
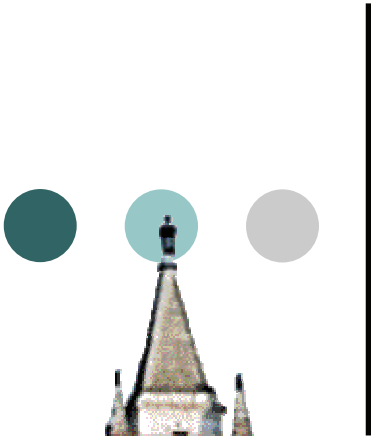




# The Future of Embedded Software

Edward A. Lee

Professor, Chair of EE, and Associate Chair of EECS  
UC Berkeley



ARTEMIS 2006 Annual Conference  
Graz, Austria  
May 22-24, 2006



# Why Embedded Software? Why Now?

“Information technology (IT) is on the verge of another revolution. Driven by the increasing capabilities and ever declining costs of computing and communications devices, IT is being embedded into a growing range of physical devices linked together through networks and will become ever more pervasive as the component technologies become smaller, faster, and cheaper... These networked systems of embedded computers ... have the potential to change radically the way people interact with their environment by linking together a range of devices and sensors that will allow information to be collected, shared, and processed in unprecedented ways. ... The use of [these embedded computers] throughout society could well dwarf previous milestones in the information revolution.”

*National Research Council Report  
Embedded Everywhere*



# The Key Obstacle to Progress: Gap Between Systems and Computing

- Traditional dynamic systems theory needs to adapt to better account for the behavior of software and networks.
- Traditional computer science needs to adapt to embrace time, concurrency, and the continuum of physical processes.



# The Next Systems Theory: Simultaneously Physical and Computational

## *The standard model:*

Embedded software is software on small computers. The technical problem is one of optimization (coping with limited resources).

## *The Berkeley model:*

Embedded software is software integrated with physical processes. The technical problem is managing time and concurrency in computational systems.



# Obstacles in Today's Technology: Consider Real Time

Electronics Technology Delivers Timeliness...

... and the overlaying software abstractions  
discard it.



# Computation in the 20<sup>th</sup> Century

$$f: \{0,1\}^* \rightarrow \{0,1\}^*$$

*A computation is a function that maps a finite sequence of bits into a finite sequence of bits.*

- *No time*
- *No concurrency*



# A Few 20<sup>th</sup> Century Innovations that Rely on Time Being Irrelevant

- Programming languages
- Caches
- Virtual memory
- Dynamic dispatch
- Speculative execution
- Power management (voltage scaling)
- Memory management (garbage collection)
- Just-in-time (JIT) compilation
- Multitasking (threads and processes)
- Networking (TCP)
- Theory (computability, complexity)



# Some Approaches Addressing Timeliness in Software

- Put time into programming languages
  - *Promising start:* Giotto, Discrete-event models, timed dataflow models
- Rethink the OS/programming language split
  - *Promising start:* TinyOS/nesc
- Rethink the hardware/software split
  - *Promising start:* FPGAs with programmable cores
- Memory hierarchy with predictability
  - *Promising start:* Scratchpad memories vs. caches
- Memory management with predictability
  - *Promising start:* Bounded pause time garbage collection
- Predictable, controllable deep pipelines
  - *Promising start:* Pipeline interleaving + stream-oriented languages
- Predictable, controllable, understandable concurrency
  - *Promising start:* Synchronous languages, SCADE
- Networks with timing
  - *Promising start:* Time triggered architectures, time synchronization
- Computational dynamical systems theory
  - *Promising start:* Hybrid systems, schedulability analysis





# Obstacles in Today's Technology: Consider Concurrency

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.*

Does this mean that it is hard for humans to reason about concurrency?

If so, we would not be able to function in the (highly concurrent) physical world.



# Most Concurrent Software is Built Using Threads

- Threads are sequential processes that share memory.
- Threads are either the implicit or explicit model in
  - Most real-time operating systems (RTOS's)
  - Device drivers
  - Most concurrent programs in C++ or Java



## 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):



## Example: Observer Pattern in Java

```
public void addListener(listener) {...}

public void setValue(newValue) {
    myValue = newValue;

    for (int i = 0; i < myListeners.length; i++) {
        myListeners[i].valueChanged(newValue)
    }
}
```

**Will this work in a  
multithreaded context?**

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



## Example: Observer Pattern With Mutual Exclusion (Mutexes)

```
public synchronized void addListener(listener) {...}

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

    for (int i = 0; i < myListeners.length; i++) {
        myListeners[i].valueChanged(newValue)
    }
}
```

