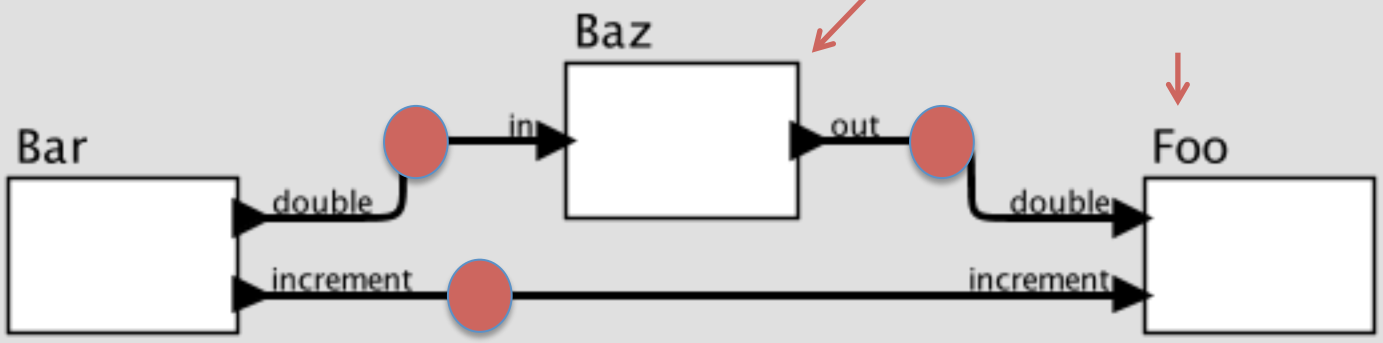


Part 3 of our Solution: Scheduling

```
main reactor Top {
    x = new Foo();
    y = new Bar();
    z = new Baz();
    y.double -> z.in;
    y.increment -> x.increment;
    z.out -> x.double;
}
```

Scheduling becomes especially interesting when production or consumption of messages is data dependent.

Ensure that Baz completes before Foo's handlers are invoked.



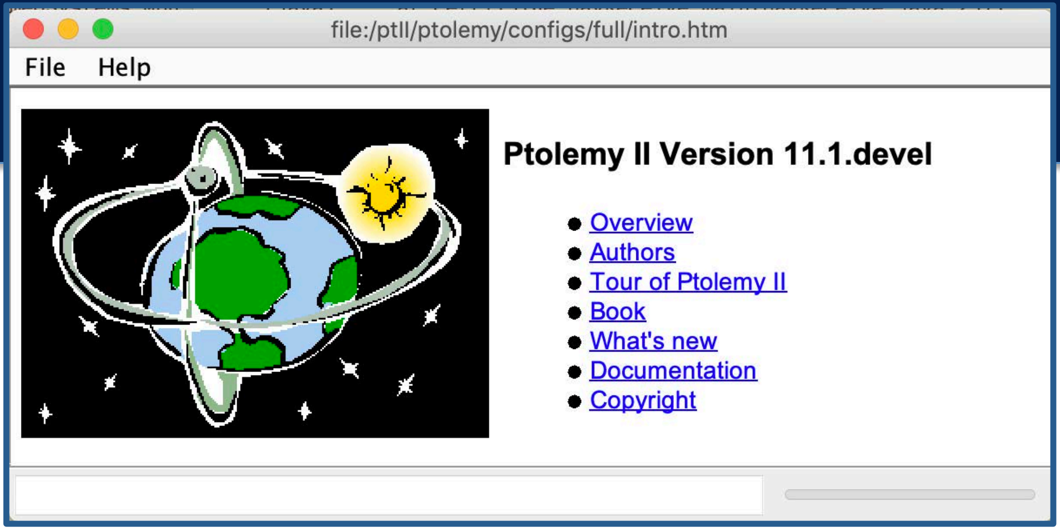
Outline

- Definitions
- Threads
- Alternatives to Threads
- Actors
- Ports, Hierarchy, and Scheduling
- Deterministic Concurrency
- Time
- Discrete-Event Languages
- Reactors

Towards Deterministic Concurrency

- Dataflow (DF)
- Process Networks (PN)
- Synchronous/Reactive (SR)
- Discrete Events (DE)

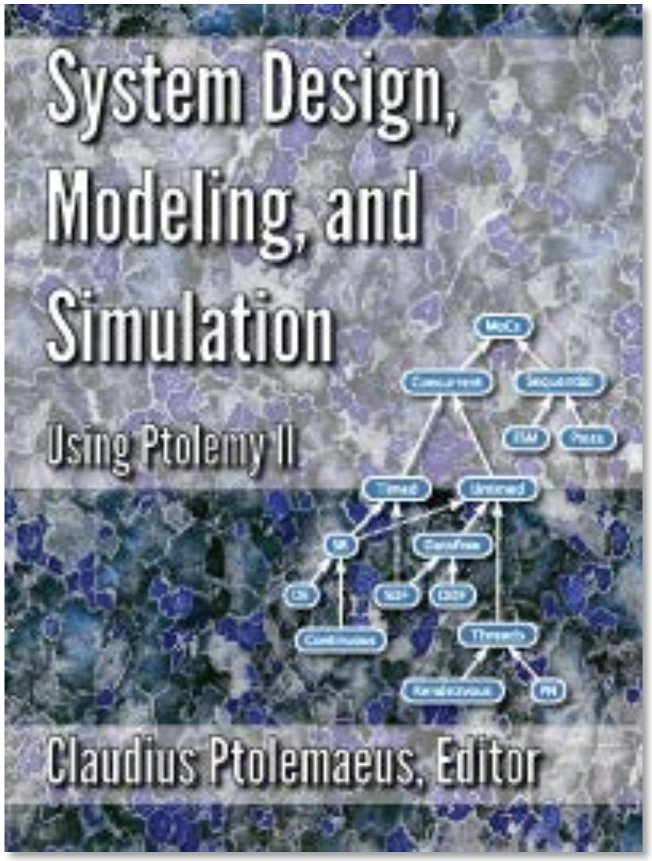
Ptolemy II



file:/ptII/ptolemy/configs/full/intro.htm

Ptolemy II Version 11.1.devel

- Overview
- Authors
- Tour of Ptolemy II
- Book
- What's new
- Documentation
- Copyright


System Design, Modeling, and Simulation
Using Ptolemy II
Claudius Ptolemaeus, Editor

http://ptolemy.org

Ptolemy II has implementations of all of these and a few more with extensive demos.

Towards Deterministic Concurrency

- Dataflow (DF)
- Process Networks (PN)
- Synchronous/Reactive (SR)
- Discrete Events (DE)

Dataflow

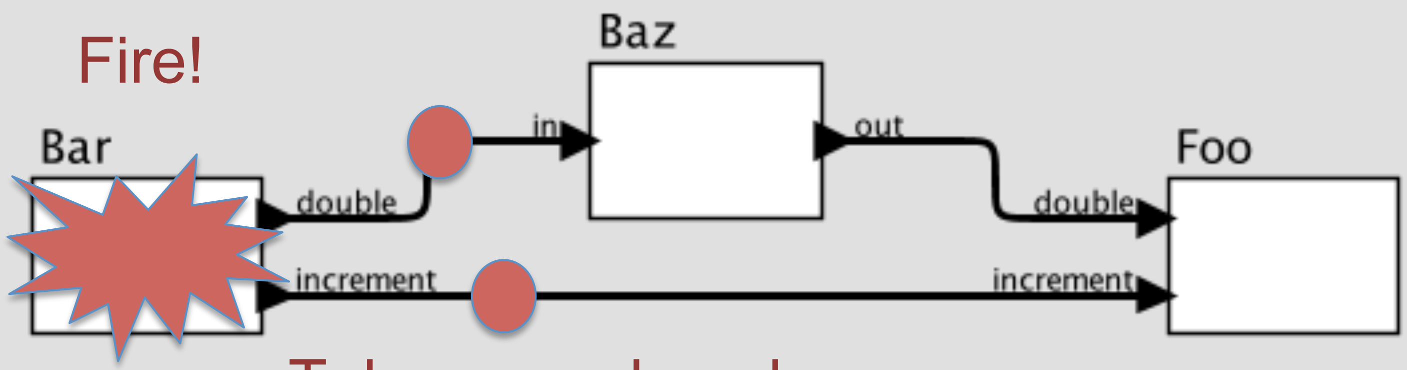
- Computation Graphs [Karp, 1966]
- Dataflow [Dennis, 1974]
- Dynamic dataflow [Arvind, 1981]
- Structured dataflow [Matwin & Pietrzykowski 1985]
- K-bounded loops [Culler, 1986]
- Synchronous dataflow [Lee & Messerschmitt, 1986]
- Structured dataflow and LabVIEW [Kodosky, 1986]
- PGM: Processing Graph Method [Kaplan, 1987]
- Dataflow synchronous languages [Lustre, Signal, 1980's]
- Well-behaved dataflow [Gao, 1992]
- Boolean dataflow [Buck and Lee, 1993]
- Multidimensional SDF [Lee, 1993]
- Cyclo-static dataflow [Lauwereins, 1994]
- Integer dataflow [Buck, 1994]
- Bounded dynamic dataflow [Lee and Parks, 1995]
- Heterochronous dataflow [Girault, Lee, & Lee, 1997]
- …

Jack Dennis

Dataflow Solution for Scheduling: Firing Rules

[Lee & Matsikoudis, 2009]

An actor with no inputs can fire at any time.

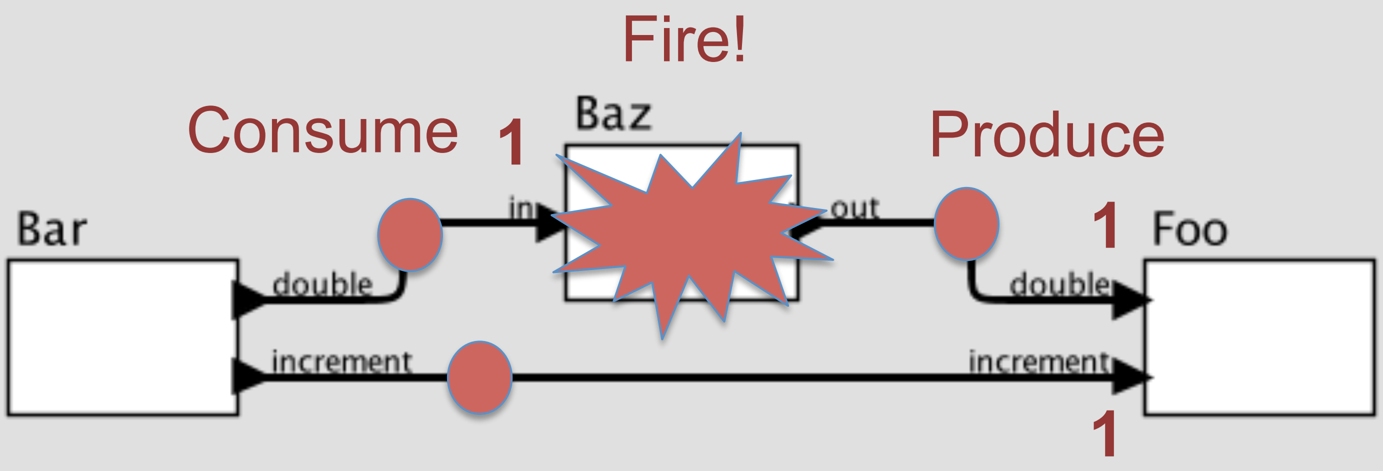


Tokens produced

Dataflow Solution for Scheduling: Firing Rules

[Lee & Matsikoudis, 2009]

An actor with inputs has to specify at all times how many tokens it needs on each input in order to fire.

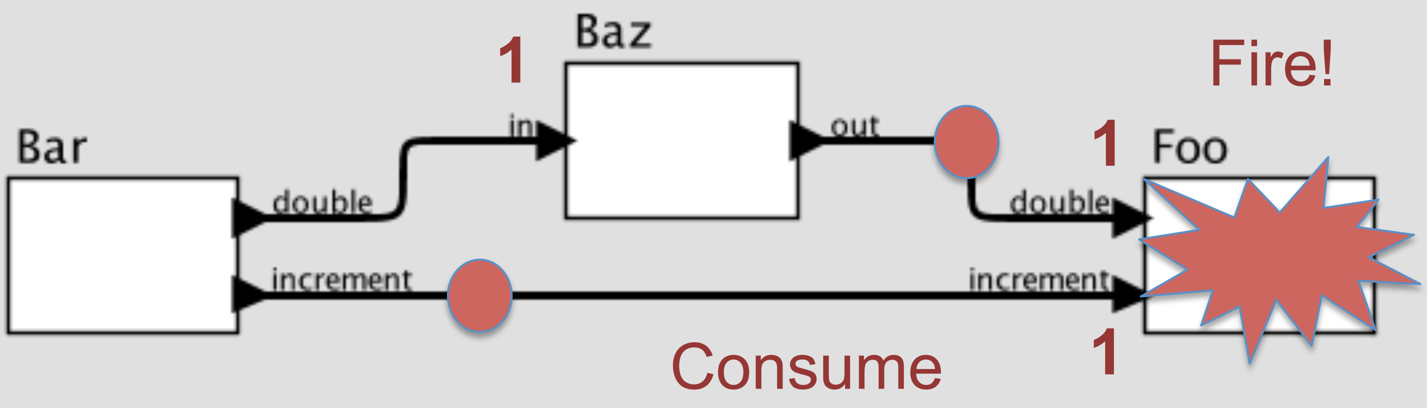


Dataflow Solution for Scheduling: Firing Rules

[Lee & Matsikoudis, 2009]

An actor with inputs has to specify at all times how many tokens it needs on each input in order to fire.

