



# Disciplined Concurrent Models of Computation for Parallel Software

**Edward A. Lee**

*Robert S. Pepper Distinguished Professor and  
UC Berkeley*

*Invited Keynote Talk*

*2008 Summer Institute*

*The Concurrency Challenge: Can We Make Parallel Programming Popular?*

*Sponsors: University of Washington and Microsoft Research*

*Blaine, WA*

*August 3 to August 7, 2008*



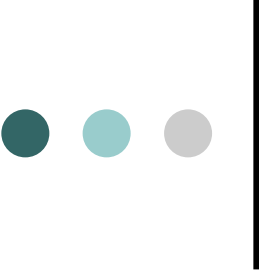


## A Tired Debate...

- Shared memory
  - Threads, semaphores, mutexes, monitors...
- Message Passing
  - Synchronous, asynchronous, buffered, ...

This debate is a red herring!

*The choice of shared memory vs. message passing should be left up to the architects and compilers, and not up to the programmers.*



Shared memory is not an acceptable  
programmer's model

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



## Consider a Simple Example

“The *Observer pattern* defines a one-to-many dependency between a subject object and any number of observer objects so that when the subject object changes state, all its observer objects are notified and updated automatically.”

*Design Patterns*, Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides (Addison-Wesley Publishing Co., 1995. ISBN: 0201633612):



# Observer Pattern in Java

```
public void addListener(Listener) {  
    myListeners.add(Listener);  
}
```

```
public void setValue(newValue) {  
    myValue = newValue;
```

```
    for (Listener : myListeners) {  
        Listener.valueChanged(newValue)  
    }  
}
```

Will this work in a  
multithreaded context?

Thanks to Mark S. Miller for the details  
of this example.



## Observer Pattern With Mutual Exclusion

```
public synchronized void addListener(listener) {  
    myListeners.add(listener);  
}
```

```
public synchronized void setValue(newValue) {  
    myValue = newValue;
```

```
    for (listener : myListeners) {  
        listener.valueChanged(newValue)  
    }  
}
```

JavaSoft recommends against this.  
What's wrong with it?



# Mutexes are Minefields

```
public synchronized void addListener(listener) {  
    myListeners.add(listener);  
}
```

```
public synchronized void setValue(newValue) {  
    myValue = newValue;
```

```
    for (listener : myListeners) {  
        listener.valueChanged(newValue)  
    }  
}
```

valueChanged() may attempt to acquire a lock on some other object and stall. If the holder of that lock calls addListener(), deadlock!





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

After years of use without problems, a Ptolemy Project code review found code that was not thread safe. It was fixed in this way. Three days later, a user in Germany reported a deadlock that had not shown up in the test suite.





Should have used tools...

Tsk, tsk...

Should have used static analysis tools to detect the deadlock potential...

But detection is only half the battle...

How to fix it?



# Simple Observer Pattern Becomes Not So Simple

```
public synchronized void addListener(listener) {  
    myListeners.add(listener);  
}
```

```
public synchronized void setValue(newValue) {  
    synchronized(this) {  
        myValue = newValue;  
        listeners = myListeners.clone();  
    }  
    for (listener : listeners) {  
        listener.valueChanged(newValue)  
    }  
}
```

*while holding lock, make  
copy of listeners to avoid  
race conditions*

*notify each listener  
outside of synchronized  
block to avoid deadlock*

This still isn't right.  
What's wrong with it?



## Simple Observer Pattern: How to Make It Right?

```
public synchronized void addListener(listener) {  
    myListeners.add(listener);  
}
```

```
public synchronized void setValue(newValue) {  
    synchronized(this) {  
        myValue = newValue;  
        listeners = myListeners.clone();  
    }  
    for (listener : listeners) {  
        listener.valueChanged(newValue)  
    }  
}
```

*Suppose two threads call setValue(). One of them will set the value last, leaving that value in the object, but listeners may be notified in the opposite order. The listeners may be alerted to the value changes in the wrong order!*



## 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



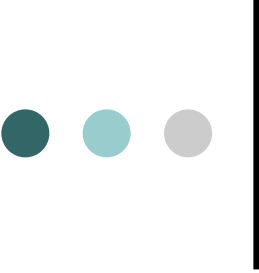
*It is not  
concurrency that  
is hard...*



...It is shared memory that is 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...*



Concurrent programs using shared memory are incomprehensible because concurrency in the physical world does not work that way.

*We have no experience!*





# 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, transactions) and limiting shared data accesses (e.g., OO design).



# We have incrementally improved threads

- Object Oriented programming
- Coding rules (Acquire locks in the same order...)
- Libraries (Stapl, Java 5.0, ...)
- PGAS (UPC, Co-array FORTRAN, Titanium, ...)
- Patterns (MapReduce, ...)
- Transactions (Databases, ...)
- Formal verification (Blast, thread checkers, ...)
- Enhanced languages (Split-C, Cilk, Guava, ...)
- Enhanced mechanisms (Promises, futures, ...)

But is it enough to refine a mechanism  
with flawed foundations?



E.g.: Verification:  
Thread checkers?

Consider what it would take for static analysis tools to detect the out-of-order notification bug in our implementation of the listener pattern...

We want to tolerate a race on calls to `setValue()` (application nondeterminism), but not on notifications to listeners. To specify this for a verifier, we have to have already solved the problem!



For a brief optimistic instant, *transactions* looked like they might save us...

“TM is not as easy as it looks (even to explain)”

Michael L. Scott, invited keynote, (EC)<sup>2</sup>  
Workshop, Princeton, NJ, July 2008

# Do we have a sound foundation for concurrent programming?

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 enterprise is held up by threads***





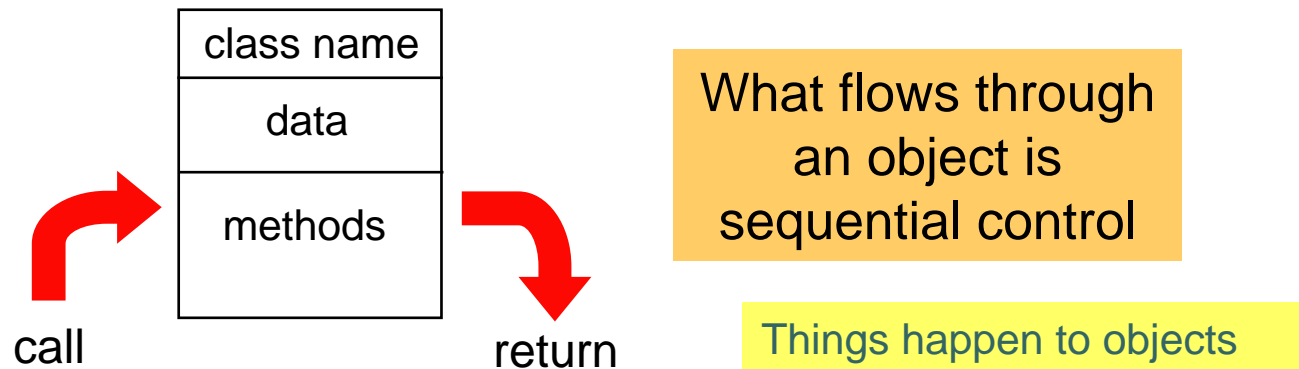
So, the answer must be message passing,  
right?

