iCyPhy

Actors Revisited for Time-Critical Systems

Edward A. Lee
Professor of the Graduate School
UC Berkeley

Invited Talk

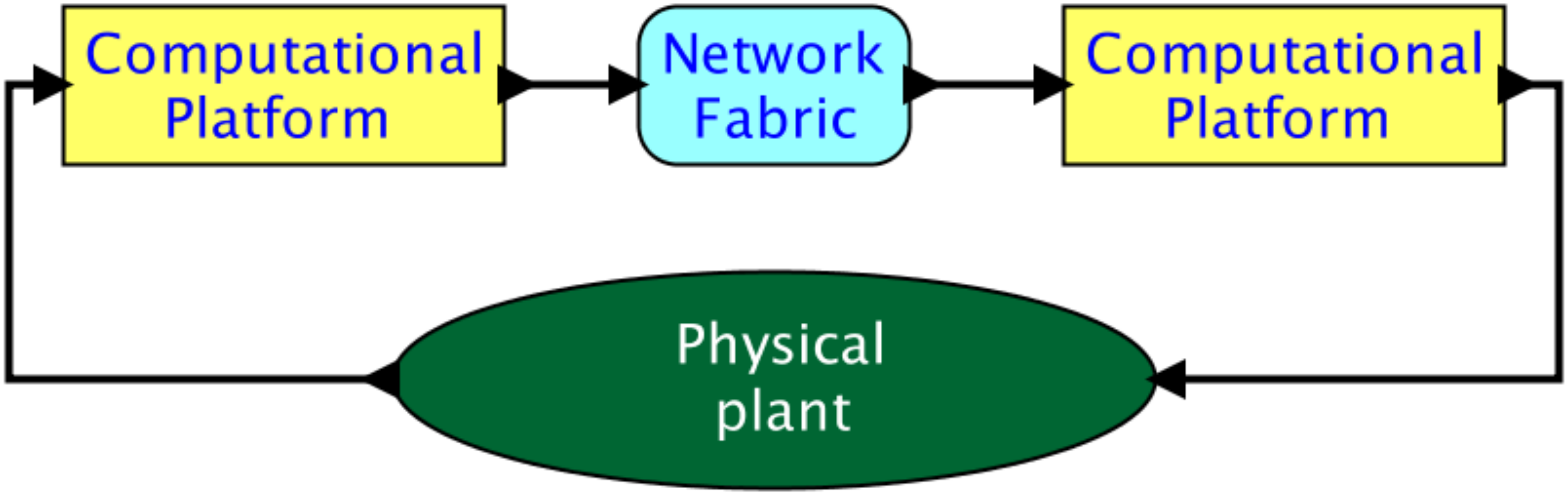
With:
Marten Lohstroh
Martin Schoeberl
Andrés Goens
Armin Wasicek
Christopher Gill
Marjan Sirjani

Software Engineering and Computer Systems (SE&CS), ITMO University

St. Petersburg, Russia, March 25, 2019

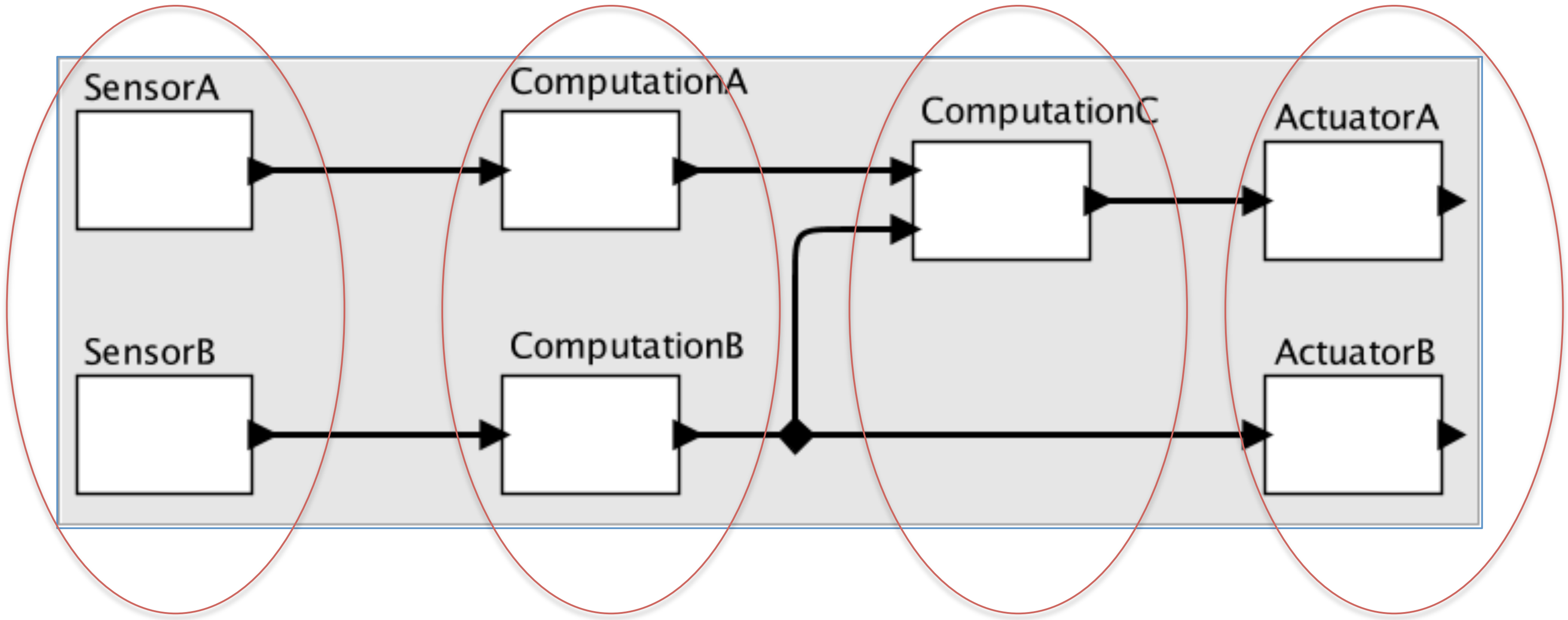
University of California at Berkeley

Cyber Physical Systems



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?

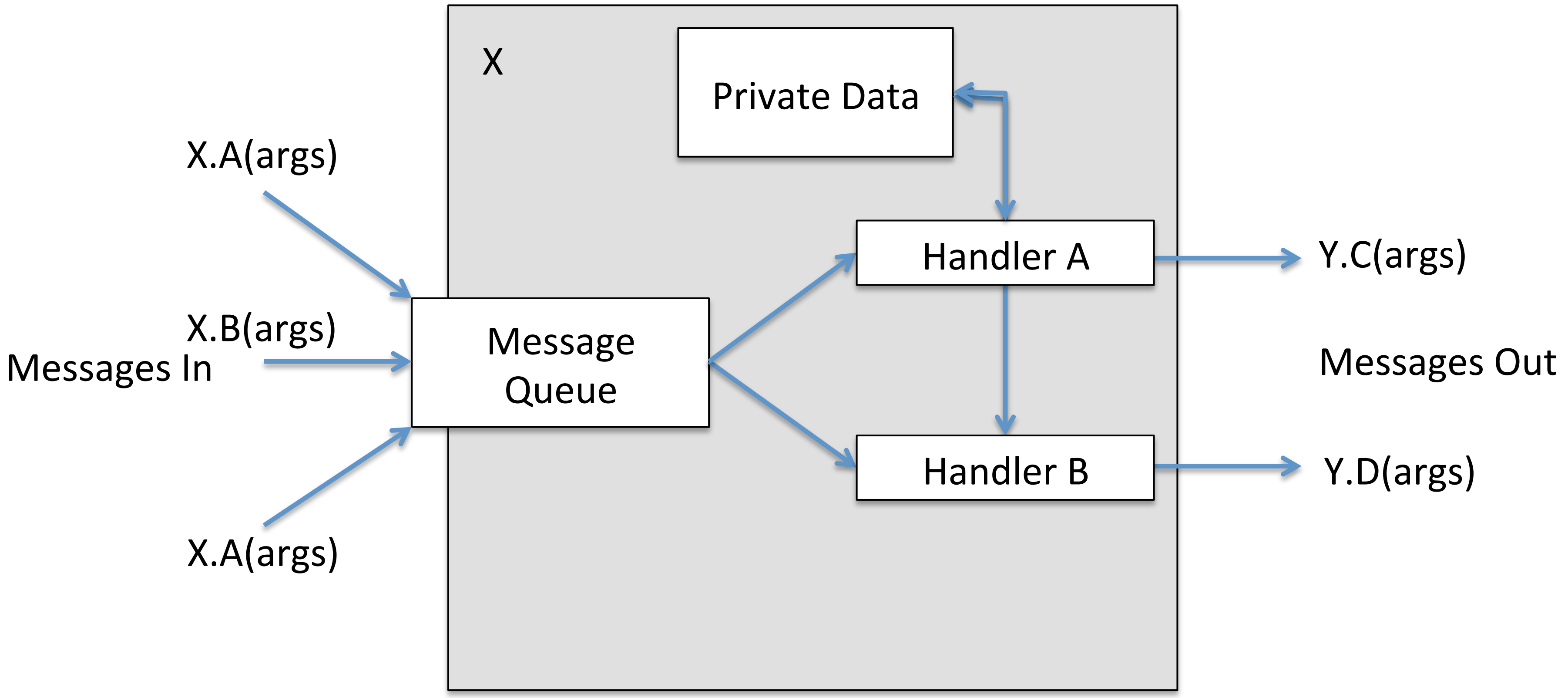
How do we specify, ensure, and enforce deadlines?

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]

Some Realizations of Hewitt/Agha Actors

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

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?

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.

