iCyPhy

Software Design for Cyber-Physical Systems

Edward A. Lee

Module 7: Concurrent Models of Computation

Technical University of Vienna
Vienna, Austria, May 2022

University of California, Berkeley

Example MoCs

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

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

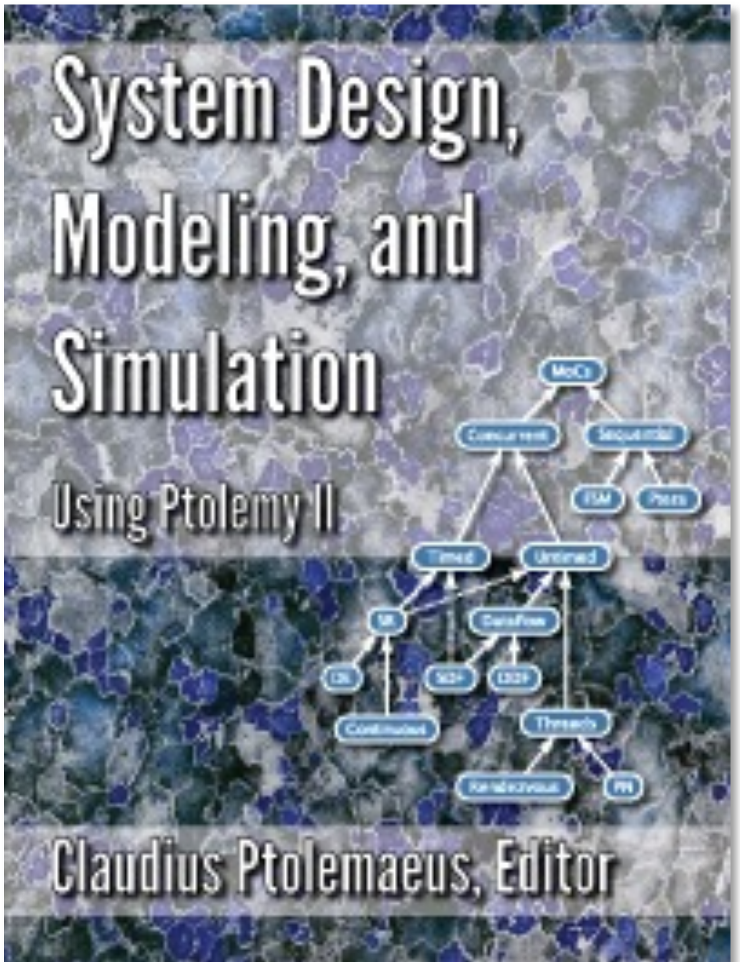
Concurrent:
- Cellular automata
- Kahn process networks
- Petri nets
- Dataflow
- Actors
- CSP (rendezvous)

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

Outline

- Definitions
- Threads
- Alternatives to Threads
- Deterministic Concurrency

Concurrent MoCs



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.

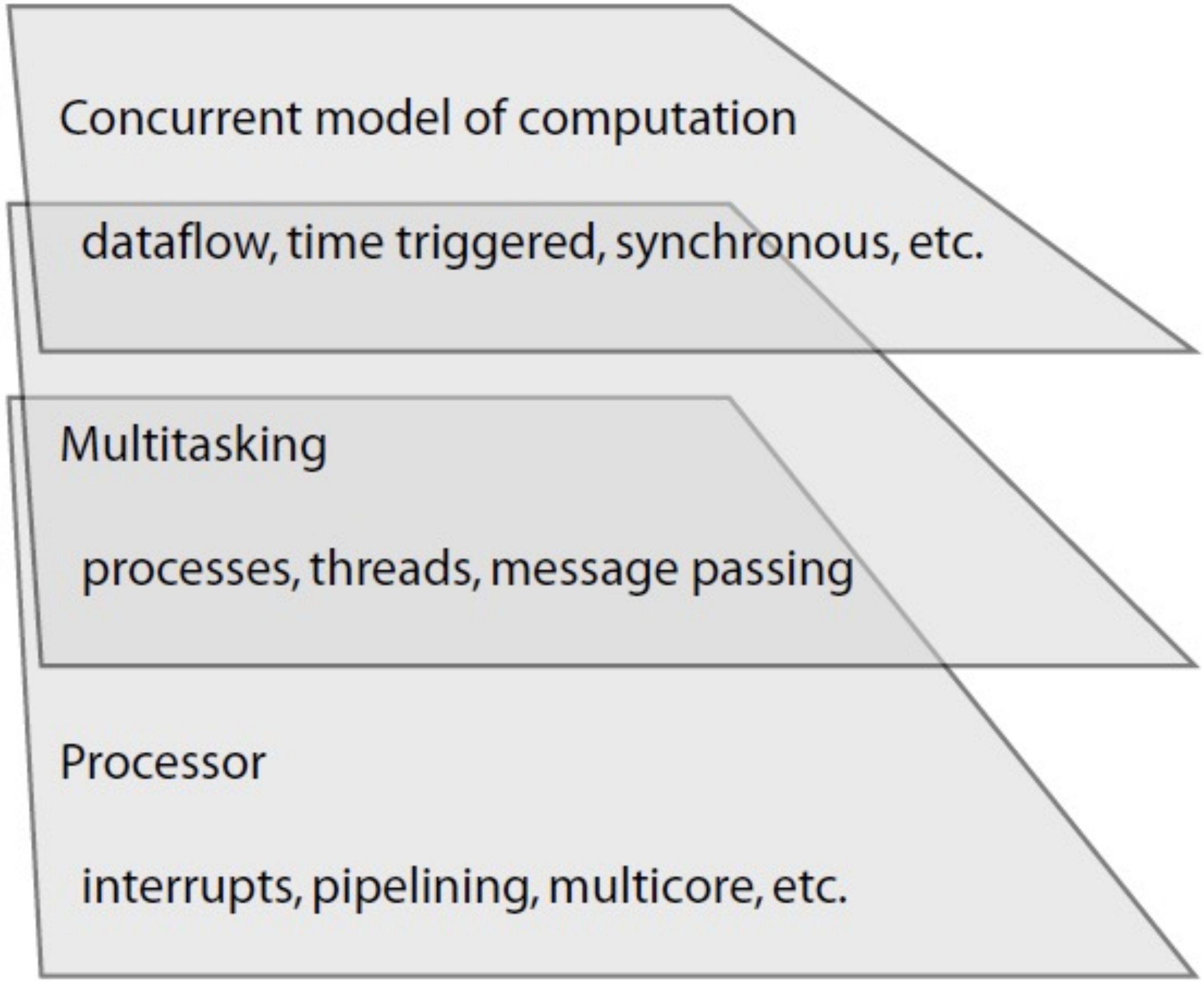
Webster:

- 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



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
- Deterministic Concurrency

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 a 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.

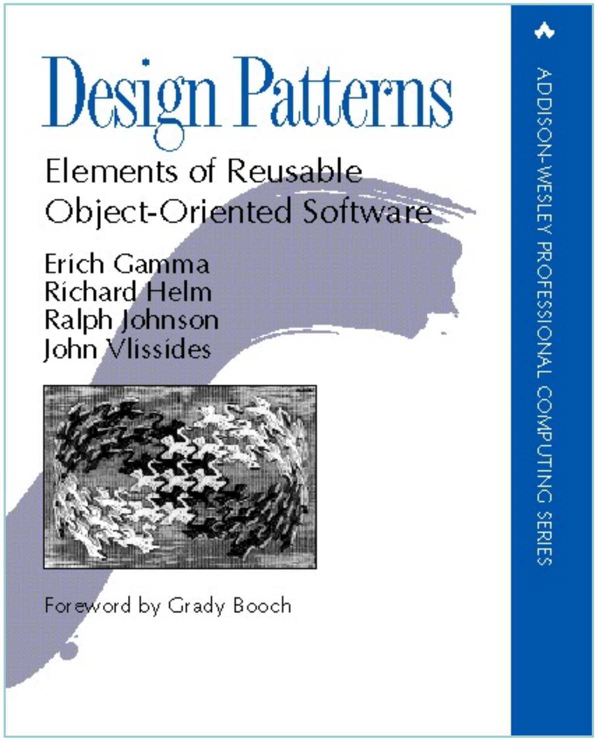
Notes

- 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.

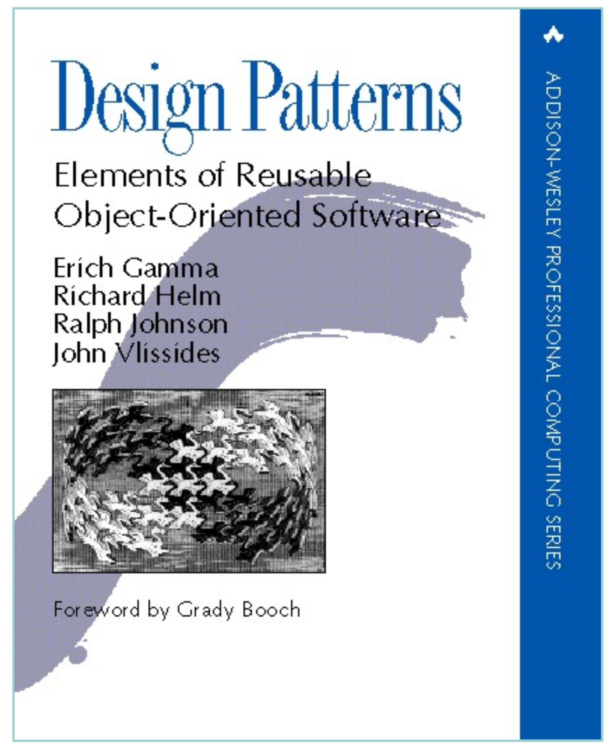


An observer process updates the cockpit display based on notifications from the engine diagnostics.