Not quite...

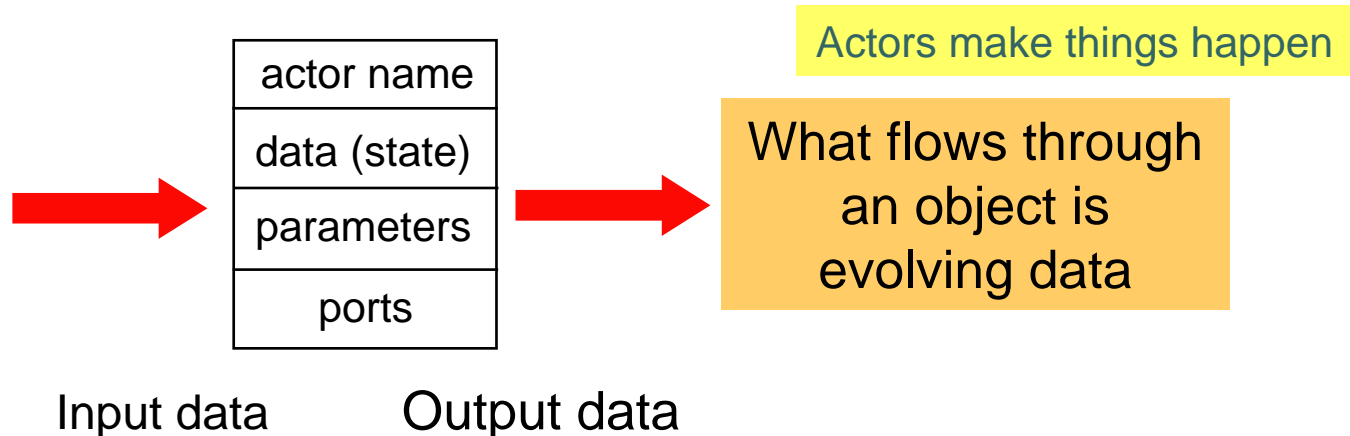
More discipline is needed than what is provided  
by today's message passing libraries.

# One approach: Rethinking software components for concurrency

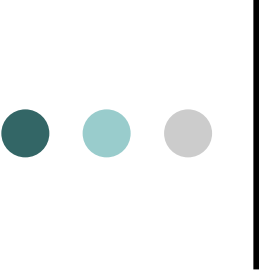
The established: Object-oriented:



The alternative: Actor oriented:







# Examples of Actor-Oriented Systems

- Unix pipes
- Dataflow systems
- CORBA event service (distributed push-pull)
- ROOM and UML-2 (dataflow, Rational, IBM)
- VHDL, Verilog (discrete events, Cadence, Synopsys, ...)
- LabVIEW (structured dataflow, National Instruments)
- Modelica (continuous-time, constraint-based, Linköping)
- OPNET (discrete events, Opnet Technologies)
- SDL (process networks)
- Occam (rendezvous)
- Simulink (Continuous-time, The MathWorks)
- SPW (synchronous dataflow, Cadence, CoWare)
- ...

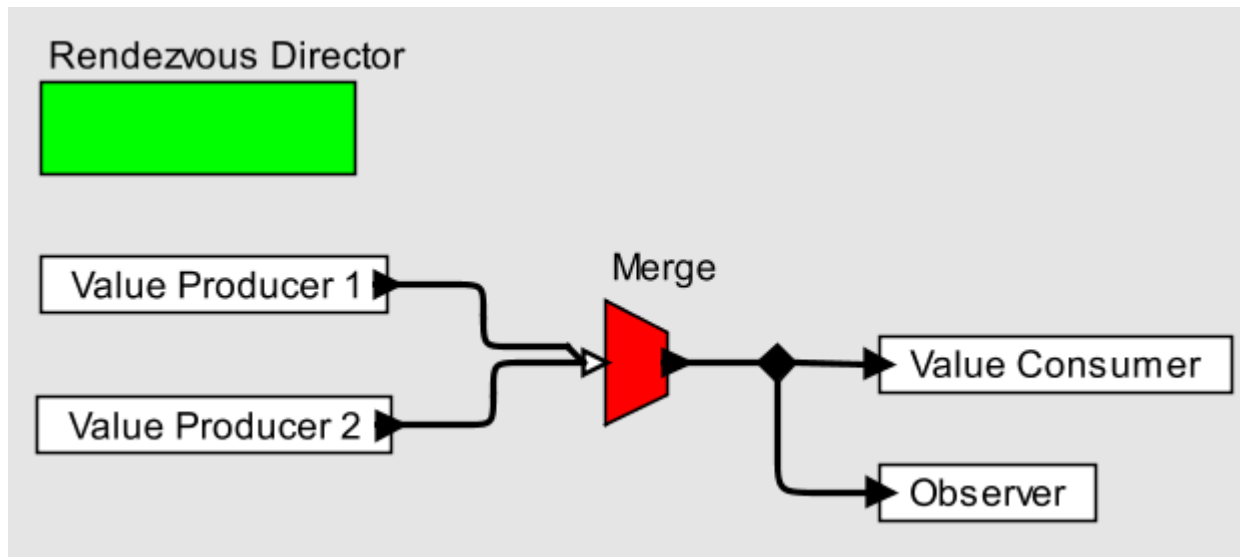
*The semantics of these differ considerably, but all provide more understandable ways of expressing concurrency.*



## Recall the Observer Pattern

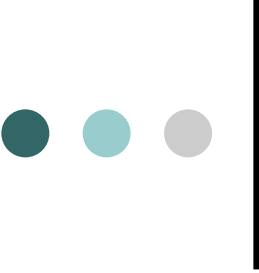
“The *Observer pattern* defines a one-to-many dependency between a subject object and any number of observer objects so that when the subject object changes state, all its observer objects are notified and updated automatically.”

# Observer Pattern using an Actor-Oriented Language with Rendezvous Semantics



Each actor is a process, communication is via rendezvous, and the Merge explicitly represents nondeterministic multi-way rendezvous.

