Overview of the Ptolemy Project

Edward A. Lee
Robert S. Pepper Distinguished Professor

EECS 249 Guest Lecture

Berkeley, CA
September 8, 2009

Elevator Speech

The Ptolemy project studies modeling, simulation, and design of concurrent, real-time, embedded systems. The focus is on assembly of concurrent components. The key underlying principle in the project is the use of well-defined models of computation that govern the interaction between components. A major problem area being addressed is the use of heterogeneous mixtures of models of computation. A software system called Ptolemy II is being constructed in Java, and serves as the principal laboratory for experimentation.

The Ptolemy Project Demographics, 2009

Sponsors:

- Government
  - National Science Foundation
  - Army Research Office
  - Air Force Research Lab
  - Air Force Office of Scientific Research
- Industry
  - Agilent
  - Bosch
  - HSBC
  - Lockheed-Martin
  - National Instruments
  - Toyota

History:

The project was started in 1990, though its mission and focus has evolved considerably. An open-source, extensible software framework (Ptolemy II) constitutes the principal experimental laboratory.

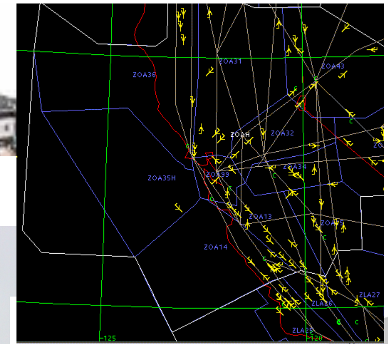
Staffing:

- 1 professor
- 9 graduate students
- 3 postdocs
- 3 full-time staff
- several visitors

Cyber-Physical Systems (CPS):

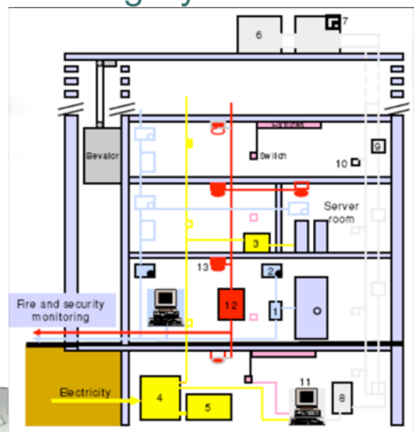
Orchestrating networked computational resources with physical systems

Avionics

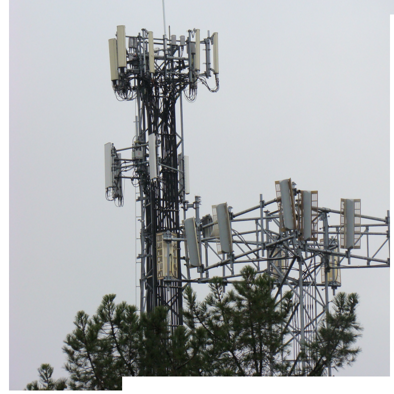


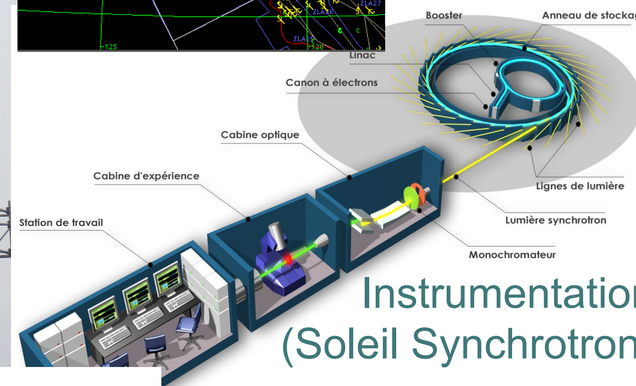
Transportation (Air traffic control at SFO)

Building Systems



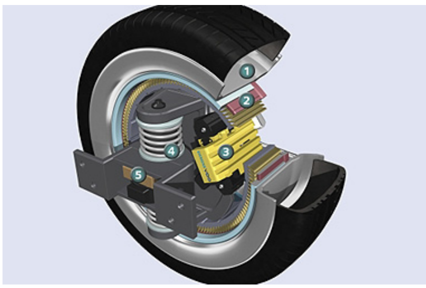
Telecommunications



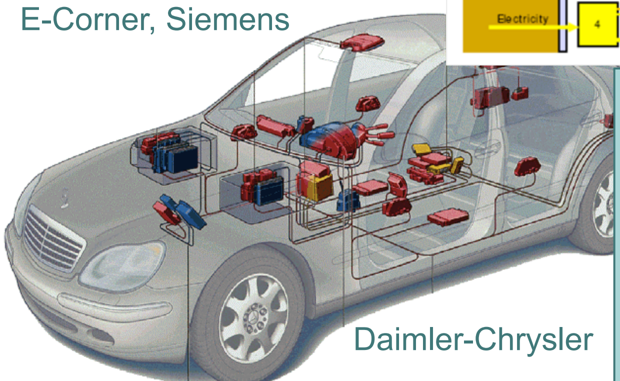


Instrumentation (Soleil Synchrotron)

