Chess

Understandable Concurrency

Edward A. Lee

Professor, Chair of EE, and Associate Chair of EECS

Director, CHESS: Center for Hybrid and Embedded Software Systems

Director, Ptolemy Project

UC Berkeley

Chess Review

November 21, 2005

Berkeley, CA

What would it take to make reliable concurrent software?

The standard concurrency model, based on:

- threads,
- semaphores, and
- mutual exclusion locks

results in programs that are *incomprehensible* to humans.

- These methods date to the 1960's [Dijkstra].
- Tauntingly simple rules (e.g. always grab locks in the same order [Lea]) are impossible to apply in practice.
- Formal methods can expose flaws, but cannot make programs understandable.

Red Herrings

- Training programmers to use threads.
- Software engineering process improvements.
- Attention to “non-functional” properties.
- Design patterns.
- Quality of service.

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

Problems with Threads: Example: Simple Observer Pattern

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

public void setValue(newValue) {
    myValue = newValue;

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

What's wrong with this (in a multithreaded context)?

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

Example: Simple Observer Pattern With Mutual Exclusion (Mutexes) using Monitors

```
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);
        }
    }
}
```

We made exactly this mistake when a code review identified exactly this concurrency flaw in Ptolemy II.

Simple Observer Pattern Becomes Not So Simple

```
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)
    }
}
```

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 perfect.
What's wrong with it?

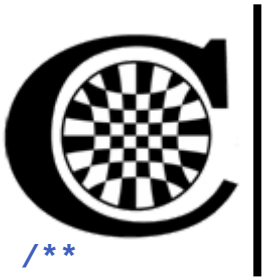
Simple Observer Pattern: Is it Even Possible to Make It Right?

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

public void setValue(newValue) {
    synchronized(this) {
        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!

This is Ridiculous…

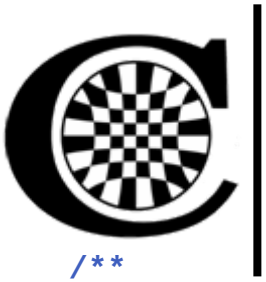
One of the simplest, textbook design patterns, commonly used throughout concurrent programs, becomes a potential Masters Thesis!



… and it Still Gets Worse…

```
/**
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;
        }
        …
    }
}
```

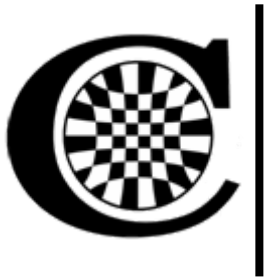
Code that had been in use for four years, central to Ptolemy II, with an extensive test suite (100% code coverage), design reviewed to yellow, then code reviewed to green in 2000, causes a deadlock during a demo on April 26, 2004.



… and Doubts Remain

```
/**
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{
        …
        private synchronized void _dissociate() {
            _unlink(); // Remove this.
            // NOTE: Deadlock risk here!  If _far is waiting
            // on a lock to this CrossRef, then we will get
            // deadlock. However, this will only happen if
            // we have two threads simultaneously modifying a
            // model. At the moment (4/29/04), we have no
            // mechanism for doing that without first
            // acquiring write permission the workspace().
            // Two threads cannot simultaneously hold that
            // write access.
            if (_far != null) _far._unlink(); // Remove far
        }
    }
```

Safety of this code depends on policies maintained by entirely unconnected classes. The language and synchronization mechanisms provide no way to talk about these systemwide properties.

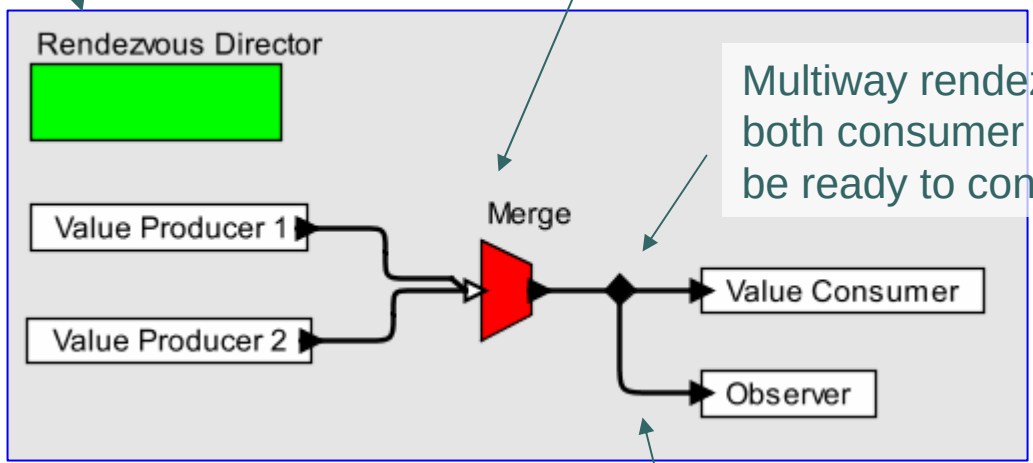


Nontrivial software written with threads, semaphores, and mutexes cannot and should not be trusted!