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)



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 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;
  }
}
```

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(in

// 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 listene

// 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.

```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.

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.

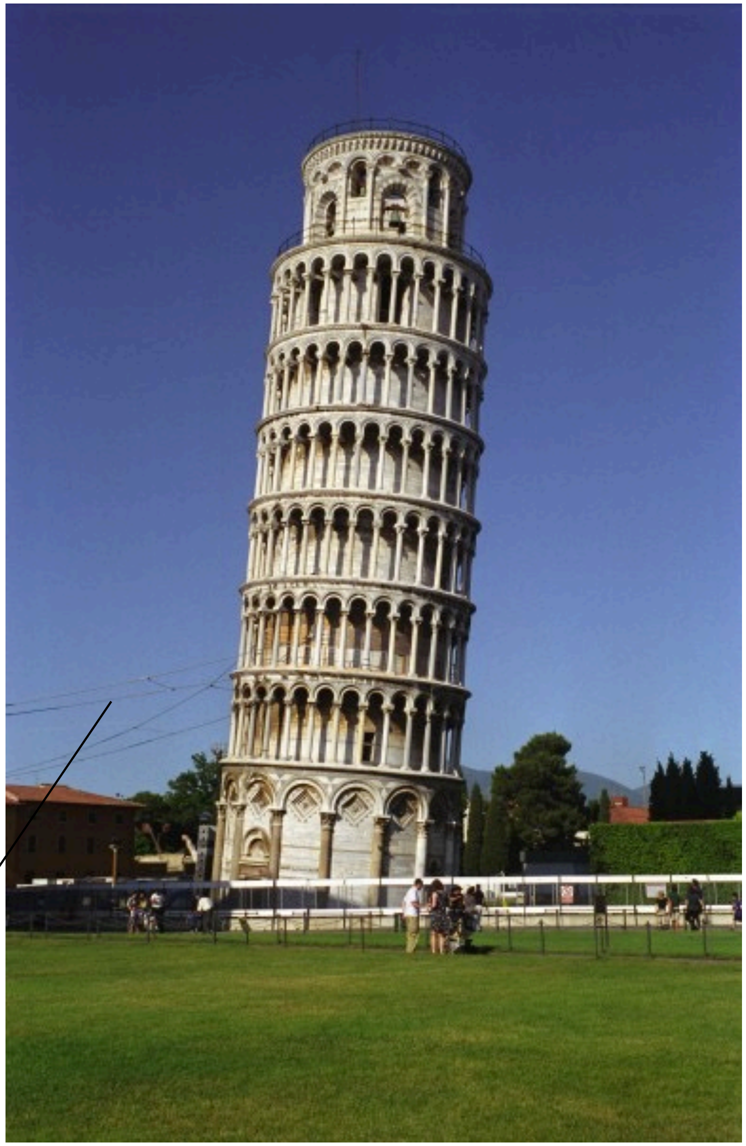
→ *Need better ways to program concurrent systems*

→ *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

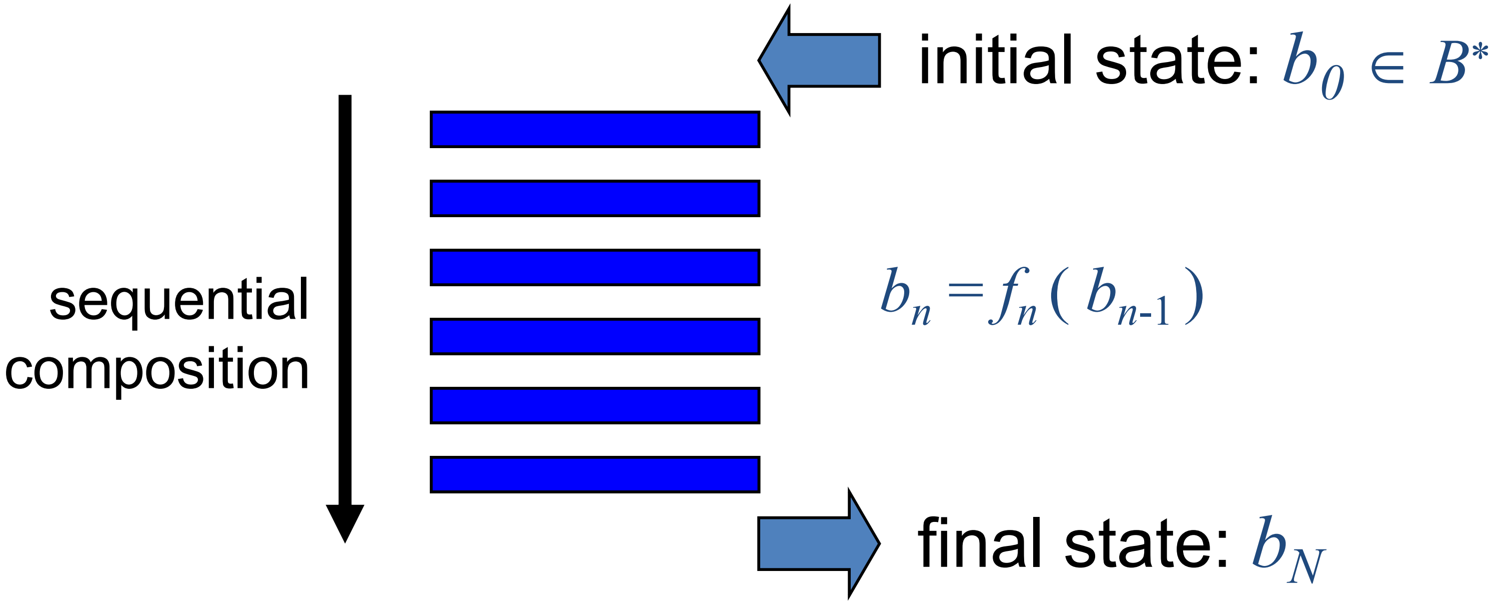
A model of computation:

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

Threads augment this model to admit concurrency.

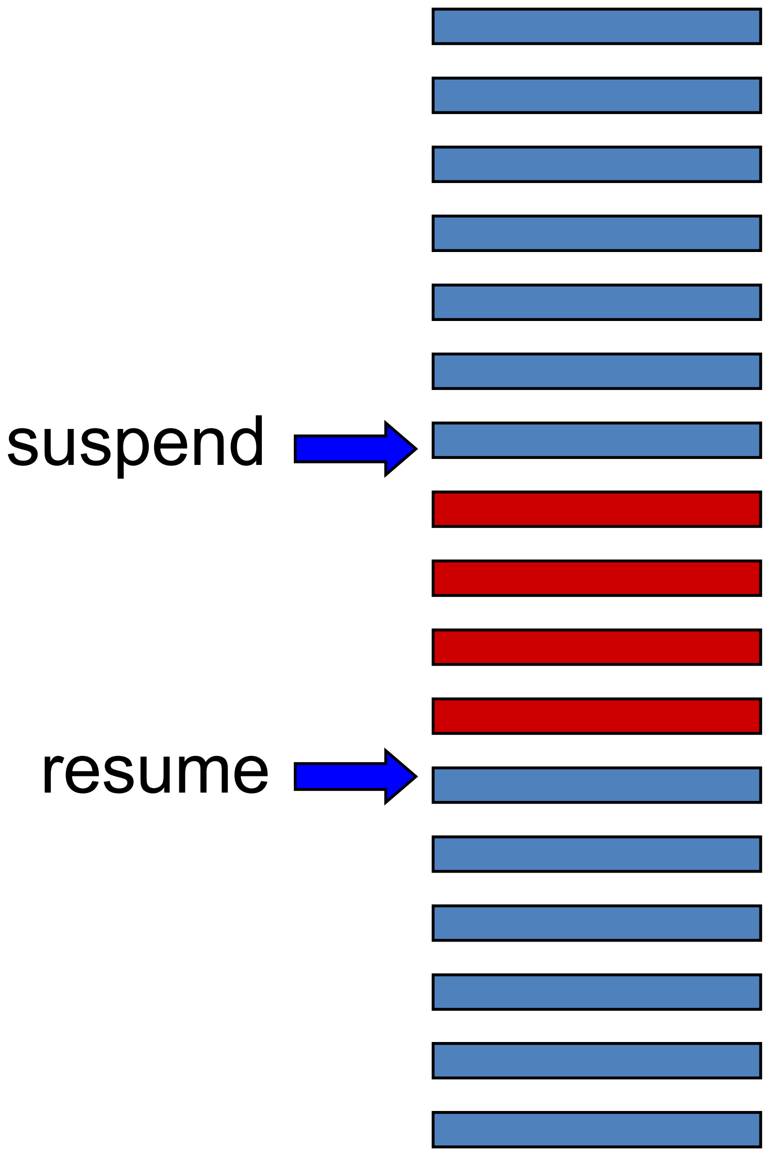
But this model does not admit concurrency gracefully.