Automotive



E-Corner, Siemens

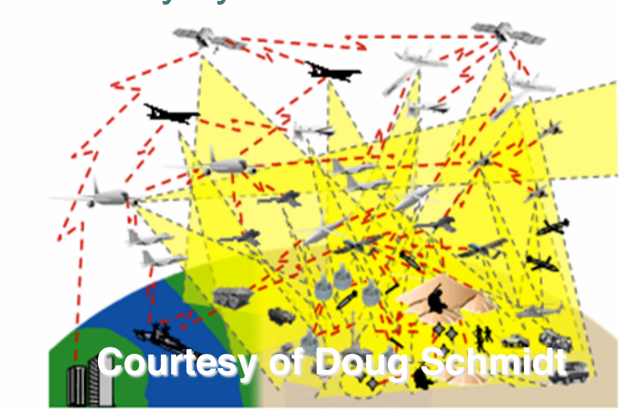


Daimler-Chrysler

Power generation and distribution

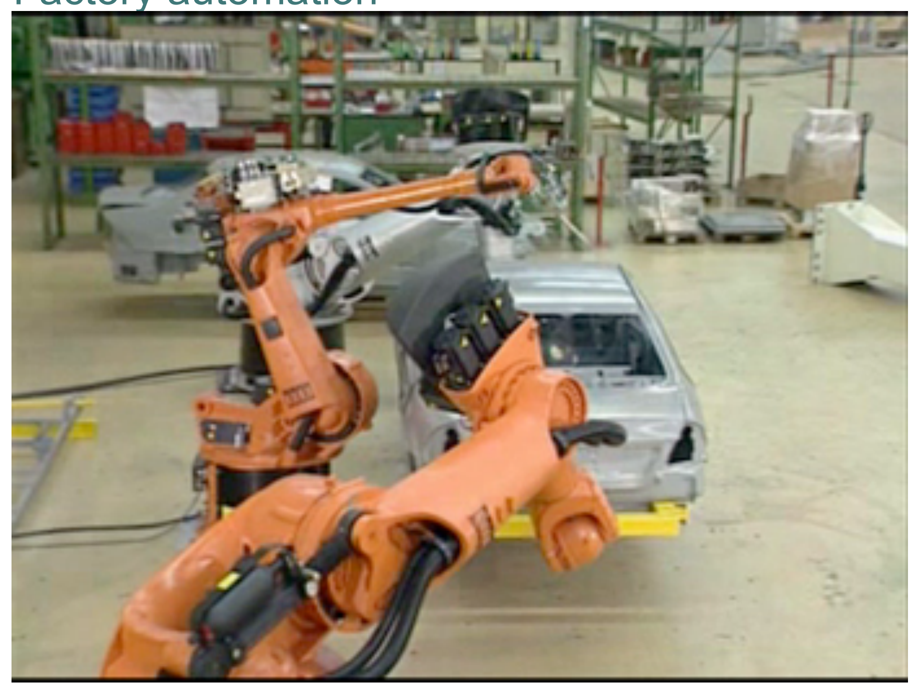
Courtesy of General Electric

Military systems:



Courtesy of Doug Schmidt

Factory automation



Courtesy of Kuka Robotics Corp.

First Challenge on the Cyber Side: Real-Time Software

Correct execution of a program in C, C#, Java, Haskell, etc. has nothing to do with how long it takes to do anything. All our computation and networking abstractions are built on this premise.

Timing of programs is not repeatable, except at very coarse granularity.

Programmers have to step outside the programming abstractions to specify timing behavior.

Second Challenge on the Cyber Side: *Concurrency*

(Needed for real time and multicore)

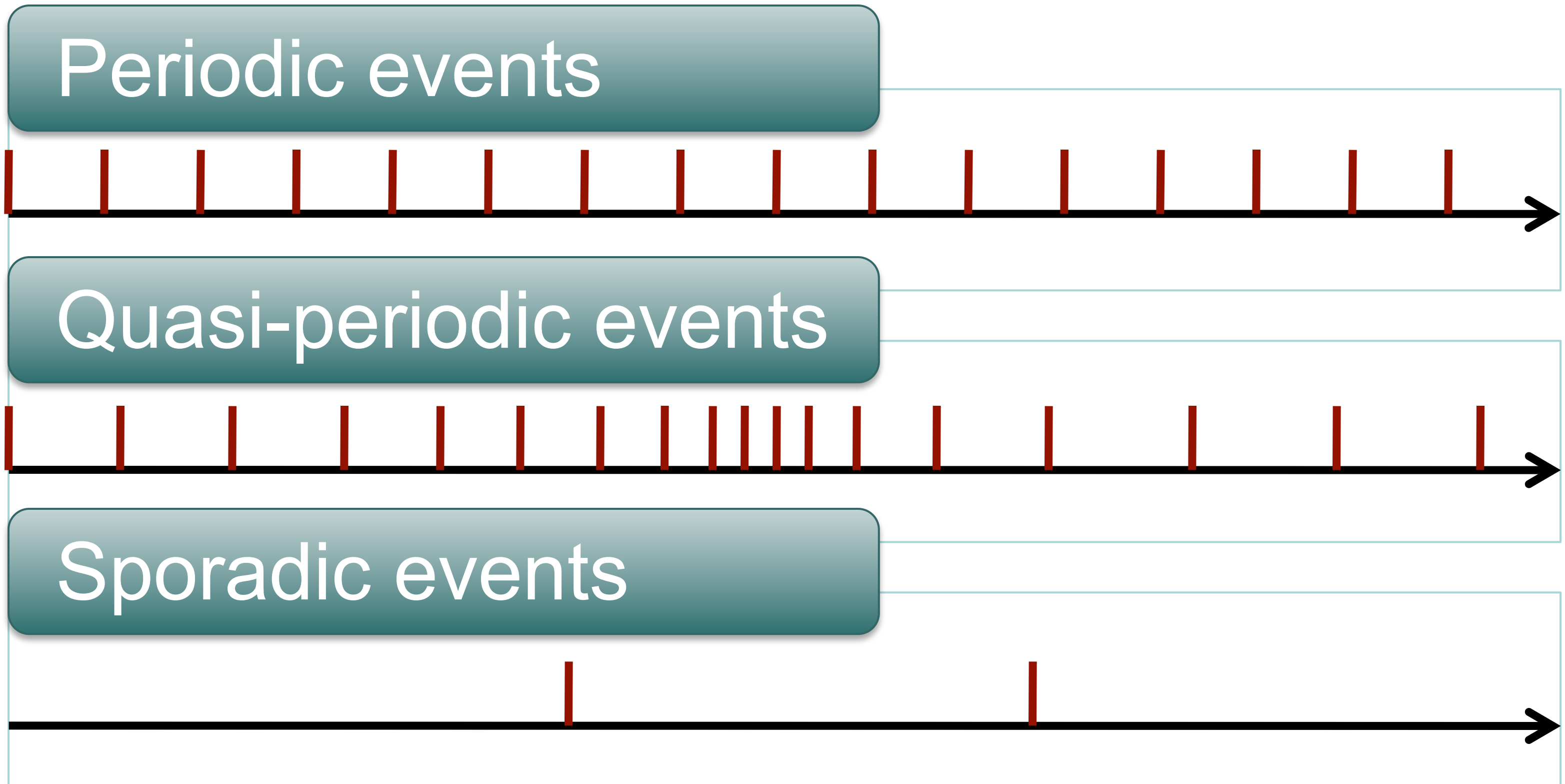
Threads dominate concurrent software.