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

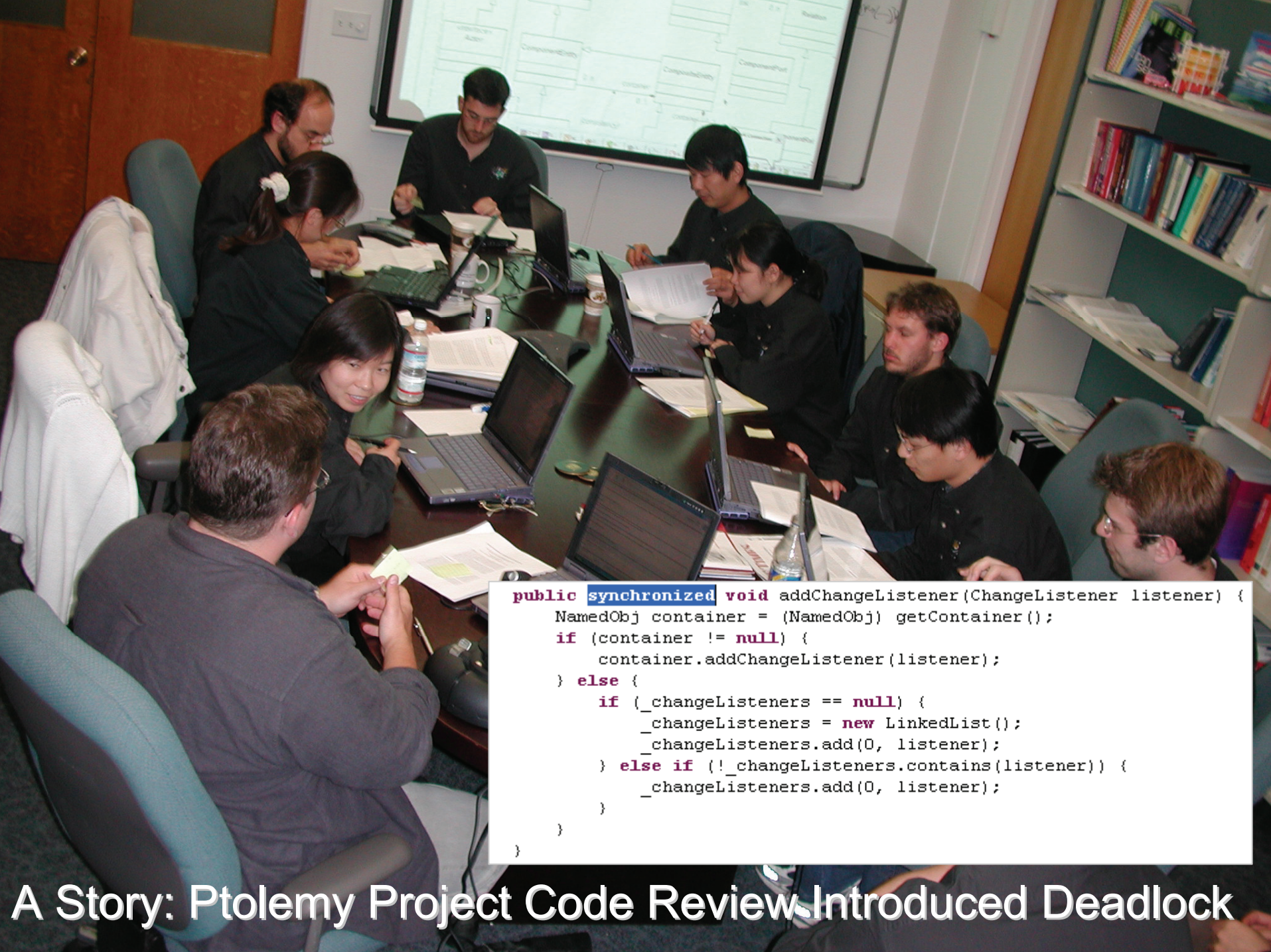


## Mutexes using Monitors are Minefields

```
public synchronized void addListener(listener) {...}  
  
public synchronized void setValue(newValue) {  
    myValue = newValue;  
  
    for (int i = 0; i < myListeners.length; i++) {  
        myListeners[i].valueChanged(newValue)  
    }  
}
```

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





```
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);  
        }  
    }  
}
```

A Story: Ptolemy Project Code Review Introduced Deadlock



# Simple Observer Pattern Becomes Not So Simple

```
public synchronized void addListener(listener) {...}
```

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

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

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

```
for (int i = 0; i < listeners.length; i++) {  
    listeners[i].valueChanged(newValue)  
}
```

```
}
```

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





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

```
public synchronized void addListener(listener) {...}
```

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

```
for (int i = 0; i < listeners.length; i++) {  
    listeners[i].valueChanged(newValue)  
}
```

```
}
```

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



# Such Problems can Linger Undetected in Code for a Very Long Time: Another Typical Story

```
/**
CrossRefList is a list that maintains pointers to other CrossRefLists.
...
@author Geroncio Galicia, Contributor: Edward A. Lee
@version $Id: CrossRefList.java,v 1.78 2004/04/29 14:50:00 eal Exp $
@since Ptolemy II 0.2
@Pt.ProposedRating Green (eal)
@Pt.AcceptedRating Green (bart)
*/
public final class CrossRefList implements Serializable {
    ...
    protected class CrossRef implements Serializable{
        ...
        // NOTE: It is essential that this method not be
        // synchronized, since it is called by _farContainer(),
        // which is. Having it synchronized can lead to
        // deadlock. Fortunately, it is an atomic action,
        // so it need not be synchronized.
        private Object _nearContainer() {
            return _container;
        }

        private synchronized Object _farContainer() {
            if (_far != null) return _far._nearContainer();
            else return null;
        }
        ...
    }
}
```

Code that had been in use for four years, central to Ptolemy II, with an extensive test suite with 100% code coverage, design reviewed to yellow, then code reviewed to green in 2000, causes a deadlock during a demo on April 26, 2004.

# What it Feels Like to Use the *synchronized* Keyword in Java



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



# Families of Possible Solutions

- Train programmers to use threads.
- Improve software engineering processes.
- Identify and apply design patterns.
- Quantify quality of service.
- Verify system properties formally.

None of these deliver a rigorous, analyzable, and understandable model of concurrency.



A stake in the ground...

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



# 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).

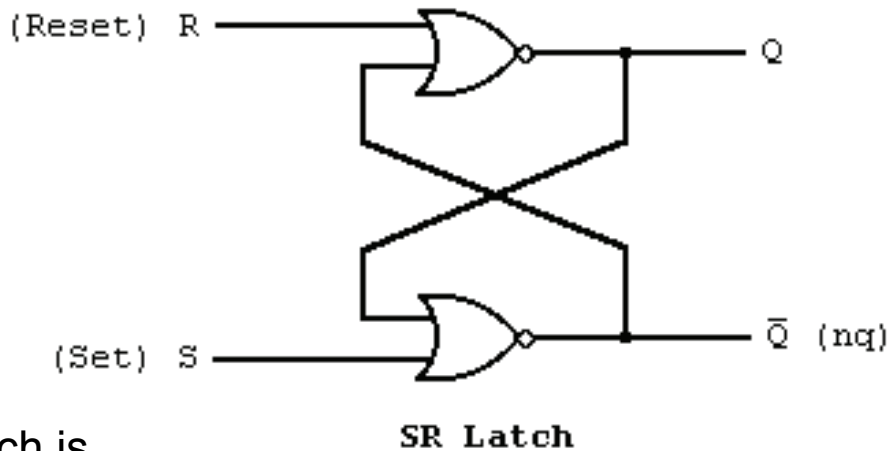


# Improve Threads? Or Replace Them?

- Improve threads
  - Pruning tools (mutexes, semaphores, ...)
  - OO programming
  - Coding rules (Acquire locks in the same order...)
  - Libraries (Stapl, Java 5.0, ...)
  - Patterns (MapReduce, Transactions, ...)
  - Formal verification (Blast, thread checkers, ...)
  - Enhanced languages (Split-C, Cilk, Guava, ...)
  - Enhanced mechanisms (Promises, futures, ...)
- Change concurrency models