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

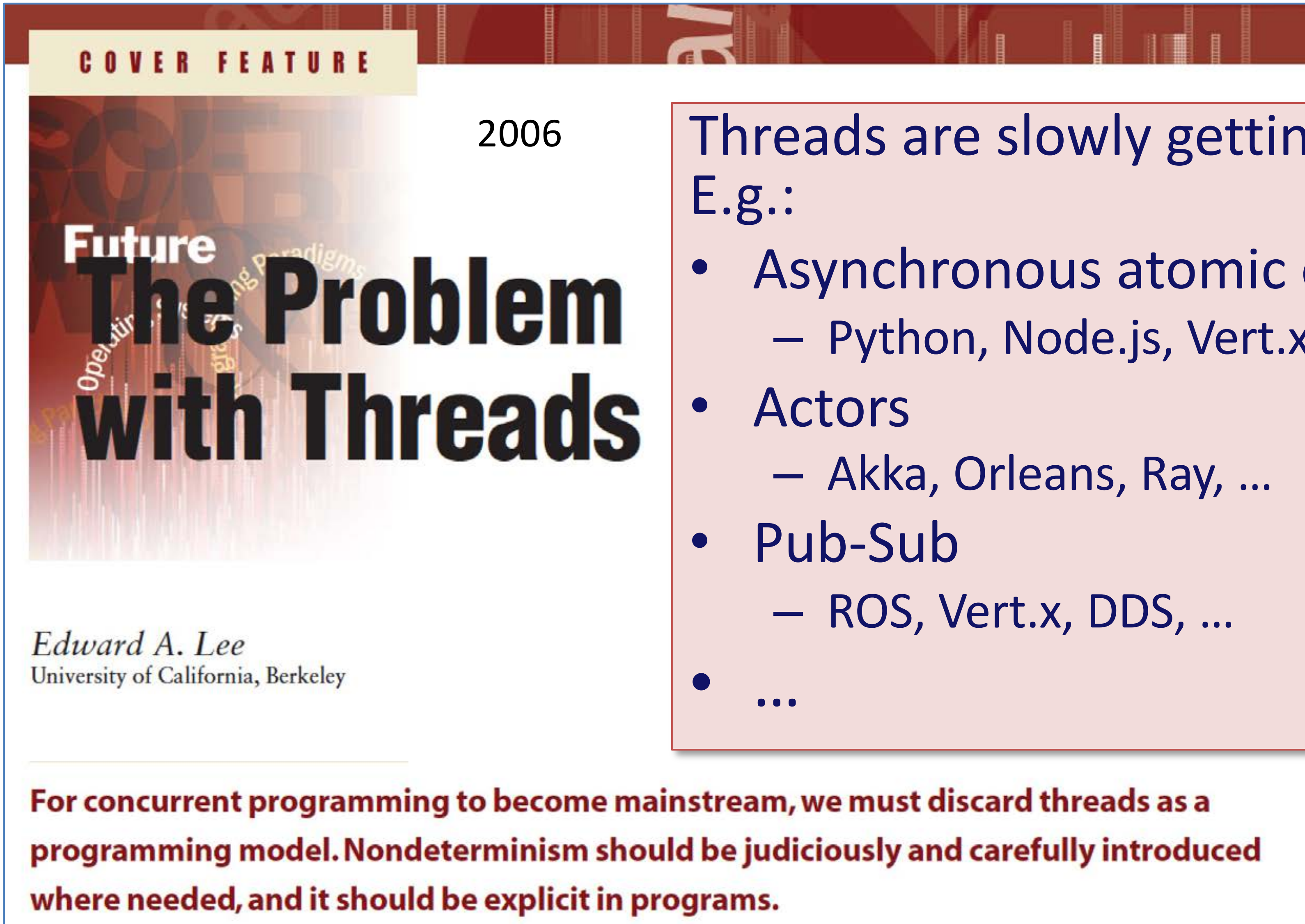
Threads are wildly nondeterministic.

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

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.

2006

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

Outline

- Definitions
- Threads
- Alternatives to Threads
- Deterministic Concurrency

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:

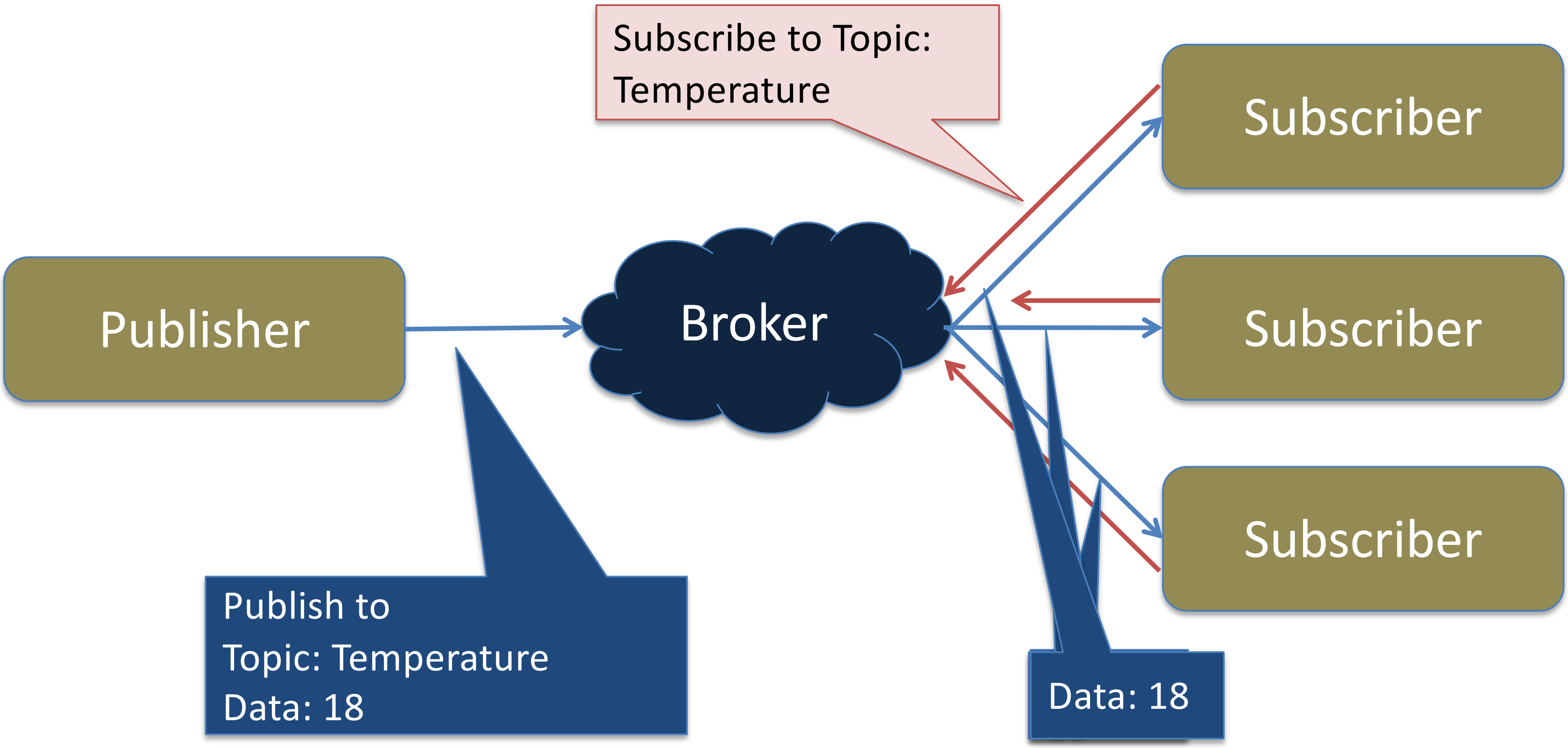
```
0, 0, 0, 0, 0, −1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, …
```

"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)



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 from *The Telegraph*, Sept. 9, 2015

Another Answer to Threads: Actors

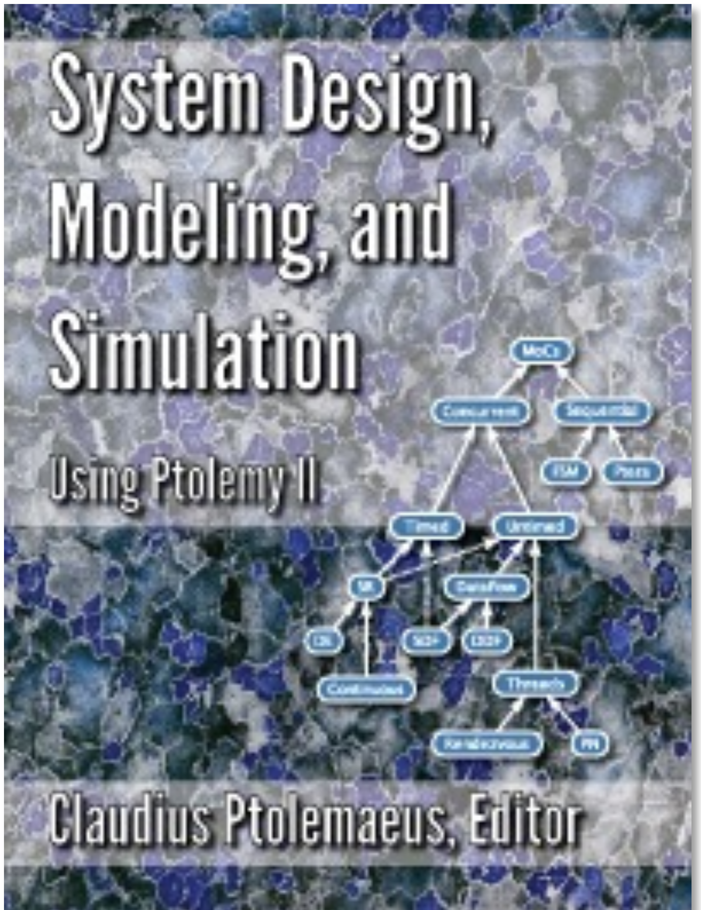
Actors are concurrent objects that communicate by sending each other messages.