- *Threads*: Sequential computation with shared memory.
- *Interrupts*: Threads started by the hardware.

Incomprehensible interactions between threads are the sources of many problems:

- Deadlock
- Priority inversion
- Scheduling anomalies
- Timing variability
- Nondeterminism
- Buffer overruns
- System crashes

Consider an Automotive Example



Consider handling this with timers, interrupts, threads, shared memory, priorities, and mutual exclusion.

This is a nightmare!

The Current State of Affairs

We build embedded software on abstractions where time is irrelevant using concurrency models that are incomprehensible.

Just think what we could do with the right abstractions!

The Answer

- Disciplined concurrent and timed models of computation (MoCs).

Today I will focus on explaining how we use Ptolemy II to study concurrent and timed MoCs.

Kahn Process Networks (PN)
A Concurrent Model of Computation (MoC)

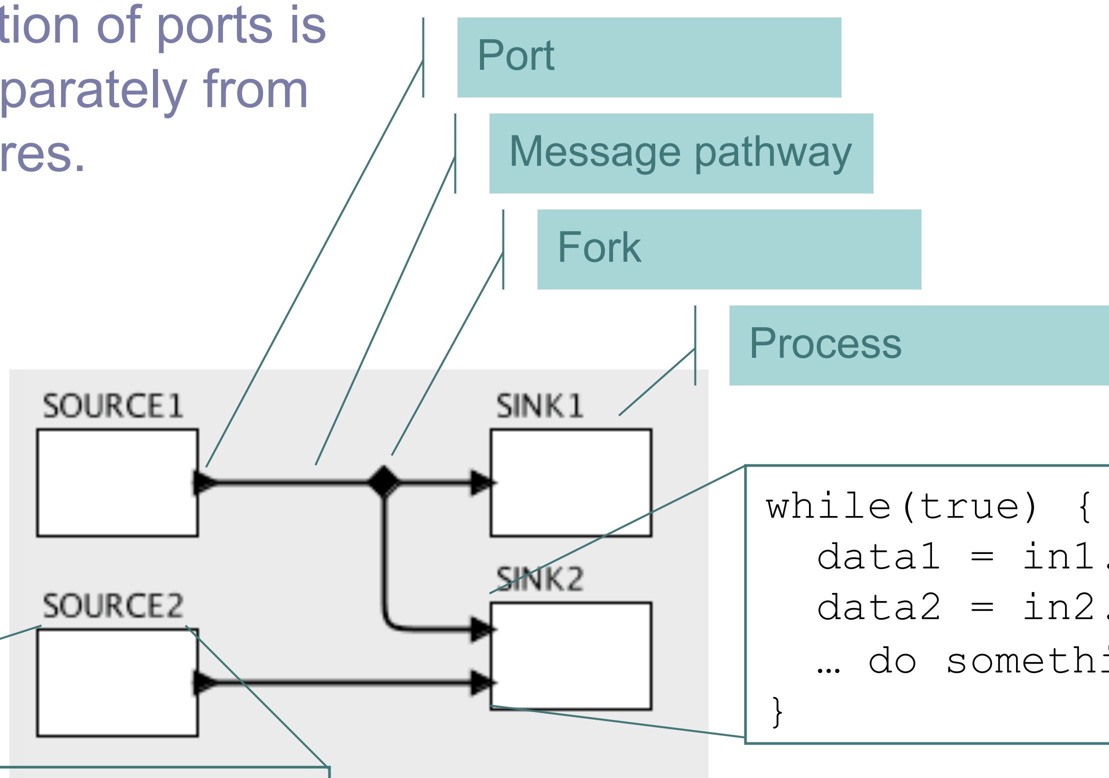
- A set of components called *actors.*
- Each representing a sequential procedure.
- Where steps in these procedures receive or send messages to other actors (or perform local operations).
- Messages are communicated asynchronously with unbounded buffers.
- A procedure can always send a message. It does not need to wait for the recipient to be ready to receive.
- Messages are delivered reliably and in order.
- When a procedure attempts to receive a message, that attempt blocks the procedure until a message is available.

Coarse History

- Semantics given by Gilles Kahn in 1974.
  - Fixed points of continuous and monotonic functions
- More limited form given by Kahn and MacQueen in 1977.
  - Blocking reads and nonblocking writes.
- Generalizations to nondeterministic systems
  - Kosinski [1978], Stark [1980s], …
- Bounded memory execution given by Parks in 1995.
  - Solves an undecidable problem.
- Debate over validity of this policy, Geilen and Basten 2003.
  - Relationship between denotational and operational semantics.
- Many related models intertwined.
  - Actors (Hewitt, Agha), CSP (Hoare), CCS (Milner), Interaction (Wegner), Streams (Broy, …), Dataflow (Dennis, Arvind, …)...