Stronger Synchronization Properties are Delivered by the Rendezvous MoC

Model of computation: Processes communicating via rendezvous.

Conditional rendezvous marries one of the input processes with the output process.



Multiway rendezvous requires both consumer processes to be ready to consume.

Deadlock is provably avoided

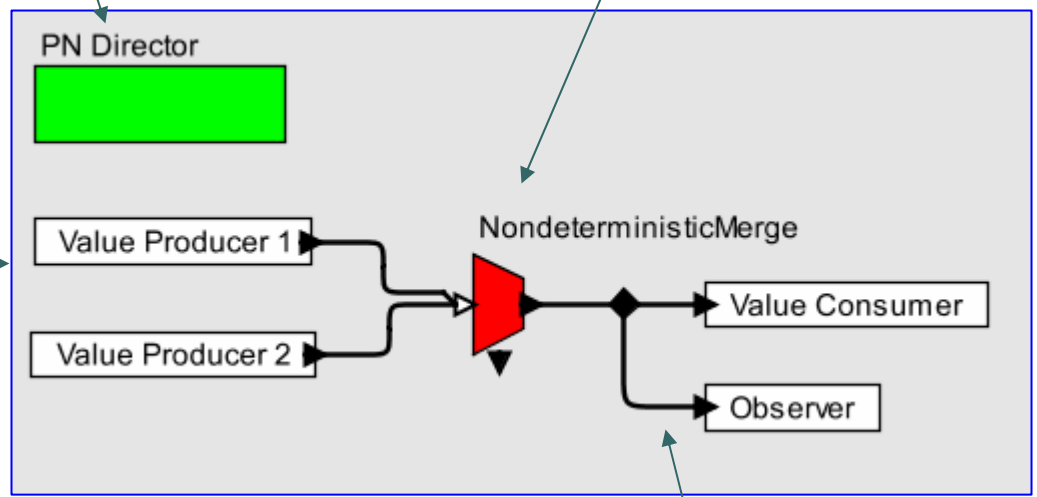
This is a generalization of CSP with multiway and conditional rendezvous, again implemented in a coordination language with a visual syntax.

Observer sees values *at the same time* as the Value Consumer

Observer Pattern in Process Networks is Trivial, as it Should Be!

Model of computation: Processes communicating via unbounded FIFO queues.

Explicit nondeterminism (vs. unexpressed race condition)



Deadlock is provably avoided

Observer sees values in the same order as the Value Consumer

You can think of this as a generalization of Unix Pipes implemented in a Coordination Language with a Visual Syntax.

