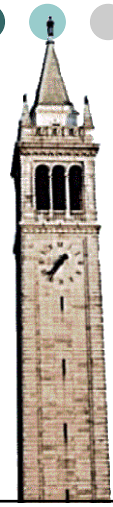
The Problem With Threads



Edward A. Lee

Robert S. Pepper Distinguished Professor and Chair of EECS
UC Berkeley
- and -
Senior Technical Adviser, director, and co-founder of
BDTI

Class #: ESC-211
Embedded Systems Conference (ESC)
Tuesday, 3 April 2007

Concurrency in Software Practice, As of 2007

- Component technologies
  - Objects in C++, C#, or Java
  - Wrappers as service definitions
- Concurrency
  - Threads (shared memory, semaphores, mutexes, …)
  - Message Passing (synchronous or not, buffered, …)
- Distributed computing
  - Distributed objects wrapped in web services, Soap, CORBA, DCOM, …

Observations

Threads and objects dominate SW engineering.

- *Threads*: Sequential computation with shared memory.
- *Objects*: Collections of state variables with procedures for observing and manipulating that state.

Even distributed objects create the illusion of shared memory through proxies.

- The components (objects) are (typically) not active.
- Threads weave through objects in unstructured ways.
- This is the source of many software problems.

The Buzz

"Multicore architectures will (finally) bring parallel computing into the mainstream. To effectively exploit them, legions of programmers must emphasize concurrency."

The vendor push:

"Please train your computer science students to do extensive multithreaded programming."

Is this a good idea?

My Claim

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

To Examine the Problems With Threads and Objects, 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) {…}

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.

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 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);
        }
    }
}
```

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.

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 <u>outside</u> 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) {…}

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!

If the simplest design patterns yield such problems, what about non-trivial designs?

```
/**
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.

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…

Lee, Berkeley & BDTI 17

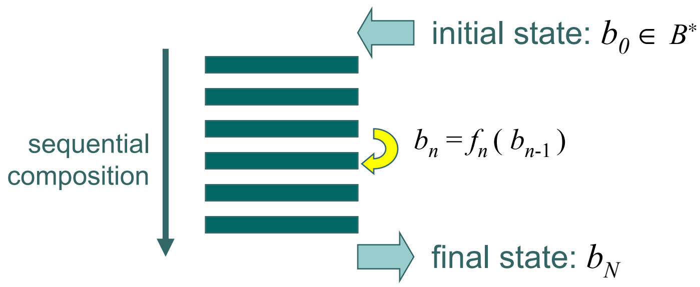
…It is Threads that are Hard!

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…

Lee, Berkeley & BDTI 18

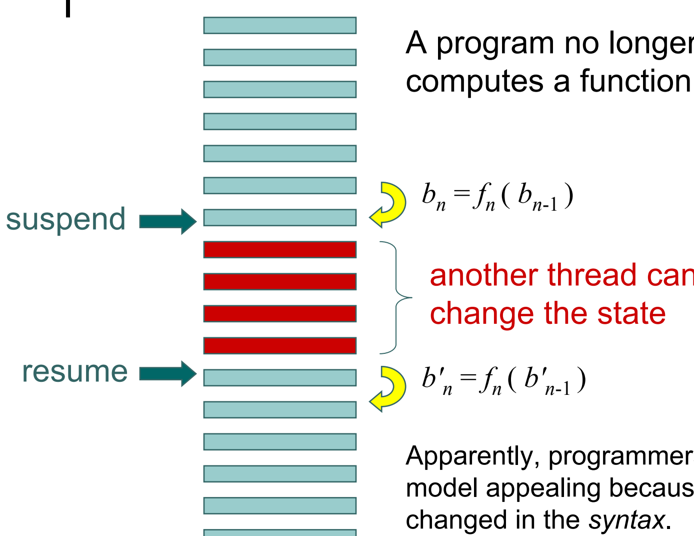
Basic Sequential Computation



Formally, composition of computations is function composition.

When There are Threads, Everything Changes

A program no longer computes a function.



Apparently, programmers find this model appealing because nothing has changed in the *syntax*.