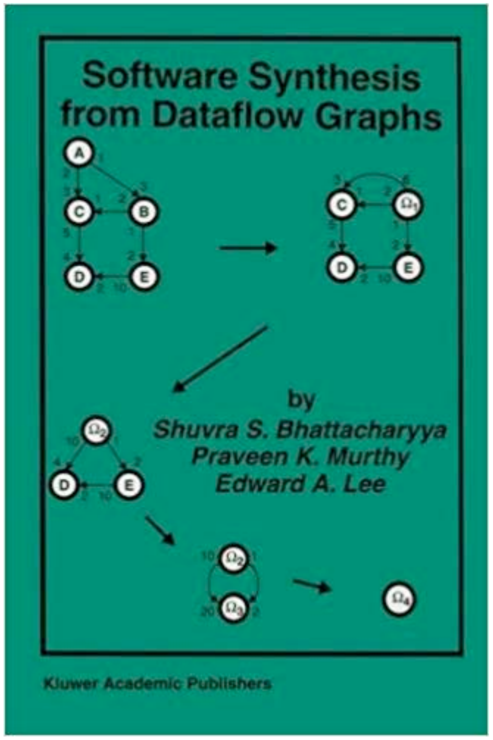
When it fires, each reaction is invoked in a deterministic order.



Synchronous Dataflow Scheduling

When the firing rules and production patterns are static integer constants, then a lot of analysis and optimization is possible.

[Lee & Messerschmitt, 1986]

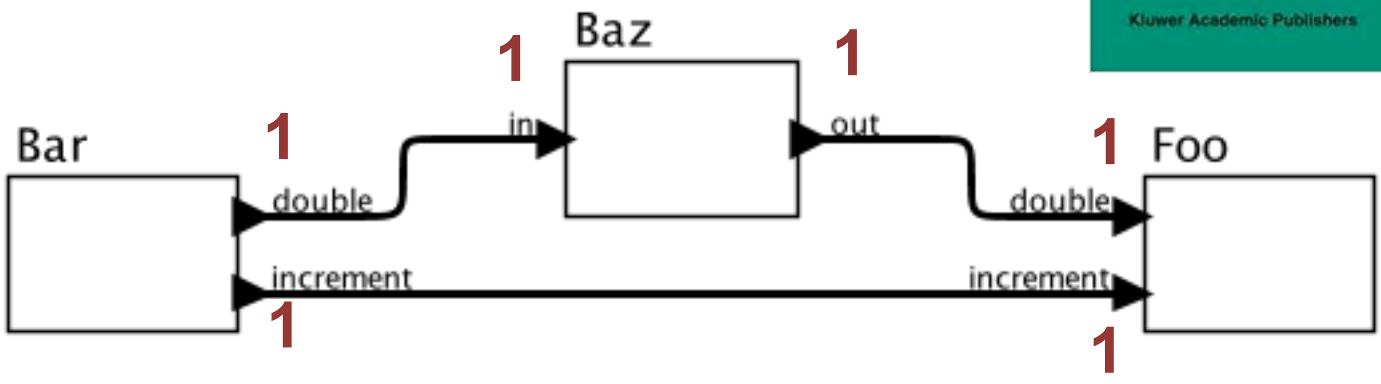


Software Synthesis from Dataflow Graphs

by Shuvra S. Bhattacharyya Praveen K. Murthy Edward A. Lee

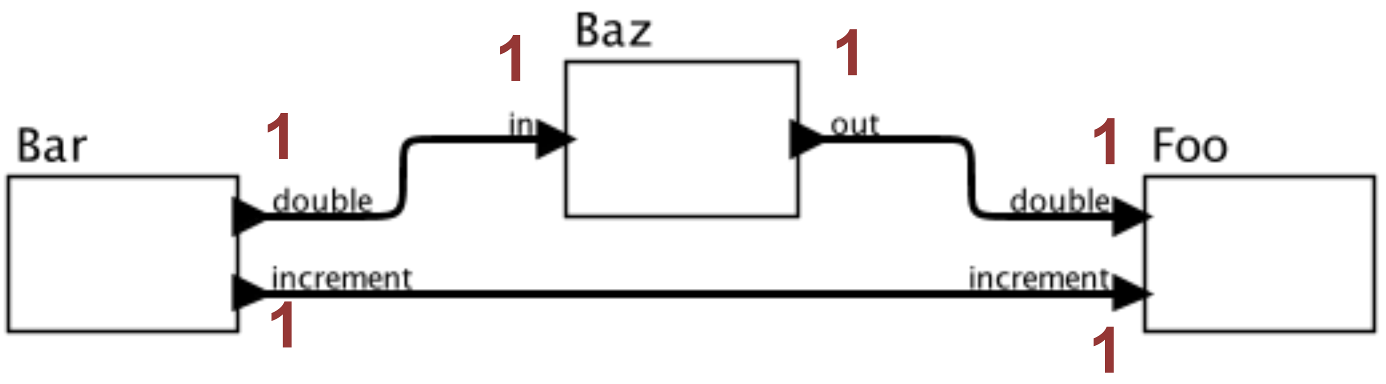
Kluwer Academic Publishers

1996



Synchronous Dataflow Scheduling with Timing

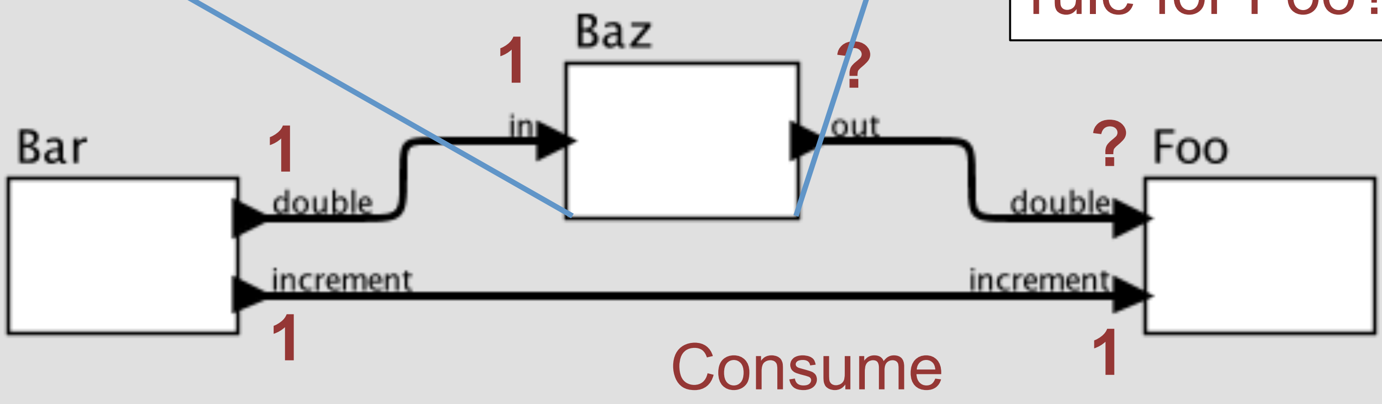
If execution times are also known, then throughput and latency bounds are derivable and optimal scheduling is possible (albeit intractable).



Dataflow Scheduling with Dynamic Firing Rules

```
reactor Baz {
    input in:bool;
    output out:bool;
    reaction(in)->out {
        if (something) {
            set(out, true);
        }
    }
}
```

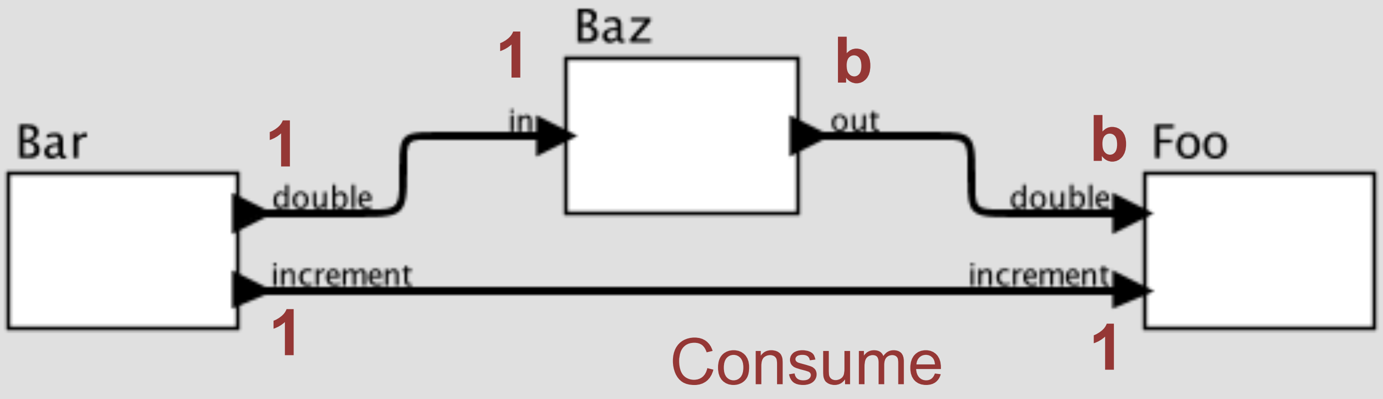
What should be the firing rule for Foo?



Boolean Dataflow

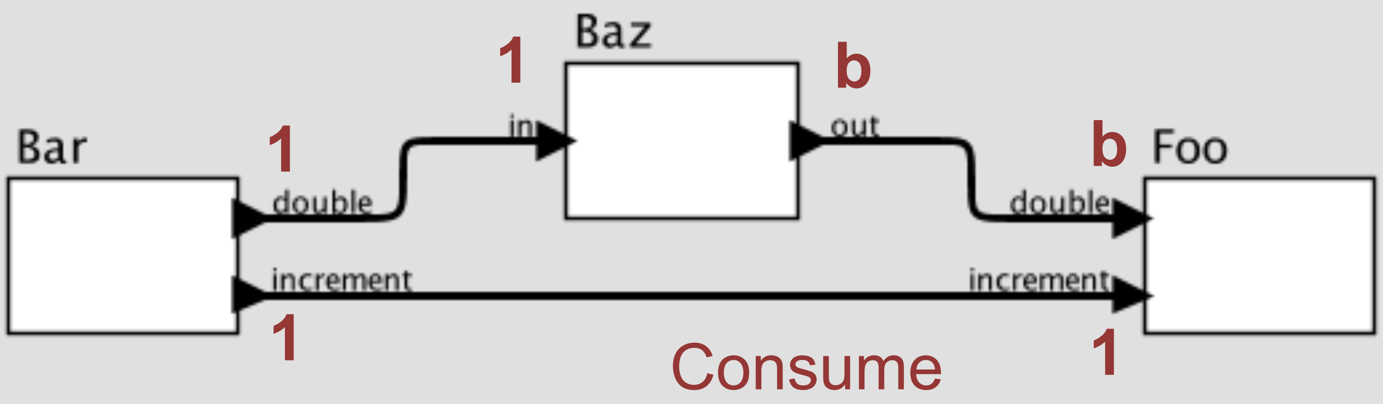
Buck [1993] showed that scheduling problems in general are undecidable in this framework.

Associate a symbolic variable with production and consumption parameters. Solve the scheduling problem symbolically.
[Buck and Lee, 1993]



Various Dataflow Variants that Remain Decidable

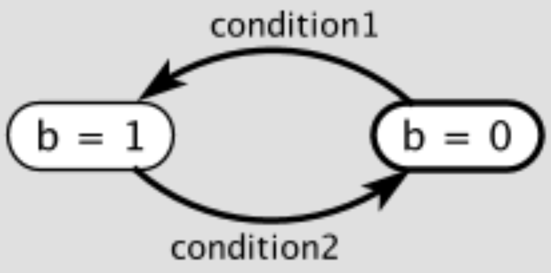
- Cyclostatic dataflow [Lauwereins 1994]
- Heterochronous dataflow [Girault, Lee & Lee, 1997]
- Parameterized dataflow [Bhattacharya & Bhattacharyya, 2001]
- Structured dataflow [Thies, 2002]
- Scenario-aware dataflow [Theelen, Geilen, Basten, et al. 2006]
- Reconfigurable dataflow [Fradet, Girault, et al., 2019]

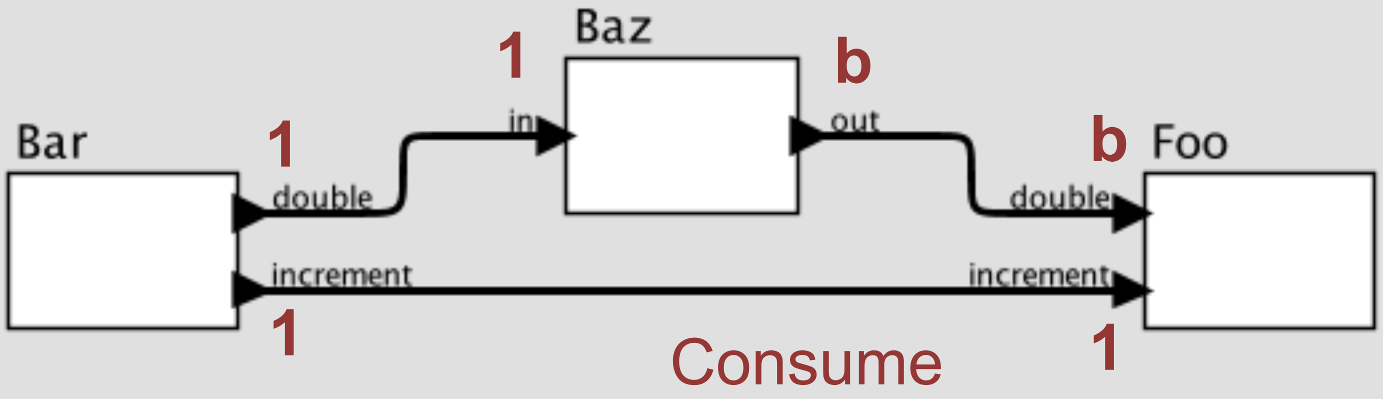


Scenario-Aware Dataflow

A state machine governs the switching between production/consumption patterns and also execution times.