# Threads are Not the Only Possibility:

## 1<sup>st</sup> example: Hardware Description Languages



*e.g. VHDL:*

```
entity latch is
  port (s,r : in bit;
        q,nq : out bit);
end latch;
```

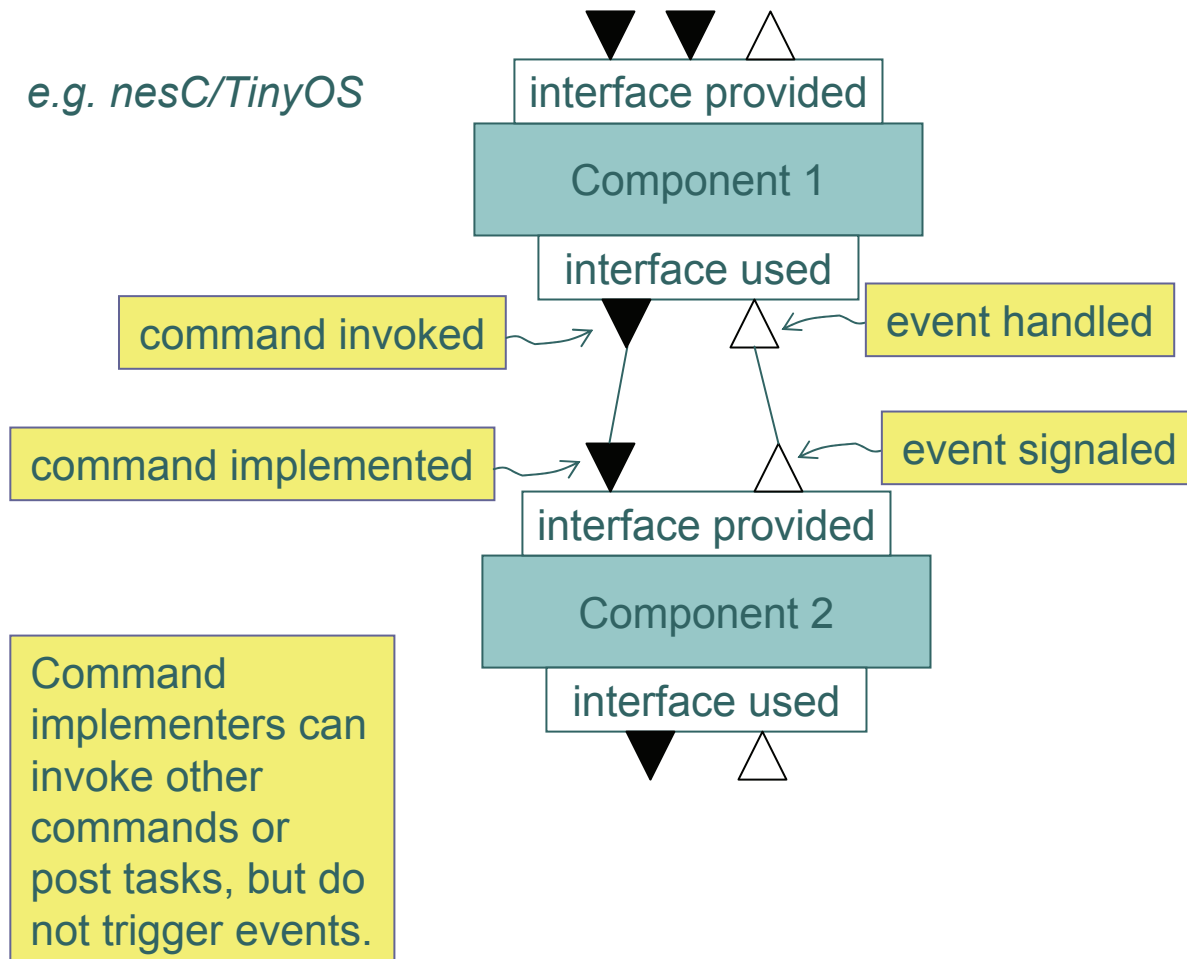
```
architecture dataflow of latch is
begin
  q<=r nor nq;
  nq<=s nor q;
end dataflow;
```