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);
    }
}
```

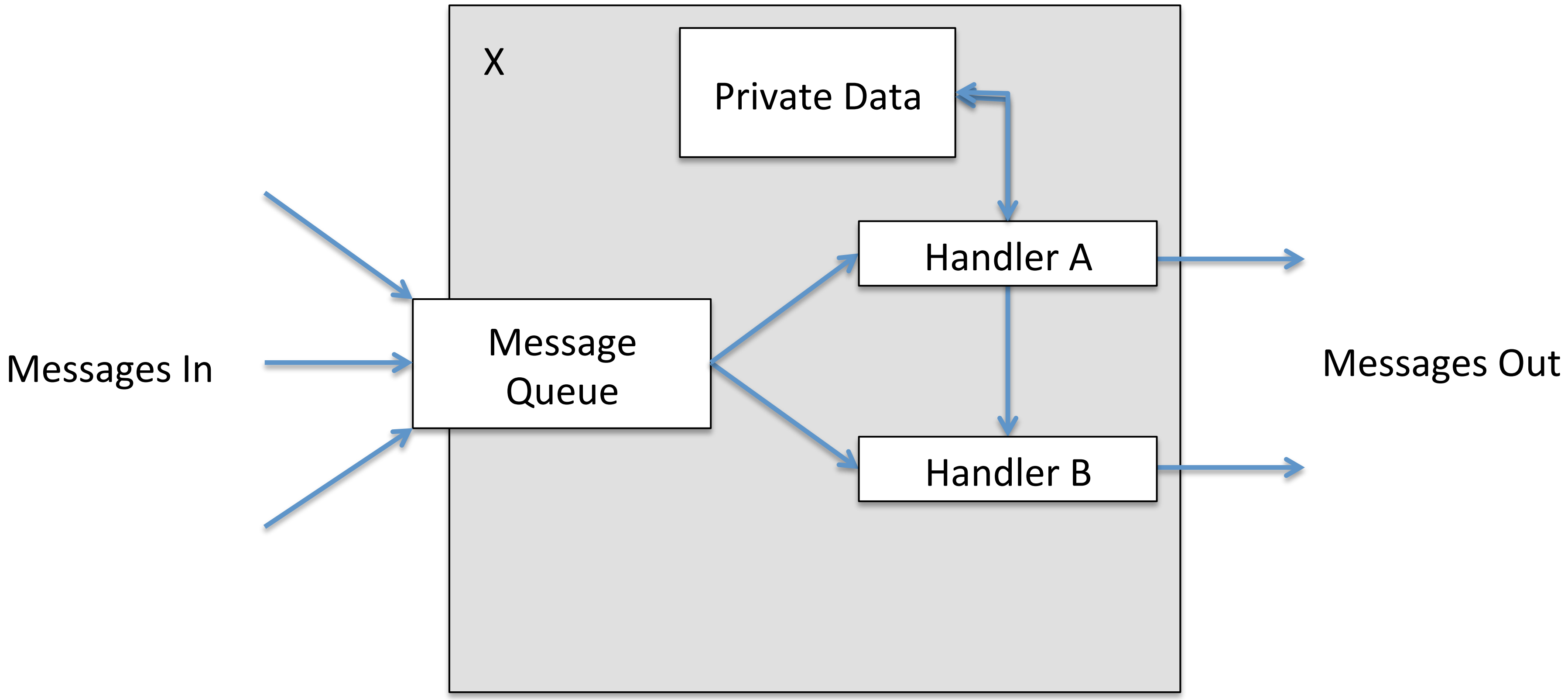
```
Actor Baz {
    handler pass(Foo x){
        x.double();
    }
}
```

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.



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);
    }
}
```

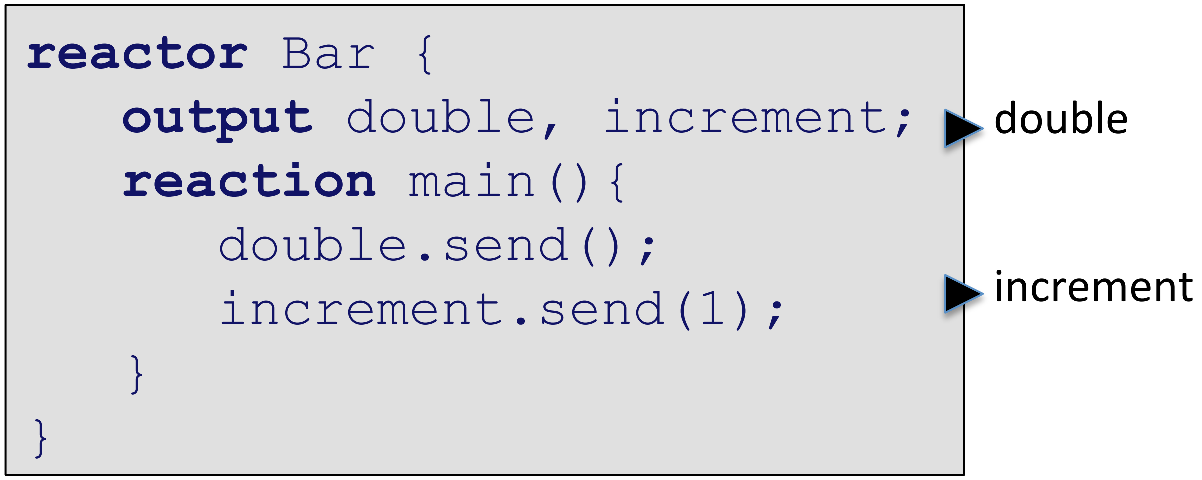
```
Actor Baz {
    handler pass(Foo x){
        if (something) {
            x.double();
        }
    }
}
```

And what if the dependence graph is data dependent?

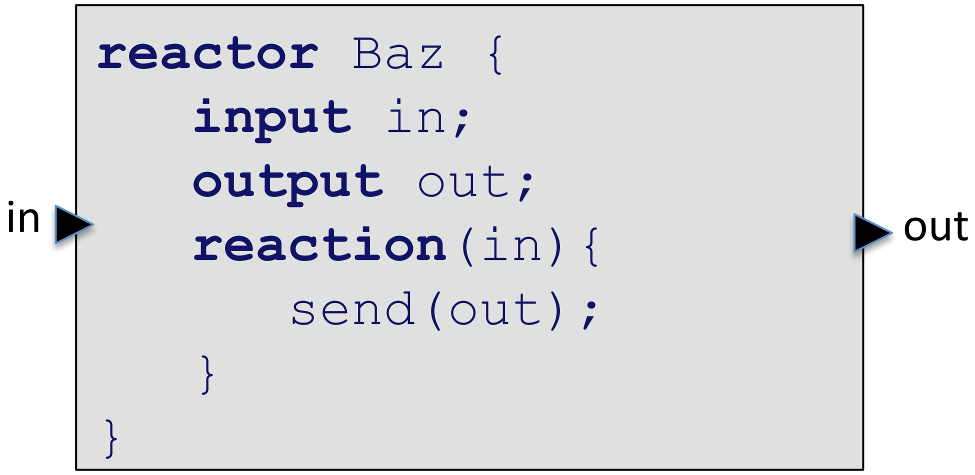
```
Actor Foo {
    int state = 1;
    handler double(){
        state *= 2;
    }
    handler increment(arg){
        state += arg;
        print state;
    }
}
```

Part 1 of our Solution: Ports

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



```
reactor Bar {
    output double, increment;
    reaction main(){
        double.send();
        increment.send(1);
    }
}
```

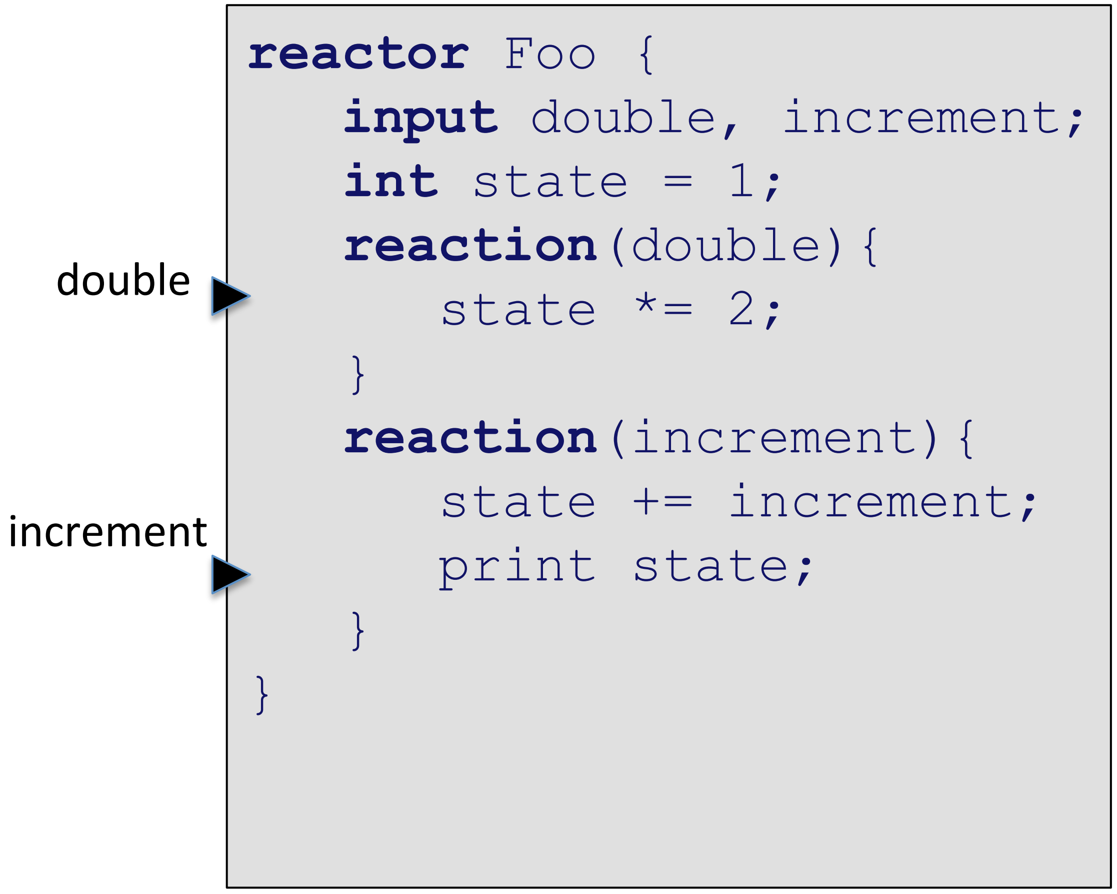


```
reactor Baz {
    input in;
    output out;
    reaction(in){
        send(out);
    }
}
```

[Ptolemeus, 2014]

Part 1 of our Solution: Ports

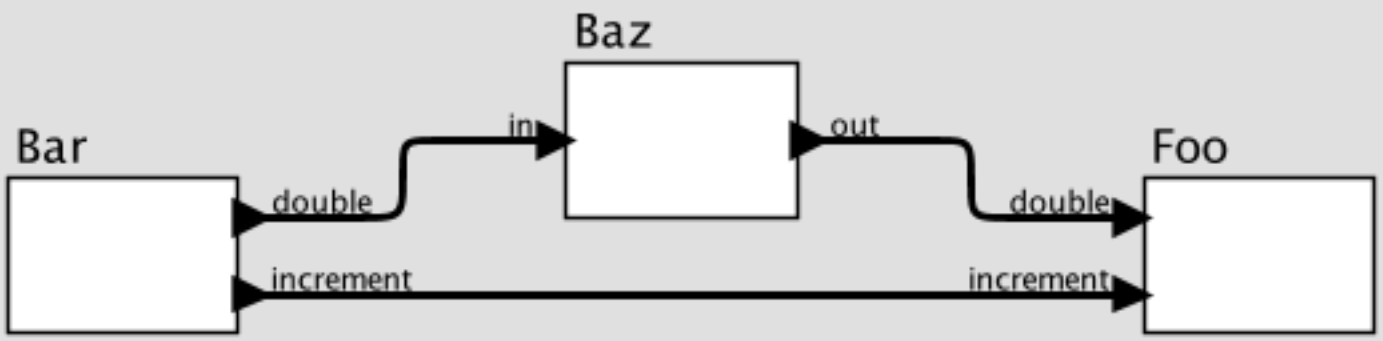
Input ports do not look much different from ordinary message handlers.