[Theelen, Geilen, Basten, et al. 2006]





Towards Deterministic Concurrency

- Dataflow (DF)
- Process Networks (PN)
- Synchronous/Reactive (SR)
- Discrete Events (DE)

A Different Solution: Blocking Reads

In Kahn Process Networks (KPN), every actor is a process that blocks on reading inputs until data is available.

double

increment

```
KPNActor Foo {
    input double:bool;
    input increment:int;
    state s:int(1);
    while(true) {
        read(double);
        s *= 2;
        x = read(increment);
        s += x;
        print(s);
    }
}
```

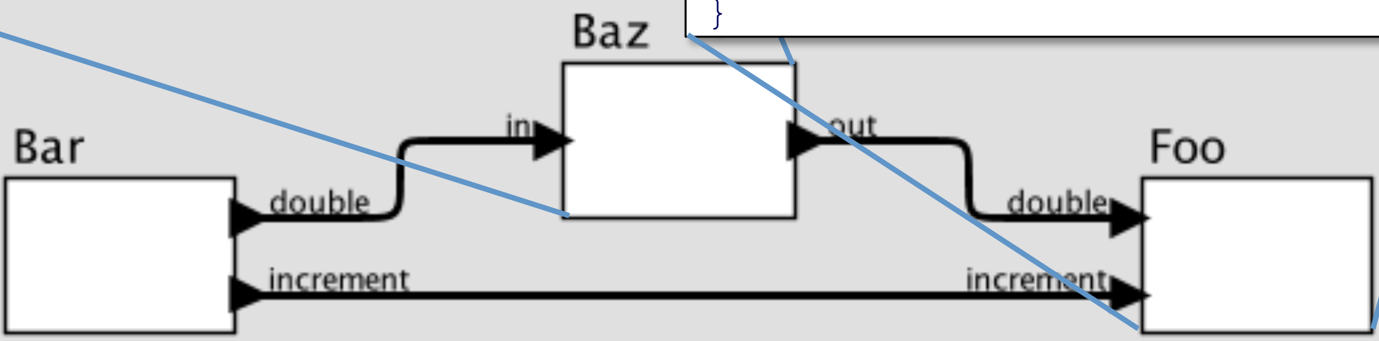
Gilles Kahn

[Kahn, 1974] [Kahn and MacQueen, 1977]

Blocking reads have trouble with data-dependent flow patterns

```
KPNActor Baz {
    input in:bool;
    output out:bool;
    while(true) {
        read(in);
        if (something) {
            send(out);
        }
    }
}
```

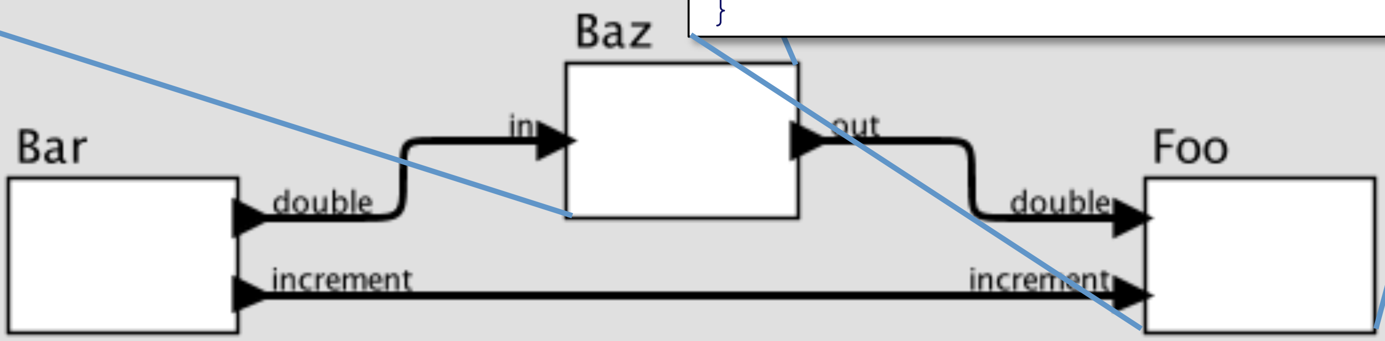
```
KPNActor Foo {
    input double:bool;
    input increment:int;
    state s:int(1);
    while(true) {
        read(double);
        s *= 2;
        x = read(increment);
        s += x;
        print(s);
    }
}
```



Solution: Coordinated Control

```
KPNActor Baz {
    input in:bool;
    output out:bool;
    while(true) {
        read(in);
        if (something) {
            send(out);
        }
    }
}
```

```
KPNActor Foo {
    input double:bool;
    input increment:int;
    state s:int(1);
    while(true) {
        if (something) {
            read(double);
        }
        s *= 2;
        x = read(increment);
        s += x;
        print(s);
    }
}
```



Towards Deterministic Concurrency

- Dataflow (DF)
- Process Networks (PN)
- Synchronous/Reactive (SR)
- Discrete Events (DE)

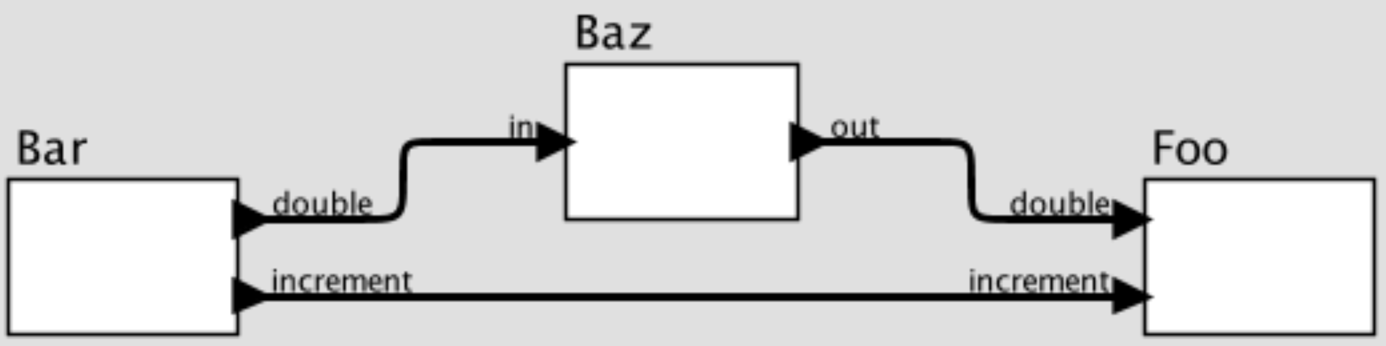
An Alternative Approach to Coordination

Make the notion of the “absence” of a message as meaningful as its presence.

A Different Approach: Synchronous Languages

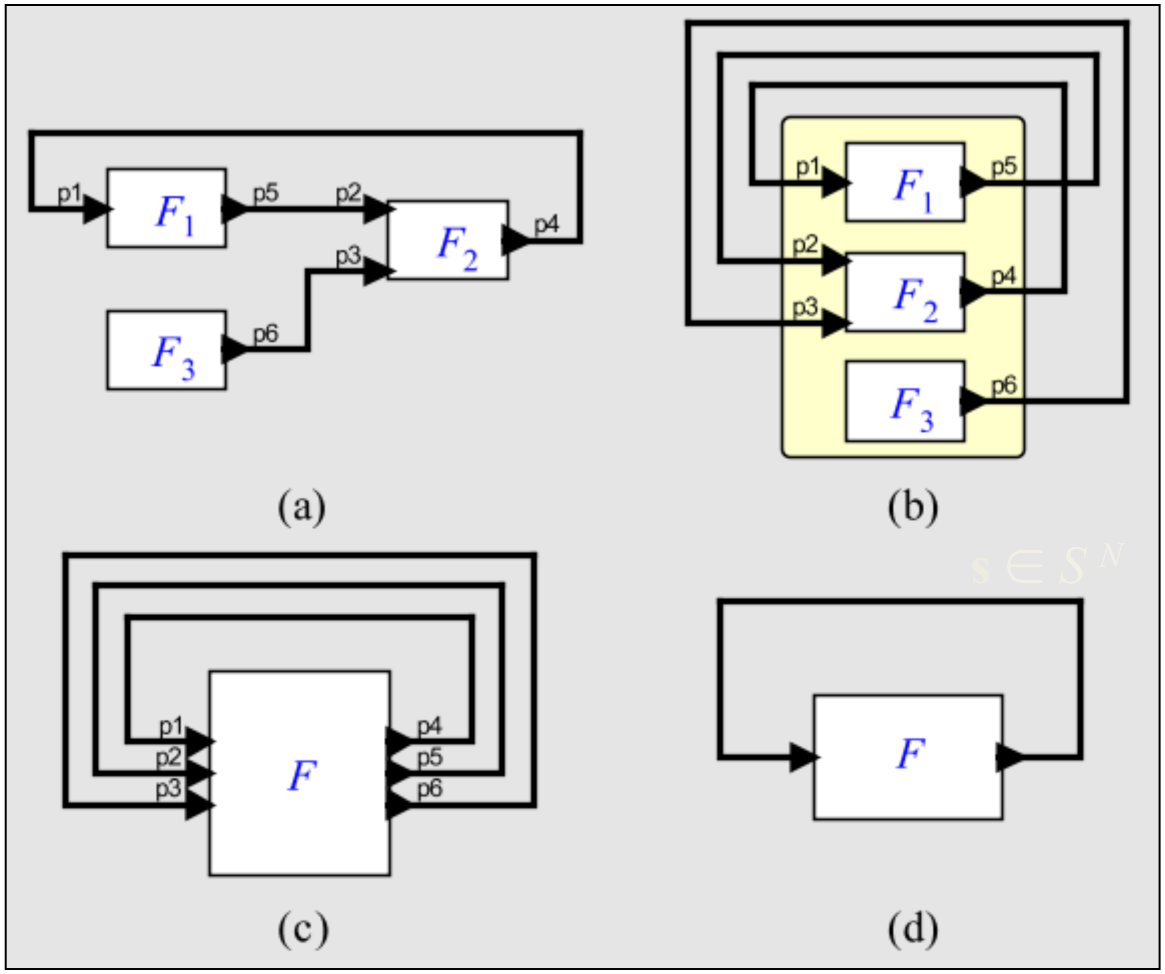
In the synchronous/reactive approach, there is a conceptual global "clock," and on each "tick" of this clock, a connection either has a well-defined value or is "absent."

Each actor realizes a time-varying function mapping inputs to outputs.



[Benveniste & Berry, 1991]

Fixed Point Semantics



[Edwards and Lee, 2003]

At each tick of the clock, the job of the execution engine is to find a valuation *s* for all signals such that *F*(*s*) = *s*.

This is called a fixed point of the function *F*. A theory of partial orders guarantees existence and uniqueness.

Distributed and Parallel Execution

Physically asynchronous,
logically synchronous (PALS)

[Sha et al., 2009]

Towards Deterministic Concurrency

- Dataflow (DF)
- Process Networks (PN)
- Synchronous/Reactive (SR)
- Discrete Events (DE)

Outline

- Definitions
- Threads
- Alternatives to Threads
- Actors
- Ports, Hierarchy, and Scheduling
- Deterministic Concurrency
- Time
- Discrete-Event Languages
- Reactors

Recall: Desirable Properties in a Model of Time

- A “present” that separates the past and future
  - Needed for a notion of “state”
- Support for causality
  - If *A* causes *B,* then every observer should see *A* before *B.*
- A well-defined “observer”
  - Otherwise, you are stuck trying to solve the physics problem.
- A notion of “simultaneity”

All are problematic in physics but useful in models.

Models in Time

Assume that "time" is about how a model *changes*.

Change may be:

1. Discrete: indivisible, atomic, an *event*.
2. Continuous: flow, motion

Representing Time

When realized in a software-based model:

1. The precision of time should be finite and the same for all observers.
2. The precision of time should be independent of the absolute magnitude of the time.
3. Addition of time should be associative. That is, for any three time intervals t_1, t_2, and t_3,

$$(t_1 + t_2) + t_3 = t_1 + (t_2 + t_3)$$

[1] Broman, et al. “Requirements for hybrid cosimulation standards. HSCC 2015.
[2] Cremona, et al., “Hybrid co-simulation: it's about time,” Software and Systems Modeling 2017.