The Following are Only Partial Solutions

- Training programmers to use threads.
- Improve software engineering processes.
- Devote attention to “non-functional” properties.
- Use design patterns.

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

We Can Incrementally Improve Threads

- Object Oriented programming
- Coding rules (Acquire locks in the same order…)
- Libraries (Stapl, Java 5.0, …)
- 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?

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 enterprise is held up by threads

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).

Succinct Solution Statement

Instead of starting with a wildly nondeterministic mechanism and asking the programmer to rein in that nondeterminism, start with a deterministic mechanism and incrementally add nondeterminism where needed.

Under this principle, even the most effective of today's techniques (OO design, transactions, message passing, …) require fundamental rethinking.

Possible Approaches

- Replace C, C++, C#, and Java with
  - Erlang
  - OCaml
  - Haskell
  - Smalltalk

or…

- Provide a new component technology (*actor-oriented design*) overlaid on existing languages.

… this is our approach

Advantages of Our Approach

- It leverages:
  - Language familiarity
  - Component libraries
  - Legacy subsystems
  - Design tools
  - The simplicity of sequential reasoning
- It allows for innovation in
  - Hybrid systems design
  - Distributed system design
  - Service-oriented architectures
- Software is intrinsically concurrent
  - Better use of multicore machines
  - Better use of networked systems
  - Better potential for robust design

Challenges

The technology is immature:

- Commercial actor-oriented systems are domain-specific
- Development tools are limited
- Little language support in C++, C#, Java
- Modularity mechanisms are underdeveloped
- Type systems are primitive
- Compilers (called “code generators”) are underdeveloped
- Formal methods are underdeveloped
- Libraries are underdeveloped

We are addressing these problems.

Object Oriented vs. Actor Oriented

The established: Object-oriented:

class name
data
methods

call

return

What flows through an object is sequential control

Things happen to objects

The alternative: Actor oriented:

Actors make things happen

actor name
data (state)
parameters
ports

What flows through an object is evolving data

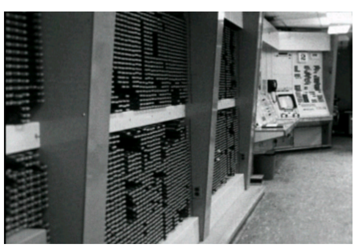
Input data

Output data

The First (?) Actor-Oriented Programming Language

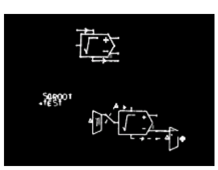
The On-Line Graphical Specification of Computer Procedures

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 language (which had a visual syntax).

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

Examples of Actor-Oriented 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, Linkoping)
- OPNET (discrete events, Opnet Technologies)
- SDL (process networks)
- Occam (rendezvous)
- Simulink (Continuous-time, The MathWorks)
- SPW (synchronous dataflow, Cadence, CoWare)
- …

Most of these are domain specific.

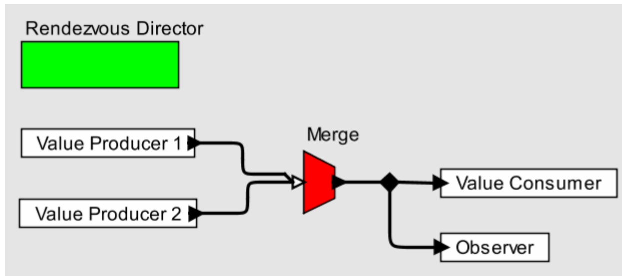
Many of these have visual syntaxes.

The semantics of these differ considerably, with significantly different approaches to 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



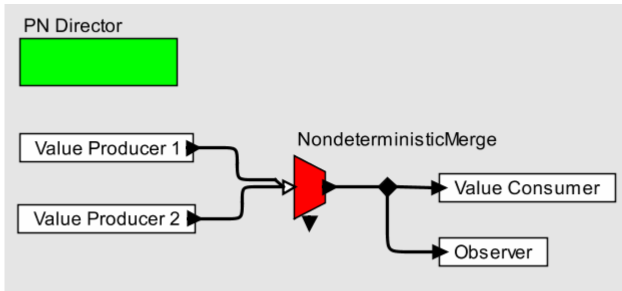
Each actor is a process, communication is via rendezvous, and the Merge explicitly represents nondeterministic multi-way rendezvous.