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

Part 2 of our Solution: Hierarchy

```
composite Top {
   reaction main(){
      Foo x = new Foo();
      Bar y = new Bar();
      Baz z = new Baz();
      connect(y.double, z.in);
      connect(y.increment, x.increment);
      connect(z.out, x.double);
   }
}
```

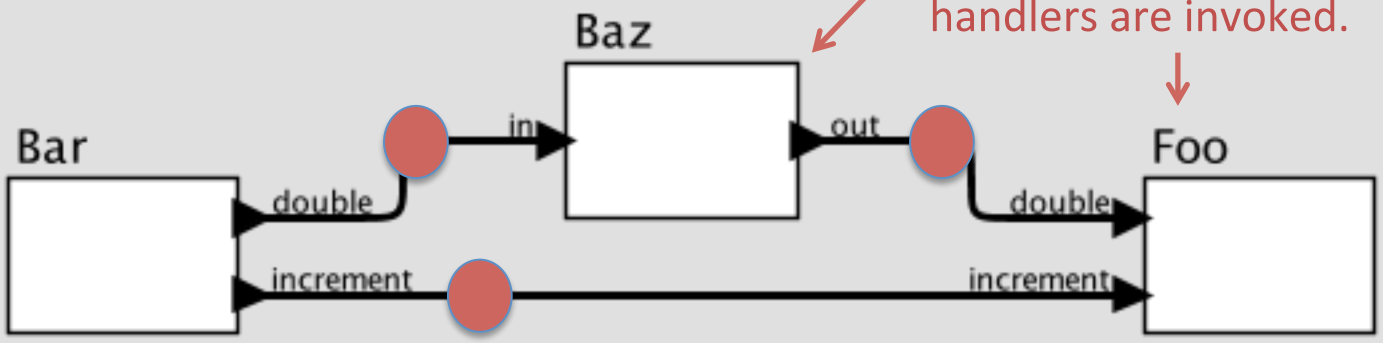


Part 3 of our Solution: Scheduling

```
composite Top {
   reaction main(){
      Foo x = new Foo();
      Bar y = new Bar();
      Baz z = new Baz();
      connect(y.double, z.in);
      connect(y.increment, x.increment);
      connect(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.



Some Strategies

- 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]
- …

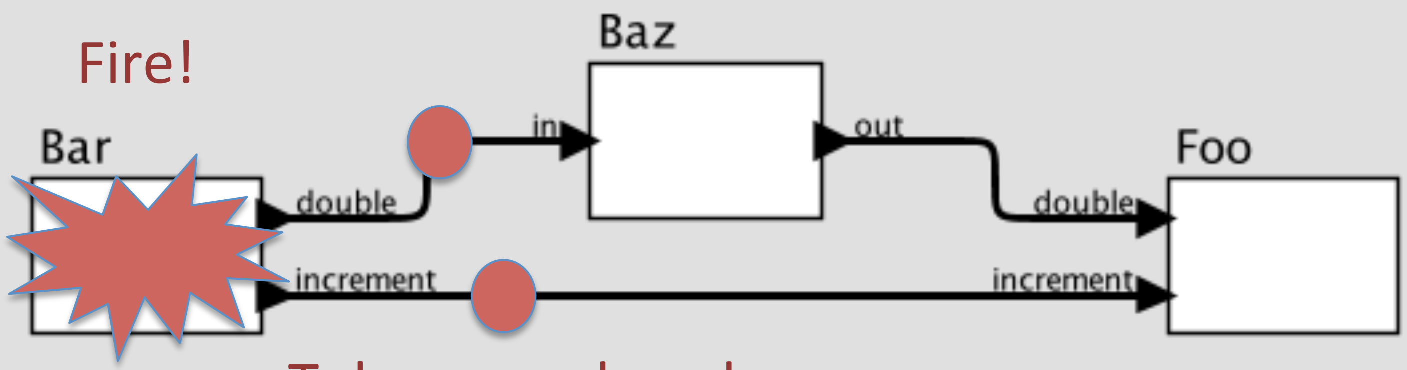
Jack Dennis

Dataflow Solution for Scheduling: Firing Rules

[Lee & Matsikoudis, 2009]

An actor with no inputs can fire at any time.

Fire!

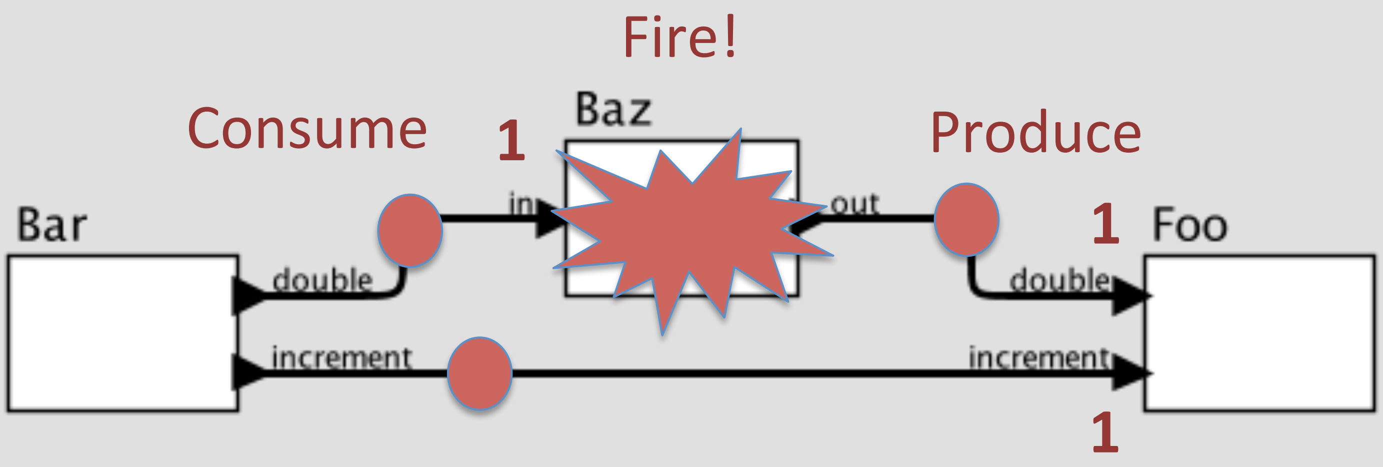


Tokens produced

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.