Representing Time

When realized in a software-based model:

1. The precision of time should be finite and the same for all observers.
2. The precision of time should be independent of the absolute magnitude of the time.
3. Addition of time should be associative. That is, for any three time intervals t_1, t_2, and t_3,

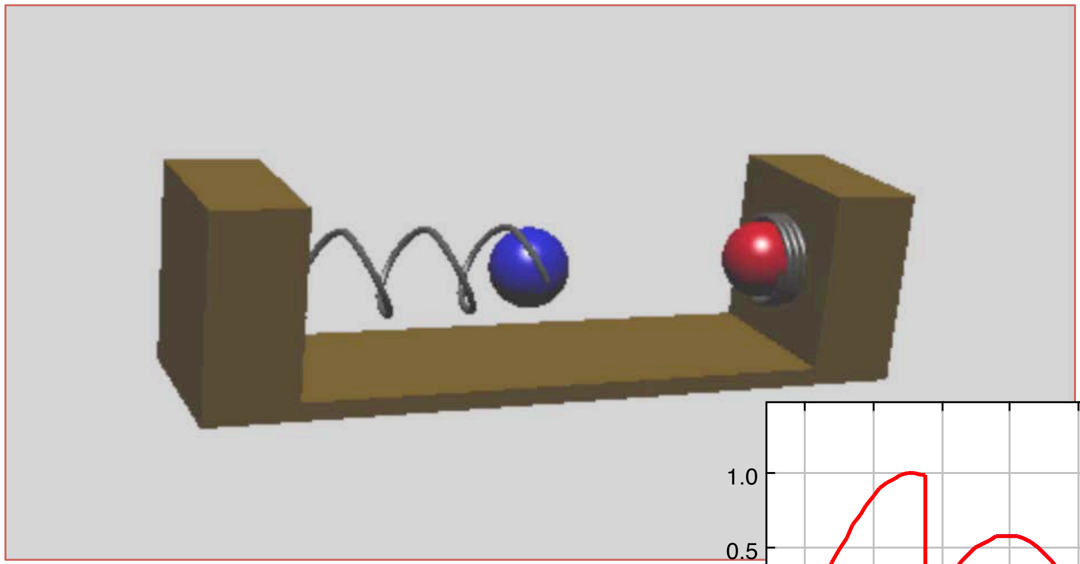
$$(t_1 + t_2) + t_3 = t_1 + (t_2 + t_3)$$

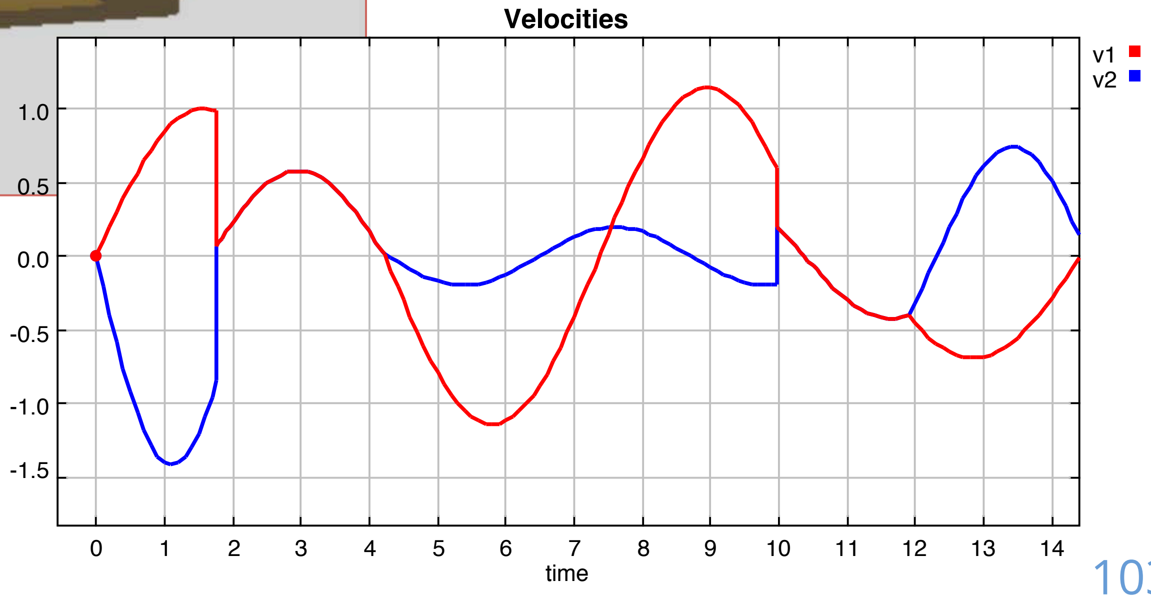
Floating point numbers do not satisfy these properties.

[1] Broman, et al. "Requirements for hybrid cosimulation standards. HSCC 2015.
[2] Cremona, et al., "Hybrid co-simulation: it's about time," Software and Systems Modeling 2017.

Representing Time

- “Continuum” does not imply “continuous.





Models Without Simultaneity

- Sometimes called “interleaving semantics”
- Simultaneity → nondeterministic ordering
- Newtonian physics no longer works
- Models are quickly intractable

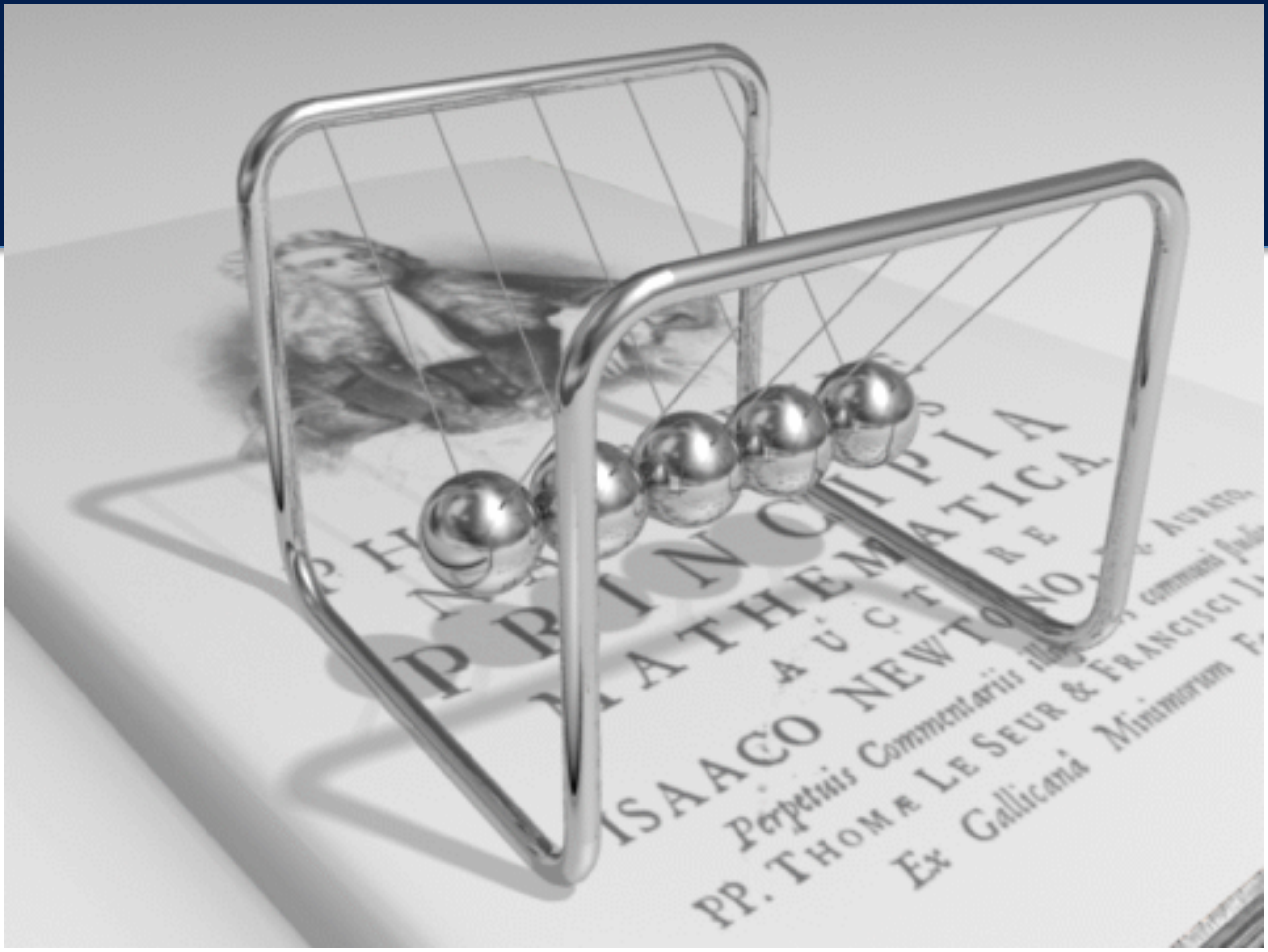
Models With Simultaneity

Event *A* is simultaneous with event *B* if no observer can see that one event occurred and the other did not.

This requires a well-defined notion of an "observer."

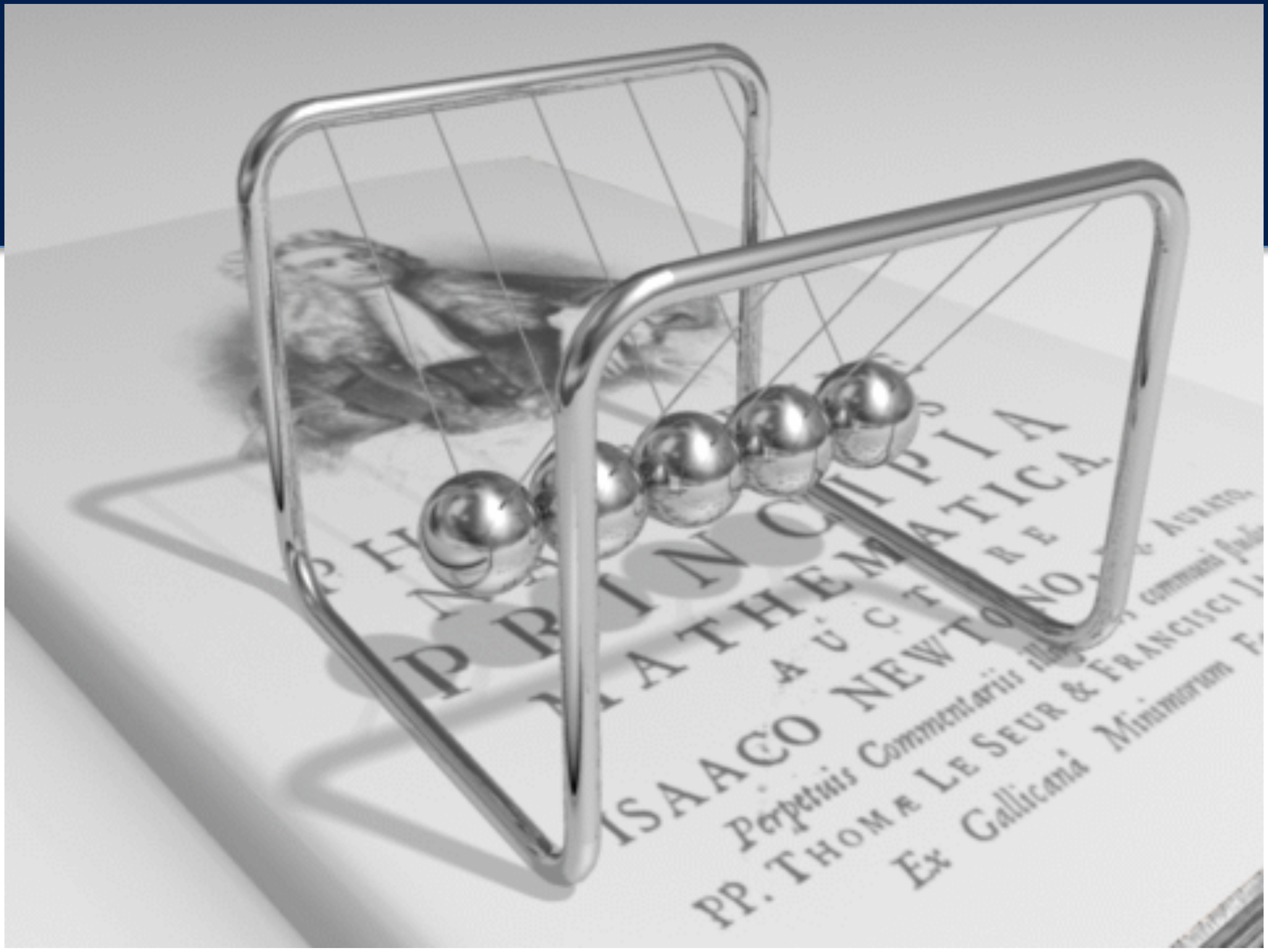
Pitfall With Simultaneity

If two events are not simultaneous, does time pass between their occurrences?