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)

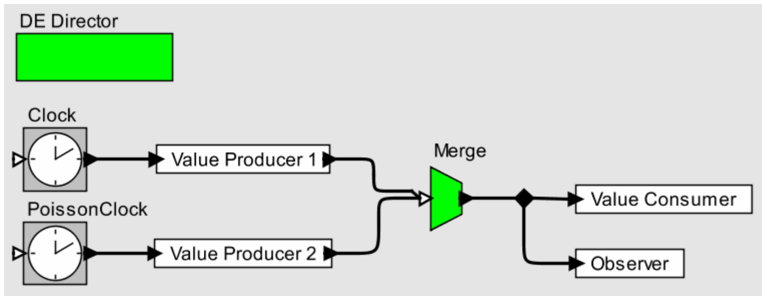


Each actor is a process, communication is via streams, and the NondeterministicMerge explicitly merges streams nondeterministically.

Again a *coordination language* with a *visual syntax*.

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

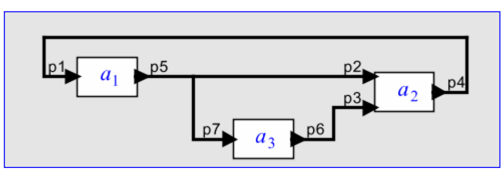
Observer Pattern using an Actor-Oriented Language with Discrete Event Semantics



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

Again a *coordination language* with a *visual syntax*.

Composition Semantics



Each actor is a function:

$$f : (T \to B^*)^m \to (T \to B^*)^n$$

Composition in three forms:

- Cascade connections
- Parallel connections
- Feedback connections

All three are function composition.

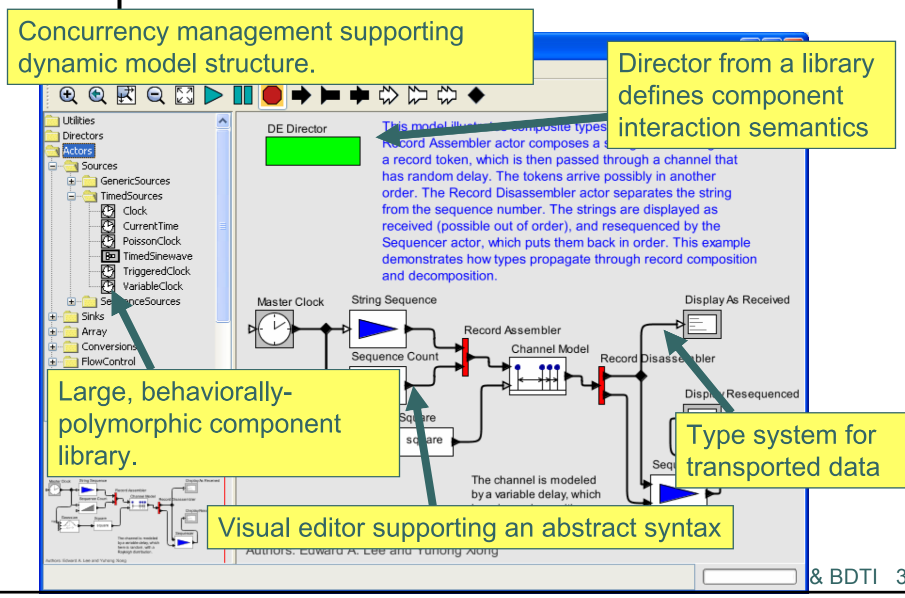
The nontrivial part of this is feedback, but we know how to handle that.

The concurrency model is called the "model of computation" (MoC).