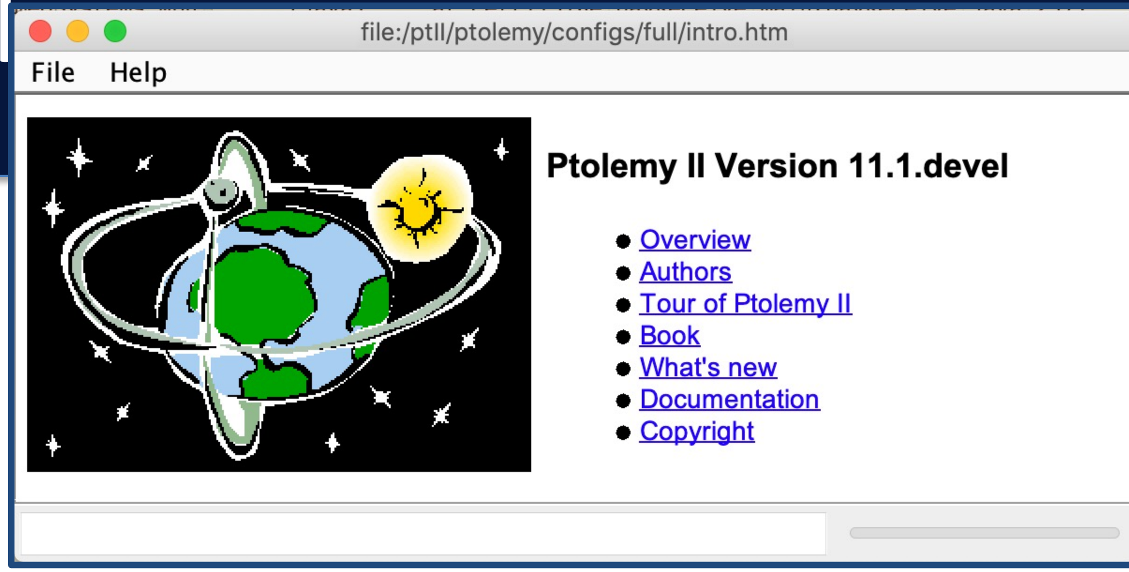
- Erlang [Armstrong, et al. 1996]
- Rebeca [Sirjani and Jaghoori, 2011]
- Akka [Roestenburg, et al. 2017]
- Ray [Moritz, et al. 2018]
- …

Outline

- Definitions
- Threads
- Alternatives to Threads
- Deterministic Concurrency

Ptolemy II





http://ptolemy.org

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

Deterministic Concurrent MoCs

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

Dataflow

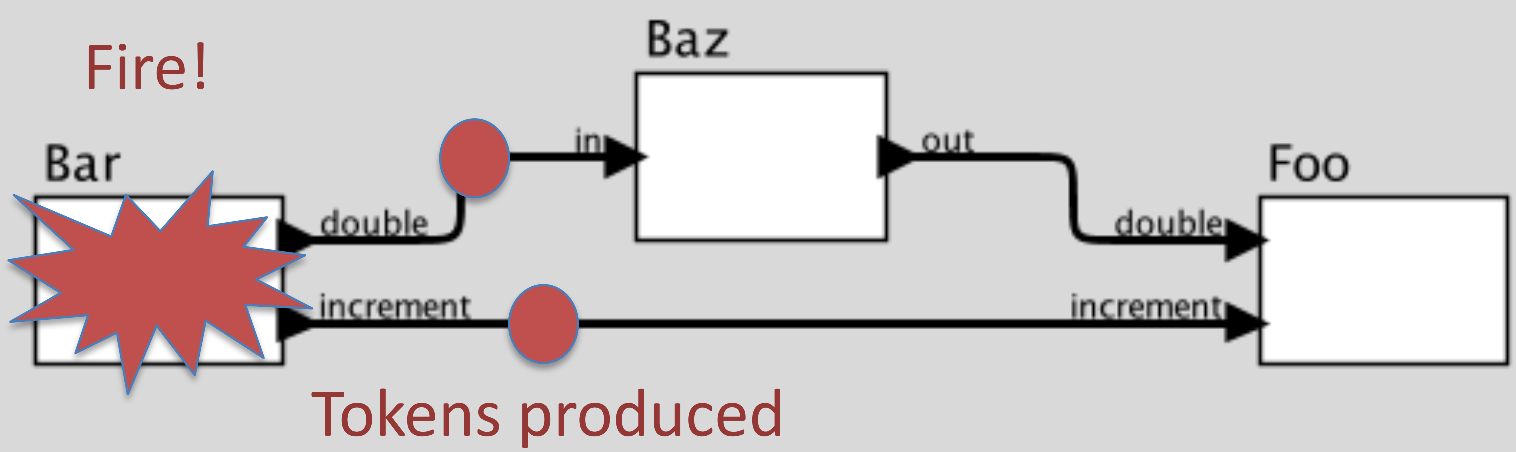
- Computation Graphs [Karp, 1966]
- Dataflow [Dennis, 1974]
- Dynamic dataflow [Arvind, 1981]
- Structured dataflow [Matwin & Pietrzykowski 1985]
- K-bounded loops [Culler, 1986]
- Synchronous dataflow [Lee & Messerschmitt, 1986]
- Structured dataflow and LabVIEW [Kodosky, 1986]
- PGM: Processing Graph 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]

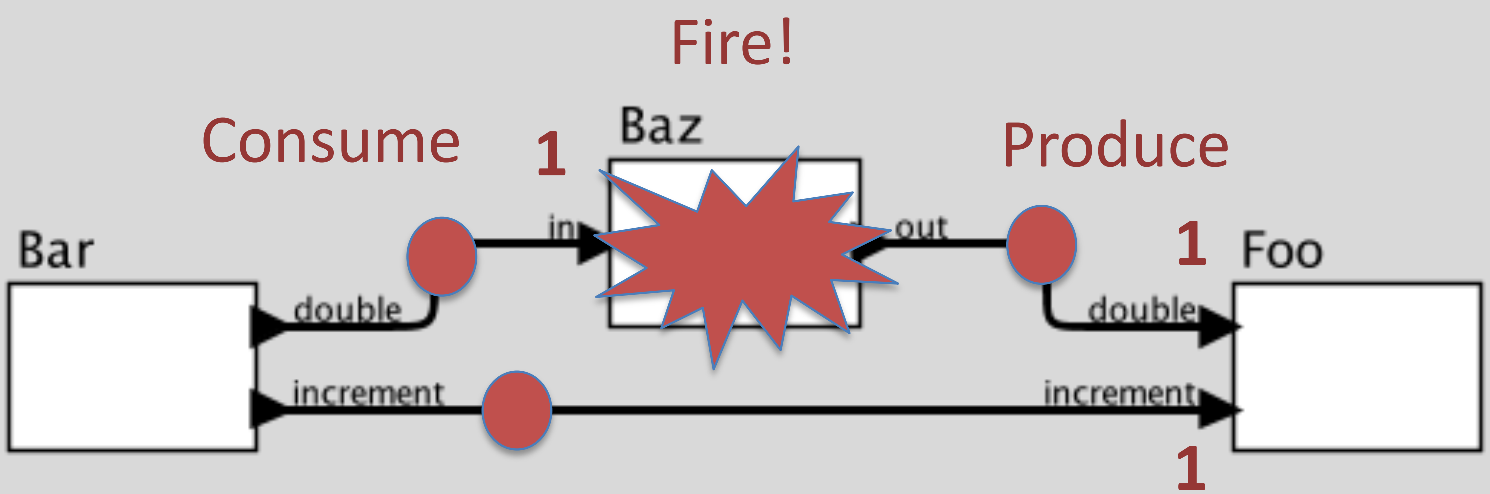
An actor with no inputs can fire at any time.



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.