The Lee Principle of Nondeterminism

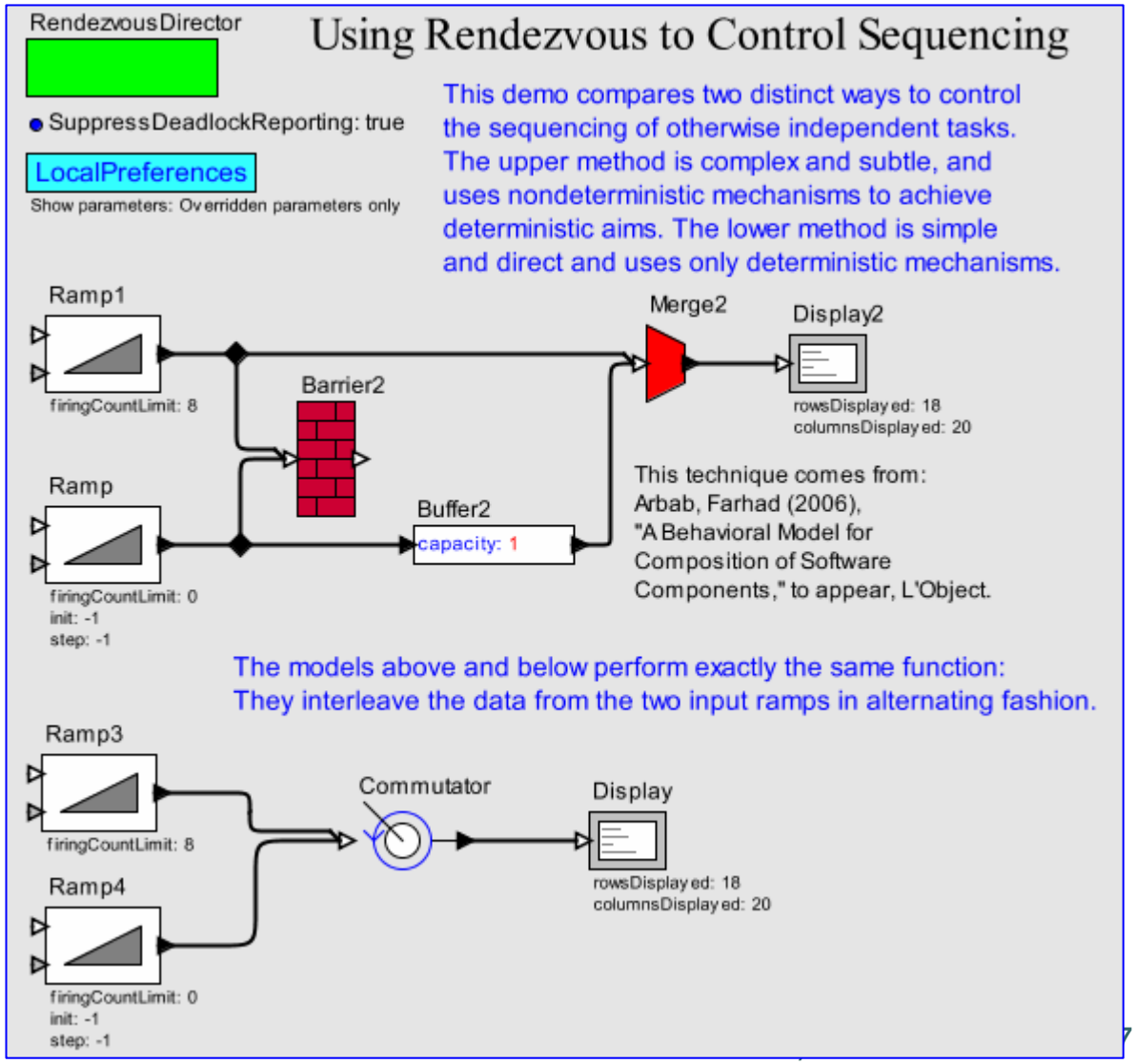
Deterministic behavior should be accomplished with deterministic mechanisms.

Only nondeterministic behavior should use nondeterministic mechanisms.

This rules out using threads for almost any deterministic behavior!

Actor-Oriented Approached Don't Guarantee Good Design

The two models at the right implement the same function, but the upper one uses subtle and complex nondeterministic mechanisms, while the one at the bottom uses simple and direct deterministic mechanisms.



Make Easy Concurrency Properties Easy, so We Can Focus on the Harder Ones

- Timing
- Controlled nondeterminacy
- Consistency in distributed programs
- Simultaneity
- Fairness
- …

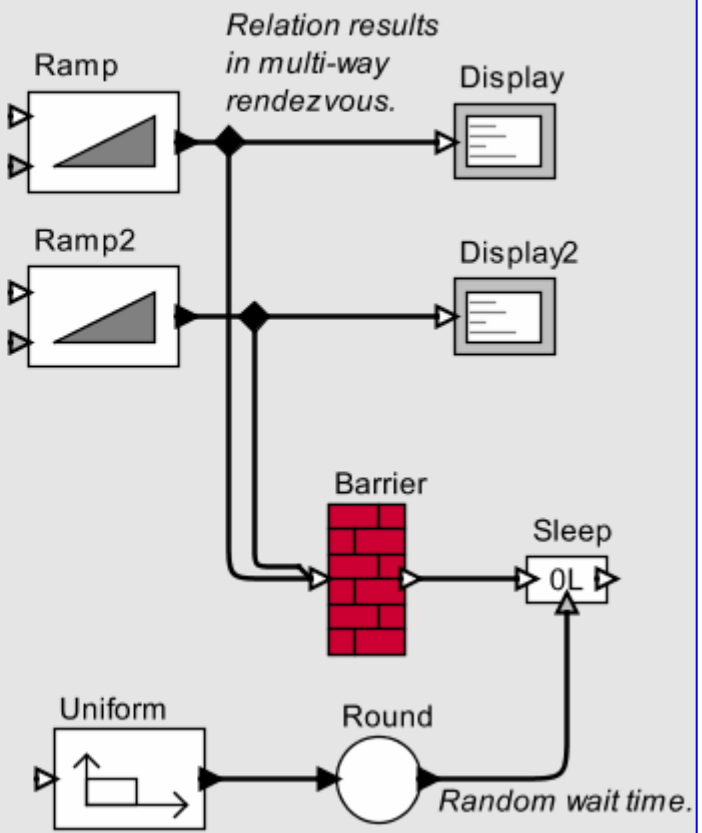
Example: Coordinating Timing of Independent Processes