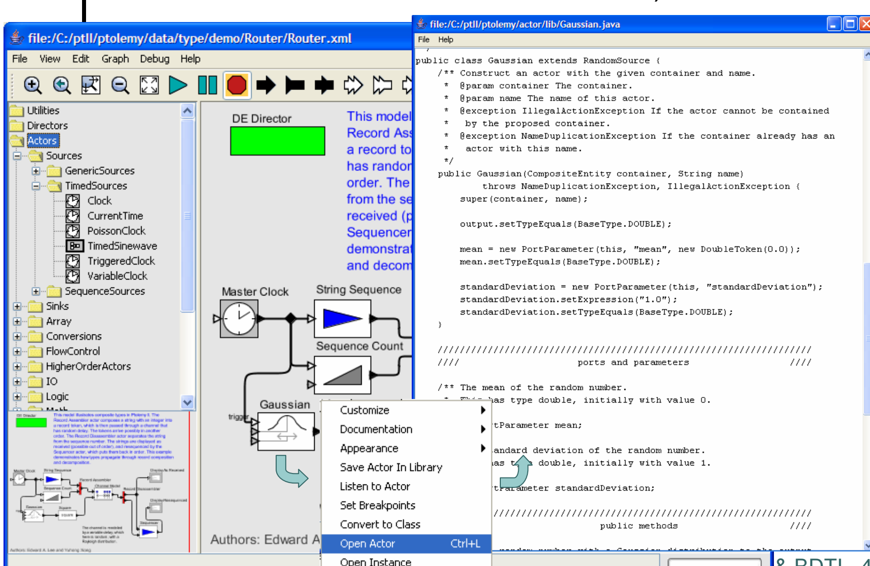
The model of computation determines the formal properties of the set T:

Useful MoCs:
- *Process Networks*
- *Synchronous/Reactive*
- *Time-Triggered*
- *Discrete Events*
- *Dataflow*
- *Rendezvous*
- *Continuous Time*
- *…*

Enter Ptolemy II: Our Laboratory for Experiments with Models of Computation

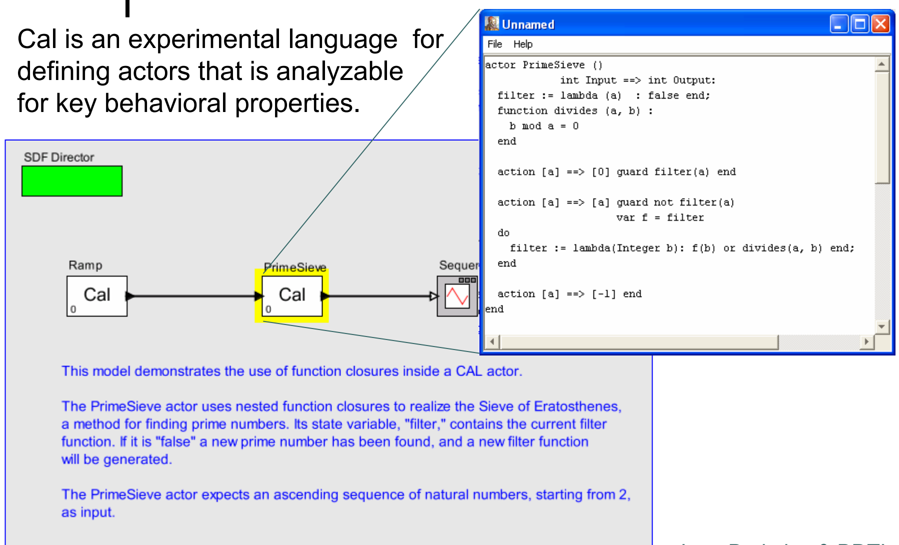


Ptolemy II: Functionality of Components is Given in Standard Java (which can wrap C, C++, Perl, Python, MATLAB, Web services, Grid services, …)

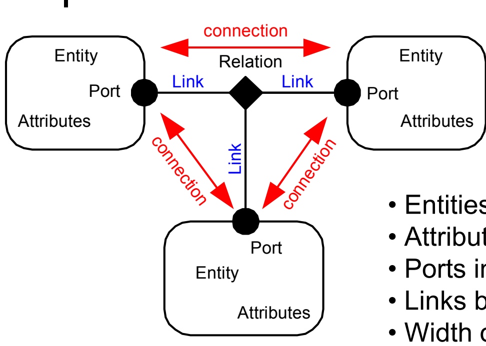


Actors can be defined in other languages. E.g. Python Actors, Cal Actors, MATLAB Actors, …

Cal is an experimental language for defining actors that is analyzable for key behavioral properties.