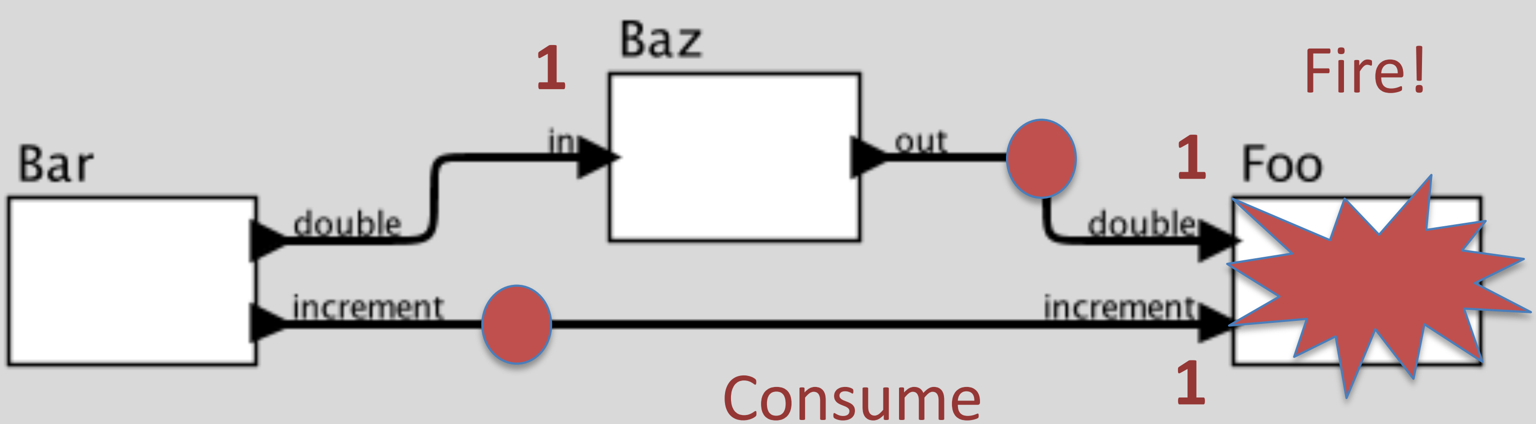


Dataflow Solution for Scheduling: Firing Rules

[Lee & Matsikoudis, 2009]

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

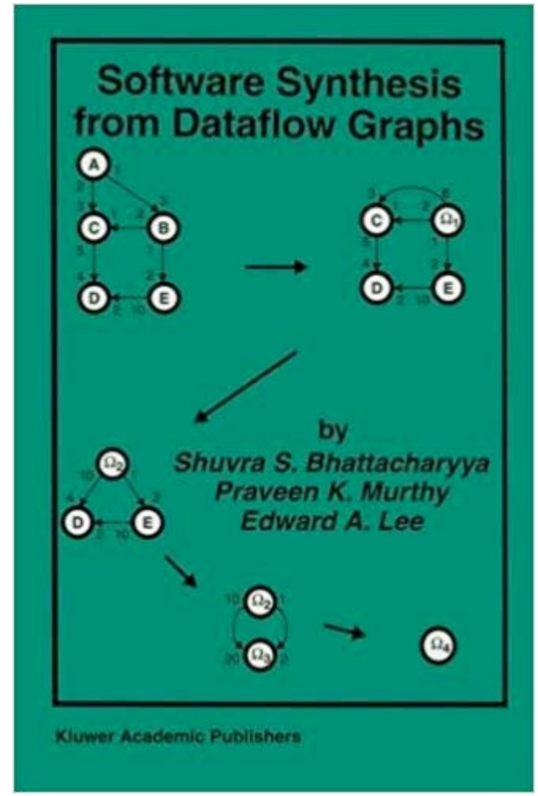
When it fires, each reaction is invoked in a deterministic order.



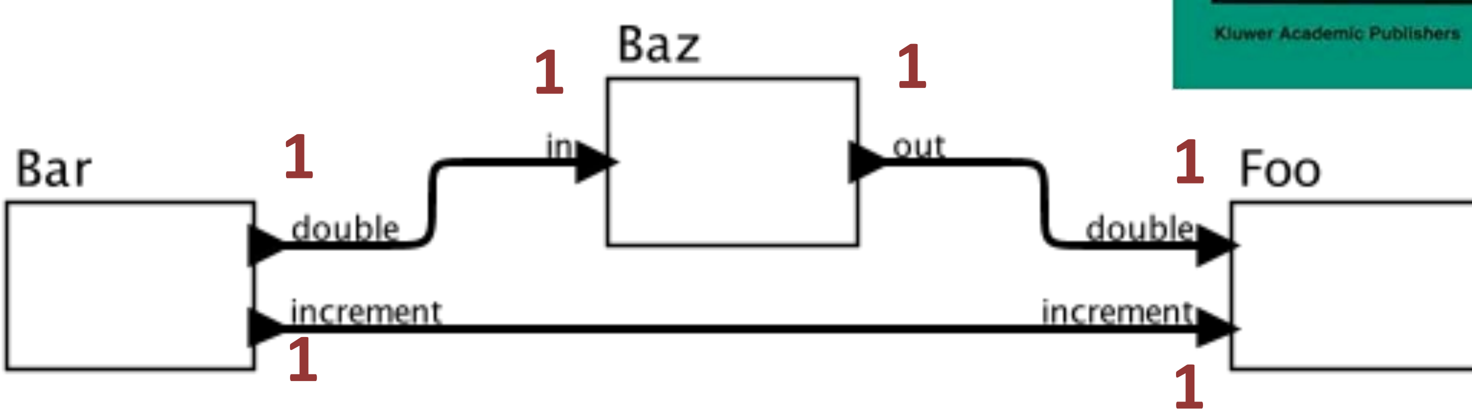
Synchronous Dataflow Scheduling

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

[Lee & Messerschmitt, 1986]



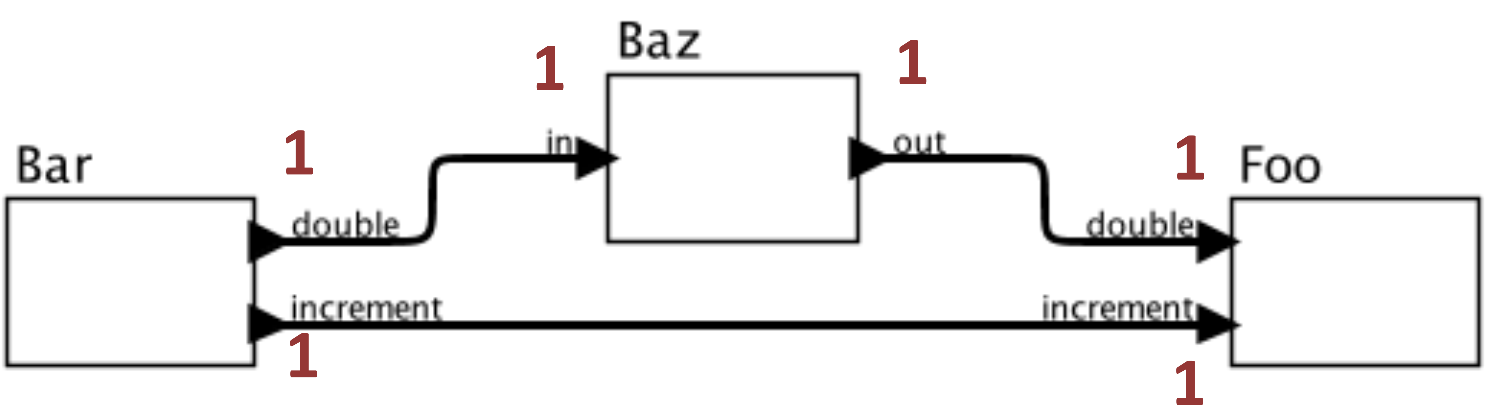
1996



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).

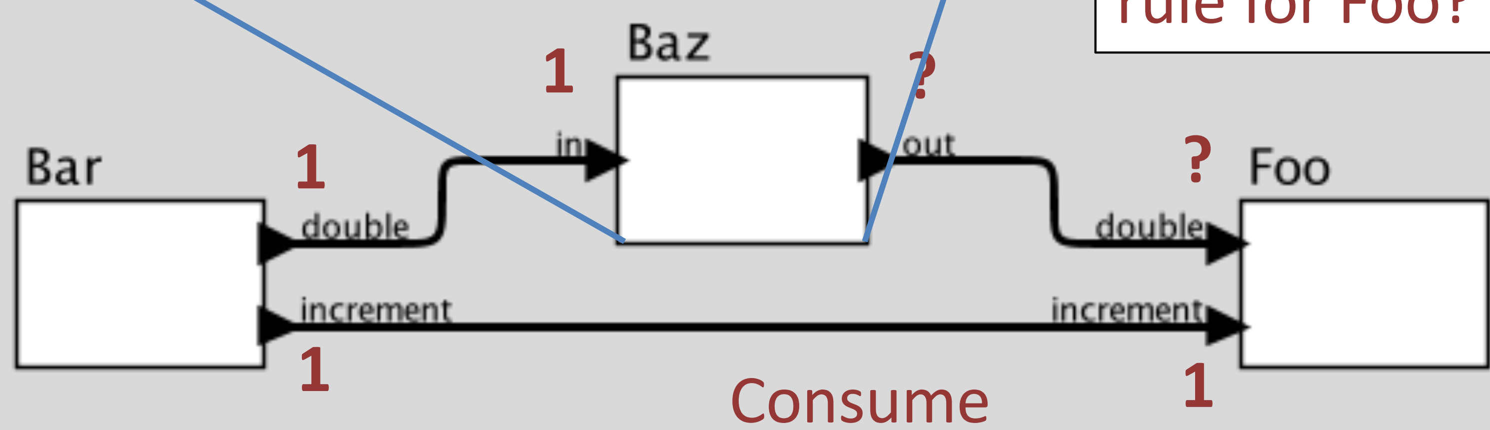
[Lee & Messerschmitt, 1986]



Dataflow Scheduling with Dynamic Firing Rules

```
reactor Baz {
    input in;
    output out;
    reaction(in){
        if (something) {
            send(out);
        }
    }
}
```