Syntax

- Processes communicate via *ports*.
- Ports are connected to one another, indicating message pathways.
- Interconnection of ports is specified separately from the procedures.




```
while(true) {
  data1 = in1.get();
  data2 = in2.get();
  … do something with it …
}
```

```
while(true) {
  data = …
  outputPort.send(data);
}
```

Discussion: What should a fork do?

Properties of PN (Two Big Topics)

- Assuming “well-behaved” actors, a PN network is determinate in that the sequence of tokens on each arc is independent of the thread scheduling strategy.
  - Making this statement precise, however, is nontrivial. See fixed-point semantics of previous lecture.
- PN is Turing complete.
  - Given only boolean tokens, memoryless functional actors, Switch, Select, and initial tokens, one can implement a universal Turing machine.
  - Whether a PN network deadlocks is undecidable.
  - Whether buffers grow without bound is undecidable.

Dataflow

Dataflow models are similar to PN models except that actor behavior is given in terms of discrete “firings” rather than processes. A firing occurs in response to inputs.

A few variants of dataflow MoCs

- Computation graphs [Karp and Miller, 1966]
- Static 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]
- 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]
- …

The Problem

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

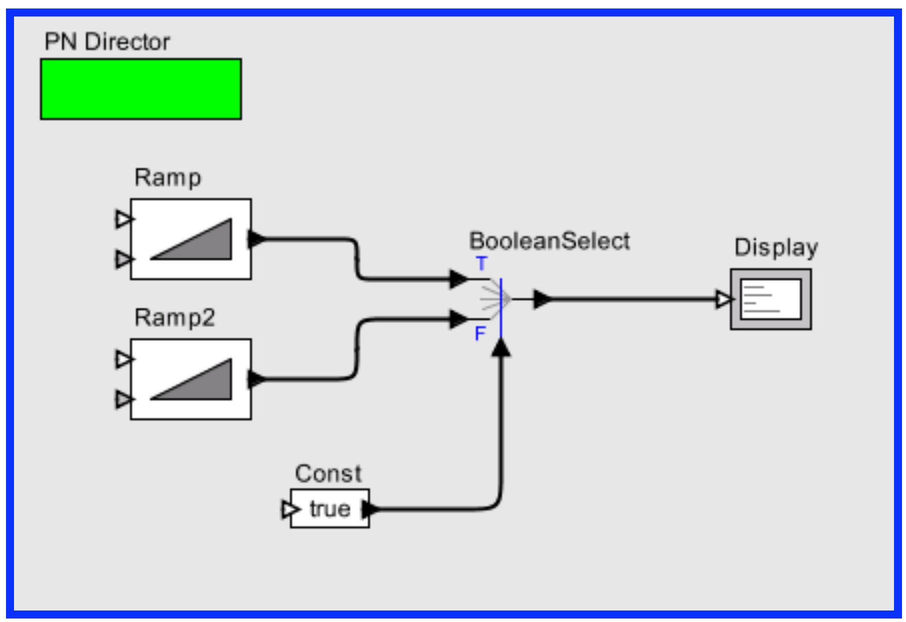
Few programmers will get it right…

Some Subtleties

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

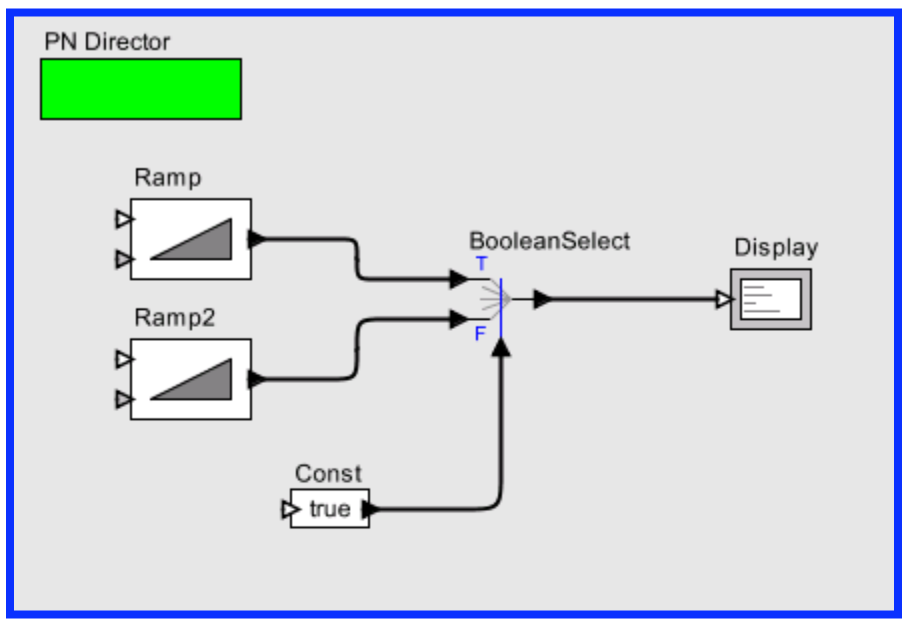
Question 1:
Is “Fair” Scheduling a Good Idea?