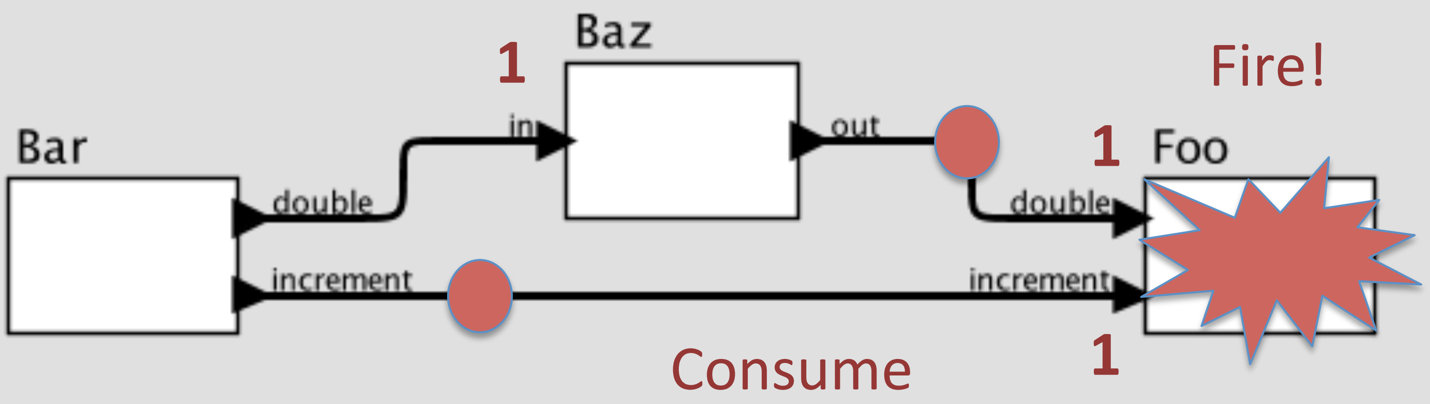


Dataflow Solution for Scheduling: Firing Rules

[Lee & Matsikoudis, 2009]

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

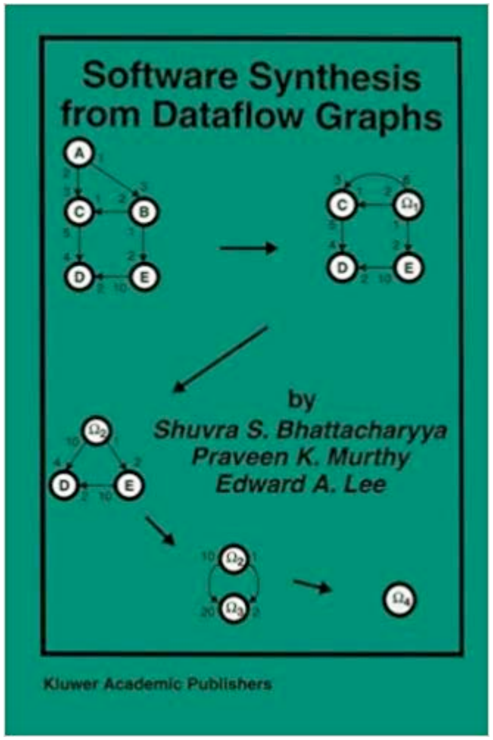
When it fires, each reaction is invoked in a deterministic order.



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]

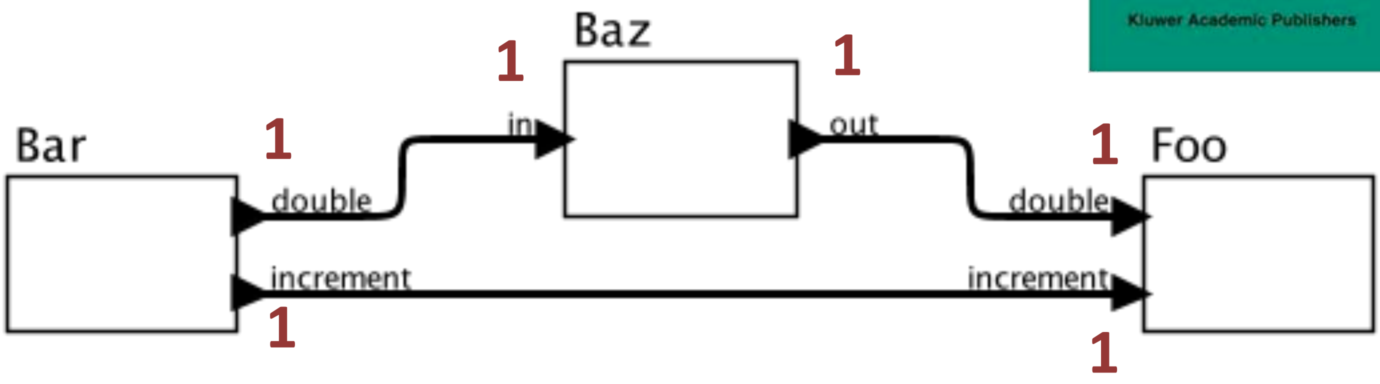


Software Synthesis from Dataflow Graphs

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

Kluwer Academic Publishers

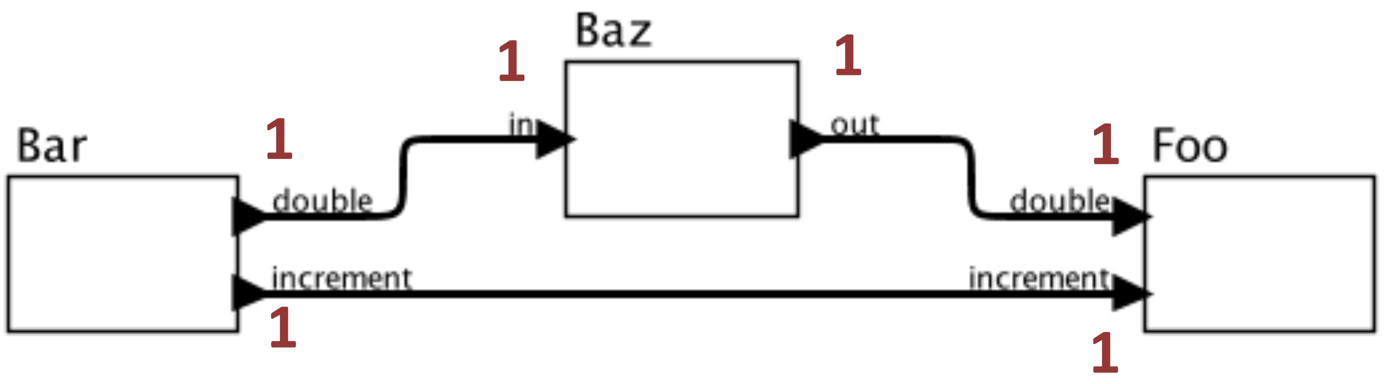
1996



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).

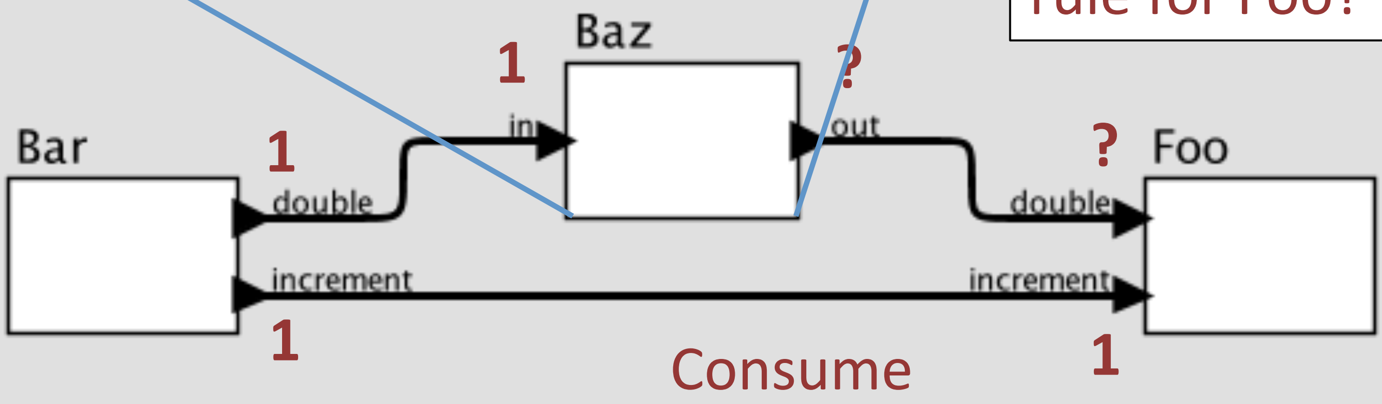
[Lee & Messerschmitt, 1986]



Dataflow Scheduling with Dynamic Firing Rules

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

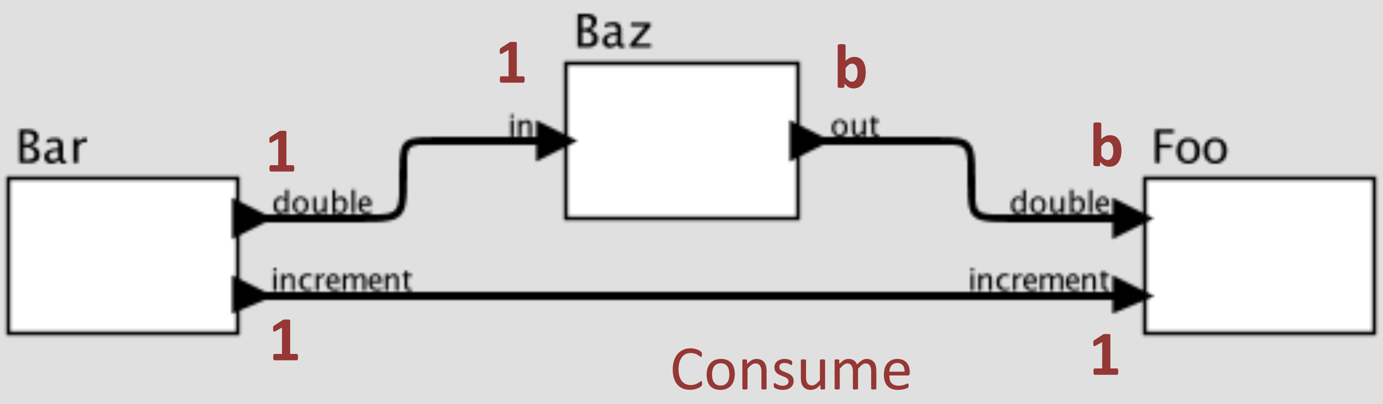
What should be the firing rule for Foo?



Boolean Dataflow

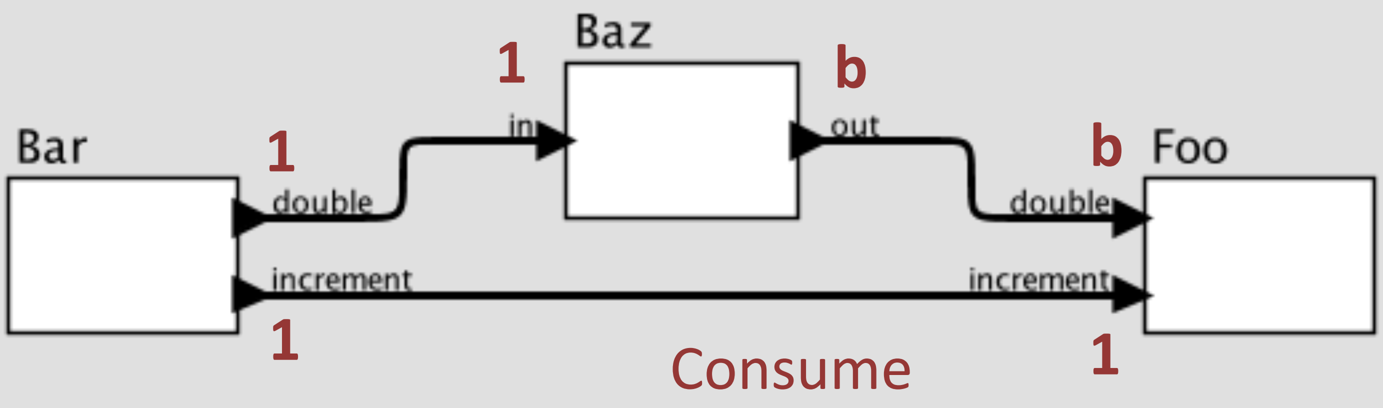
Buck [1993] showed that scheduling problems in general are undecidable in this framework.