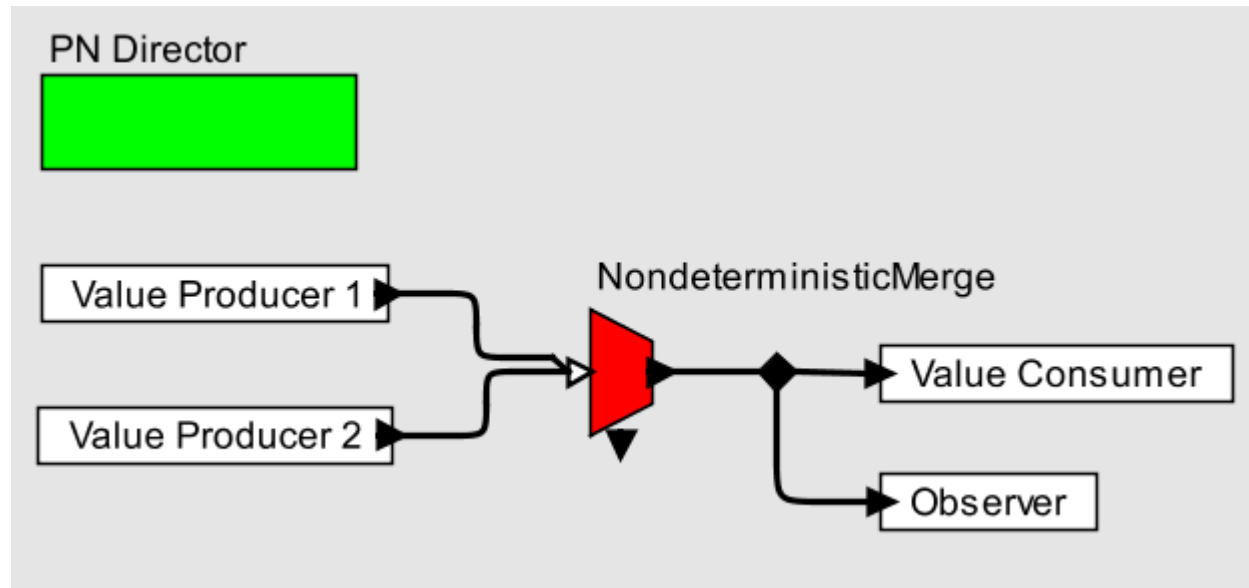
This is realized here in a *coordination language with a visual syntax*.



Now that we've made a trivial design pattern trivial, we can work on more interesting aspects of the design.

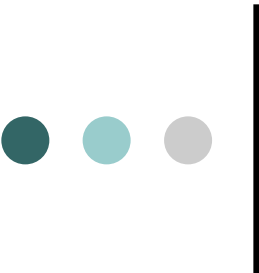
E.g., suppose we don't care how long notification of the observer is deferred, as long as the observer is notified of all changes in the right order?

# Observer Pattern using an Actor-Oriented Language with Kahn Semantics (Extended with Nondeterministic Merge)



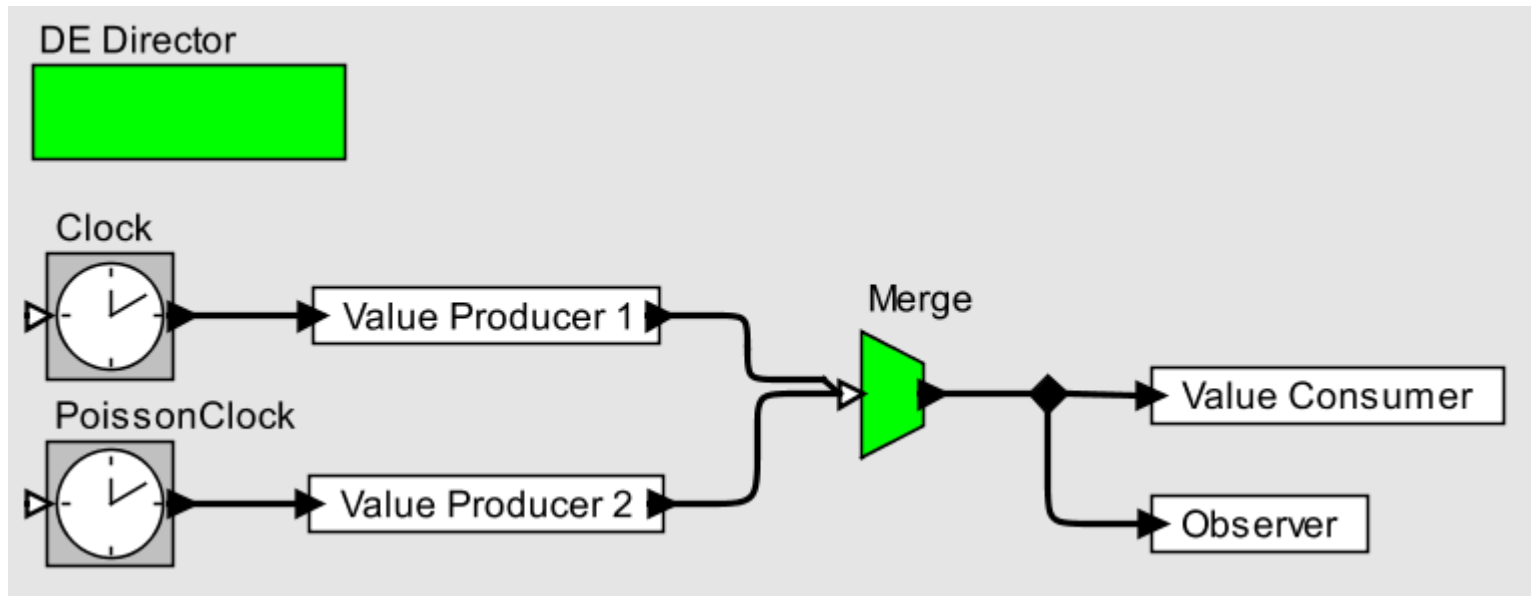
Each actor is a process, communication is via streams, and the **NondeterministicMerge** explicitly merges streams nondeterministically.

*Again a coordination language with a visual syntax.*



Suppose further that we want to explicitly specify the timing?

# Observer Pattern using an Actor-Oriented Language with Discrete Event Semantics



Messages have a (semantic) time, and actors react to messages chronologically. Merge now becomes deterministic.

*Again a coordination language with a visual syntax.*





Isn't this just message passing?

Each of these realizations of the listener pattern can be implemented with a message passing library (or with threads, for that matter).

*But a message passing library allows too much flexibility to yield comprehensible designs.*

Its capabilities need to be used judiciously...



## Consider for example Determinism

Most programs specify a particular computation, expecting the same input to yield the same output.

*Occasionally, programs require nondeterminism, where the input/output relation is not a function. Multiple outputs are possible for the same input.*