The Basic Abstract Syntax for Composition



- Entities
- Attributes on entities (parameters)
- Ports in entities
- Links between ports
- Width on links (channels)
- Hierarchy

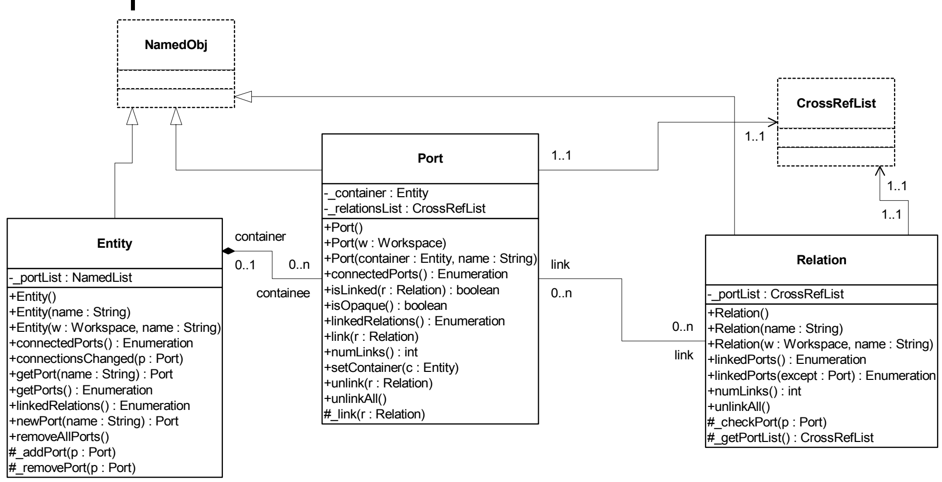
Concrete syntaxes:
- XML
- Visual pictures
- Actor languages (Cal, StreamIT, …)

MoML
XML Schema for this Abstract Syntax

Ptolemy II designs are represented in XML:

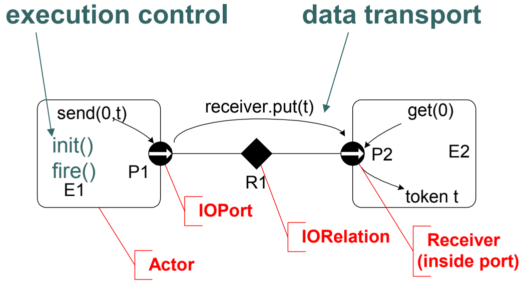
```
...
<entity name="FFT" class="ptolemy.domains.sdf.lib.FFT">
    <property name="order" class="ptolemy.data.expr.Parameter" value="order">
    </property>
    <port name="input" class="ptolemy.domains.sdf.kernel.SDFIOPort">
       ...
    </port>
    ...
</entity>
...
<link port="FFT.input" relation="relation"/>
<link port="AbsoluteValue2.output" relation="relation"/>
...
```


Meta Model: Kernel Classes
Supporting the Abstract Syntax



These get subclassed for specific purposes.