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



Various Dataflow Variants that Remain Decidable

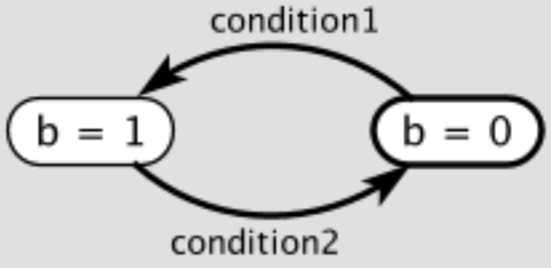
- Cyclostatic dataflow [Lauwereins 1994]
- Heterochronous dataflow [Girault, Lee & Lee, 1997]
- Parameterized dataflow [Bhattacharya & Bhattacharyya, 2001]
- Structured dataflow [Thies, 2002]
- Scenario-aware dataflow [Theelen, Geilen, Basten, et al. 2006]
- Reconfigurable dataflow [Fradet, Girault,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

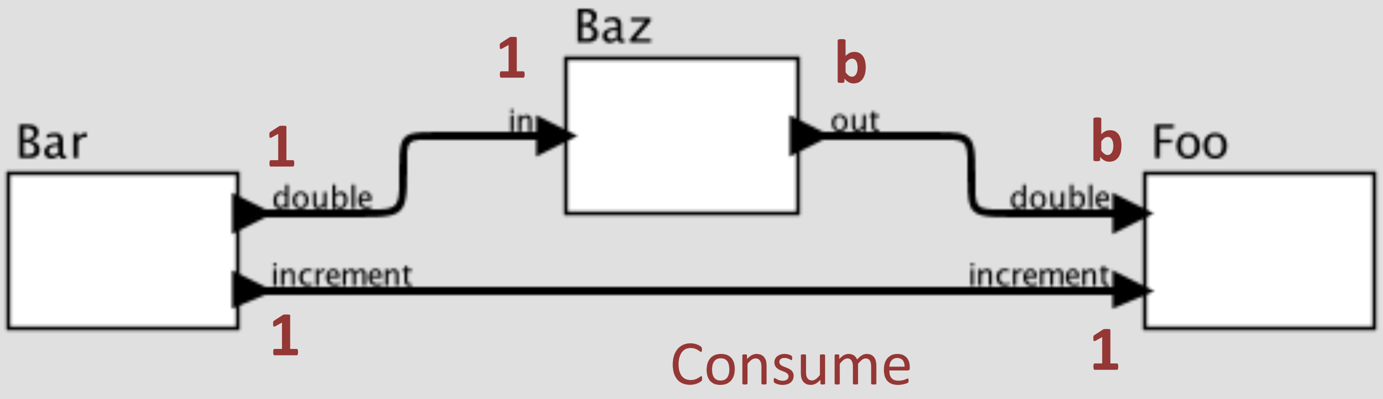


Scenario-Aware Dataflow

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

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



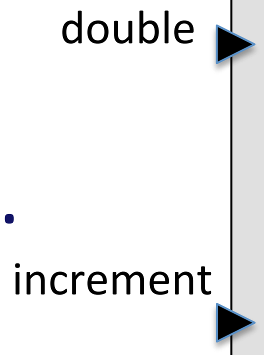


Some Strategies

- 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.



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

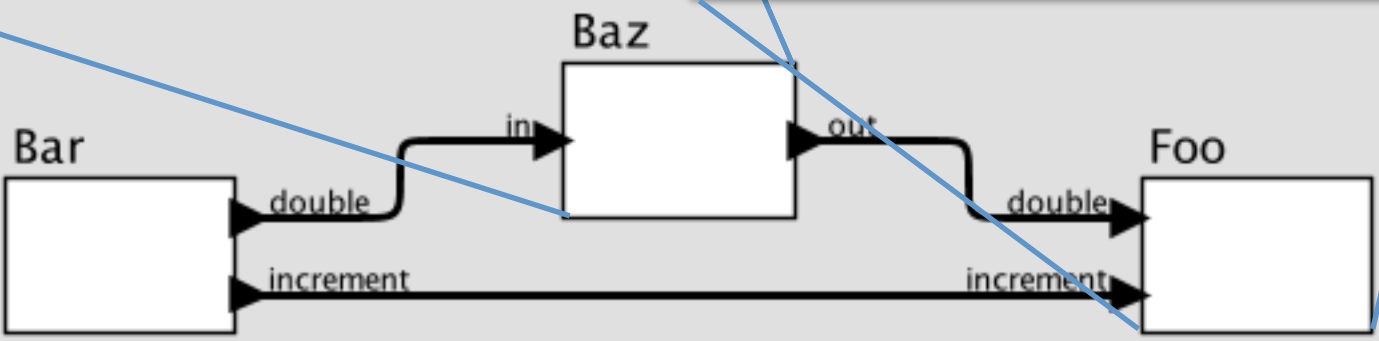
Gilles Kahn

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

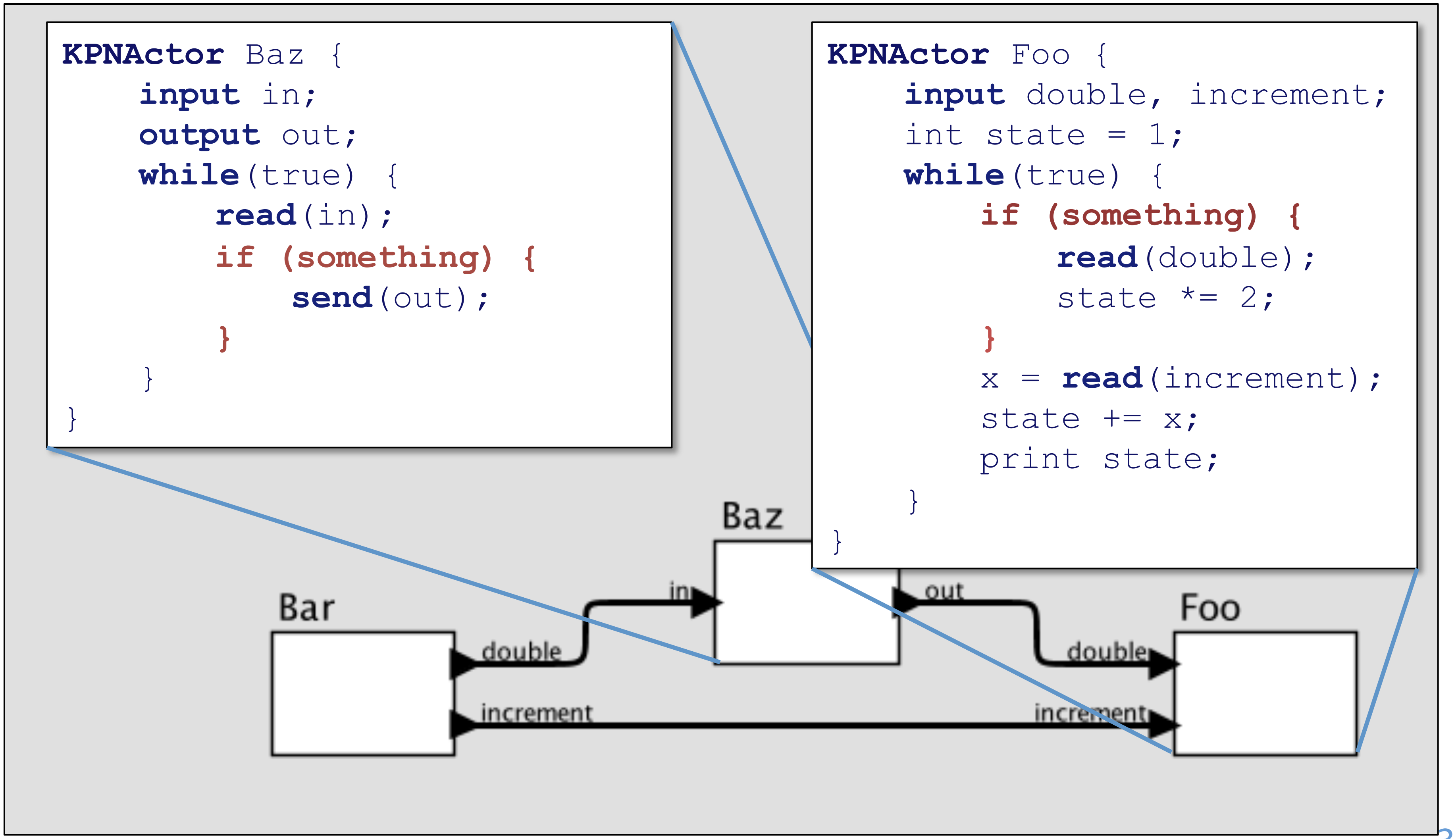
Blocking reads have trouble with data-dependent flow patterns