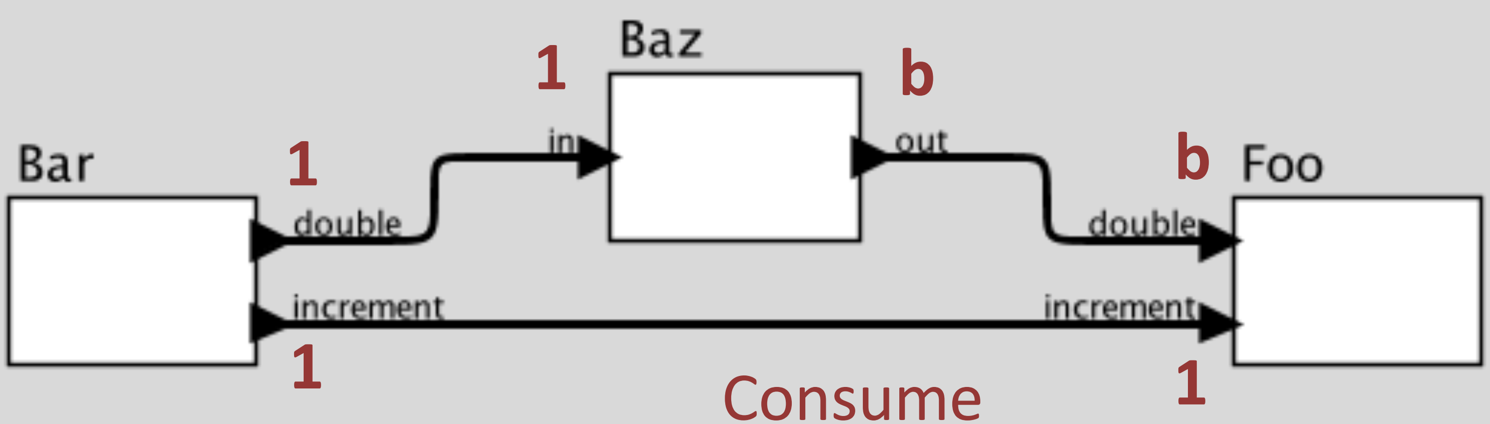
What should be the firing rule for Foo?



Boolean Dataflow

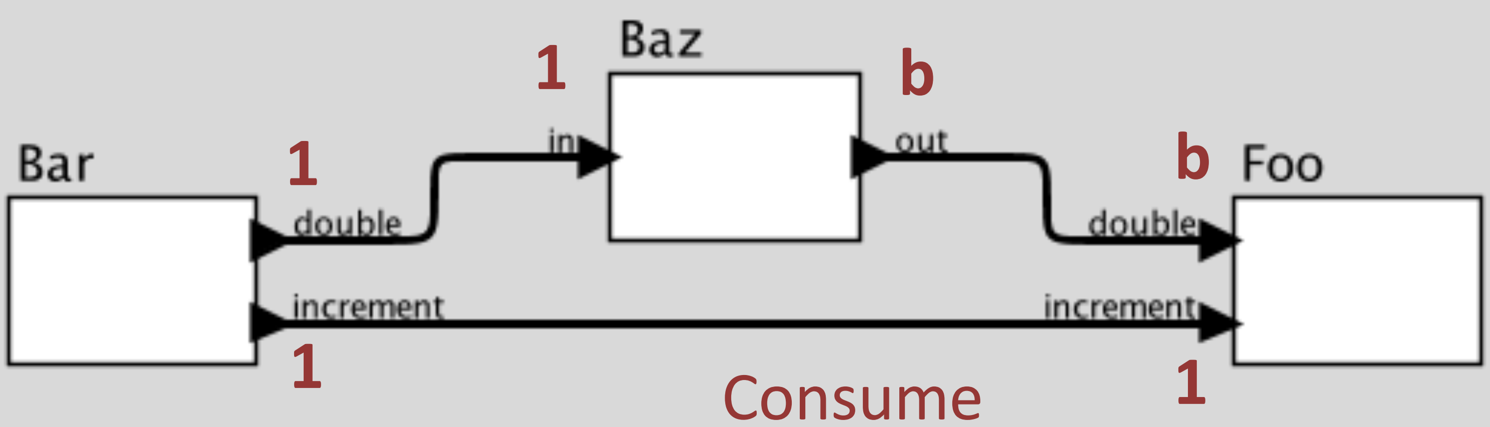
Buck [1993] showed that scheduling problems in general are undecidable in this framework.

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



Various Dataflow Variants that Remain Decidable

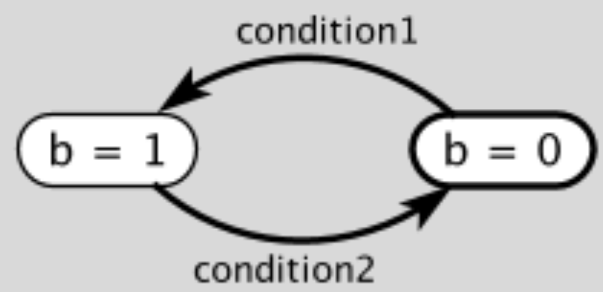
- Cyclostatic dataflow [Lauwereins 1994]
- Heterochronous dataflow [Girault, Lee & Lee, 1997]
- Parameterized dataflow [Bhattacharya & Bhattacharyya, 2001]
- Structured dataflow [Thies, 2002]
- Scenario-aware dataflow [Theelen, Geilen, Basten, et al. 2006]
- Reconfigurable dataflow [Fradet, Girault,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