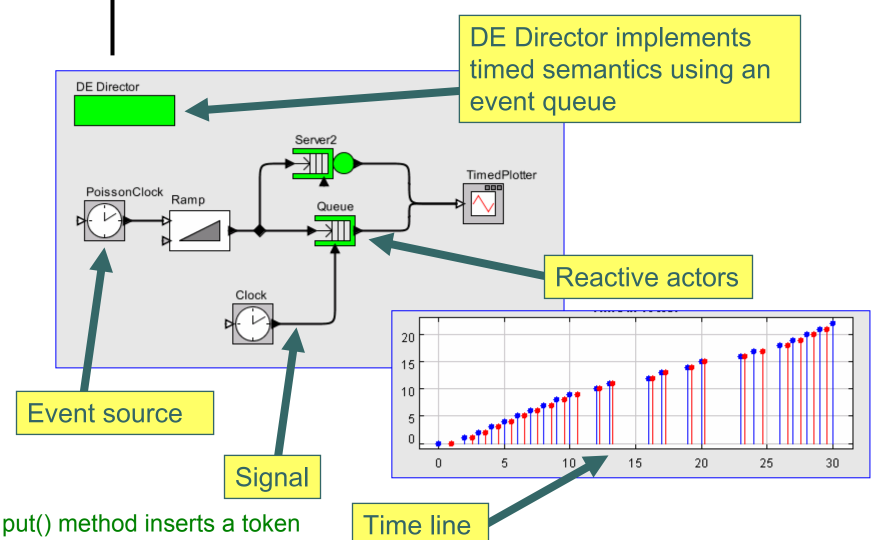
Abstract Semantics (Informally)
of *Actor-Oriented* Models of Computation



Actor-Oriented Models of Computation that we have implemented:

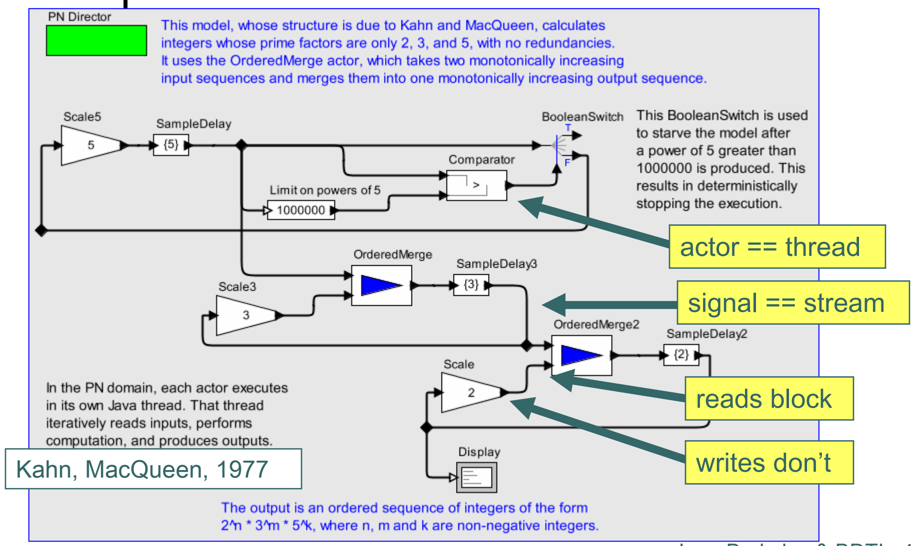
- dataflow (several variants)
- process networks
- distributed process networks
- Click (push/pull)
- continuous-time
- CSP (rendezvous)
- discrete events
- distributed discrete events
- synchronous/reactive
- time-driven (several variants)
- …

Example: Discrete Event Models



put() method inserts a token into the event queue.

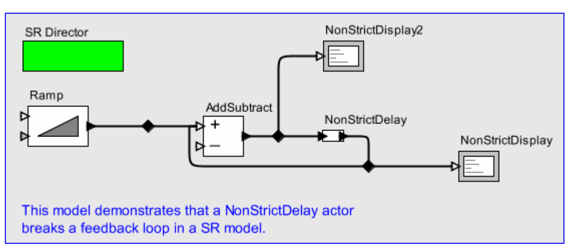
Example: Process Networks (PN) in Ptolemy II

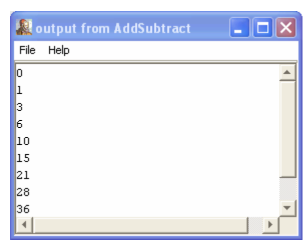


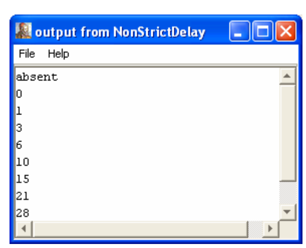
Lee, Berkeley & BDTI 47

Example: Synchronous/Reactive (SR)

At each tick of a global "clock," every signal has a value or is absent.