```
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



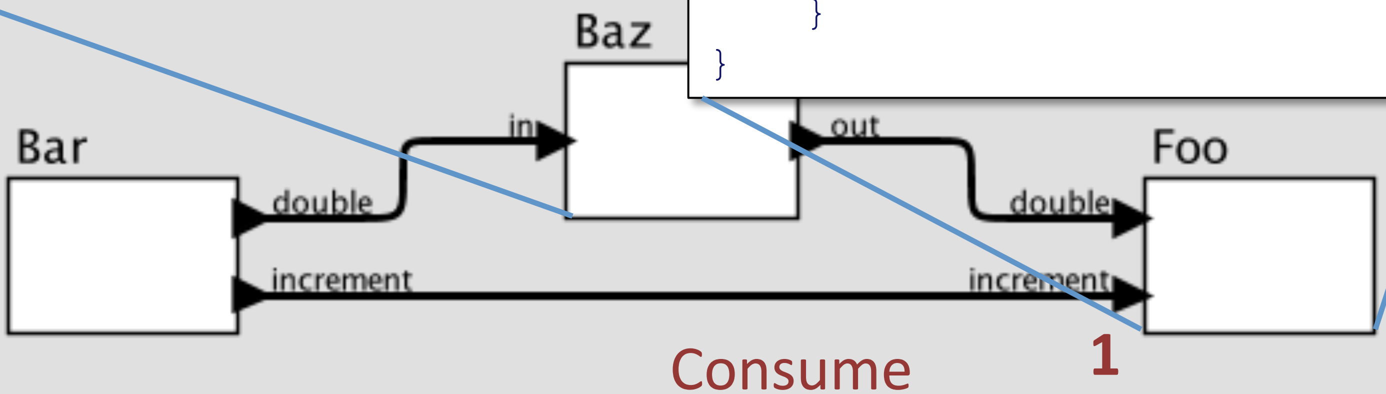
```
KPNActor Baz {
    input in;
    output out;
    while(true) {
        read(in);
        if (something) {
            send(out);
        }
    }
}
```

```
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

Some Strategies

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

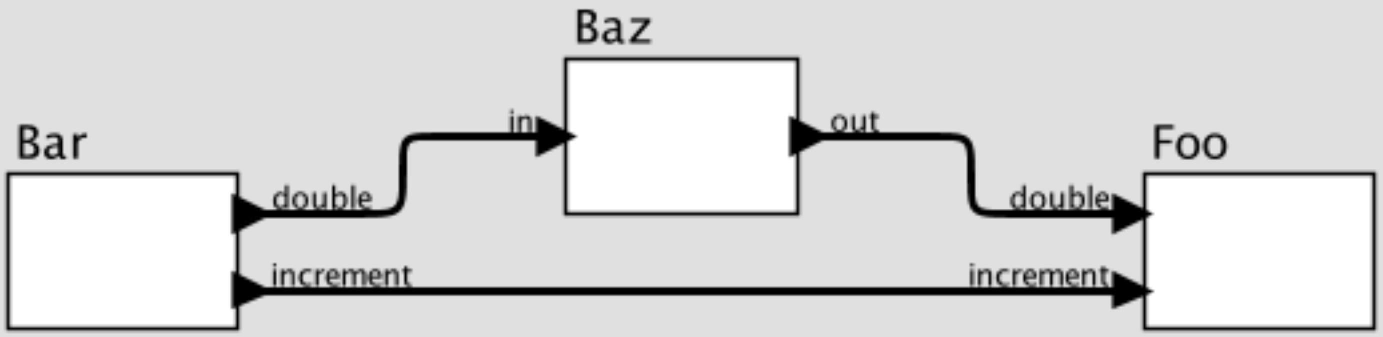
An Alternative Approach to Coordination

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

A Different Approach: Synchronous Languages

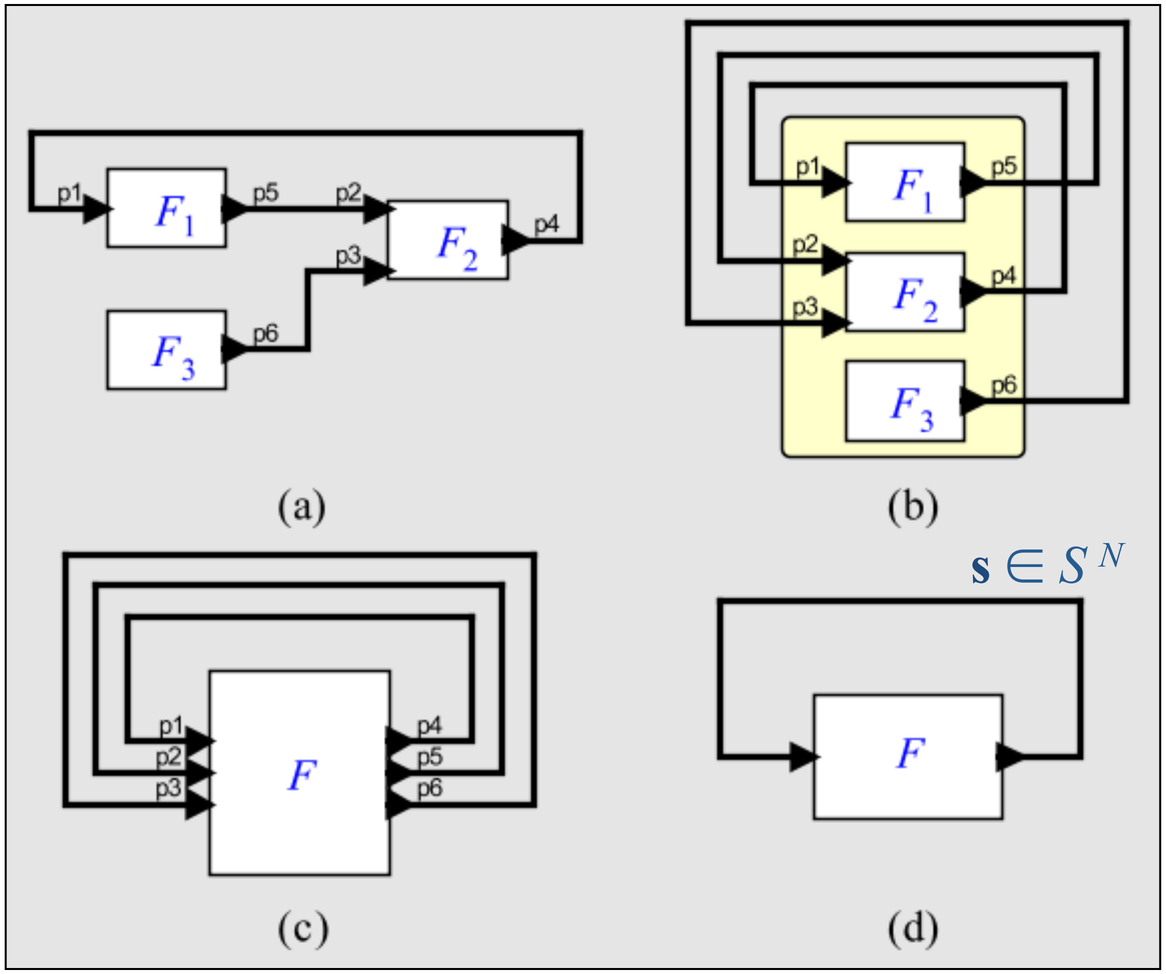
In the synchronous/reactive approach, there is a conceptual global “clock,” and on each “tick” of this clock, a connection either has a well-defined value or is “absent.”

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



[Benveniste & Berry, 1991]

Fixed Point Semantics



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.

[Edwards and Lee, 2003]

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)

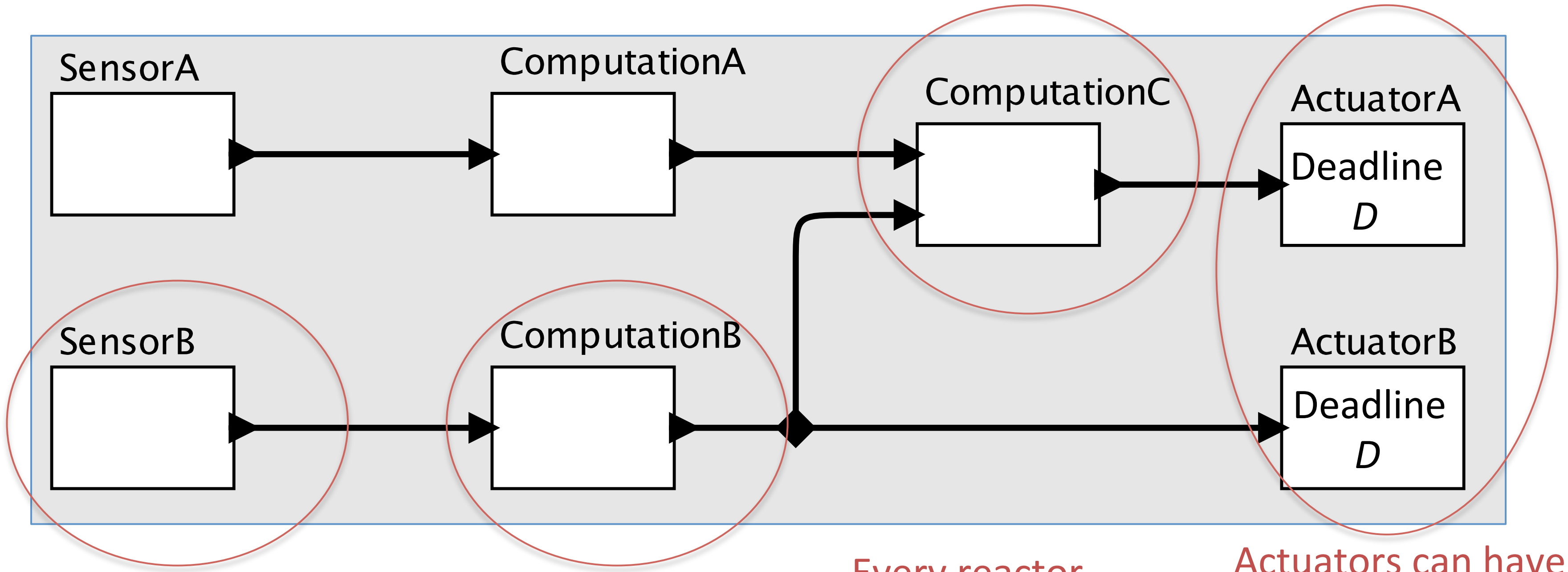
Discrete-Event Languages

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]

Finally! We can talk about the motivating example.



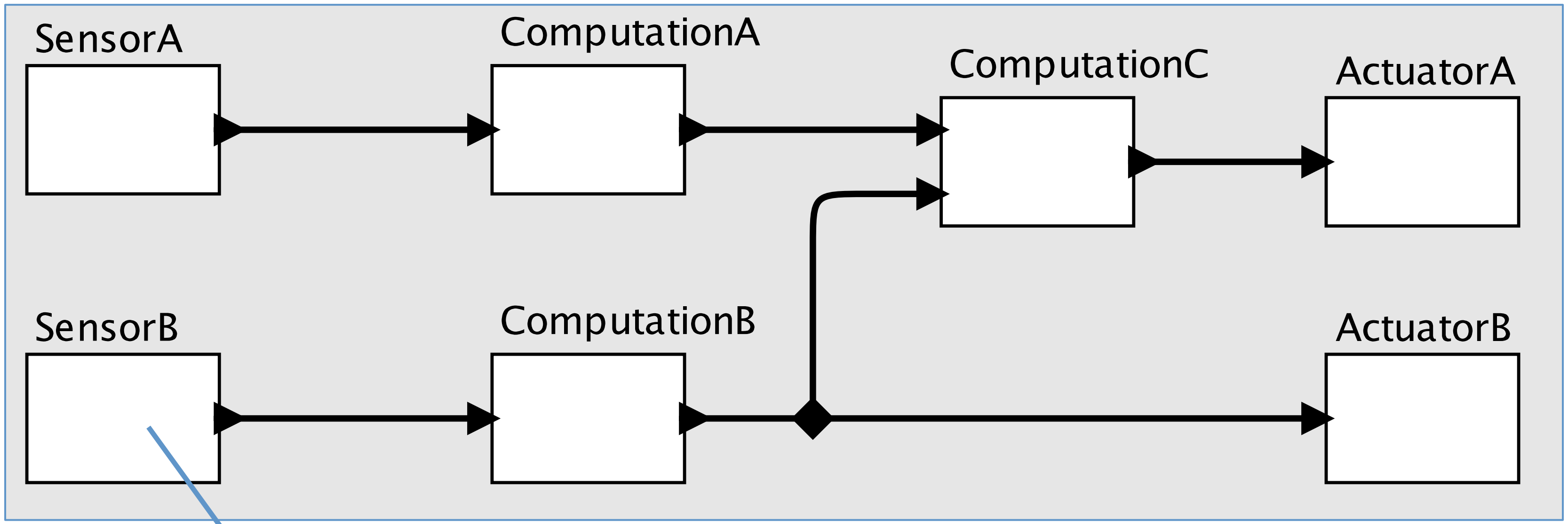