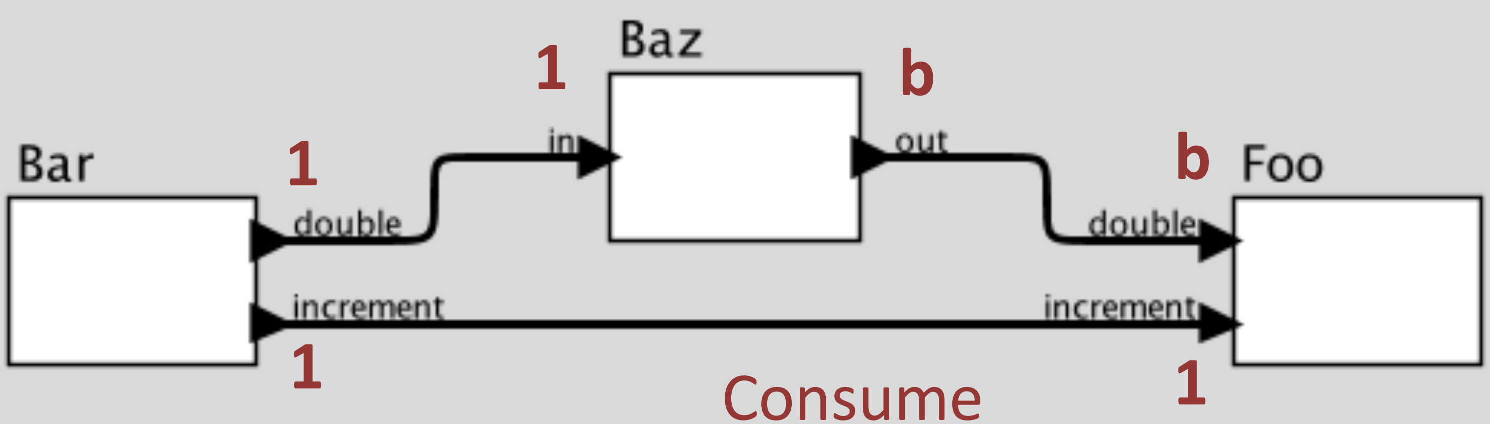


Scenario-Aware Dataflow

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

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





Some Strategies

- 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, increment;
    int state = 1;
    while(true) {
        read(double);
        state *= 2;
        x = read(increment);
        state += x;
        print state;
    }
}
```

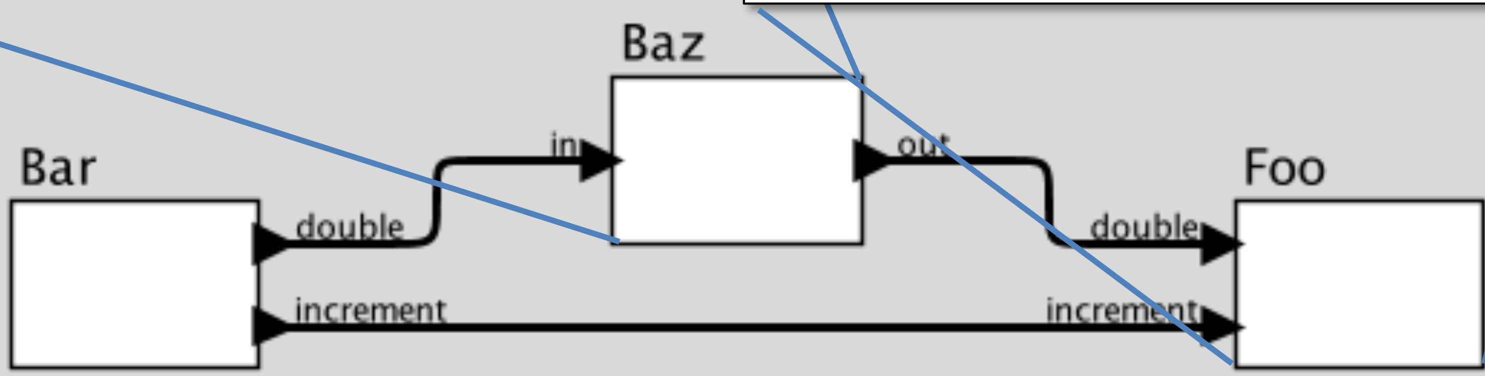
Gilles Kahn

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

Blocking reads have trouble with data-dependent flow patterns

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

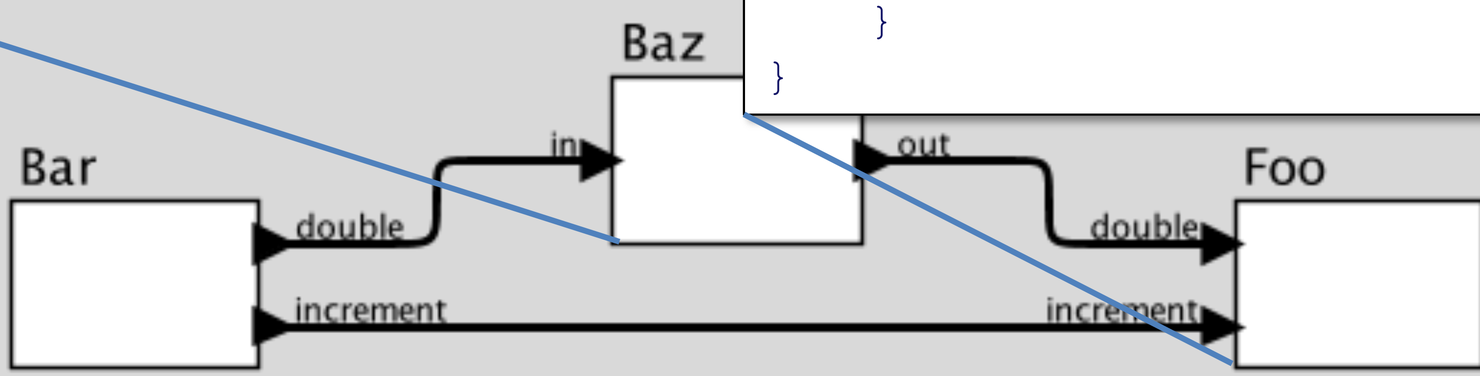
```
KPNActor Foo {
    input double, increment;
    int state = 1;
    while(true) {
        read(double);
        state *= 2;
        x = read(increment);
        state += x;
        print state;
    }
}
```



Blocking reads have trouble with data-dependent flow patterns

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

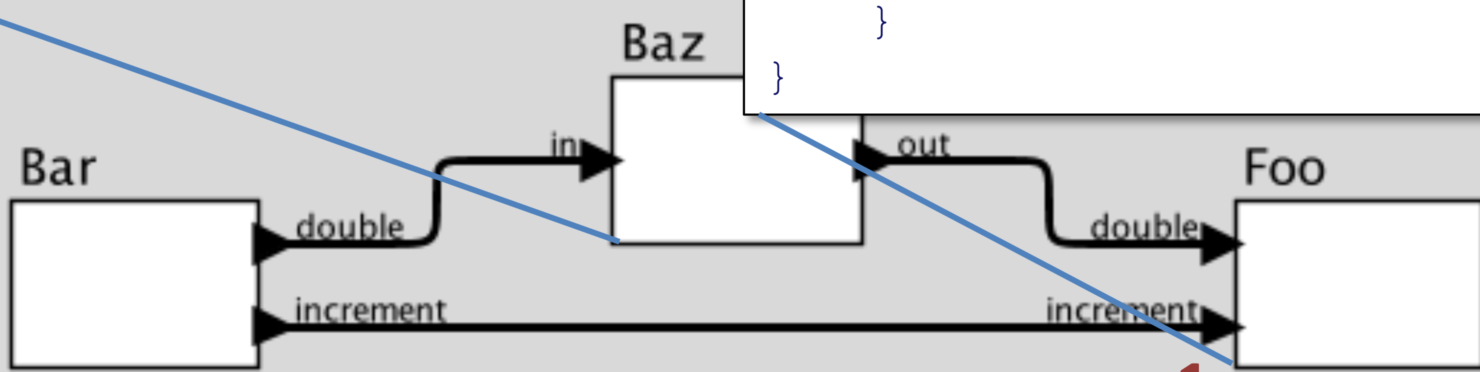
```
KPNActor Foo {
    input double, increment;
    int state = 1;
    while(true) {
        if (something) {
            read(double);
            state *= 2;
        }
        x = read(increment);
        state += x;
        print state;
    }
}
```



Solution: Coordinated Control