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



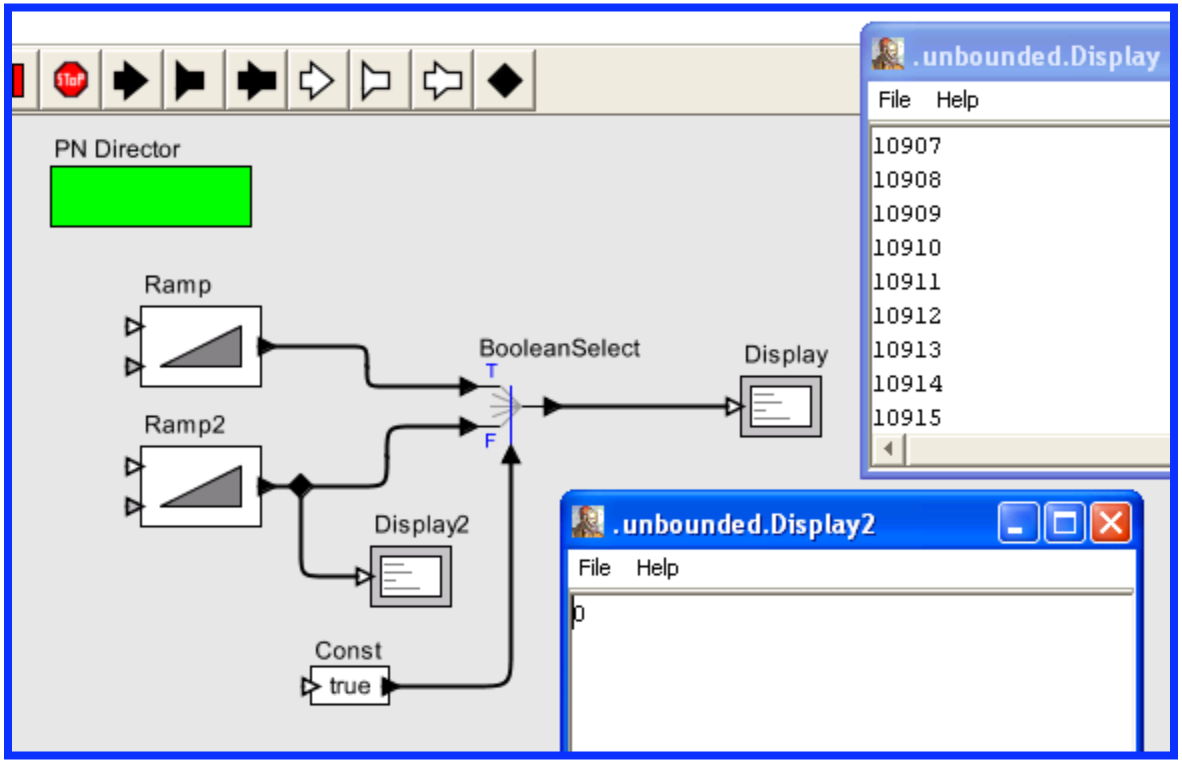
Question 2:
Is “Data-Driven” Execution a Good Idea?

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



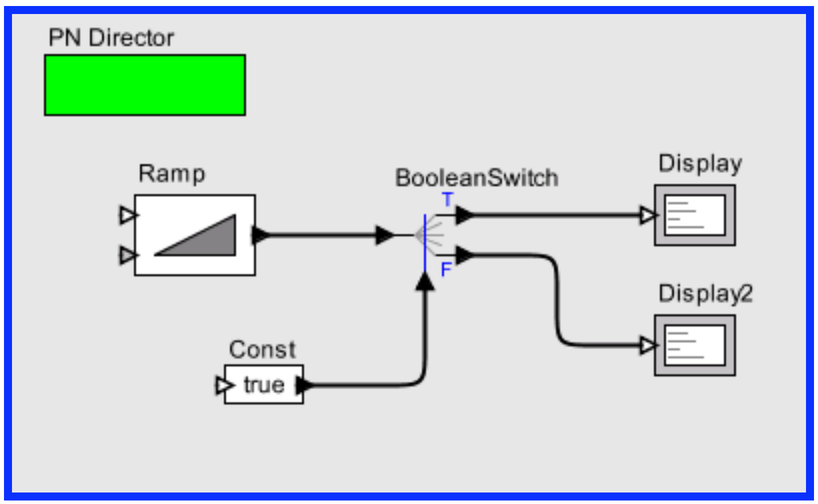
Question 3:
When are Outputs Required?