Sporadic events are assigned a time stamp based on the local physical-time clock

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*.

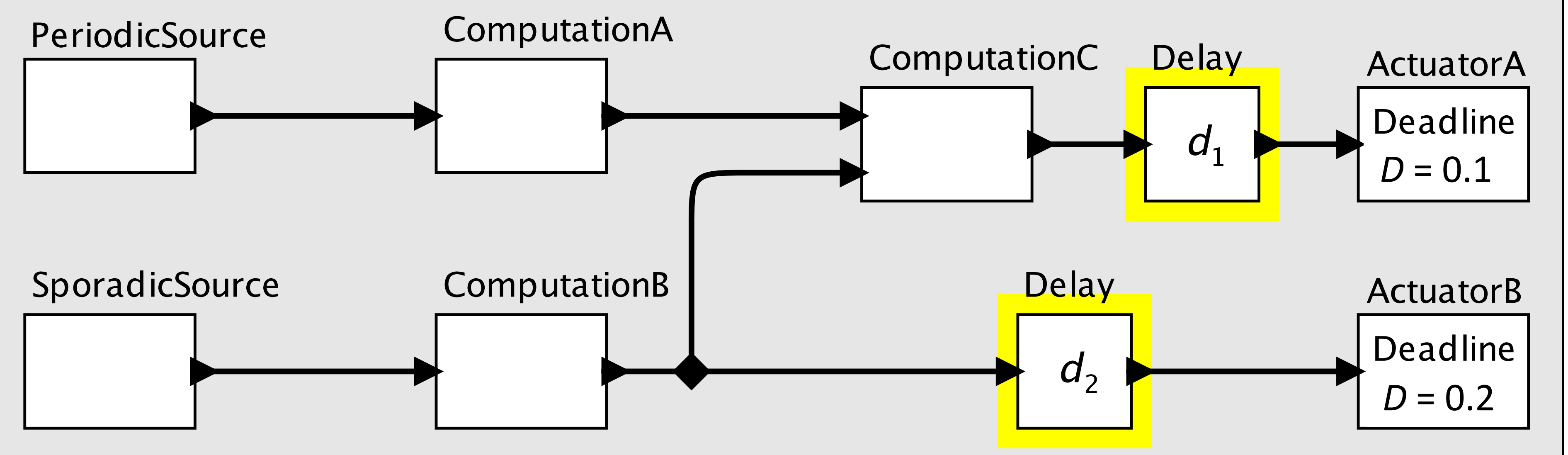
Simple, Single-Machine Realization



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.

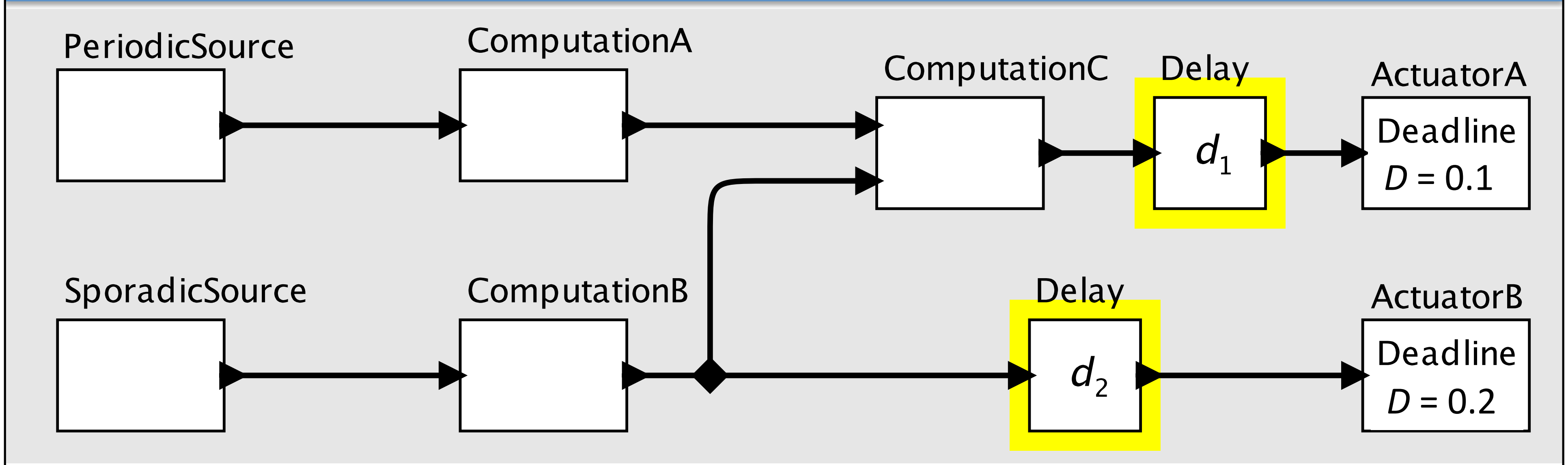
- Sort reactors topologically based on precedences.
- Global notion of "current time" *t.*
- Event queue containing future events.
- Choose earliest time stamp *t'* on the queue.
- Wait for the real-time clock to match *t'.*
- Execute reactors in topological sort order.

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.

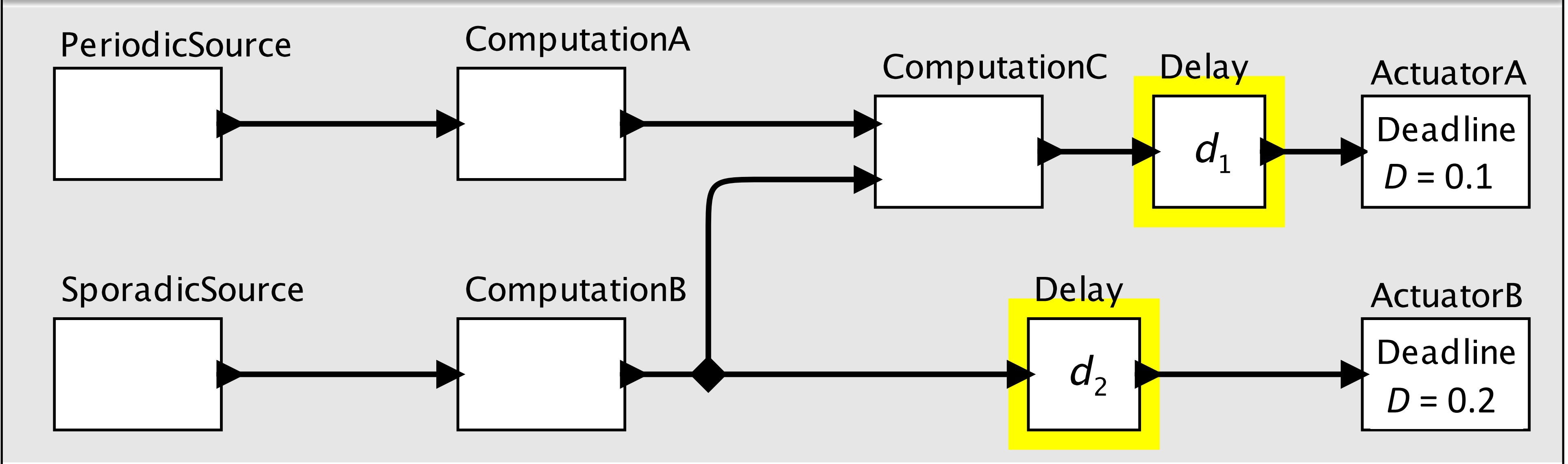
Real-Time Systems



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

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

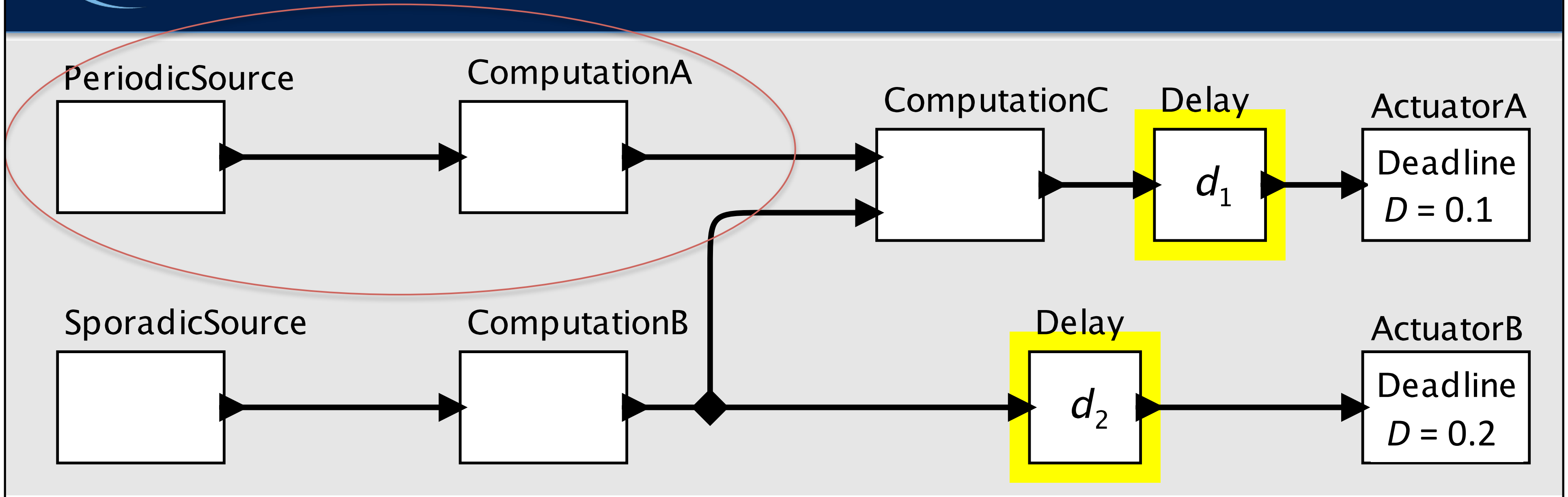
Iron-Clad Guarantees with PRET Machines



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.

Models of Time: 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: $\mathbf{v}(t_i, 1) = \mathbf{K}$

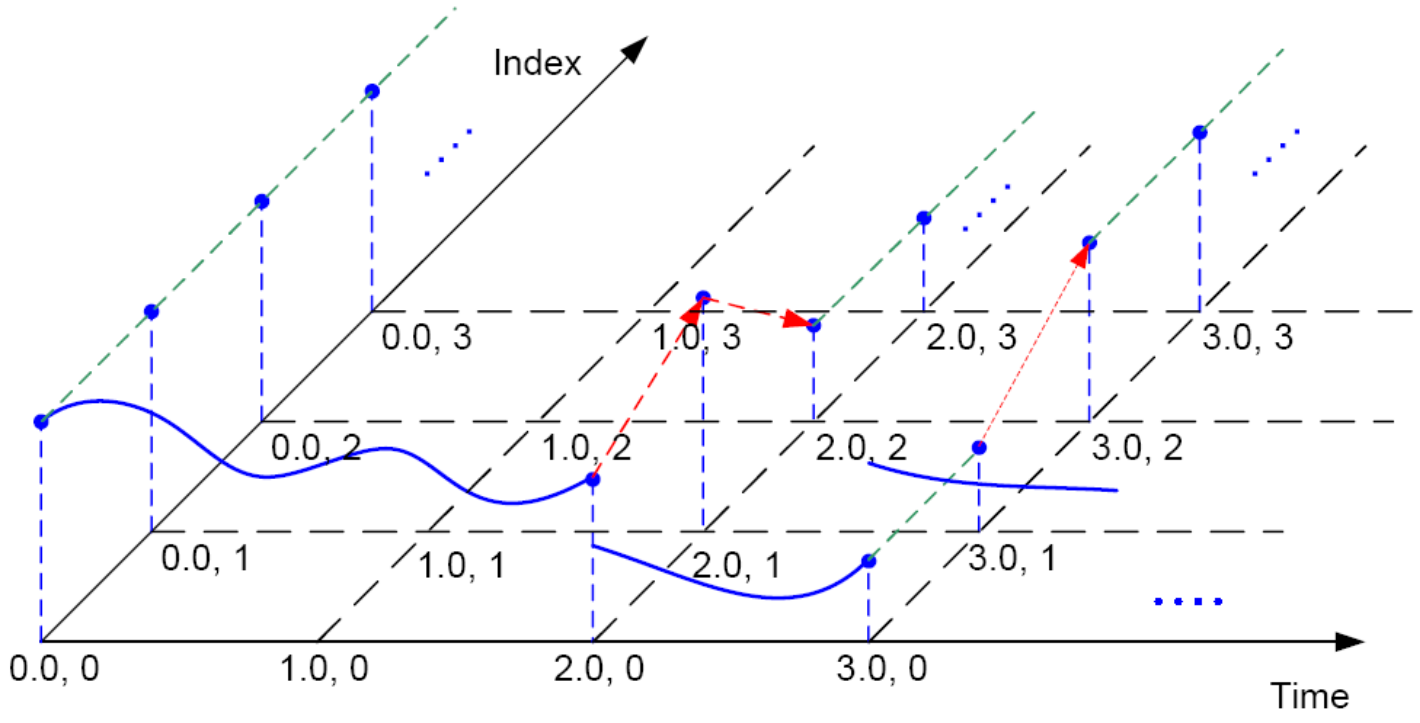
Final value: $\mathbf{v}(t_i, n) = 0, \quad n \geq 2$

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

[Lee & Zheng, 2005]