PRINCIPIA
MATHEMATICA
AUCTORE
ISAACO NEWTONO, Eq. Aurato,
Perpetuis Commentariis illustrata, communi studio
PP. THOMÆ LE SEUR & FRANCISCI JACQUIER
Ex Gallicanâ Minimorum Familiâ

What is the momentum of the middle ball as a function of time?

$$\mathbf{p}(t) = m\mathbf{v}(t)$$

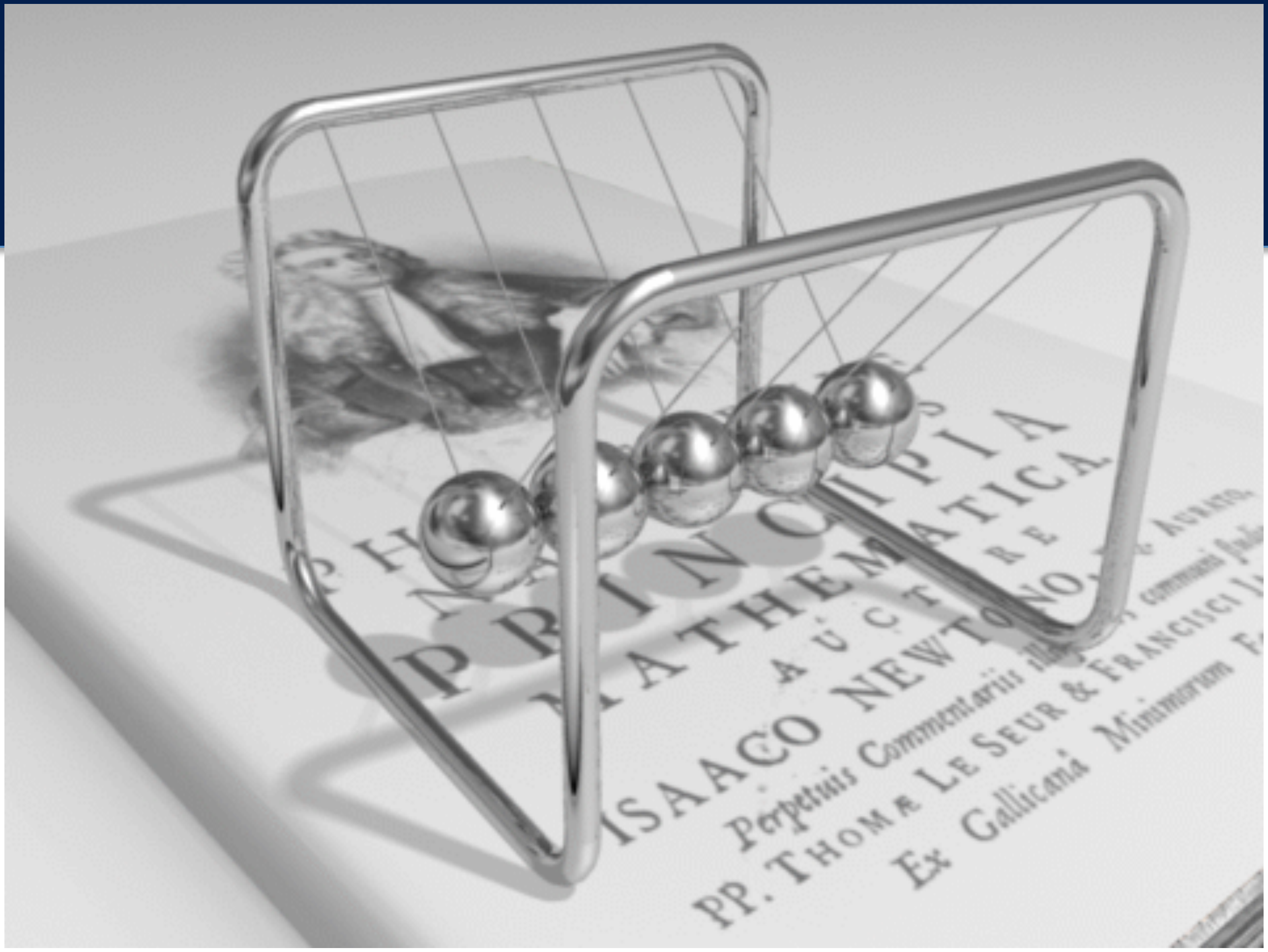

PHILOSOPHIAE NATURALIS
PRINCIPIA
MATHEMATICA
AUCTORE
ISAACO NEWTONO, Eq. Aurato.
Perpetuis Commentariis illustrata, communi studio
PP. THOMÆ LE SEUR & FRANCISCI JACQUIER
Ex Gallicanâ Minimorum Familiâ

What is the momentum of the middle ball as a function of time?

$$\mathbf{p}(t) = m\mathbf{v}(t)$$

It might seem:

$$\mathbf{v}(t) = 0 \quad \Rightarrow \quad \mathbf{p}(t) = 0$$

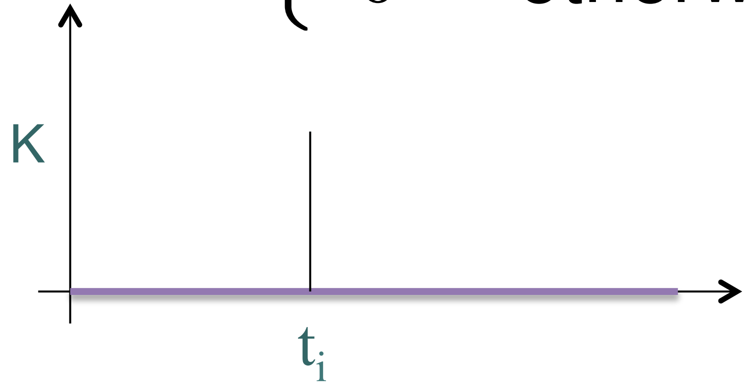


But no, it is:

$$\mathbf{v}(t) = \begin{cases} \mathbf{K}, & t = t_i \\ 0 & \text{otherwise} \end{cases}$$

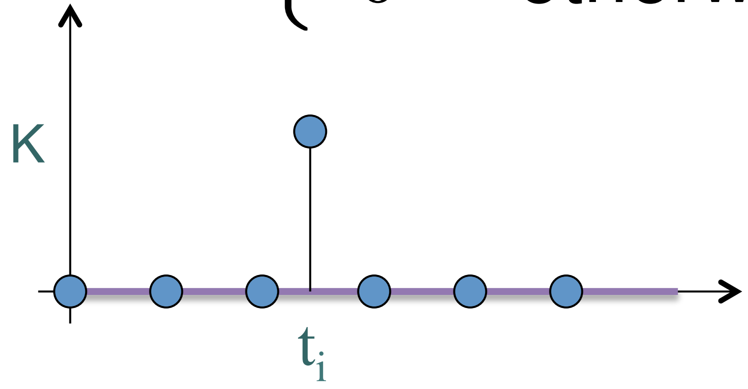
where t_i is the time of collision

$$\mathbf{v}(t) = \begin{cases} \mathbf{K}, & t = t_i \\ 0 & \text{otherwise} \end{cases}$$



Since position is the integral of velocity, and the integral of **v** is zero, the ball does not move.

$$\mathbf{v}(t) = \begin{cases} \mathbf{K}, & t = t_i \\ 0 & \text{otherwise} \end{cases}$$

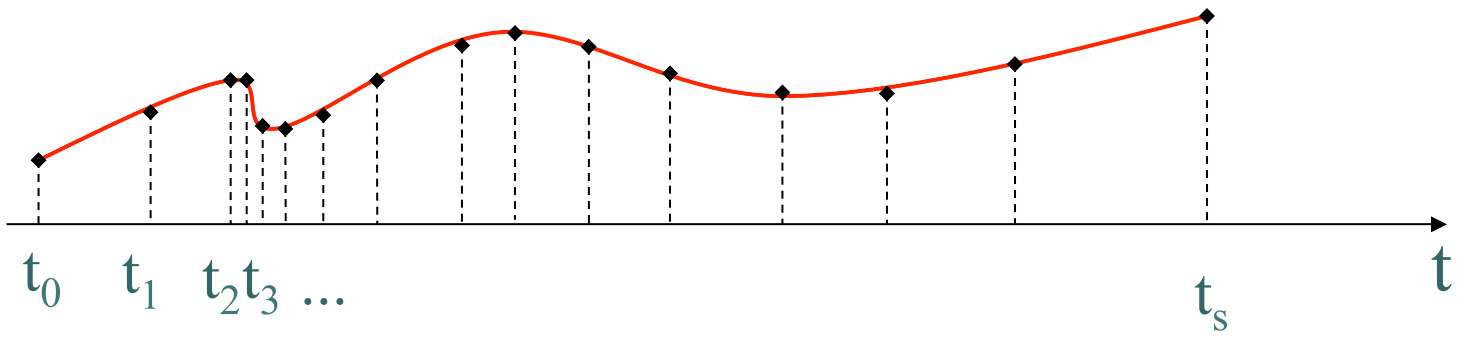


A *discrete* representation of this signal with *samples* is inadequate.

Samples yield discrete signals

A signal $s : T \to D$ is sampled at tags

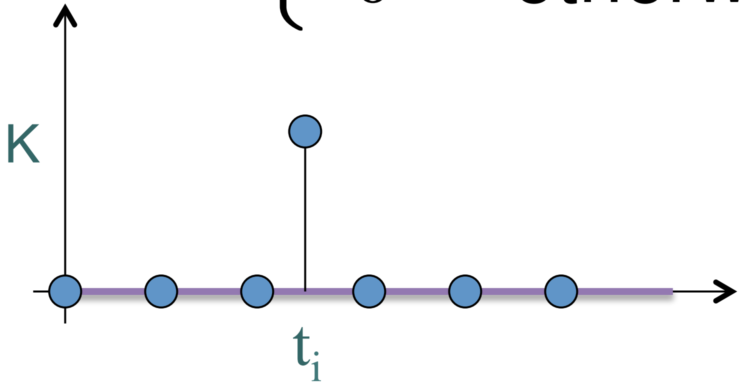
$$\pi(s) = \{t_0, t_1, \dots\} \subset T$$



A signal s is discrete if there is an order embedding from its tag set $\pi(s)$ (the tags for which it is defined and not absent) to the natural numbers (under their usual order).

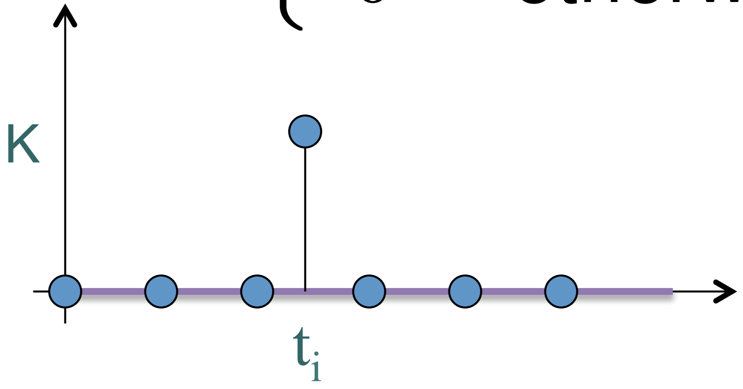
PHILOSOPHIÆ NATURALIS
PRINCIPIA MATHEMATICA
AUCTORE ISAACO NEWTONO, Eq. Aurato.
Perpetuis Commentariis illustrata, communi studio
PP. THOMÆ LE SEUR & FRANCISCI J
Ex Gallicanâ Minimorum F

$$\mathbf{v}(t) = \begin{cases} \mathbf{K}, & t = t_i \\ 0 & \text{otherwise} \end{cases}$$



No discrete subset of real-valued times is adequate to unambiguously represent this signal.

$$\mathbf{v}(t) = \begin{cases} \mathbf{K}, & t = t_i \\ 0 & \text{otherwise} \end{cases}$$



There is no *semantic distinction* between a discrete event and a rapidly varying continuous signal.