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

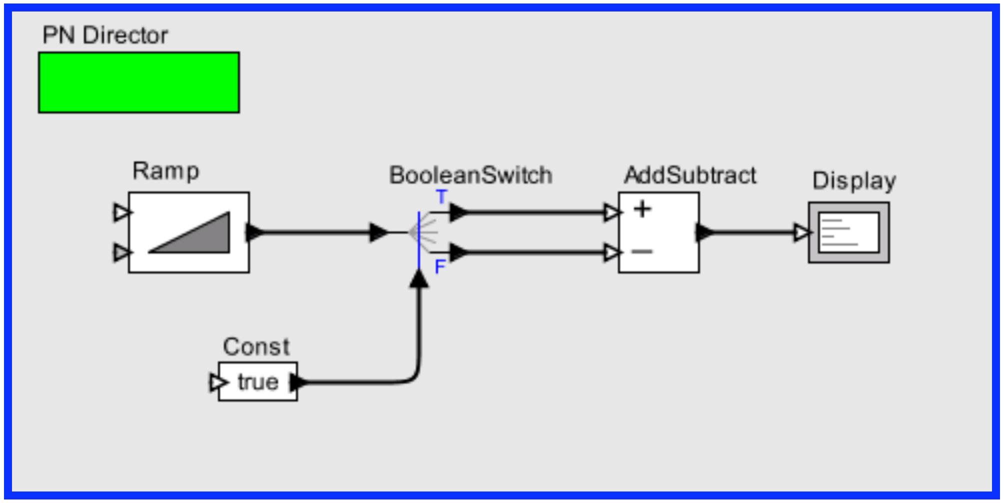


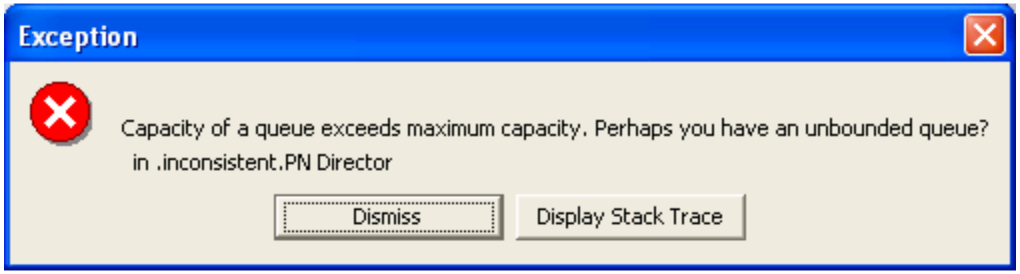
Question 4: Is "Demand-Driven" Execution a Good Idea?

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

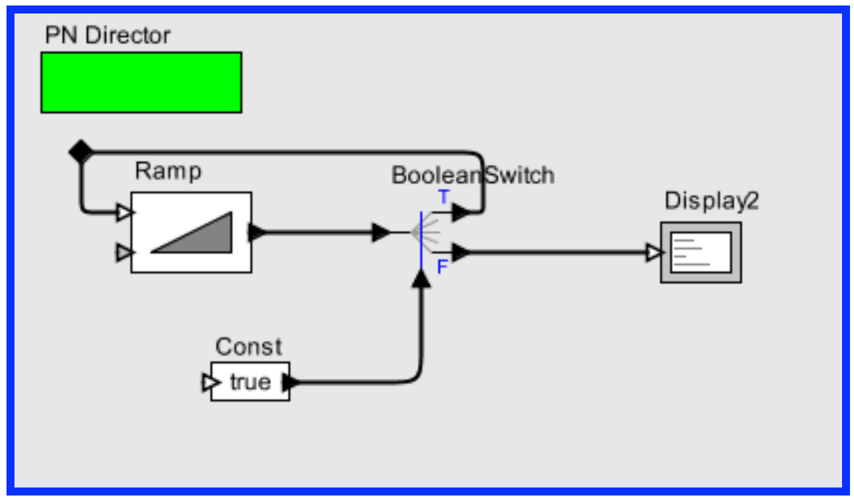


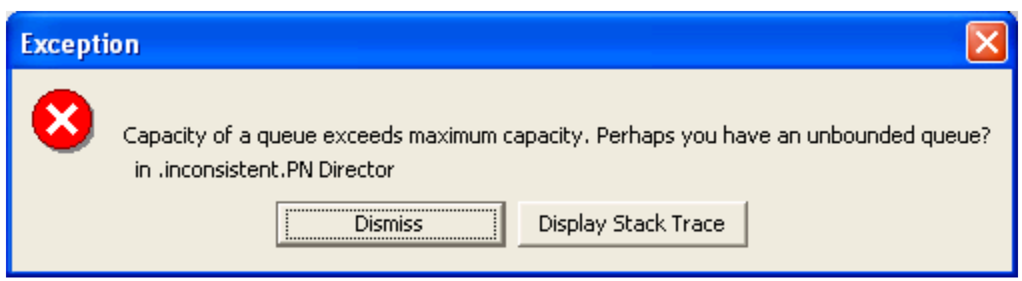
Question 5: What is the "Correct" Execution of This Program?





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



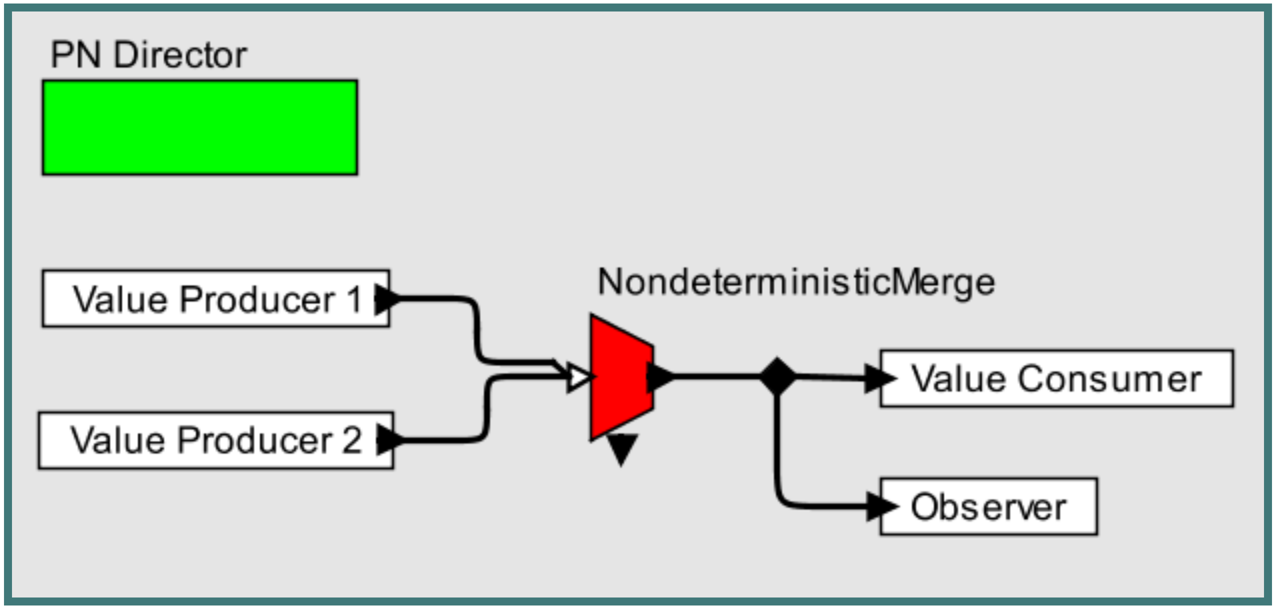


Naïve Schedulers Fail

- 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?