# Threads are Not the Only Possibility:

## 2<sup>nd</sup> example: Sensor Network Languages

*e.g. nesC/TinyOS*

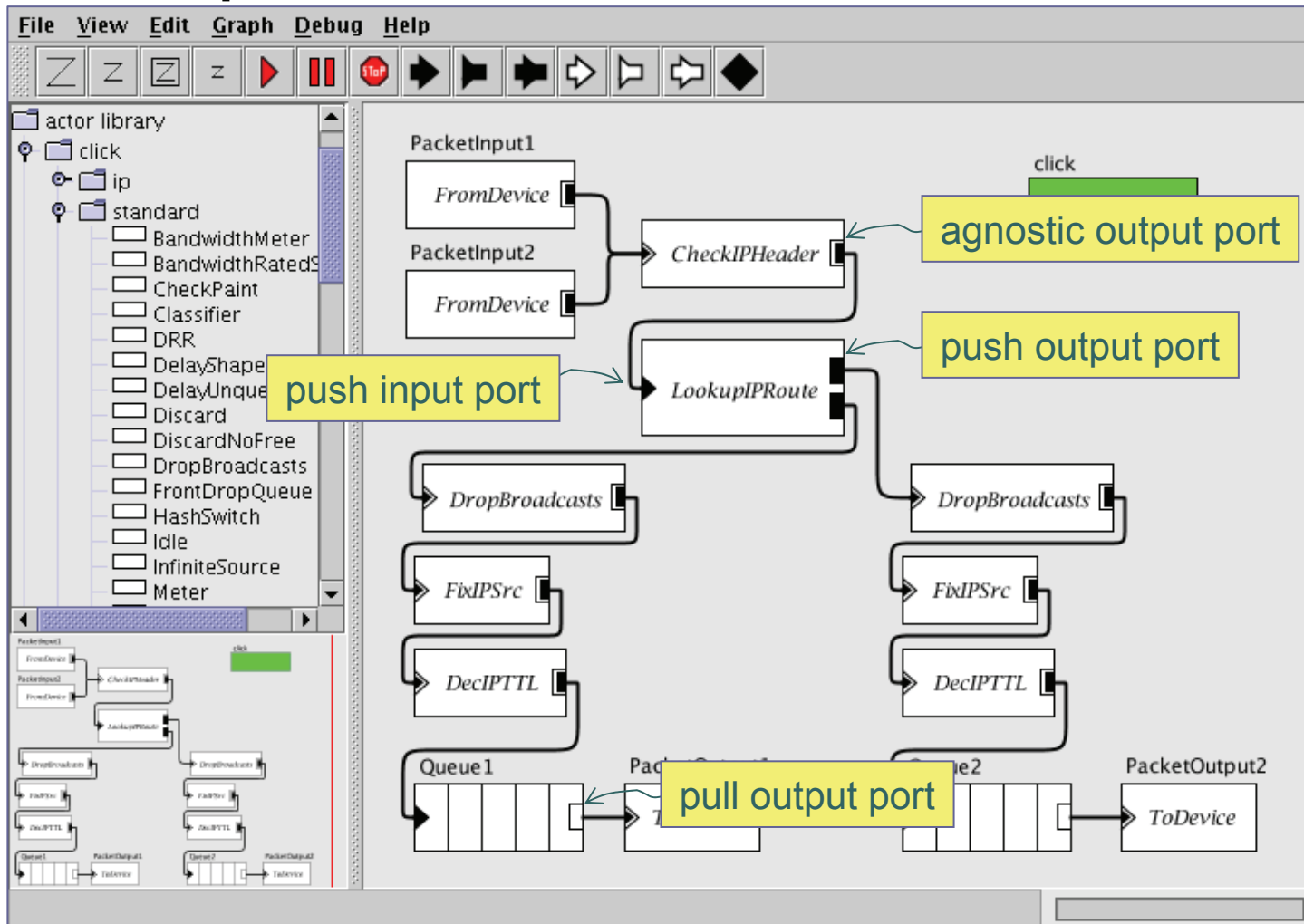


Typical usage pattern:

- hardware interrupt signals an event.
- event handler posts a task.
- tasks are executed when machine is idle.
- tasks execute atomically w.r.t. one another.
- tasks can invoke commands and signal events.
- hardware interrupts can interrupt tasks.
- exactly one monitor, implemented by disabling interrupts.

# Threads are Not the Only Possibility:

## 3<sup>rd</sup> example: Network Languages

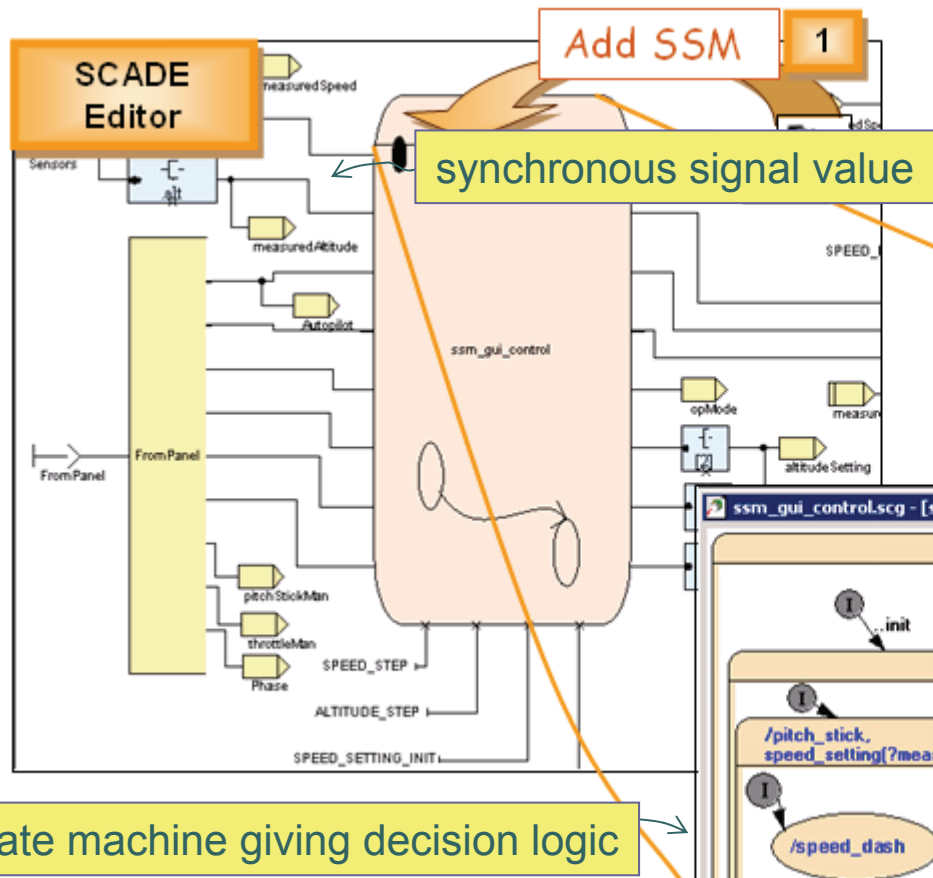


Typical usage pattern:

- queues have push input, pull output.
- schedulers have pull input, push output.
- thin wrappers for hardware have push output or pull input only.