Superdense Time

$$\cancel{\mathbf{v}: \mathbb{R} \to \mathbb{R}^3}$$

$$\mathbf{v}: (\mathbb{R} \times \mathbb{N}) \to \mathbb{R}^3$$

Initial value: $\mathbf{v}(t_i, 0) = 0$

Intermediate value: $\mathbf{v}(t_i, 1) = \mathbf{K}$

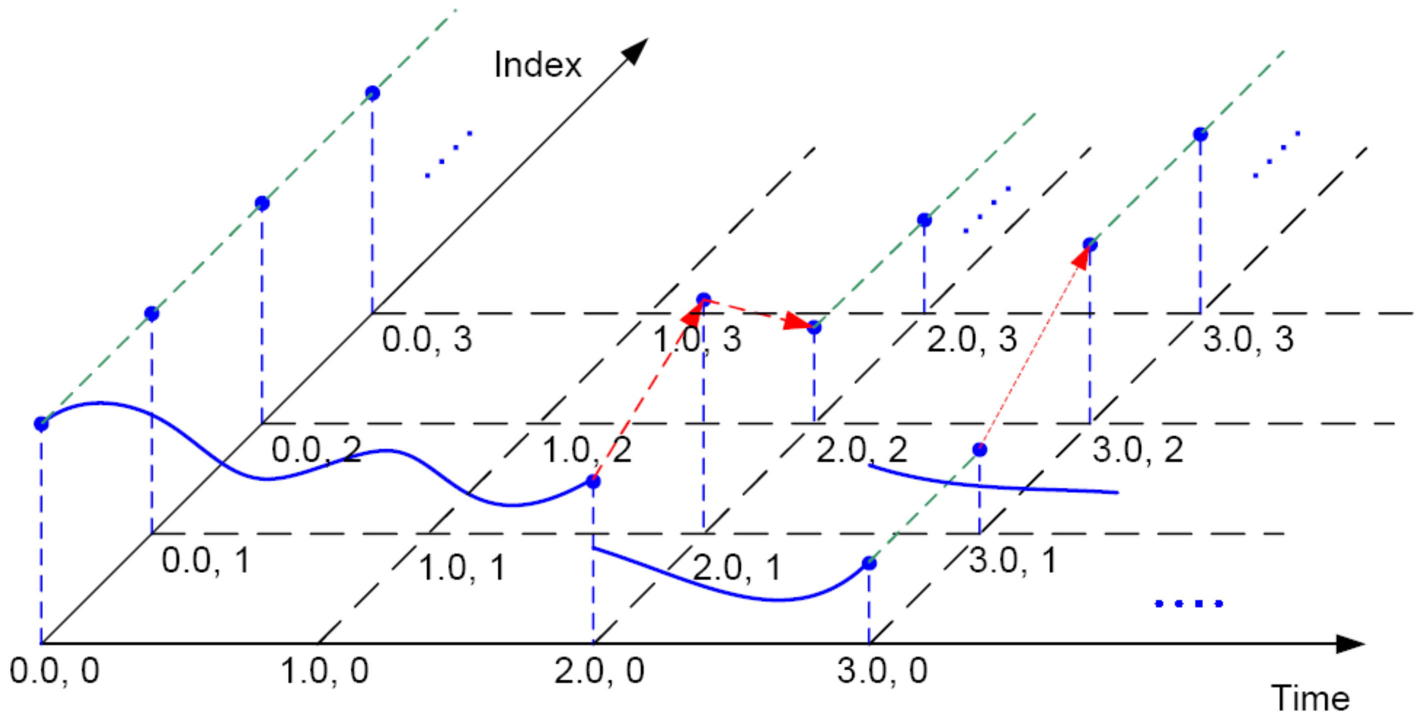
Final value: $\mathbf{v}(t_i, n) = 0, \quad n \geq 2$

At each **tag**, the signal has *exactly one value*.
At each time point, the signal has a *sequence of values*.

[Lee & Zheng, 2005]

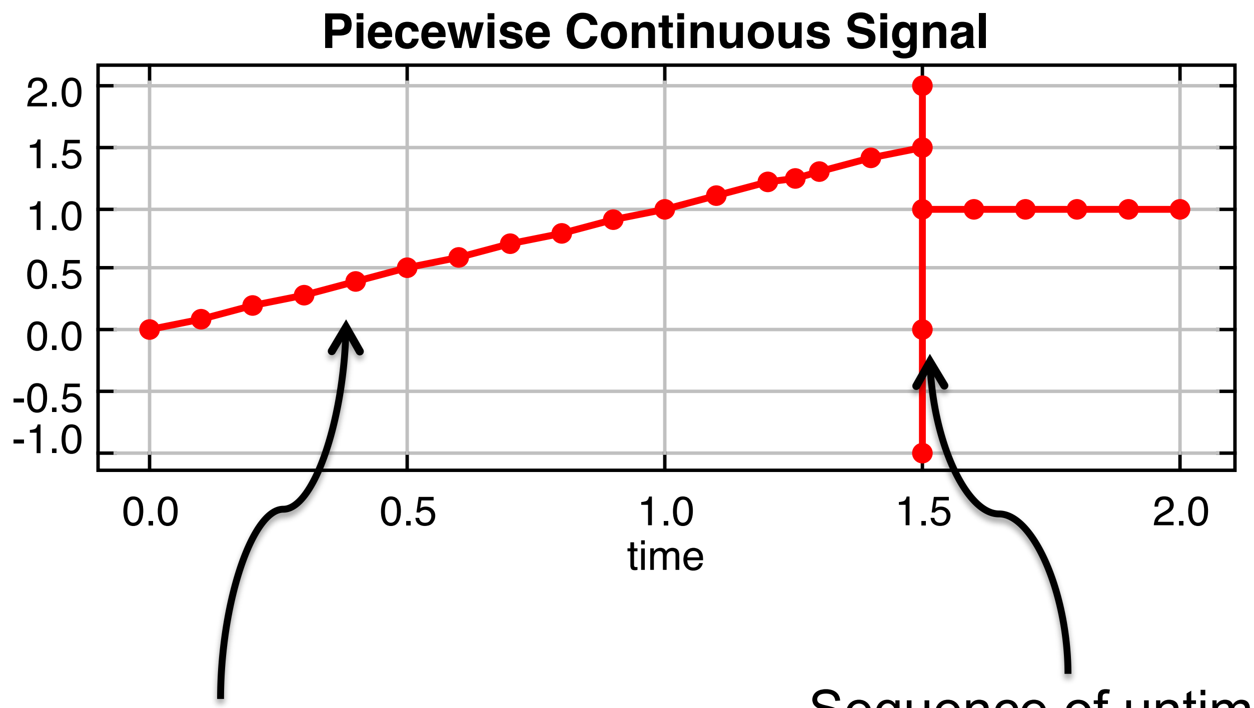
[Lee, "CPS Foundations," DAC, 2010]
[Maler, Manna, Pnuelli, 92]

Superdense Time



The red arrows indicate value changes between tags, which correspond to discontinuities. Signals are continuous from the left *and* continuous from the right at points of discontinuity.

Superdense Time in Ptolemy II



Samples from numerical ODE solver.

Sequence of untimed events.

[1] Cardoso, et al., "Continuous-Time Models," in *System Design, Modeling, and Simulation using Ptolemy II*, Claudius Ptolemaeus (ed.), ptolemy.org, 2014.

Outline

- Definitions
- Threads
- Alternatives to Threads
- Actors
- Ports, Hierarchy, and Scheduling
- Deterministic Concurrency
- Time
- Discrete-Event Languages
- Reactors

Discrete-Event Languages

DE is a generalization of SR, where there is a notion of “time between ticks.”

WARNING: immediately have (at least) two time lines: logical time and physical time(s).

[Lee & Zheng, 2007]

Recall: Asynchronous Atomic Callbacks: Periodic Actions

```
var x = 0;
function increment() {
  x = x + 1;
}
function decrement() {
  x = x - 2;
}
function observe() {
  console.log(x);
}
setInterval(increment, 1000);
setInterval(decrement, 2000);
setInterval(observe, 4000);
```

- Shared variable x
- Timed actions on x

- +1 every second
- −2 every two seconds
- Observe every 4 seconds

On Node.js v5.3.0, MacOS Sierra:

```
0, 0, 0, 0, 0, −1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, …
```

Our Contribution: Logical Clocks

```
var x = 0;
function increment() {
  x = x + 1;
}
function decrement() {
  x = x - 2;
}
function observe() {
  console.log(x);
}
setInterval(increment, 1000);
setInterval(decrement, 2000);
setInterval(observe, 4000);
```

- Shared variable x
- Timed actions on x

- +1 every second
- −2 every two seconds
- Observe every 4 seconds

Make Times ***logical*** not ***physical.***

On Node.js v5.3.0, MacOS Sierra:

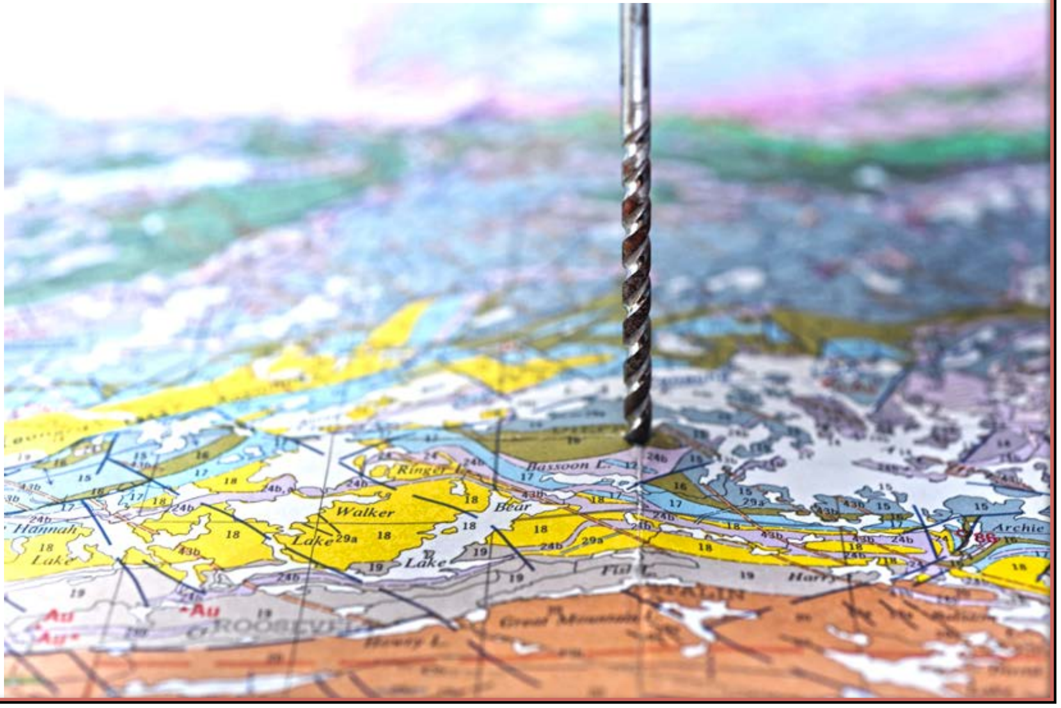
```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
```

Do not confuse the map with the territory!

You will never strike oil by drilling through the map!

Solomon Wolf Golomb

But this does not in any way diminish the value of a map!



Lee, Berkeley

Atomic Execution #1

```
var x = 0;
function increment() {
  x = x + 1;
}
function decrement() {
  x = x - 2;
}
function observe() {
  console.log(x);
}
setInterval(increment, 1000);
setInterval(decrement, 2000);
setInterval(observe, 4000);
```

At initialization, this executes atomically and the time origin T_A for logical clock domain ‘A’ is set to the current measurement of physical time.

```
x = 0
```

Atomic Execution #2

```
var x = 0;
function increment() {
  x = x + 1;
}
function decrement() {
  x = x - 2;
}
function observe() {
  console.log(x);
}
setInterval(increment, 1000);
setInterval(decrement, 2000);
setInterval(observe, 4000);
```

At time approximately T_A + 1000ms, increment x by 1.

```
x = 1
```

Atomic Execution #3

```
var x = 0;
function increment() {
  x = x + 1;
}
function decrement() {
  x = x - 2;
}
function observe() {
  console.log(x);
}
setInterval(increment, 1000);
setInterval(decrement, 2000);
setInterval(observe, 4000);
```

At time approximately T_A + 2000ms, increment x by 1, then decrement x by 2, *atomically.*

```
x = 0
```

Atomic Execution #4

```
var x = 0;
function increment() {
  x = x + 1;
}
function decrement() {
  x = x - 2;
}
function observe() {
  console.log(x);
}
setInterval(increment, 1000);
setInterval(decrement, 2000);
setInterval(observe, 4000);
```

At time approximately T_A + 3000ms, increment x by 1.

```
x = 1
```

Atomic Execution #5

```
var x = 0;
function increment() {
  x = x + 1;
}
function decrement() {
  x = x - 2;
}
function observe() {
  console.log(x);
}
setInterval(increment, 1000);
setInterval(decrement, 2000);
setInterval(observe, 4000);
```

At time approximately T_A + 4000ms, increment x by 1, then decrement x by 2, then print the value of x, *atomically.*

```
x = 0
```

A *Semantic* Notion of Simultaneity

Event *A* is simultaneous with event *B* if no observer can see that one event occurred and the other did not.

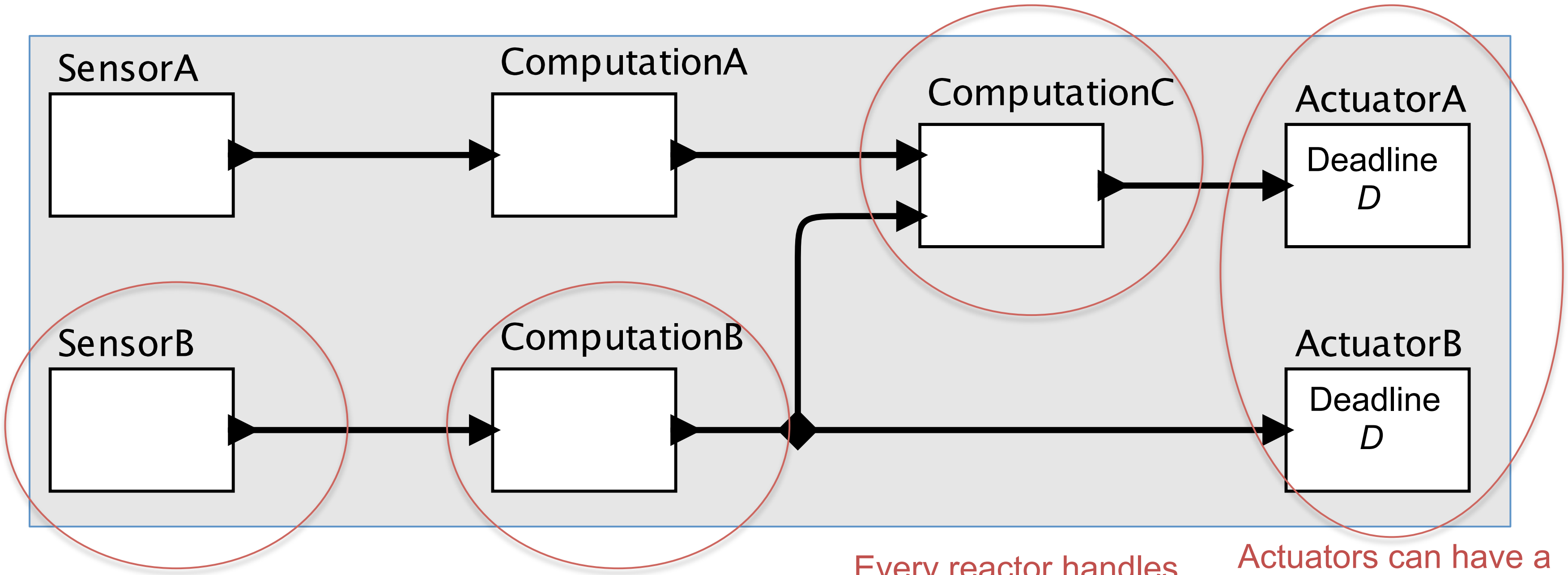
This requires a well-defined notion of an "observer."