```
Actor Baz {
    input in;
    output out;
    handler in(){
        if (something) {
            out.send();
        }
    }
}
```

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



Consume 1

Some Strategies

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

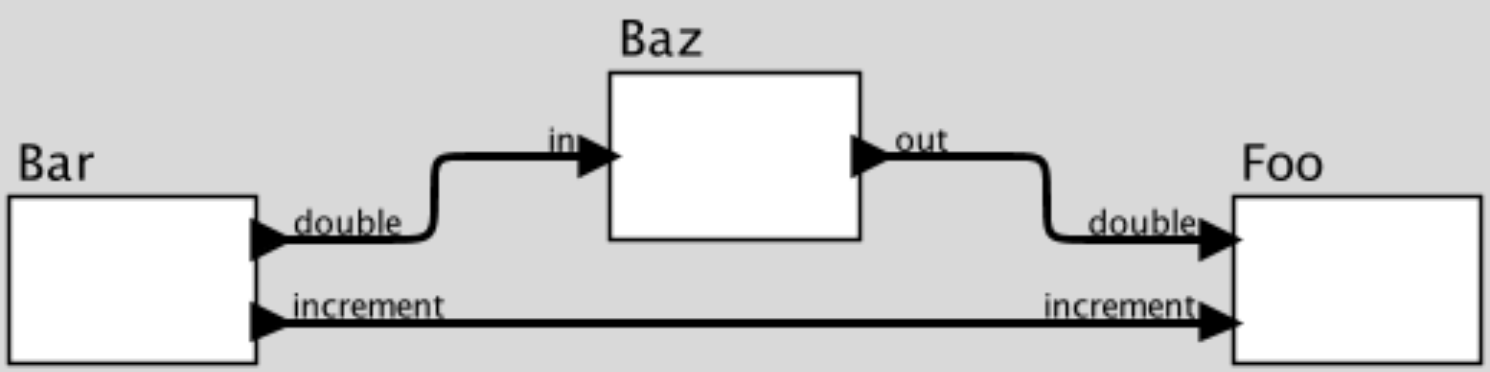
An Alternative Approach to Coordination

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

A Different Approach: Synchronous Languages

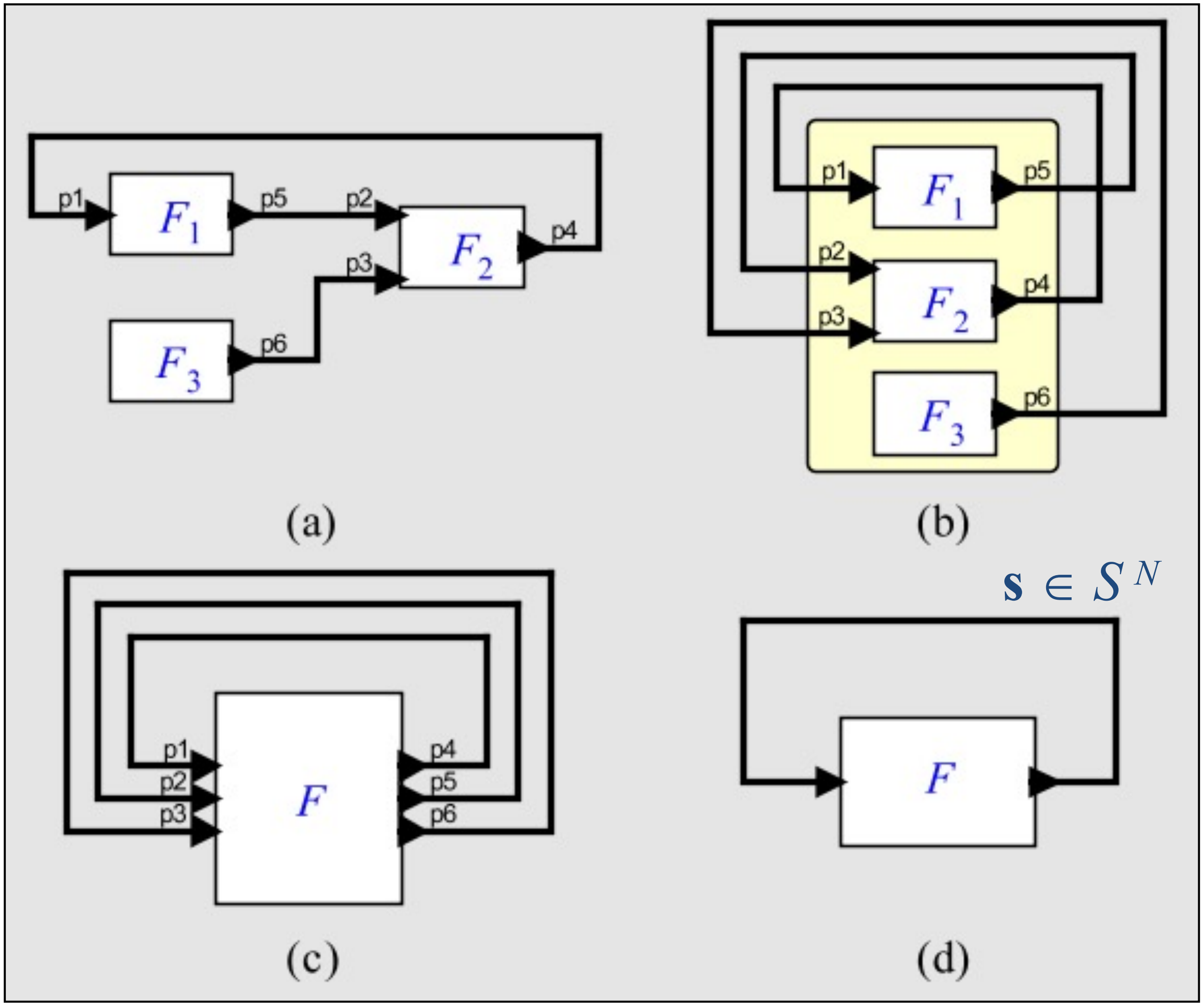
In the synchronous/reactive approach, there is a conceptual global "clock," and on each "tick" of this clock, a connection either has a well-defined value or is "absent."

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]

Some Strategies

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

Discrete-Event Languages

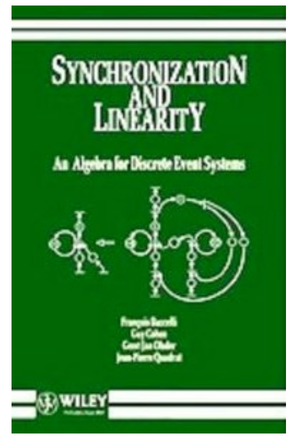
DE is a generalization of SR, where there is a notion of "time between ticks."

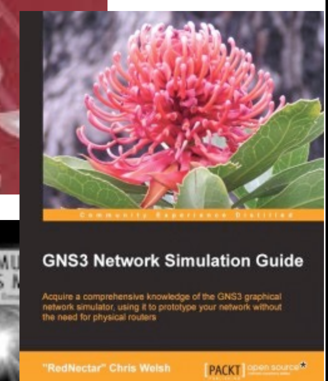
WARNING: immediately have (at least) two time lines: logical time and physical time(s).

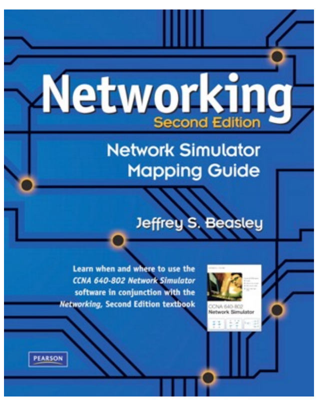
[Lee & Zheng, 2007]

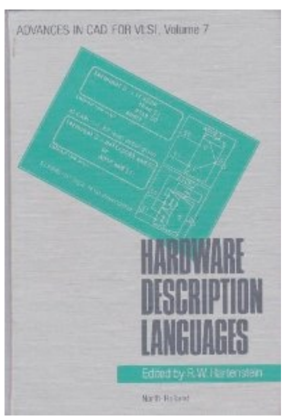
Discrete Events (DE)

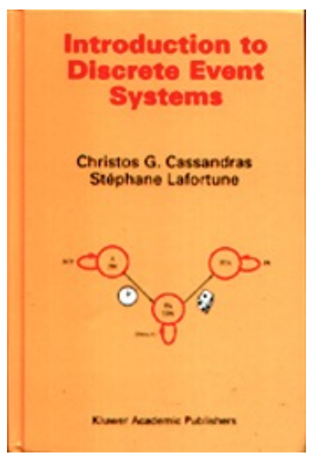
- Events that are processed in timestamp order.
- Widely used in simulation
- Foundation of hardware description languages.
- A deterministic concurrent MoC.
- But how to realize on distributed machines?

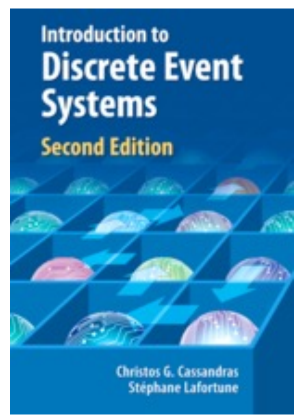


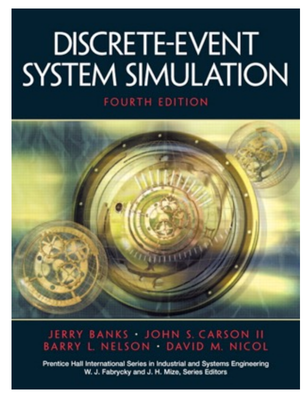












And Lingua Franca!

References

Many dataflow papers: https://ptolemy.berkeley.edu/publications/dataflow.htm

- Agha, G. A. (1997). Abstracting Interaction Patterns: A Programming Paradigm for Open Distributed Systems. Formal Methods for Open Object-based Distributed Systems, IFIP Transactions, Chapman and Hall.
- Agha, G. (1990). "Concurrent object-oriented programming." Communications of the ACM 33(9): 125-140.
- Agha, G. (1986). ACTORS: A Model of Concurrent Computation in Distributed Systems. Cambridge, MA, MIT Press.
- Armstrong, J., et al. (1996). Concurrent programming in Erlang, Prentice Hall.
- Benveniste, A. and G. Berry (1991). "The Synchronous Approach to Reactive and Real-Time Systems." Proceedings of the IEEE 79(9): 1270-1282.
- Bhattacharya, B. and S. S. Bhattacharyya (2000). Parameterized Dataflow Modeling of DSP Systems. International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Istanbul, Turkey.
- Buck, J. T. and E. A. Lee (1993). Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP).
- Buttazzo, G. C. (2005). Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Springer.

References

- Edwards, S. A. and E. A. Lee (2007). The Case for the Precision Timed (PRET) Machine. Design Automation Conference (DAC), San Diego, CA.
- Edwards, S. A. and E. A. Lee (2003). "The Semantics and Execution of a Synchronous Block-Diagram Language." Science of Computer Programming 48(1): 21-42.
- Fradet, P., et al. (2019). RDF: Reconfigurable Dataflow. Design Automation in Europe (DATE), Florence, Italy.
- Girault, A., et al. (1999). "Hierarchical Finite State Machines with Multiple Concurrency Models." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 18(6): 742-760.
- Hewitt, C. (1977). "Viewing control structures as patterns of passing messages." Journal of Artificial Intelligence 8(3): 323-363.
- Kahn, G. (1974). The Semantics of a Simple Language for Parallel Programming. Proc. of the IFIP Congress 74, North-Holland Publishing Co.
- Kahn, G. and D. B. MacQueen (1977). Coroutines and Networks of Parallel Processes. Information Processing, North-Holland Publishing Co.

References

- Lee, E. A., et al. (2017). Abstract {PRET} Machines. IEEE Real-Time Systems Symposium (RTSS), Paris, France.
- Lee, E. A. and E. Matsikoudis (2009). The Semantics of Dataflow with Firing. From Semantics to Computer Science: Essays in memory of Gilles Kahn. G. Huet, G. Plotkin, J.-J. Levy and Y. Bertot, Cambridge University Press.
- Lee, E. A. and D. G. Messerschmitt (1987). "Synchronous Data Flow." Proceedings of the IEEE 75(9): 1235-1245.
- Lee, E. A. and H. Zheng (2007). Leveraging Synchronous Language Principles for Heterogeneous Modeling and Design of Embedded Systems. EMSOFT, Salzburg, Austria, ACM.
- Lee, E. A. and H. Zheng (2005). Operational Semantics of Hybrid Systems. Hybrid Systems: Computation and Control (HSCC), Zurich, Switzerland, Springer-Verlag.
- Bilsen, G., et al. (1996). "Cyclo-static dataflow." IEEE Transactions on Signal Processing 44(2): 397-408.
- Moritz, P., et al. (2018). "Ray: A Distributed Framework for Emerging AI Applications." Xiv:1712.05889v2 [cs.DC] 30 Sep 2018.
- Ptolemaeus, C., Ed. (2012). System Design, Modeling, and Simulation Using Ptolemy II. Berkeley, CA, USA, Ptolemy.org.

References

- Sha, L., et al. (2009). PALS: Physically Asynchronous Logically Synchronous Systems, Univ. of Illinois at Urbana Champaign (UIUC).
- Sirjani, M. and M. M. Jaghoor (2011). Ten Years of Analyzing Actors: Rebeca Experience. Formal Modeling: Actors, Open Systems, Biological Systems. Agha G., Danvy O. and M. J. Berlin, Heidelberg, Springer. Lecture Notes in Computer Science, vol 7000.
- Theelen, B. D., et al. (2006). A Scenario-Aware Data Flow Model for CombinedLong-Run Average and Worst-Case Performance Analysis. Formal Methods and Models for Co-Design.
- Thies, W., et al. (2002). {StreamIt}: A Language for Streaming Applications. 11th International Conference on Compiler Construction, Grenoble, France, Springer-Verlag.
- Wilhelm, R., et al. (2008). "The worst-case execution-time problem - overview of methods and survey of tools." ACM Transactions on Embedded Computing Systems (TECS) 7(3): 1-53.