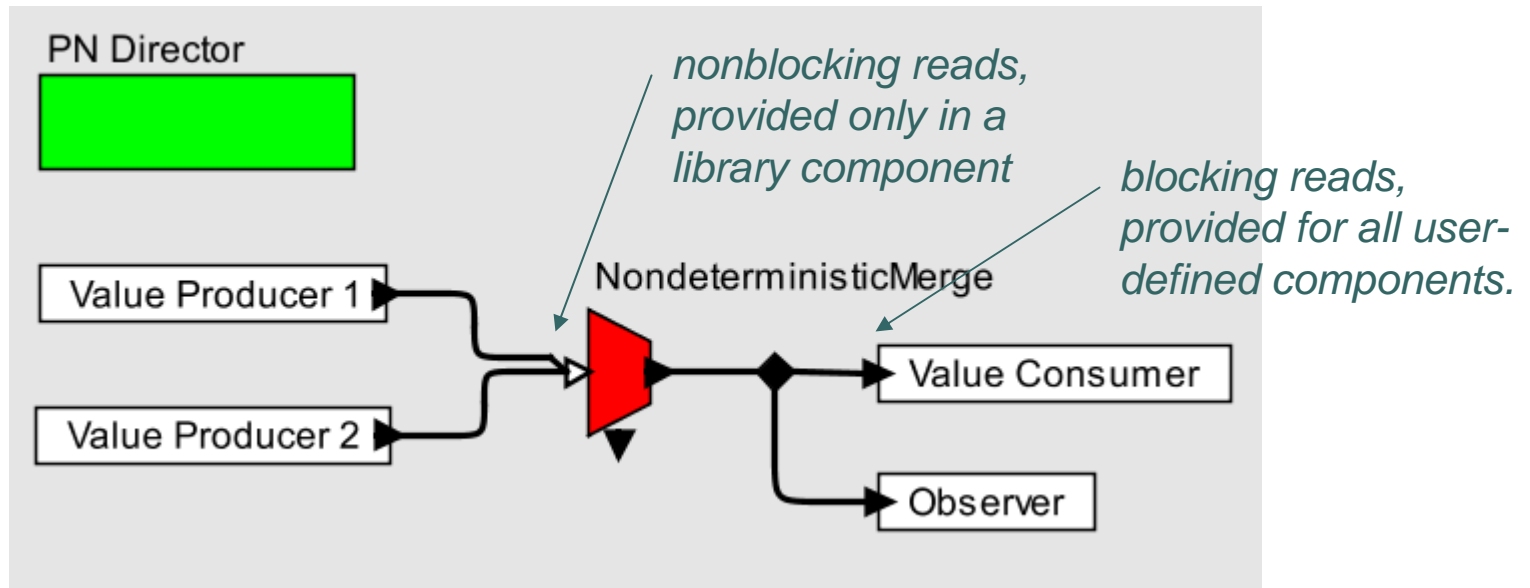
Regrettably, without considerable sophistication, message passing libraries often yield inadvertently nondeterminate programs.



# Consider Streams

- In 1974, Gilles Kahn showed that prefix-monotonic functions on streams composed deterministically.
- In 1977, Kahn and MacQueen showed that prefix monotonic functions could be implemented with blocking reads.
- Unix pipes use such blocking reads, and achieve determinate composition, but have limited expressiveness.
- Message passing libraries, however, are more flexible, and unless the programmer has studied Kahn, he is likely to mess up...

● ● ● | A *disciplined* use of streams follows Kahn-MacQueen semantics, except where *explicitly* requested by the programmer.



This becomes a *disciplined model of computation*, trivially easy for the programmer to understand, with excellent analogies in the physical world.



# A few disciplined concurrent models of computation

- Kahn process networks
- Dataflow
- Synchronous/reactive systems
- Rendezvous
- Discrete-events
- ...

*Each of these has many variants with sometimes subtle differences due to differing constraints imposed on the programmer.*



# A few variants of dataflow, for example

- 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]
- 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]
- ...



Isn't this just message passing again?

Dataflow models can be built with message passing libraries (and with threads). But should the programmer be asked to handle the considerable subtleties?

*Few programmers will get it right...*





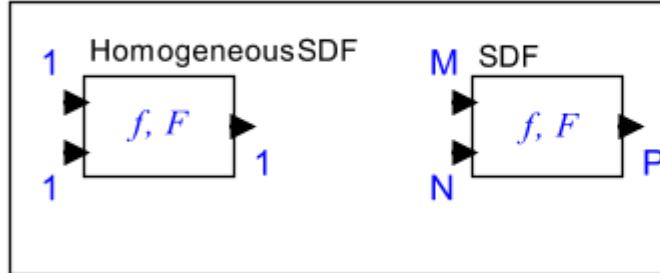
## Some Subtleties

- Termination, deadlock, and livelock (halting)
- Bounding the buffers.
- Fairness
- Parallelism
- Data structures and shared data
- Determinism
- Syntax

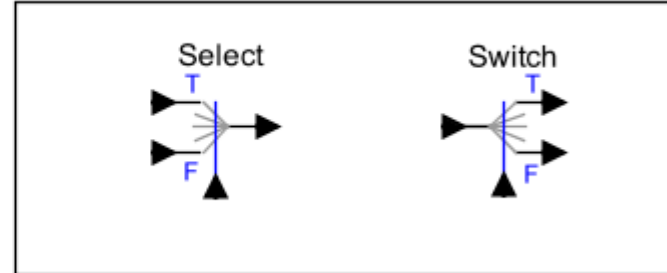
# Dennis-Style Dataflow

Firing rules:  
the number of  
tokens  
required to fire  
an actor.

Synchronous Dataflow



Dynamic Dataflow



Communication between actors is via potentially unbounded streams of *tokens*. Each actor has *firing rules*, which specify how availability of input tokens enables a computation. Constraints on the firing rules can yield very nice properties.

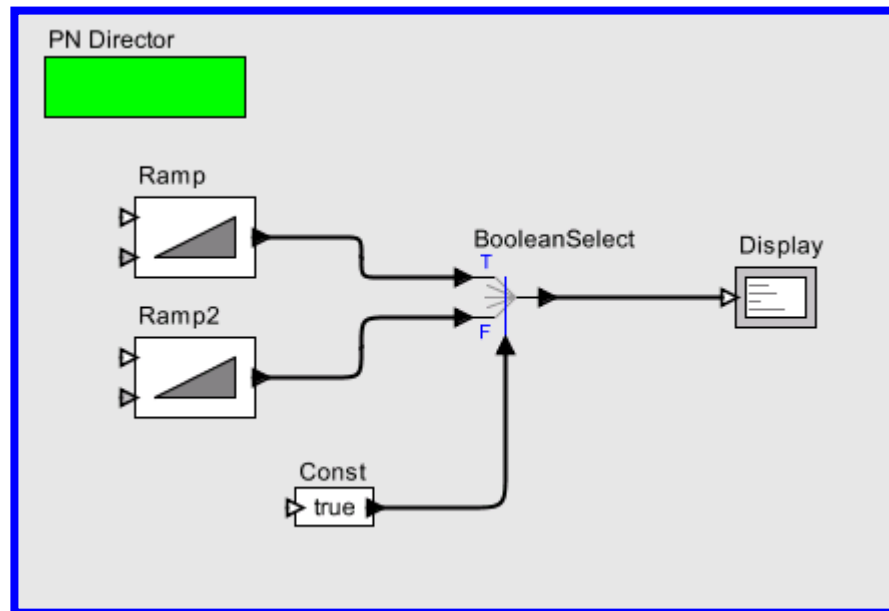
Without these nice properties, scheduling policy is a very subtle question.

A *signal* or *stream* is a (potentially infinite) sequence of communicated data tokens..