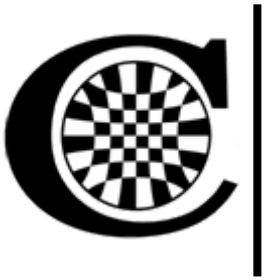
RendezvousDirector

Illustration of Barrier Synchronization using Rendezvous

This model illustrates a design pattern with rendezvous called a "barrier synchronization." In this example, the two Ramps are sending increasing sequences of integers to the Displays. However, the transfer is constrained to occur only when the Barrier actor reads inputs. Thus, a multi-way rendezvous between the two Ramp actors, the two Display actors, and the Barrier actor constrains the two transfers to the Display actors to occur simultaneously. The Barrier actor, after reading inputs, will rendezvous with the Sleep actor connected to its output. This sleep actor will sleep a random amount of time after reading its input, and during that time will not accept additional inputs. Thus, after the first two transfers to the Display actors (why two?) the time between transfers is controlled by the Sleep actor.

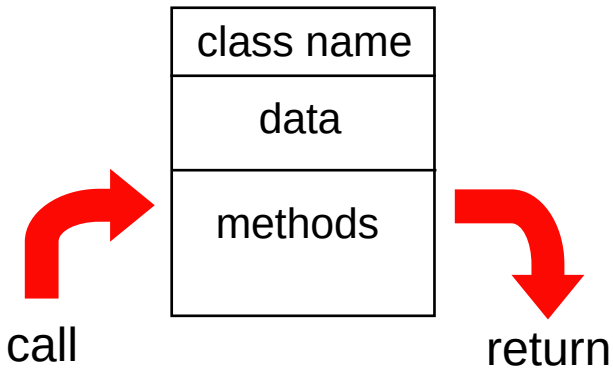
Author: Edward A. Lee





These Techniques are Examples of a Family of Concurrent Component Technologies

The old: Object-oriented:

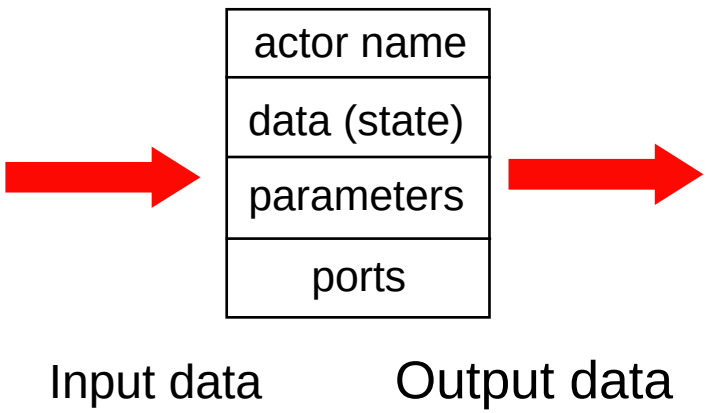


What flows through an object is sequential control

Things happen to objects

The new: Actor oriented:

Actors make things happen



What flows through an object is streams of data

Useful Actor-Oriented Models of Computation

- Synchronous/reactive (SCADE, Esterel, Statecharts, …)
- Time triggered (Giotto, Simulink, …)
- Dataflow (Labview, SPW, …)
- Discrete events (VHDL, Verilog, Opnet, Visualsense, …)
- Distributed discrete-events (Croquet, Time Warp, …)
- Continuous time (Simulink, …)
- Hybrid systems (HyVisual, Charon, …)
- Event triggered (xGiotto, nesC/TinyOS, …)
- …

None of these have threads!

Conclusion

The time is right to create a 21-st century technology for concurrent computing.