Code is Testable!

Outline

- Definitions
- Threads
- Alternatives to Threads
- Actors
- Ports, Hierarchy, and Scheduling
- Deterministic Concurrency
- Time
- Discrete-Event Languages
- Reactors

Finally! We can talk about the motivating example.



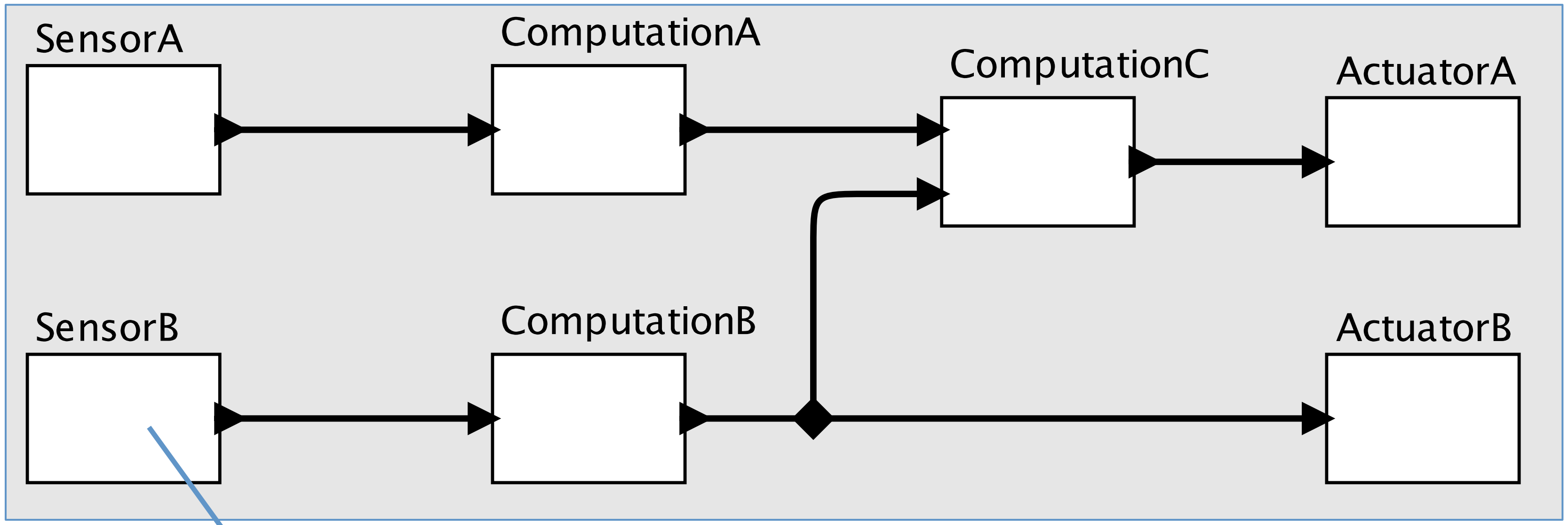
Sporadic events are assigned a time stamp based on the local physical-time clock

Computations have logically zero delay.

Every reactor handles events in time-stamp order. If time-stamps are equal, events are "simultaneous"

Actuators can have a deadline *D*. An input with time stamp *t* is required to be delivered to the actuator before the local clock hits *t* + *D*.

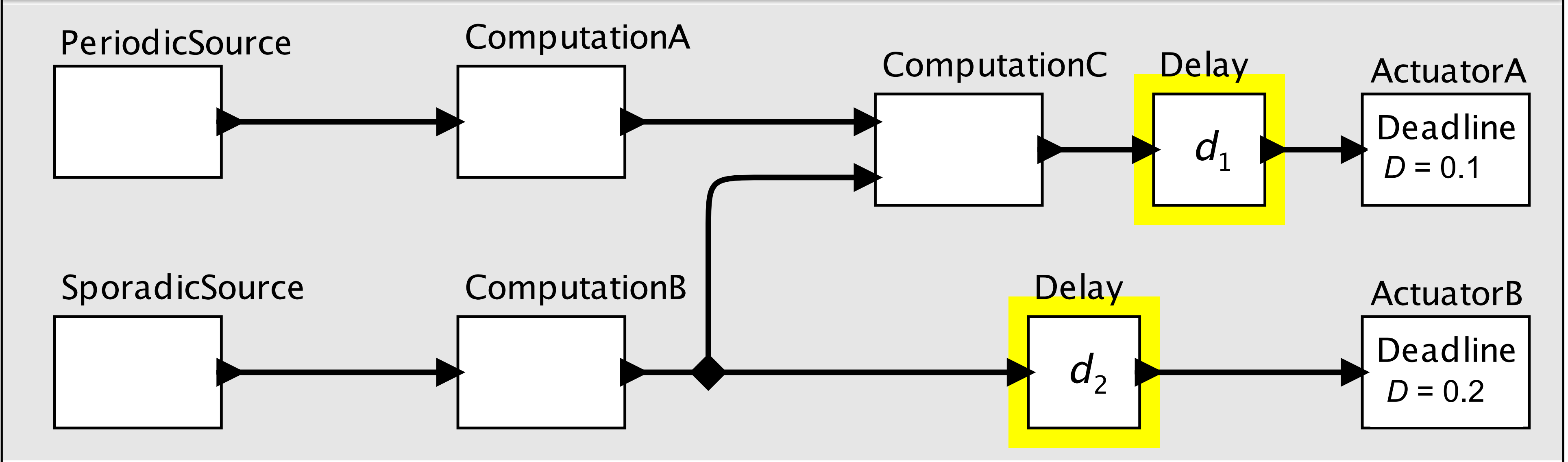
Simple, Single-Machine Realization



When a sporadic sensor triggers (or an asynchronous event like a network message arrives), assign a time stamp based on the local physical-time clock.

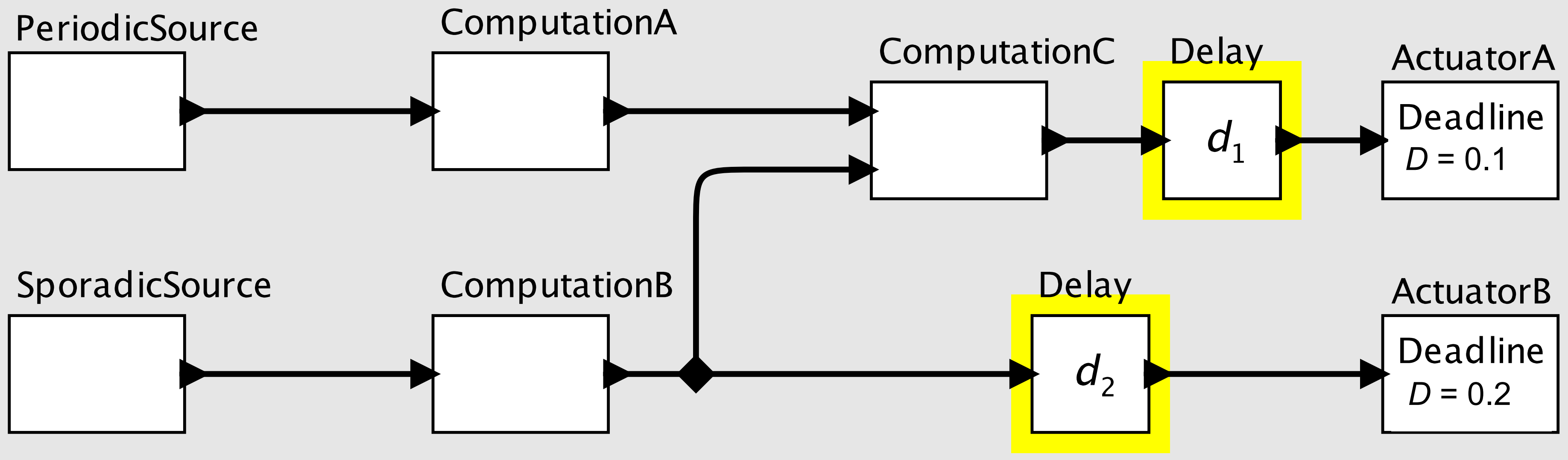
- Sort reactions topologically based on precedences.
- Global notion of "current logical time" *t.*
- Event queue containing future events.
- Choose earliest time stamp *t'* on the queue.
- Wait for the physical time clock to match *t'.*
- Execute reactors in topological sort order.

Temporal Operators (Logical Time)



This example has a pre-defined latency from physical sensing to physical actuation, thereby delivering a closed-loop deterministic cyber-physical model.

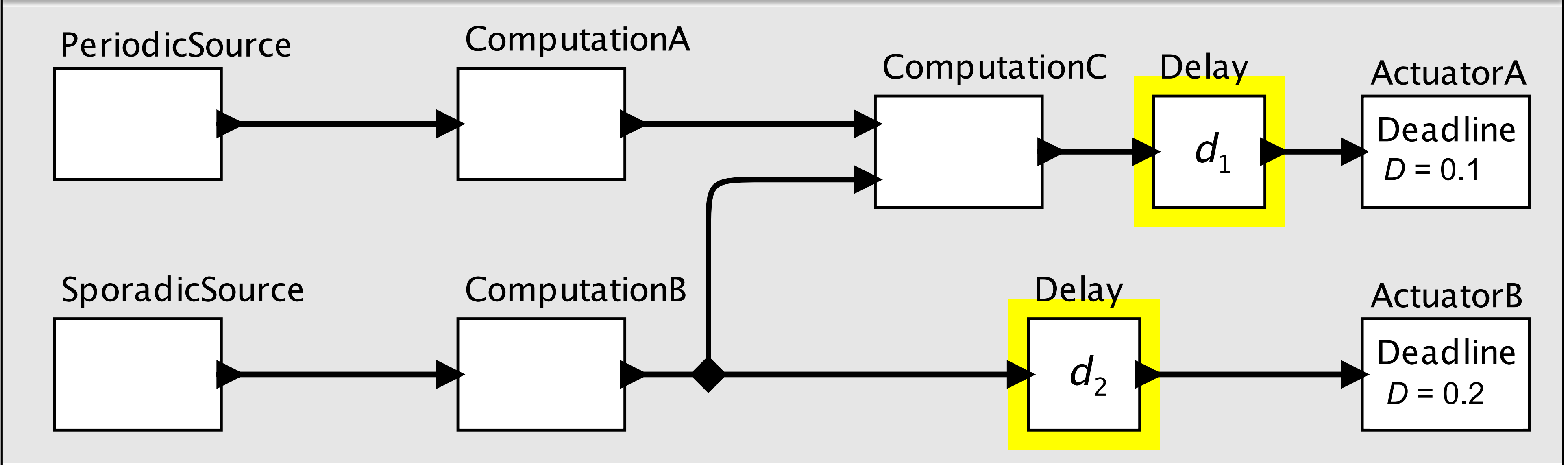
Real-Time Systems



Classical real-time systems scheduling and execution-time analysis determines whether the specification can be met.