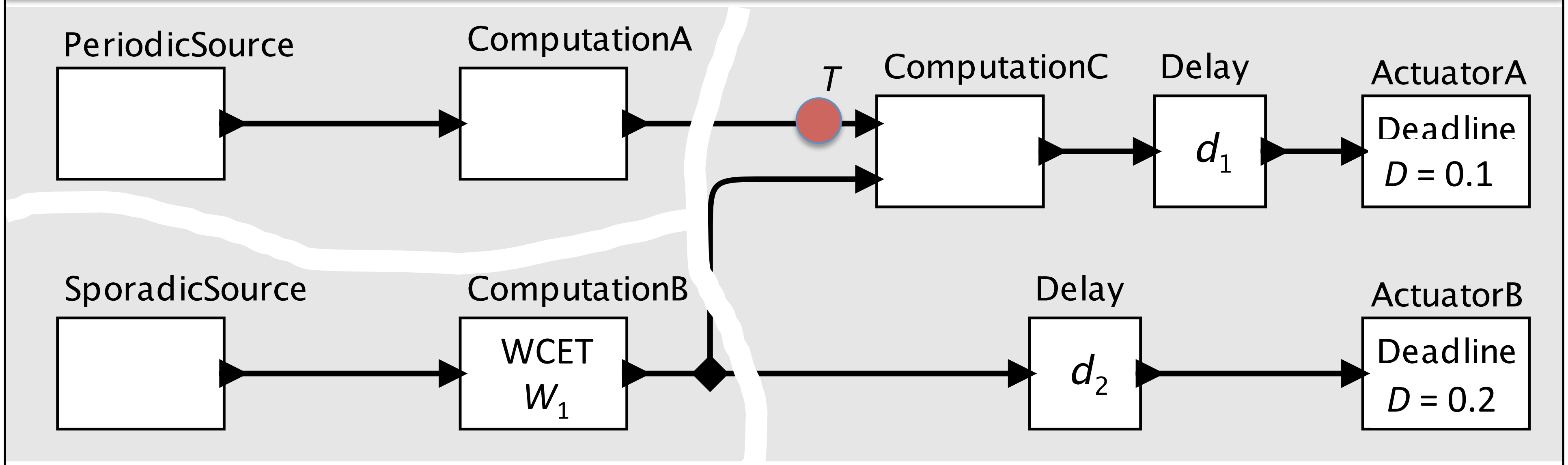
[Lee, "CPS Foundations," DAC, 2010]
[Maler, Manna, Pnuelli, 92]

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.

Networked Scheduling: PTides



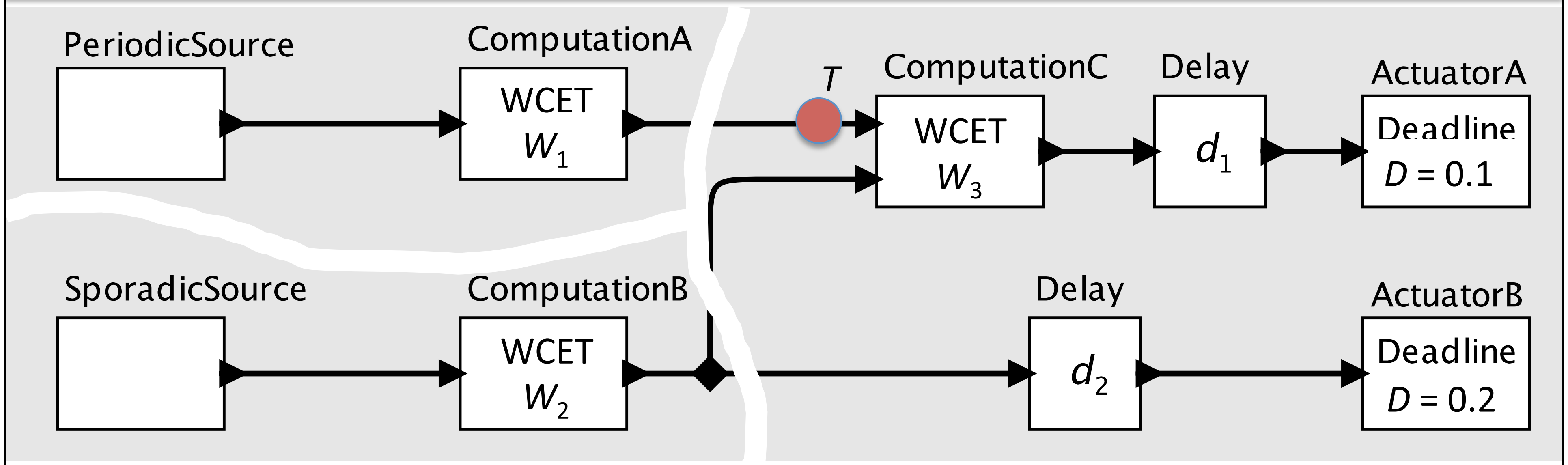
When is this “safe to process”?

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

[Zhao et al., 2007]
[Edison et al., 2012]
[Corbett et al., 2012]

Networked Scheduling: PTides

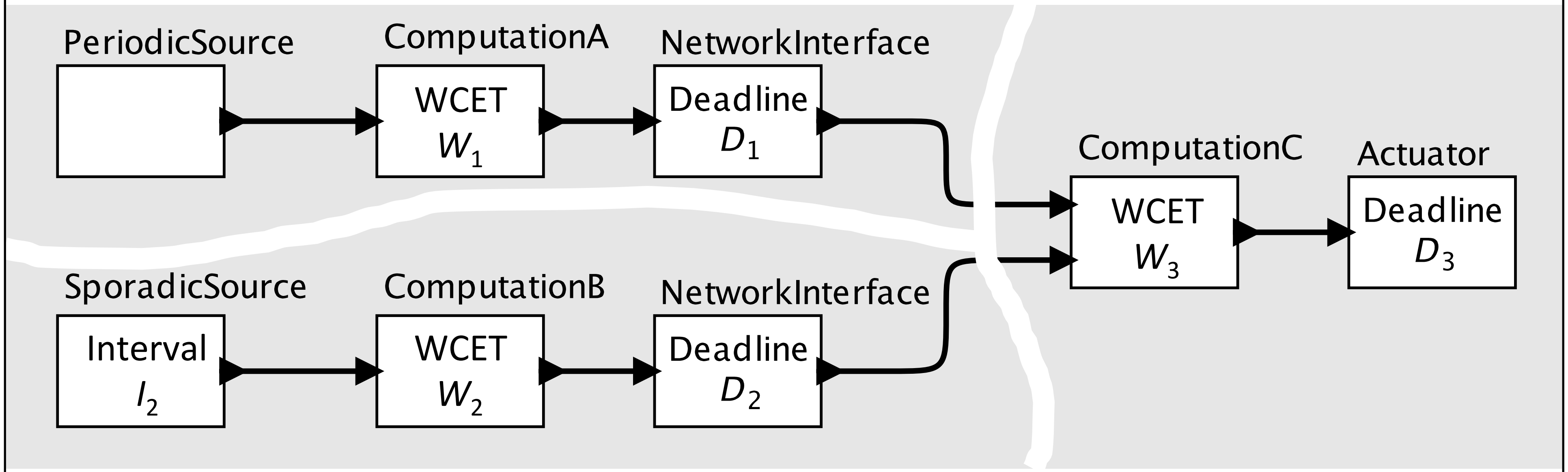


Will the deadline at ActuatorA be met?

Yes if $D + d_1 \geq \max(W_1, W_2) + E + N + W_3$

[Zhao et al., 2007]
[Edison et al., 2012]
[Corbett et al., 2012]

Decoupling Real-Time Analysis with Networked Scheduling

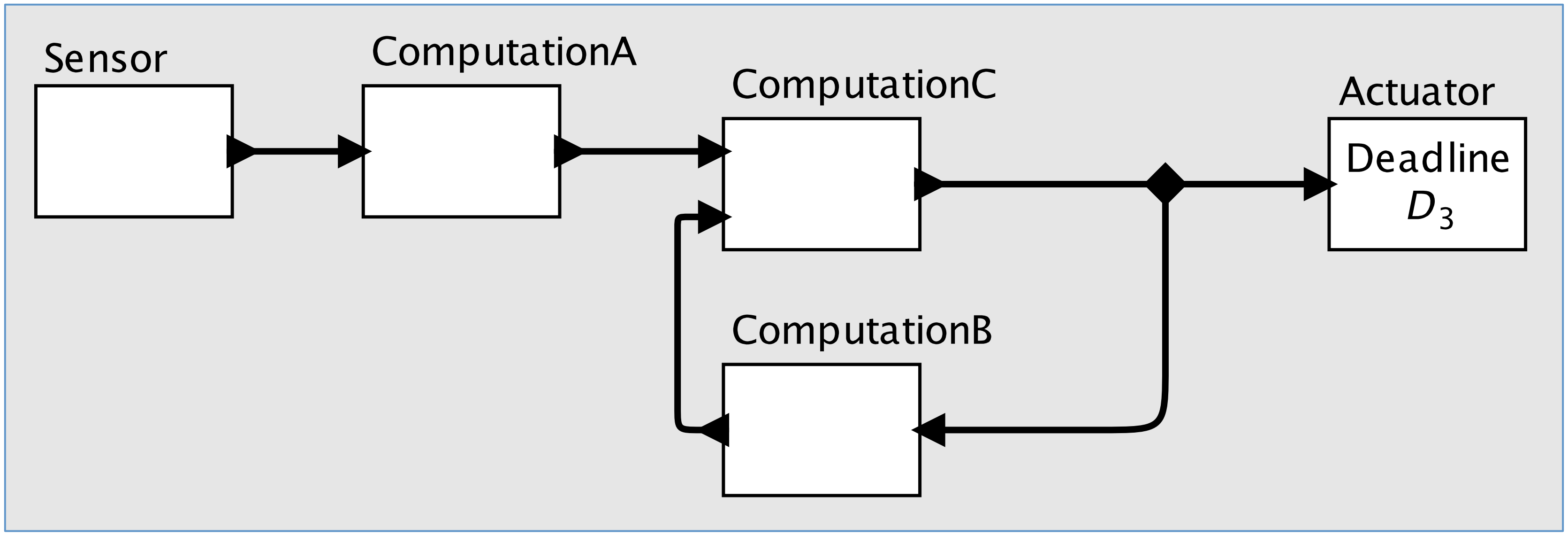


Imposing deadlines on network interfaces decouples the real-time analysis problem. Each execution platform can be individually verified for meeting deadlines.

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
- …

Conclusion

- Hewitt/Agha actors are nondeterministic
- Some solutions:
  - Dataflow
  - Process networks
  - Synchronous/Reactive models
  - Discrete-Event
- Reactors are actors revisited with DE semantics

Pseudo code shown is based on 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.