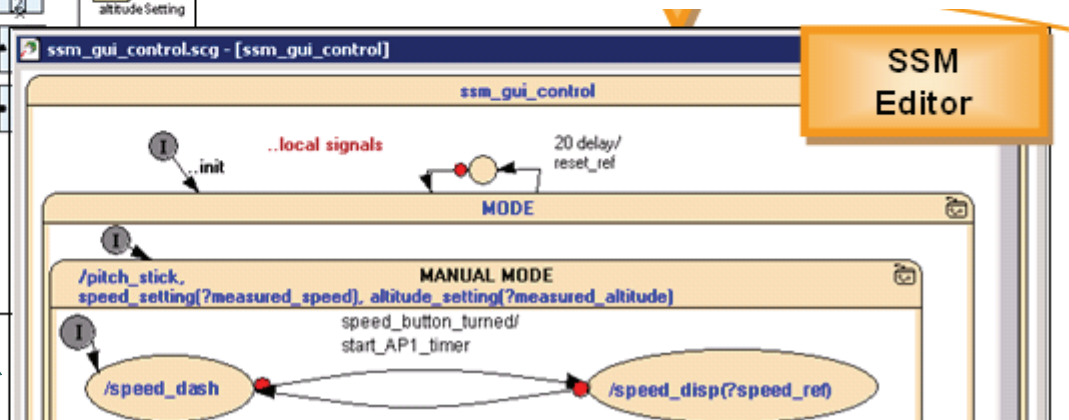
# Threads are Not the Only Possibility:

## 4<sup>th</sup> example: Synchronous Languages

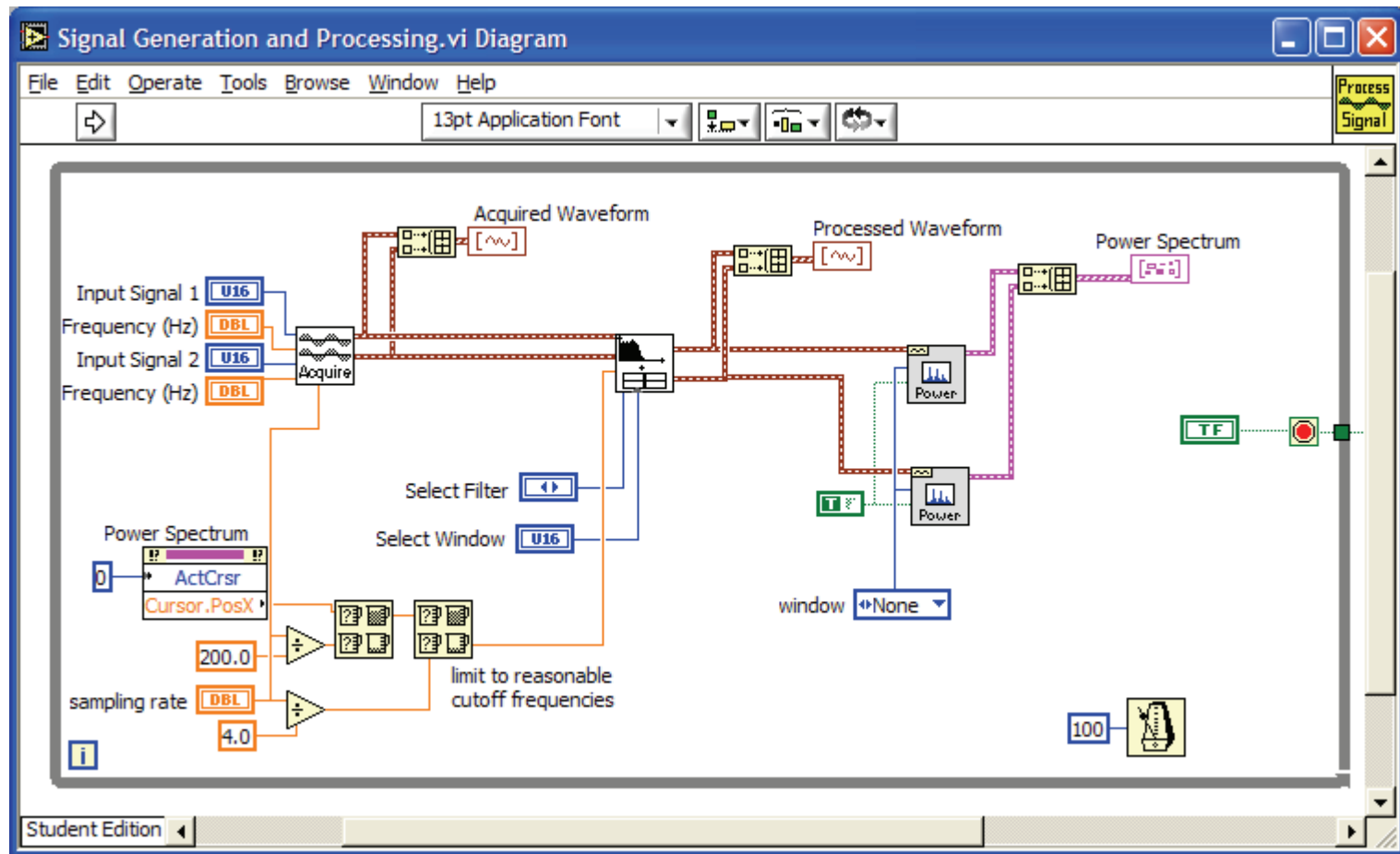


Typical usage pattern:

- specify tasks aligned to a master “clock” and subclocks
- clock calculus checks for consistency and deadlock
- decision logic is given with hierarchical state machines.