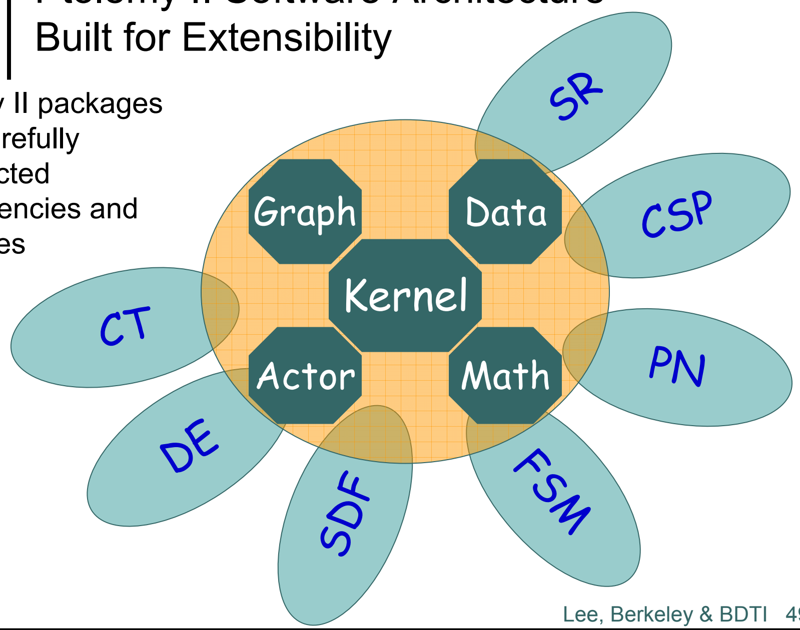


The job of the SR director is to find the value at each tick (it iterates to a fixed point).

Lee, Berkeley & BDTI 48

Ptolemy II Software Architecture Built for Extensibility

Ptolemy II packages have carefully constructed dependencies and interfaces

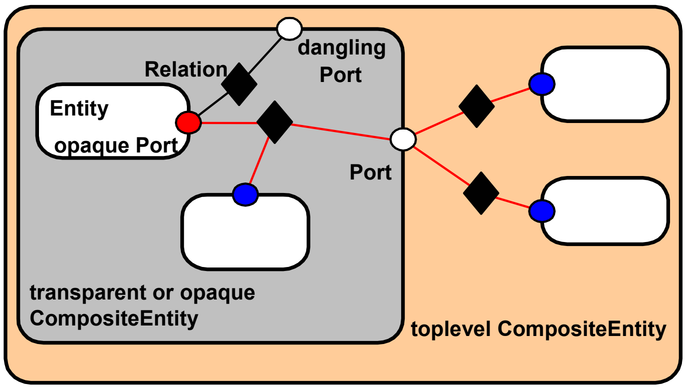


Models of Computation Implemented in Ptolemy II

- CI – Push/pull component interaction
- Click – Push/pull with method invocation
- CSP – concurrent threads with rendezvous
- Continuous – continuous-time modeling with fixed-point semantics
- CT – continuous-time modeling
- DDF – Dynamic dataflow
- DE – discrete-event systems
- DDE – distributed discrete events
- DPN – distributed process networks
- FSM – finite state machines
- DT – discrete time (cycle driven)
- Giotto – synchronous periodic
- GR – 3-D graphics
- PN – process networks
- Rendezvous – extension of CSP
- SDF – synchronous dataflow
- SR – synchronous/reactive
- TM – timed multitasking

Most of these are actor oriented.