Question 7: How to support nondeterminism?



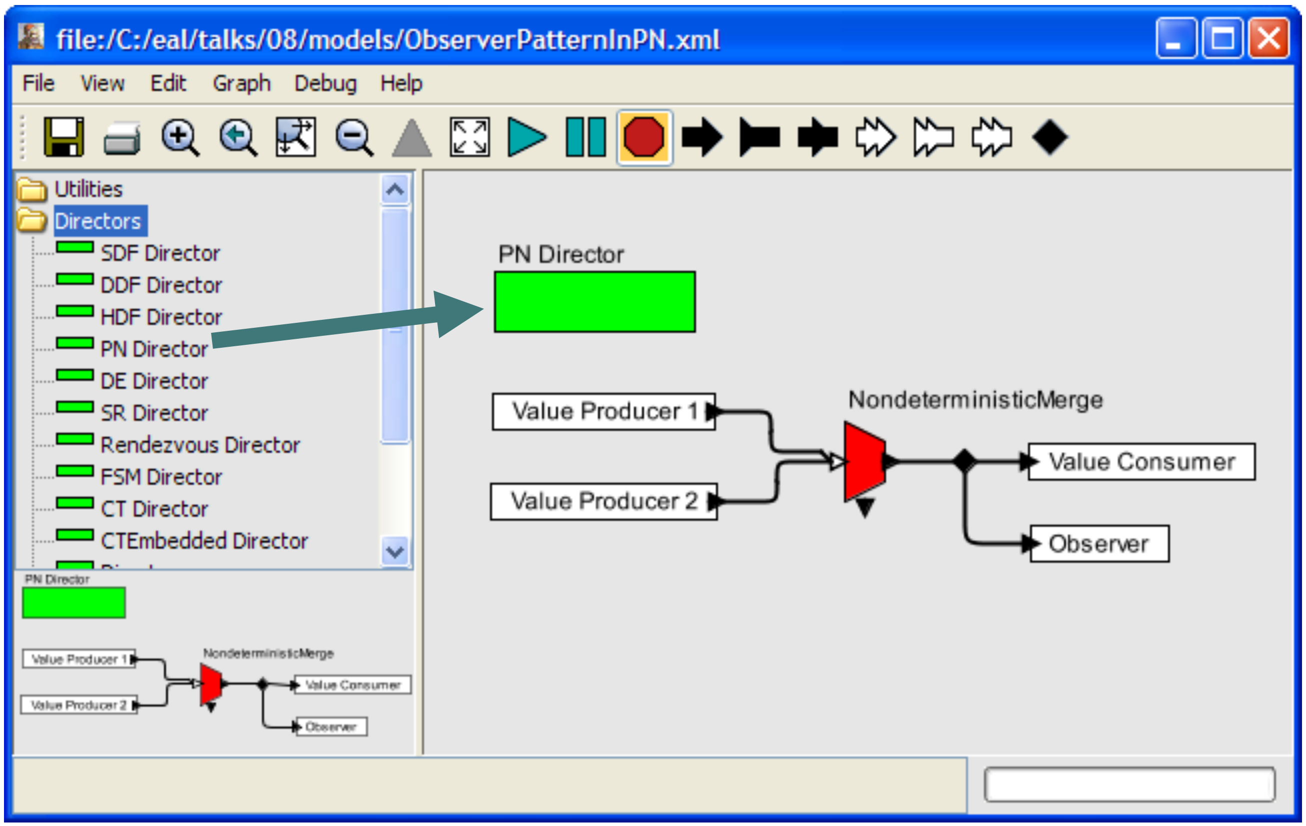
Merging of streams is needed for some applications. Does this require fairness? What does fairness mean?

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.

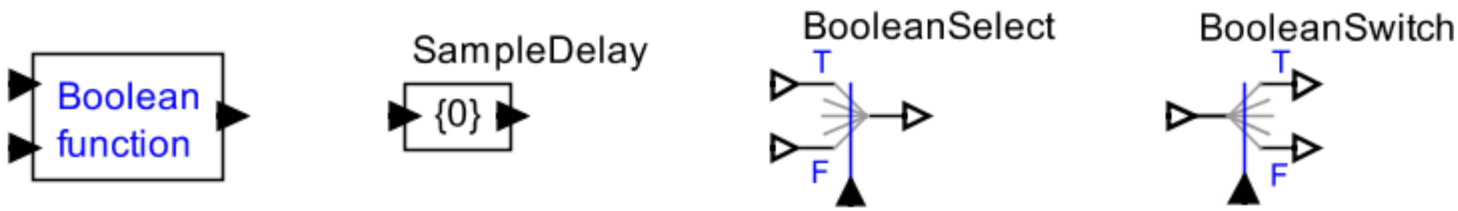
Define a ***useful execution*** to be a correct execution that satisfies the following criteria:

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).