# Threads are Not the Only Possibility: 5<sup>th</sup> example: Instrumentation Languages



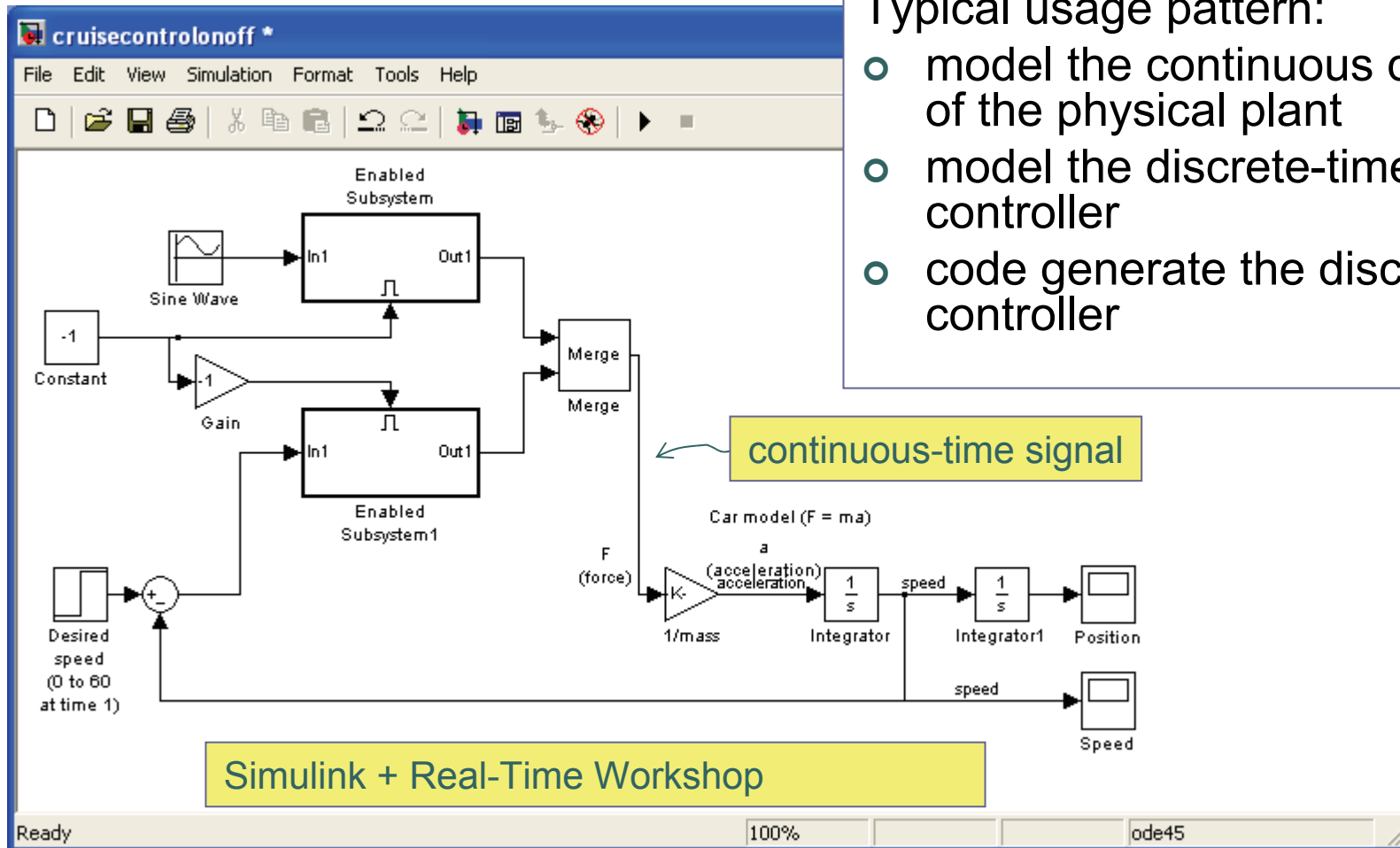
e.g. LabVIEW, Structured dataflow model of computation

# Threads are Not the Only Possibility:

## 6<sup>th</sup> example: Continuous-Time Languages

Typical usage pattern:

- model the continuous dynamics of the physical plant
- model the discrete-time controller
- code generate the discrete-time controller





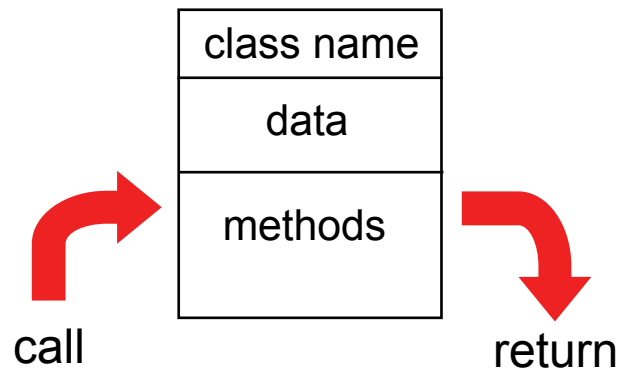
# A Common Feature

- None is mainstream in computing.
- All are domain-specific.
- Emphasis on concurrent composition with determinism:
  - Composability
  - Security
  - Robustness
  - Resource management
  - Evolvability

Compared with message passing schemas, such as PVM, MPI, OpenMP, these impose stricter interaction patterns that yield determinism in the face of concurrency.

# Many of These and Other Concurrent Component Models are *Actor Oriented*

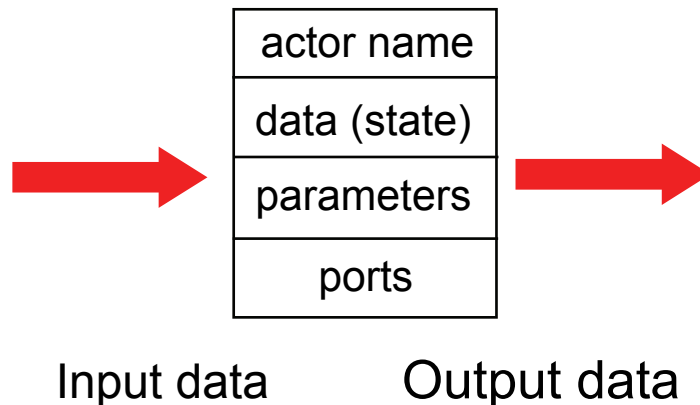
The established: Object-oriented:



What flows through  
an object is  
sequential control

Things happen to objects

The alternative: Actor oriented:



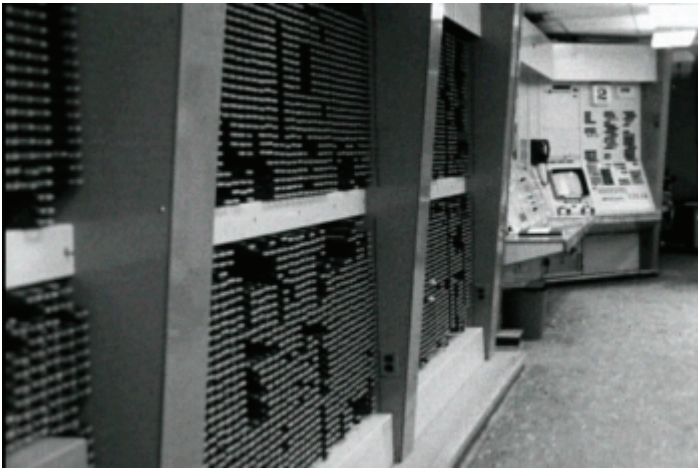
Actors make things happen

What flows through  
an object is  
streams of data

# The First (?) Actor-Oriented Platform

*The On-Line Graphical Specification of Computer Procedures*

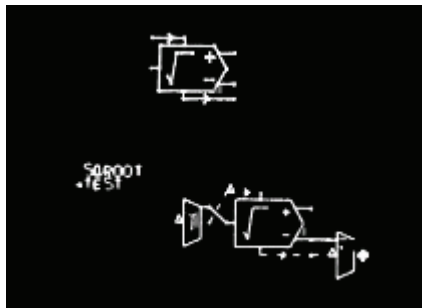
W. R. Sutherland, Ph.D. Thesis, MIT, 1966



MIT Lincoln Labs TX-2 Computer



Bert Sutherland with a light pen



Bert Sutherland used the first acknowledged object-oriented framework (Sketchpad, created by his brother, Ivan Sutherland) to create the first actor-oriented programming framework.

Partially constructed actor-oriented model with a class definition (top) and instance (below).

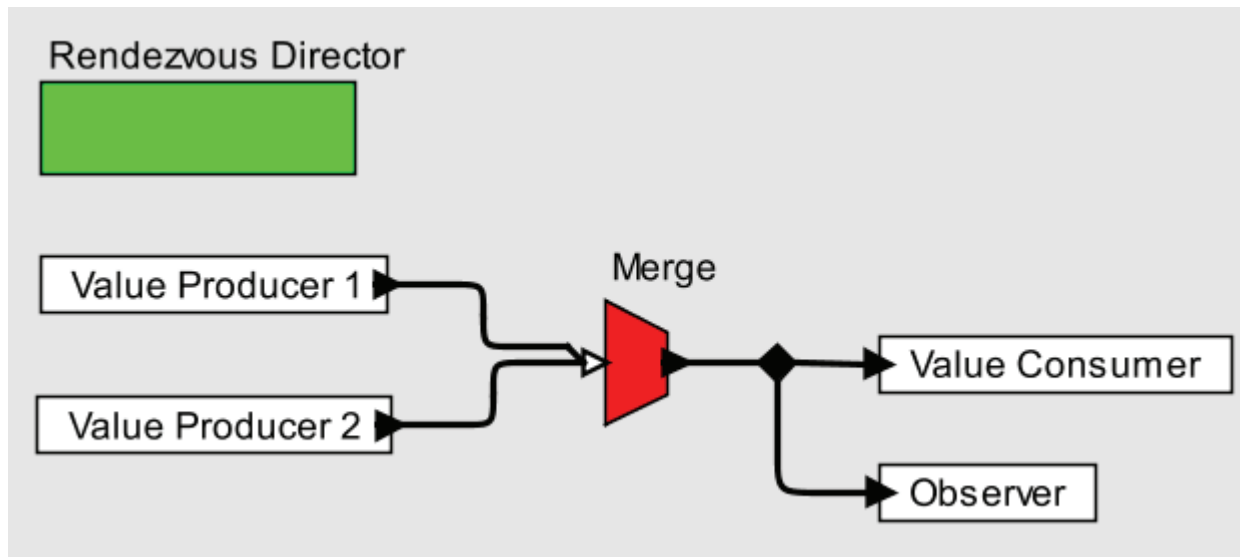




## Recall the Observer Pattern

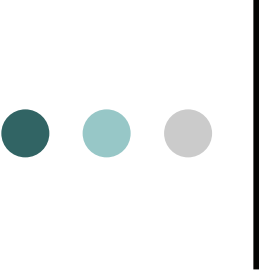
“The *Observer pattern* defines a one-to-many dependency between a subject object and any number of observer objects so that when the subject object changes state, all its observer objects are notified and updated automatically.”

# Observer Pattern using CSP-like Rendezvous



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

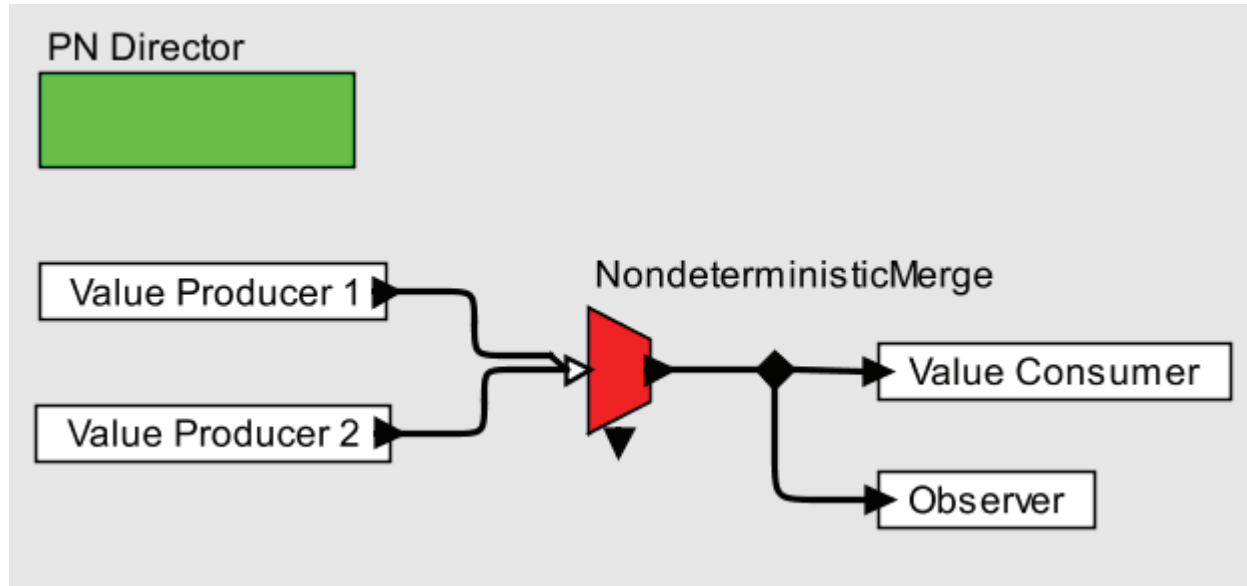
The above diagram is an expression in a *composition language* with a visual syntax.