## Question 1:

### Is “Fair” Scheduling a Good Idea?

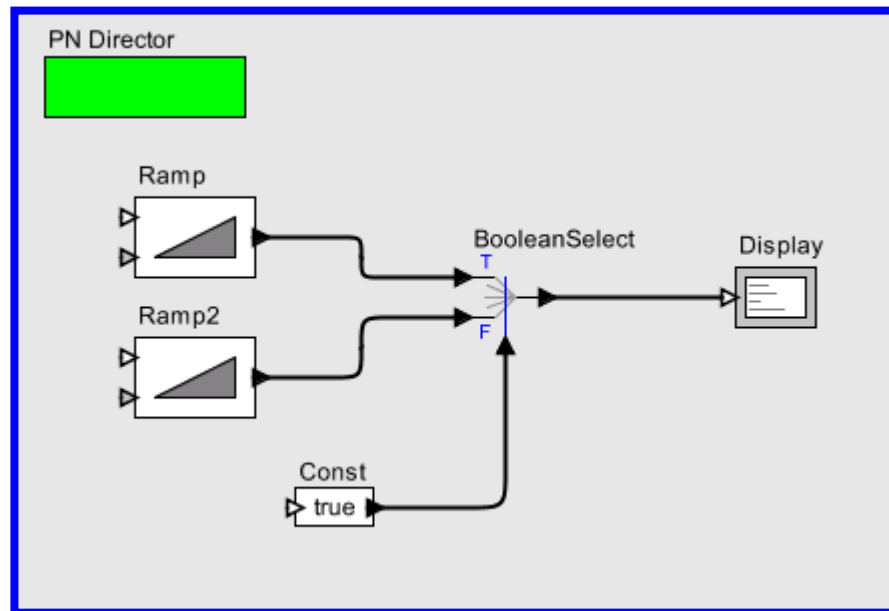
In the following model, what happens if every actor is given an equal opportunity to run?



## Question 2:

### Is “Data-Driven” Execution a Good Idea?

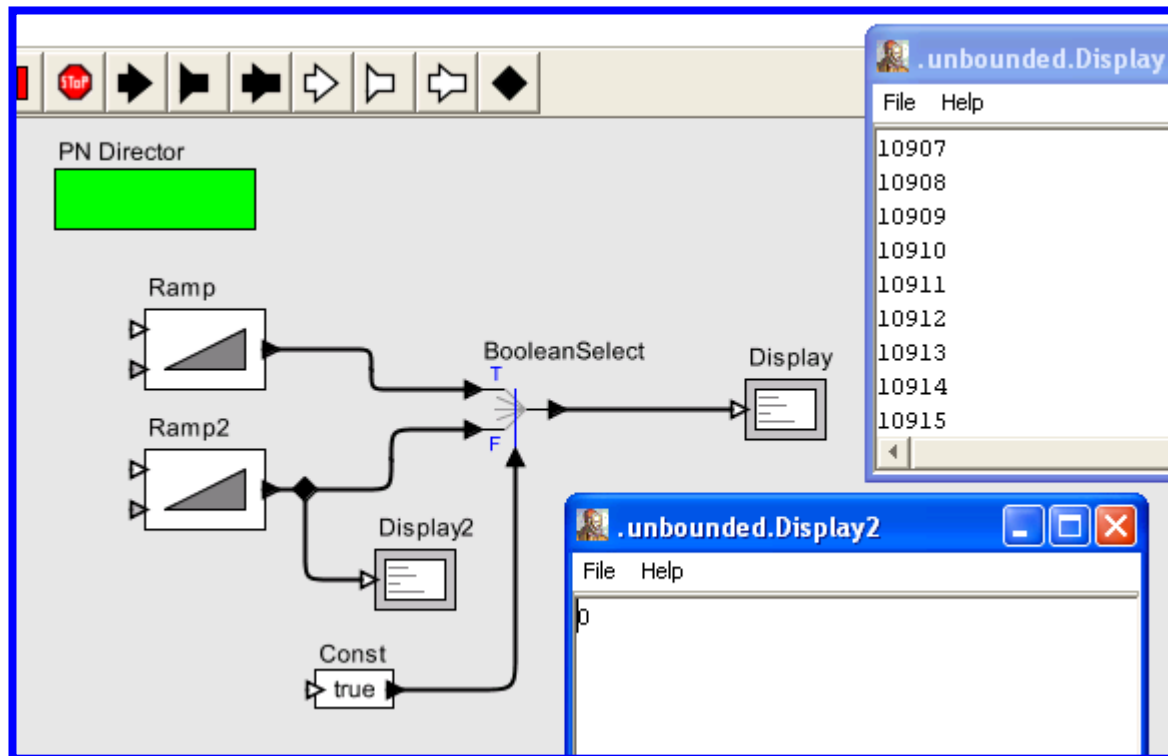
In the following model, if actors are allowed to run when they have input data on connected inputs, what will happen?



### Question 3:

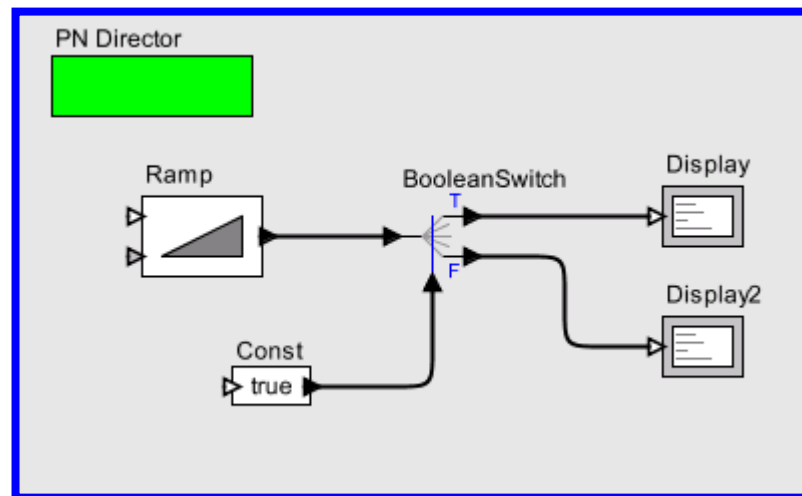
## When are Outputs Required?

Is the execution shown for the following model the “right” execution?

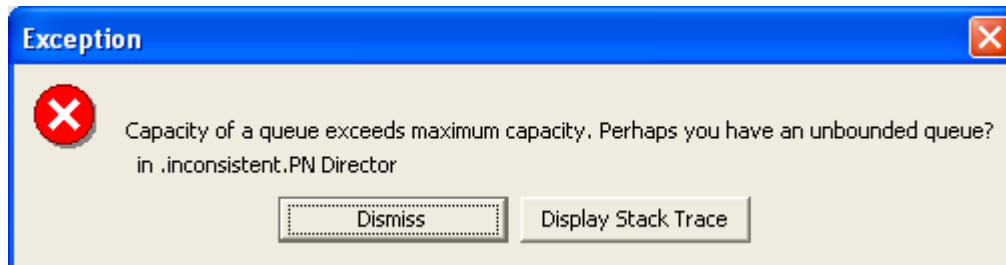
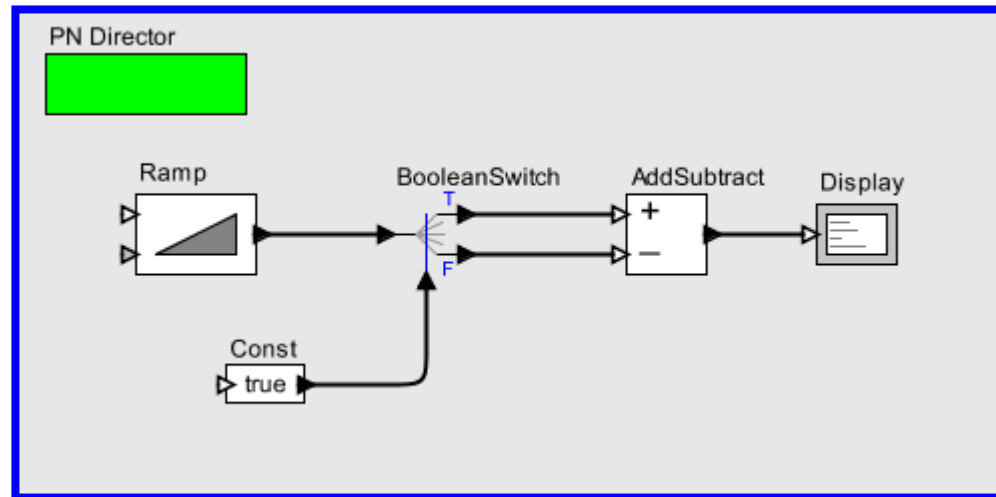