[Buttazzo, 2005] [Wilhelm et al., 2008]

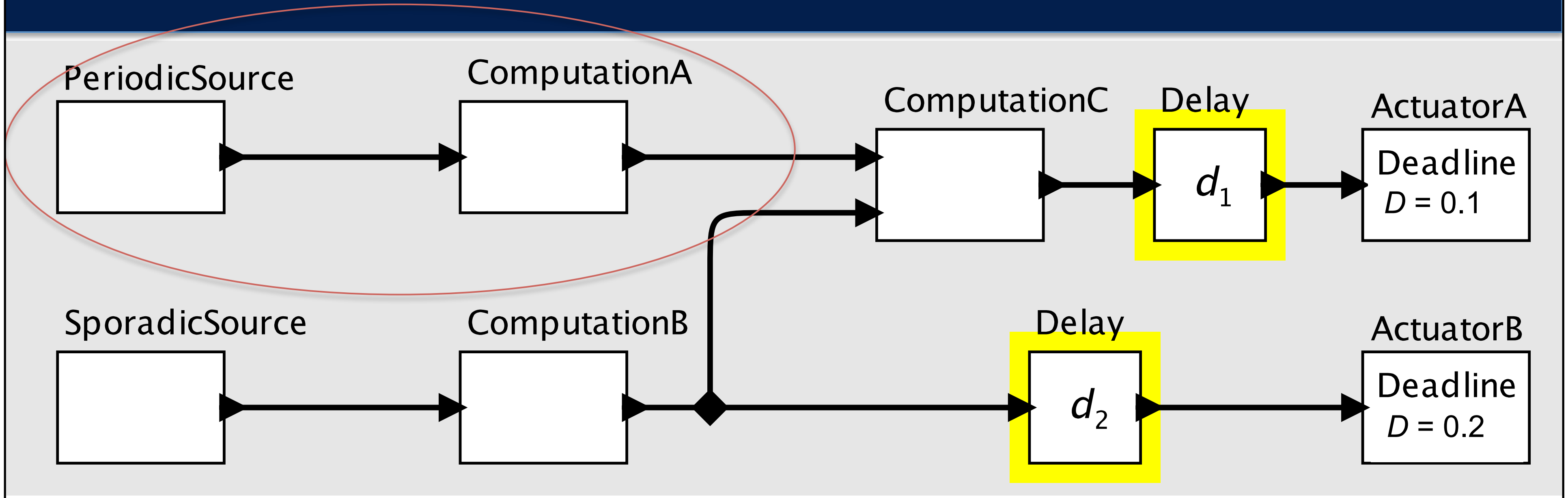
Iron-Clad Guarantees with PRET Machines



Precision-timed (PRET) machines deliver deterministic clock-cycle-level repeatable timing with no loss of performance on sporadic workloads.

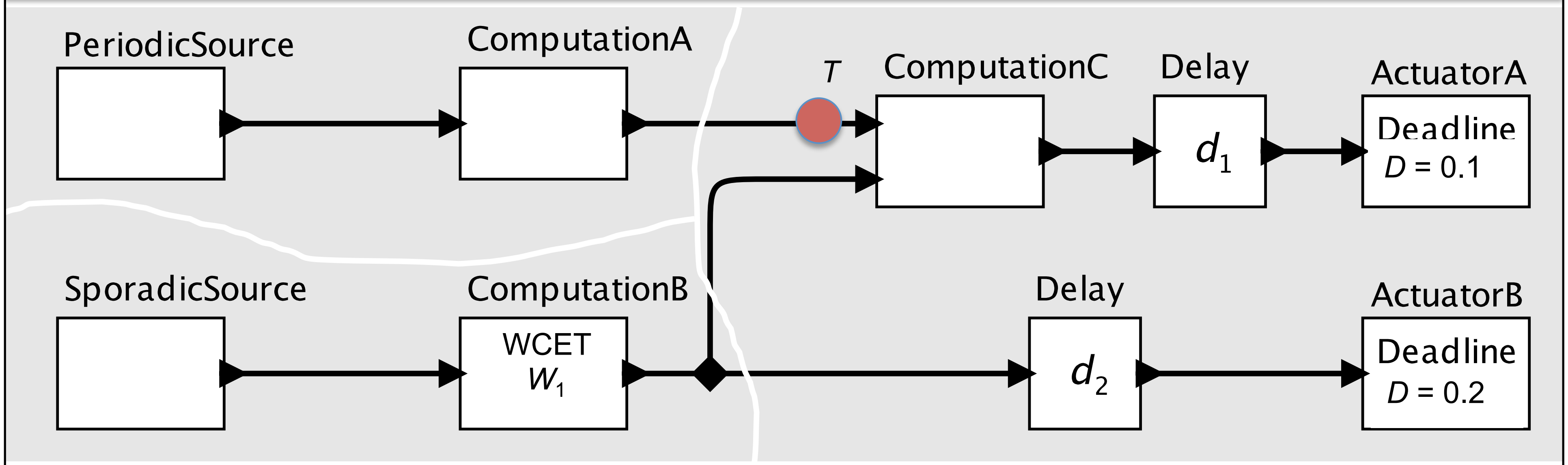
[Edwards & Lee, 2007] [Lee et al., 2017]

Opportunity for Optimization



If the PeriodicSource does not depend on physical inputs, then pre-computing (logical time ahead of physical time) becomes possible, based on dependence analysis.

Networked Scheduling: PTides



When is this "safe to process"?

[Zhao et al., 2007]

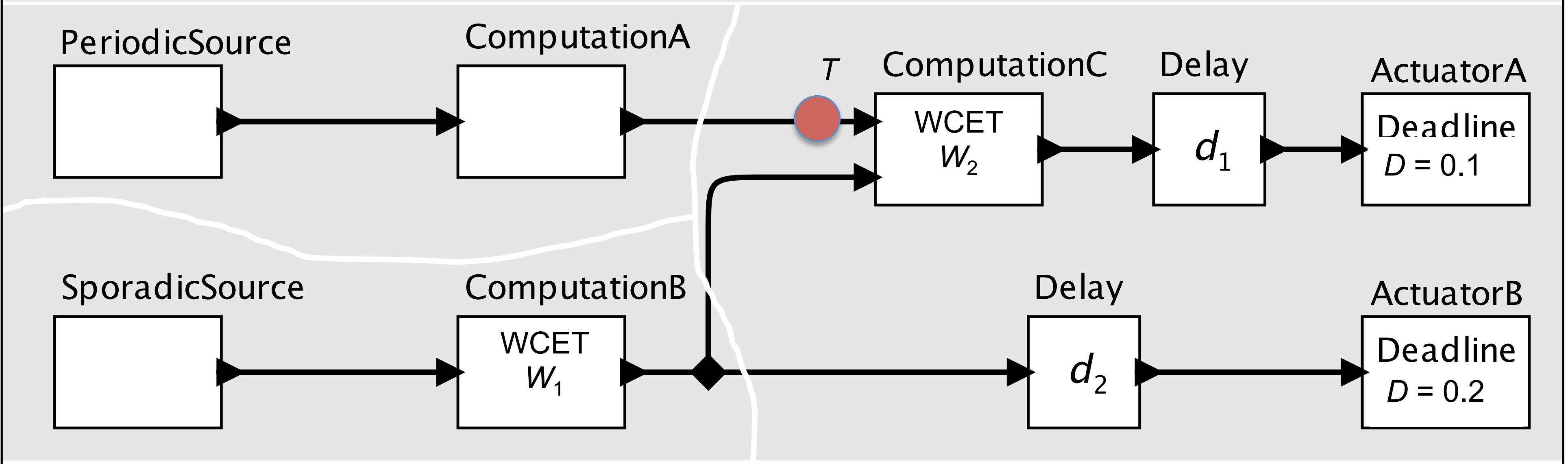
[Edison et al., 2012]

[Corbett et al., 2012]

When $\tau \geq T + W_1 + E + N$, where

- τ is the local physical clock time
- W_1 is worst-case execution time
- E is the bound on the clock synchronization error
- N the bound on the network delay

Networked Scheduling: PTides



Will the deadline at ActuatorA be met?

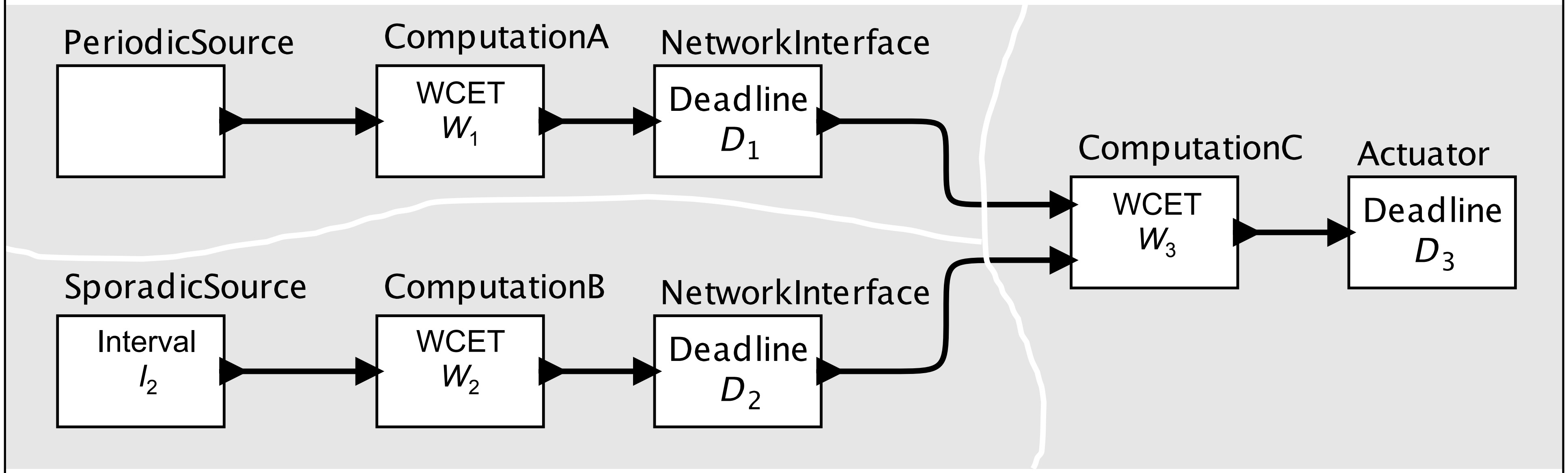
Yes if $D + d_1 \geq T + W_1 + E + N + W_2$

[Zhao et al., 2007]

[Edison et al., 2012]

[Corbett et al., 2012]

Decoupling Real-Time Analysis with Networked Scheduling

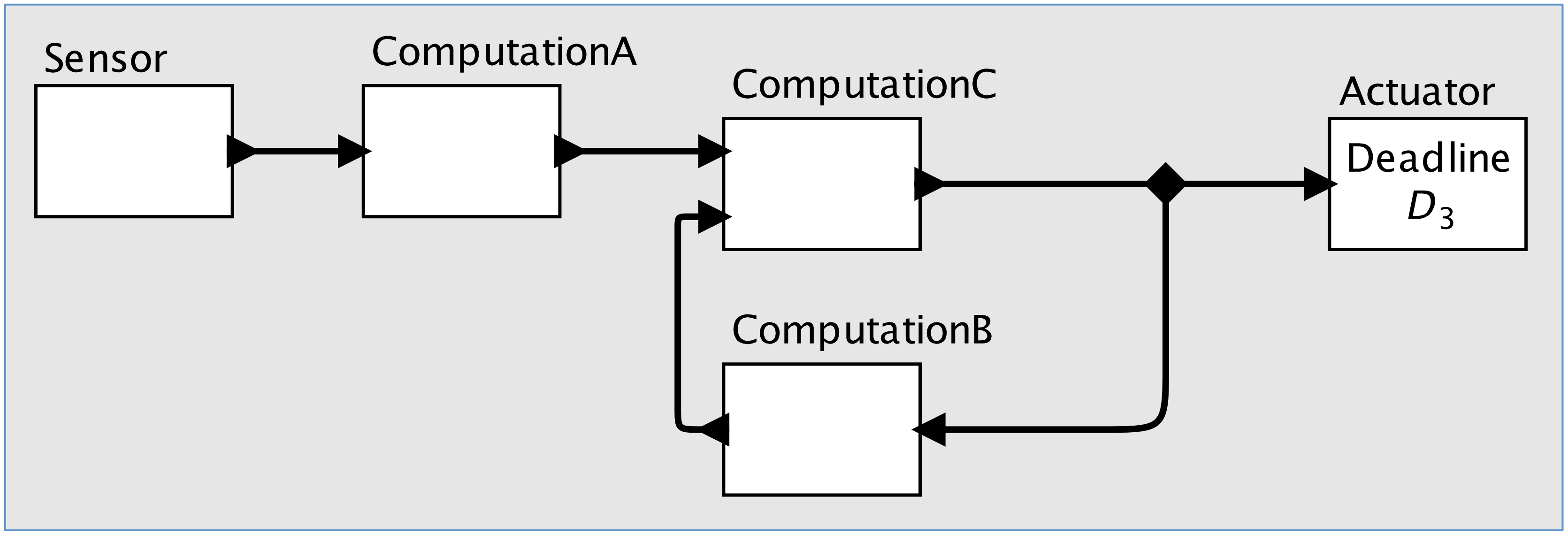


Imposing deadlines on network interfaces decouples the real-time analysis problem. Each execution platform can be individually verified for meeting deadlines.

E.g., $I_2 \geq W_2$, $D_2 \geq W_2$, $D_3 \geq D_2 + W_3$, …

[Zhao et al., 2007]

Other Issues: Feedback



- Fixed-point semantics
- Causality loops
- Superdense time
- …

Projects at Berkeley Focused on *Engineering Models* for CPS

Deterministic models for CPS:

- PTIDES: distributed real-time software
  - http://chess.eecs.berkeley.edu/ptides
- PRET: time-deterministic architectures
  - http://chess.eecs.berkeley.edu/pret
- Lingua Franca: a programming model
  - https://github.com/icyphy/lingua-franca

Together, these technologies give a model for distributed and concurrent real-time systems that is deterministic, has controlled timing, and is implementable.

Model-Based Design of Cyber-Physical Systems

Changing the Question:

Is the question whether we can build models describing the behavior of cyber-physical systems?

Or

Is the question whether we can make cyber-physical systems that behave like our models?