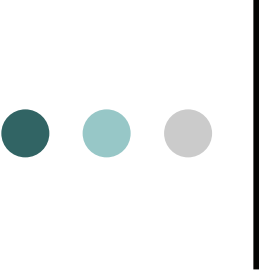
Now that we've made a trivial design pattern trivial, we can work on more interesting aspects of the design.

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 Process Networks [Kahn 1974] Extended with Nondeterministic Merge

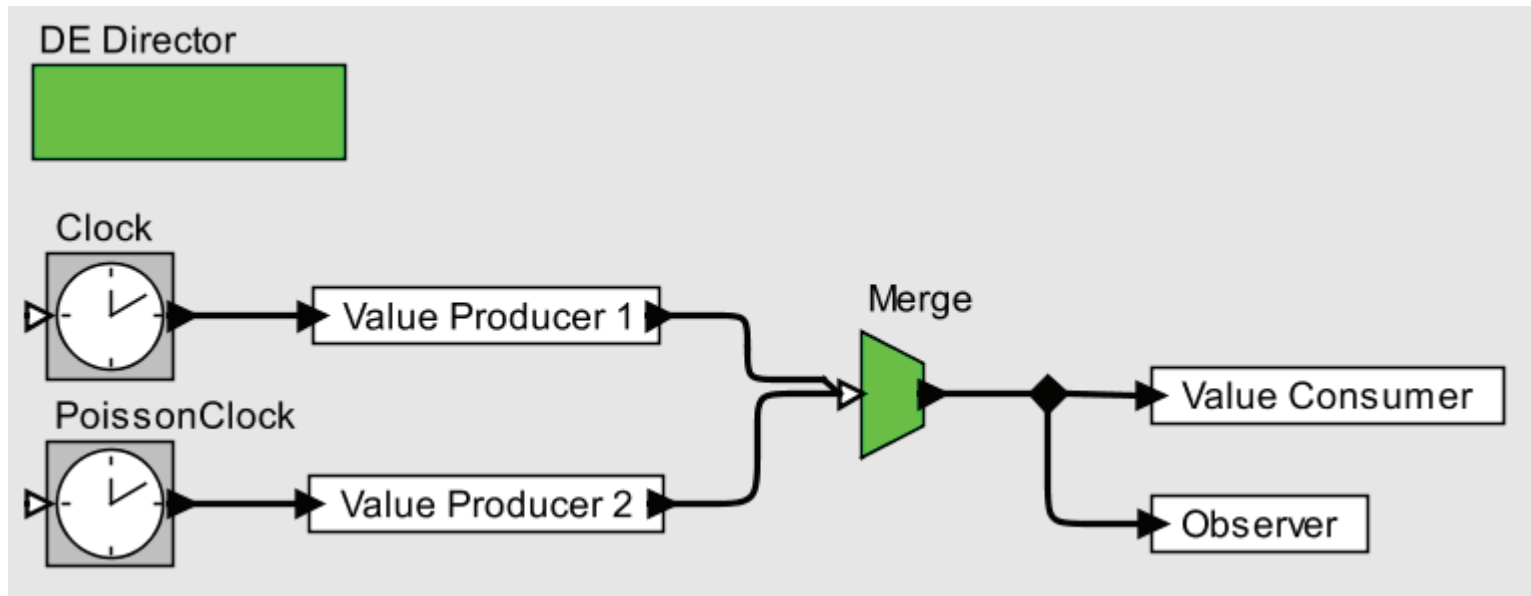


Each actor is a process, communication is via streams, and the **NondeterministicMerge** explicitly merges streams nondeterministically.



Suppose further that we want to explicitly specify the timing of producers?

# Observer Pattern using Discrete Events



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



Instead of a Program Being...

$$f: B^{**} \rightarrow B^{**}$$



... a Program Can Be

$$f: (T \rightarrow B^*)^P \rightarrow (T \rightarrow B^*)^P$$

*For some partially ordered set  $T$ .*

A computation is a function that maps an evolving pattern of bits into an evolving pattern of bits.

*Composition of concurrent components becomes function composition, resulting in well-founded determinate computation (**composability!**)*





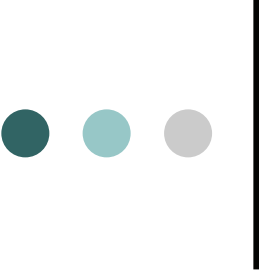
# Challenges

- Computation is deeply rooted in the sequential paradigm.
  - Threads appear to adhere to this paradigm, but throw out its essential attractiveness.
- Programmers are reluctant to accept new syntax
  - Regrettably, syntax has a bigger effect on acceptance than semantics, as witnessed by the wide adoption of threads.
- Only general purpose languages are interesting
  - A common litmus test: must be able to write the compiler for the language in the language.



# Opportunities

- New syntaxes can be accepted when their purpose is orthogonal to that of established languages.
  - Witness UML, a family of languages for describing object-oriented design, complementing C++ and Java.
- Composition languages can provide capabilities orthogonal to those of established languages.
  - The syntax can be noticeably distinct (as in the diagrams shown before).
- Patterns of composition can be codified
  - E.g.: MapReduce.



# So What is the Future of Embedded Software?

I don't know...

But I know what it should be:

Foundational architectures that combine software and models of physical dynamics with composition languages that have concurrency and time in a rigorous, composable, semantic framework.





# Conclusion

- Many innovations in computation lose timing predictability.
- If timing predictability is important, many things have to change.
- Threads are the dominant concurrency model for programmers.
- Threads discard the most essential features of programs.
- Threads are incomprehensible to humans.
- Threads  $\neq$  concurrency.
- Deterministic aims should be achieved with deterministic means.
- Nondeterminism should be used judiciously and explicitly.
- Actor orientation offers alternative component models.
- Composition languages can realize actor models.
- There *are* opportunities for language design.