## Question 4: Is “Demand-Driven” Execution a Good Idea?

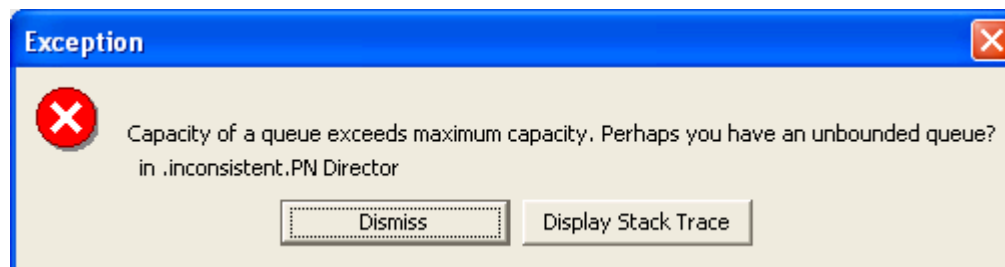
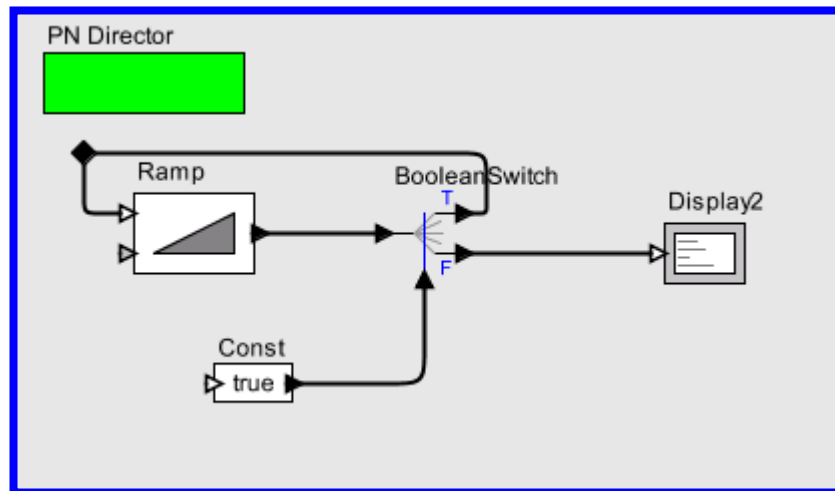
In the following model, if actors are allowed to run when another actor requires their outputs, what will happen?



## Question 5: What is the “Correct” Execution of This Program?



## Question 6: What is the Correct Behavior of this Program?







## Naïve Schedulers Fail

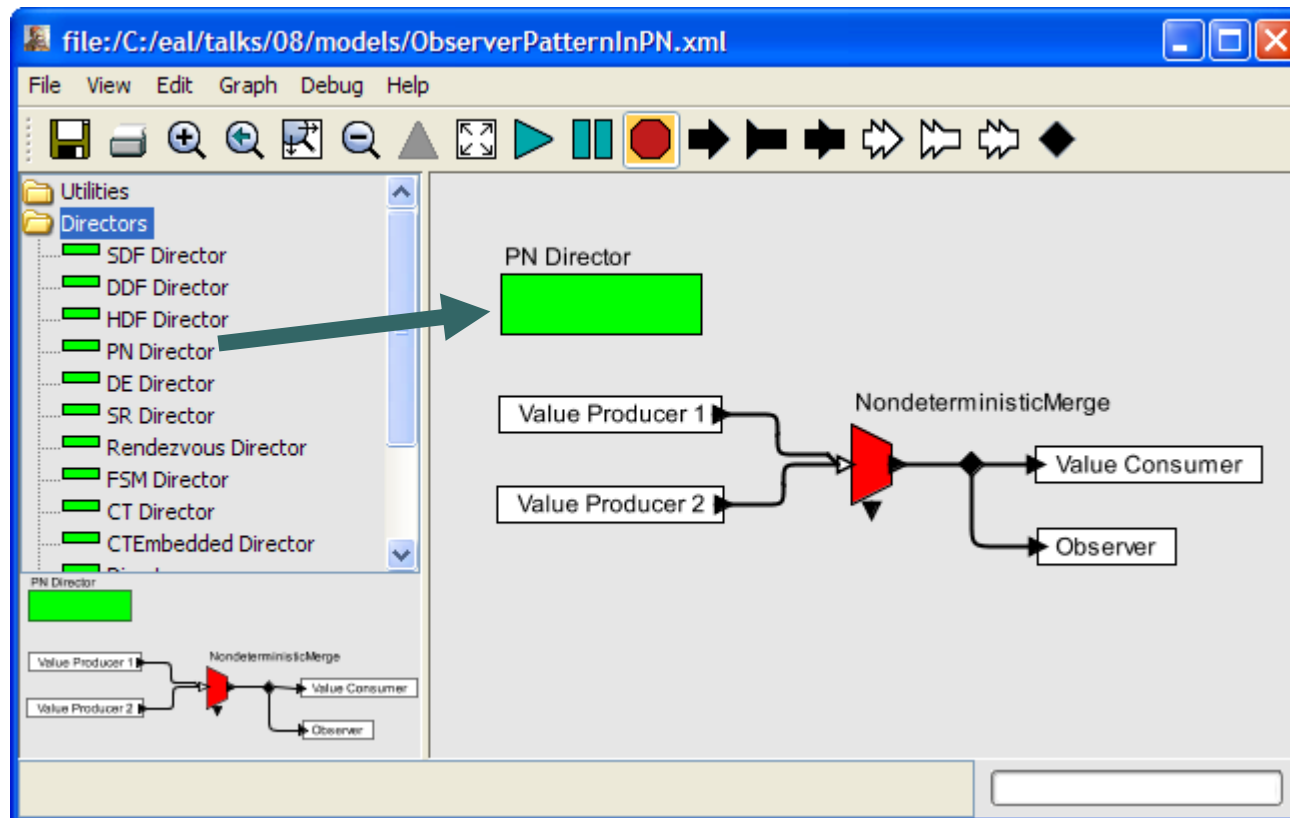
- Fair
- Demand driven
- Data driven
- Most mixtures of demand and data driven

*If programmers are building such programs with message passing libraries or threads, what will keep them from repeating these mistakes that have been made by top experts in the field?*

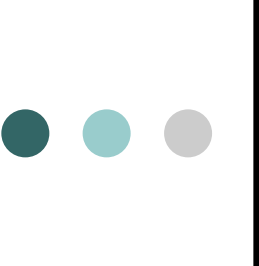
These problems have been solved!  
*Let's not make programmers re-solve  
them for every program.*

Library of  
directors

Program using actor-oriented  
components and a PN MoC



In Ptolemy II, a programmer specifies a *director*, which provides much more structure than message-passing or thread library. It provides a concurrent *model of computation* (MoC).