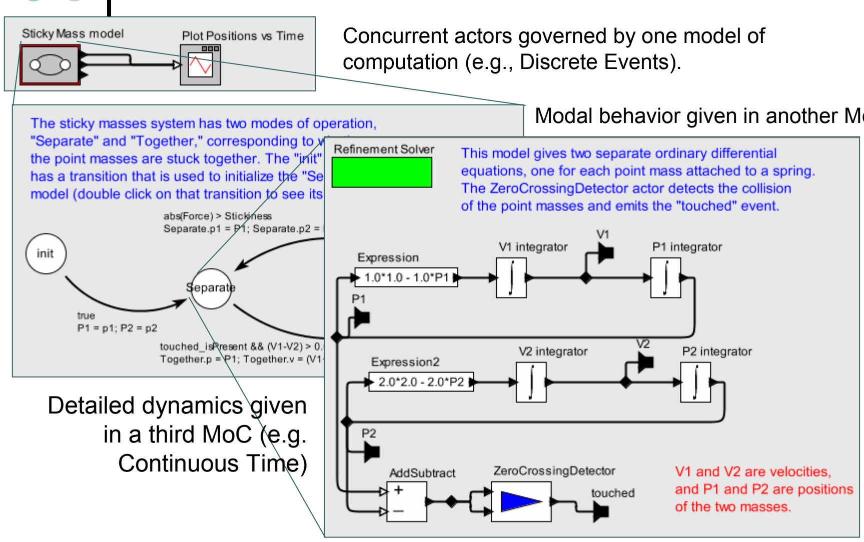
Scalability 101:
Hierarchy - Composite Components



Ptolemy II Hierarchy Supports Heterogeneity

Concurrent actors governed by one model of computation (e.g., Discrete Events).

Modal behavior given in another MoC.

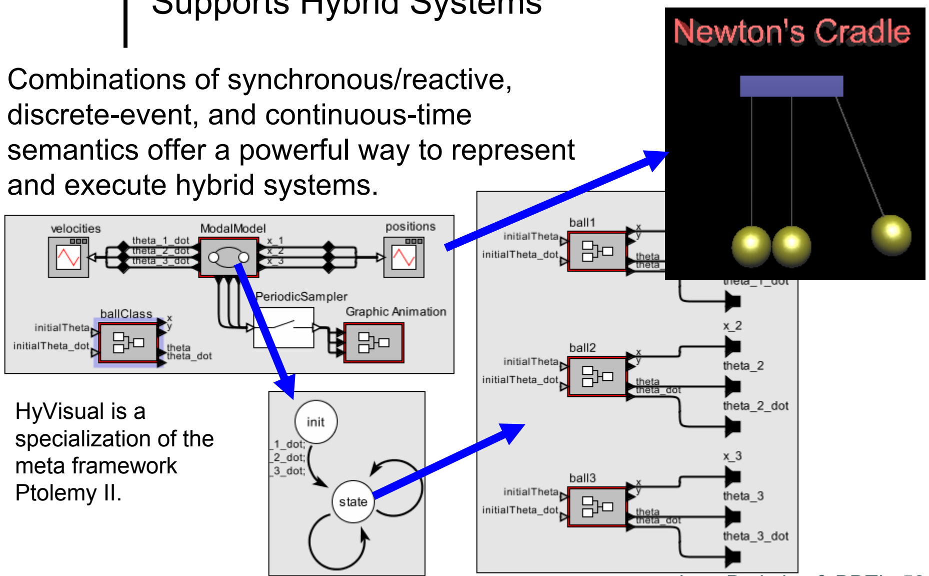


Detailed dynamics given in a third MoC (e.g. Continuous Time)

This requires a composable abstract semantics.

Hierarchical Heterogeneity (HH) Supports Hybrid Systems

Combinations of synchronous/reactive, discrete-event, and continuous-time semantics offer a powerful way to represent and execute hybrid systems.



HyVisual is a specialization of the meta framework Ptolemy II.

Do Not Ignore the 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 get attention.
  - 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.
- Coordination languages can provide capabilities orthogonal to those of established languages.
  - The syntax can be noticeably distinct (as in the diagrams shown before).

Actor-oriented design can be accomplished through *coordination languages* that complement rather than replace existing languages.

The Solution

Actor-oriented component architectures implemented in *coordination languages* that complement rather than replace existing languages.

With good design of these coordination languages, this will deliver understandable concurrency.

See the Ptolemy Project for explorations of several such languages: http://ptolemy.org