The PN Director solves the above problems by implementing a “useful execution”

Define a **correct execution** to be any execution for which after any finite time every signal is a prefix of the signal given by the (Kahn) least-fixed-point semantics.

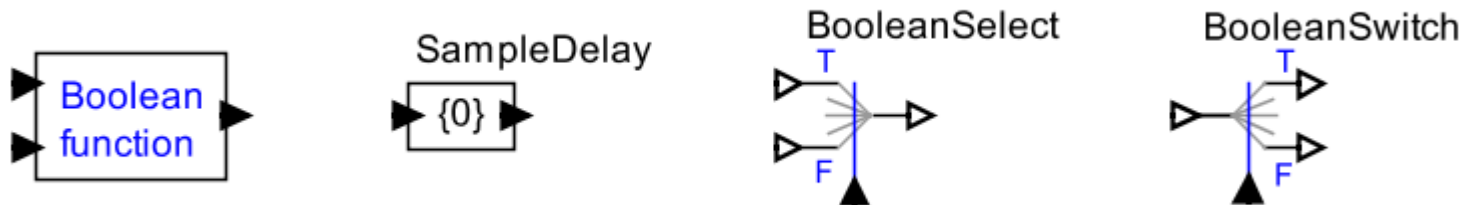
Define a **useful execution** to be a correct execution that satisfies the following criteria:

1. For every non-terminating model, after any finite time, a useful execution will extend at least one stream in finite (additional) time.
2. If a correct execution satisfying criterion (1) exists that executes with bounded buffers, then a useful execution will execute with bounded buffers.

● ● ● | **Programmers should not have to figure out how to solve these problems!**

*Undecidability and Turing Completeness* [Buck 93]

Given the following four actors and Boolean streams, you can construct a universal Turing machine:



Hence, the following questions are undecidable:

- Will a model deadlock (terminate)?
- Can a model be executed with bounded buffers?



## Our solution: Parks' Strategy [Parks 95]

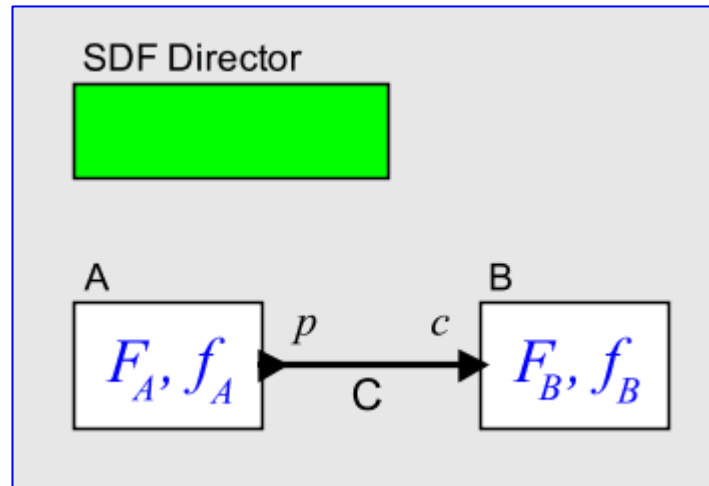
### This “solves” the undecidable problems:

- Start with an arbitrary bound on the capacity of all buffers.
- Execute as much as possible.
- If deadlock occurs and at least one actor is blocked on a write, increase the capacity of at least one buffer to unblock at least one write.
- Continue executing, repeatedly checking for deadlock.

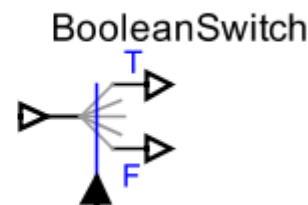
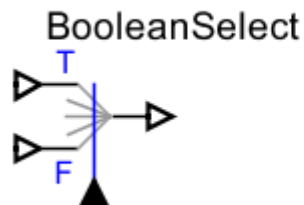
This delivers a useful execution (possibly taking infinite time to tell you whether a model deadlocks and how much buffer memory it requires).

More constrained MoCs yield better to static analysis. E.g.

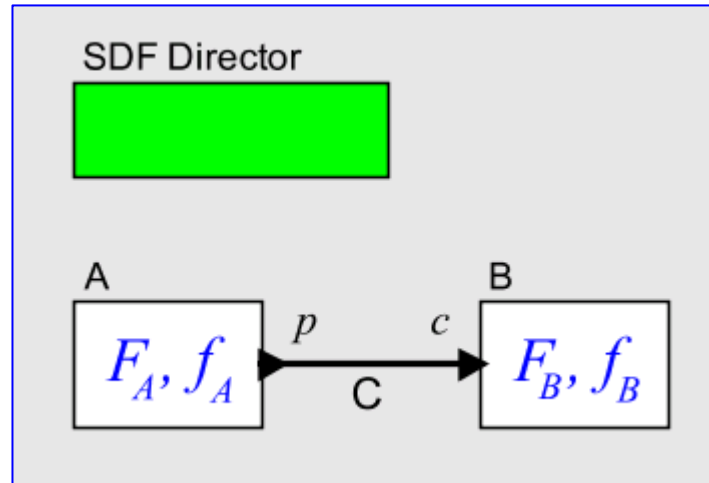
*Synchronous Dataflow (SDF)* [Lee & Messerschmitt, 87]



Limit the expressiveness by constraining the number of tokens consumed and produced on each firing to be constant. Eliminates:



# Balance Equations



Let  $q_A, q_B$  be the number of firings of actors A and B.

Let  $p_C, c_C$  be the number of token produced and consumed on a connection C.

Then the system is *in balance* if for all connections C

$$q_A p_C = q_B c_C$$

where A produces tokens on C and B consumes them.



# Decidable Models

For SDF, boundedness and deadlock are decidable. Moreover, parallel scheduling can be done statically, and useful optimization problems can be solved. See for example:

1. Ha and Lee, "Compile-Time Scheduling and Assignment of Dataflow Program Graphs with Data-Dependent Iteration," *IEEE Trans. on Computers*, November, 1991.
2. Sih and Lee, "Declustering: A New Multiprocessor Scheduling Technique," *IEEE Trans. on Parallel and Distributed Systems*, June 1993.
3. Sih and Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Trans. on Parallel and Distributed Systems*, February 1993.

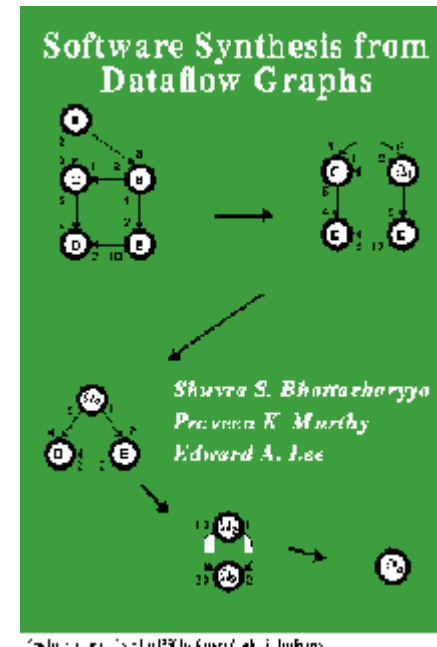


Although this makes scheduling decidable,  
complex optimization problems remain.

*Programmers should not have to solve these!*

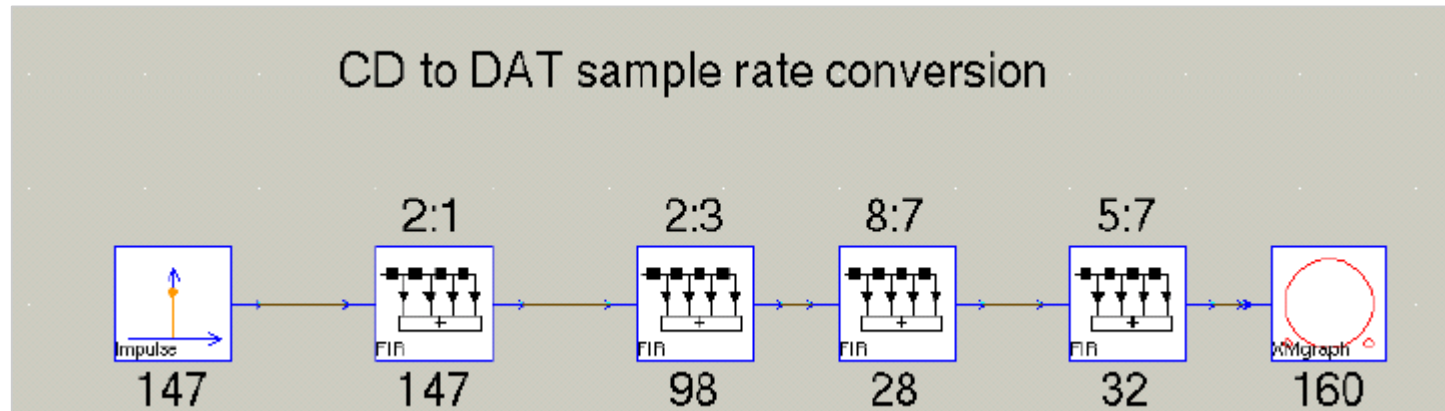
Optimization criteria that might be applied:

- Minimize buffer sizes.
- Minimize the number of actor activations.
- Minimize the size of the representation of the schedule (code size).
- Maximize the throughput.
- Minimize latency.



See Bhattacharyya, Murthy, and Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Press, 1996.

# Example: Minimum Buffer Schedule for a 6-Actor Dataflow Model



ABABCABABCABABCDEAFFFFFFBABCABABCABABCDE  
AFFFFFFBCABABCABABCDEAFFFFFFBCABABCABABC  
DEAFFFFFFBABCABABCABABCDEAFFFFFFBABCABCA  
BABCDEAFFFFFFBCABABCABABCDEAFFFFFFFEBCA  
FFFFFFBABCABABCDEAFFFFFFBABCABABCABABCDEAF  
FFFFFFBABCABABCABABCDEAFFFFFFBCABABCABABC  
DEAFFFFFFBCABABCABABCDEAFFFFFFBABCABABCABCA  
BCDEAFFFFFFBABCABABCABABCDEAFFFFFFFEBCAFFFFFFB  
ABCABABCABABCDEAFFFFFFBCABABCABABCDEAFFFFFFBA  
BCABABCABABCABABCDEAFFFFFFBABCABABCABABCDEAFFF  
FFBCABABCABABCABABCDEAFFFFFFBCABABCABABCDEAF  
FFFFFFBABCABABCABABCABABCDEAFFFFFFFEBAFFFFFFBCABC  
ABABCDEAFFFFFFBCABABCABABCABABCDEAFFFFFFBCA  
BABCABABCDEAFFFFFFBABCABABCABABCABABCDEAFFFFFFB  
ABCABABCABABCDEAFFFFFFBCABABCABABCABABCDEAF  
FFFFFFBCABABCABABCDEFFFFFFEFFFFFFF



SDF, by itself, is too restrictive.  
*Extensions improve expressiveness.*

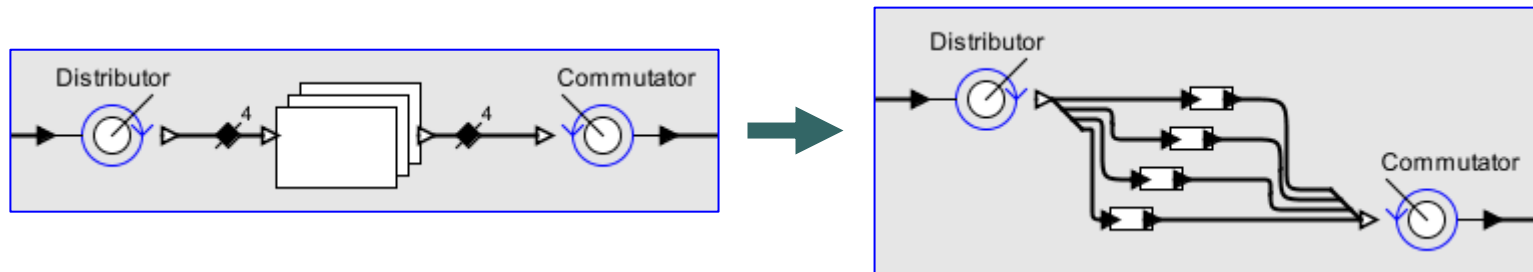
- Heterochronous Dataflow [Girault, Lee, and Lee, 97]
- Structured Dataflow [Kodosky 86, Thies et al. 02]
- (the other) Synchronous Dataflow [Halbwachs et al. 91]
- Cyclostatic Dataflow [Lauwereins 94]
- Multidimensional SDF [Lee & Murthy 96]
- Parameterized Dataflow [Bhattacharya et al. 00]
- Teleport Messages [Thies et al. 05]

All of these remain decidable

*And there are many other non-dataflow actor-oriented MoCs to bring into the mix!*

# Work to be done

- Develop language support for actor-oriented design (like what C++ did for object-oriented design).
- Generalize parallel schedulers to work for more expressive MoCs.
- Support mixing MoCs to enable to exploiting static analysis where possible (this is partially done in Ptolemy II, but much work is left).
- Develop and support design patterns that expose parallelism in actor-oriented designs (e.g. gather-scatter, MapReduce, etc.)





## Conclusion:

### *Disciplined Concurrent Models of Computation*

- Do not use nondeterministic programming models to accomplish deterministic ends.
- Use concurrency models that have analogies in the physical world (actors, not threads).
- Provide these in the form of models of computation (MoCs) with well-developed semantics and tools.
- Use specialized MoCs to exploit semantic properties (avoid excess generality).
- Leave the choice of shared memory or message passing to the compiler.