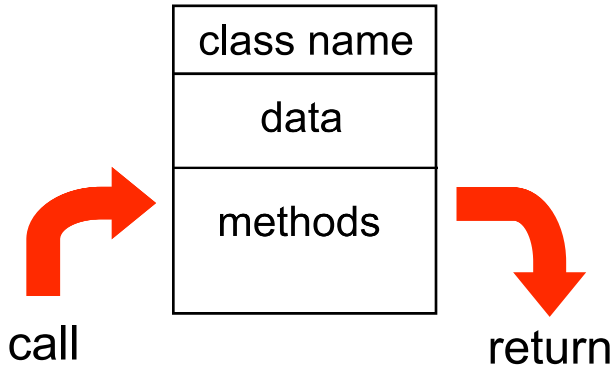
There are many more subtleties!
We need disciplined concurrent models of computation, not arbitrarily flexible libraries.

Some principles:

- 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.

Our Premise: Software Components are Actors rather than Objects

The established: Object-oriented:

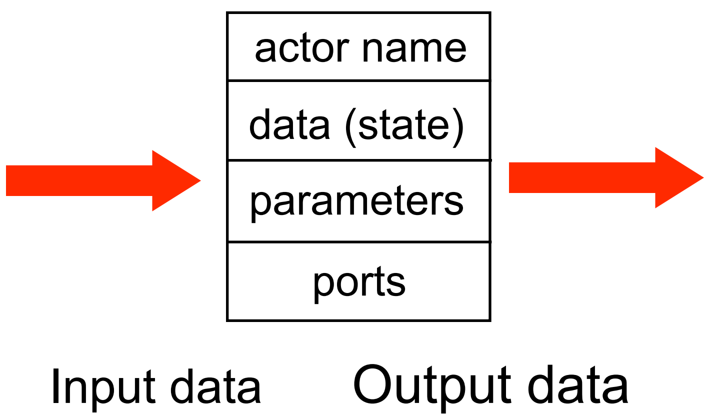


What flows through an object is sequential control

Things happen to objects

The alternative: Actor oriented:

Actors make things happen

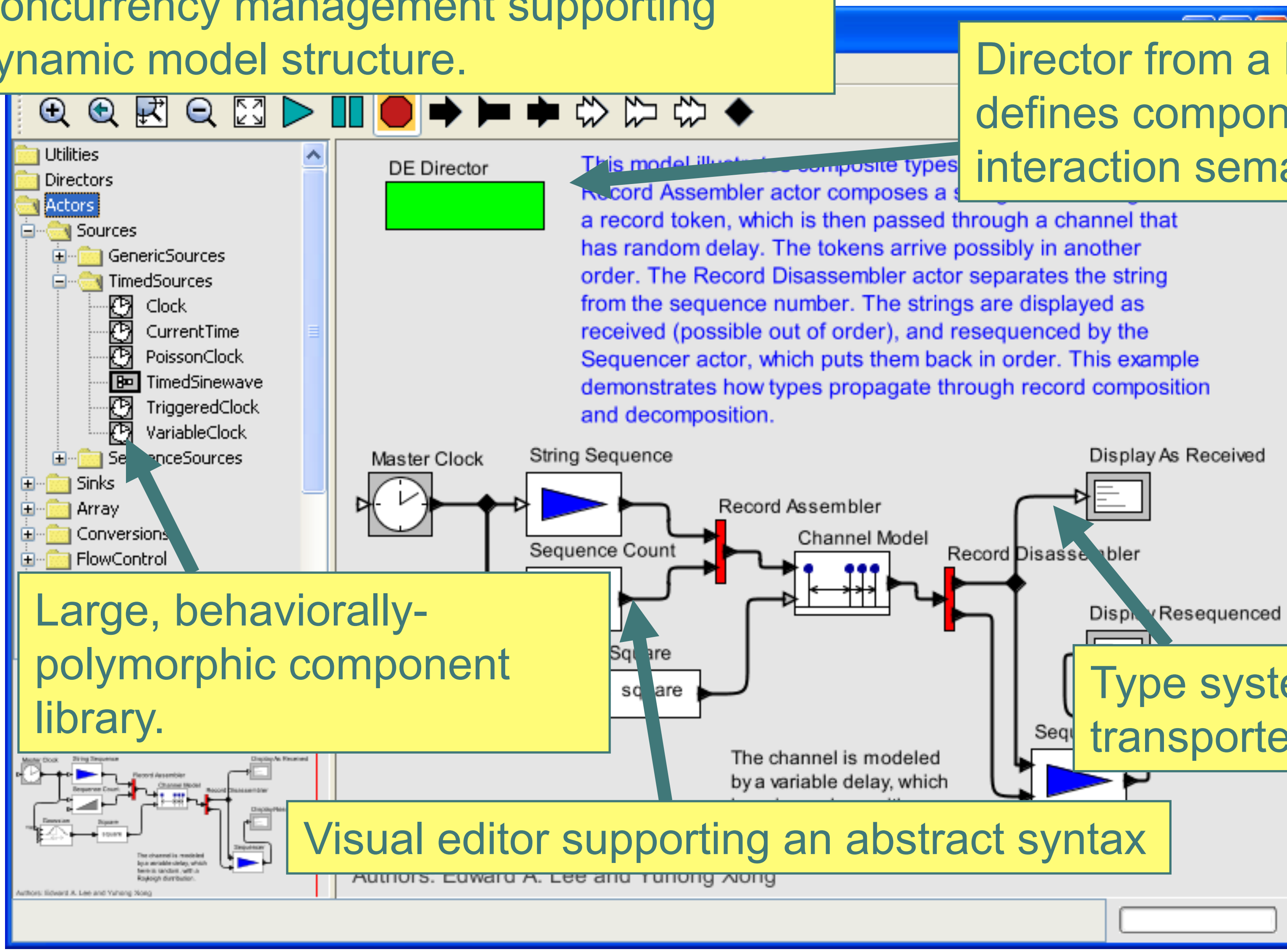


What flows through an object is evolving data

Ptolemy II: Our Laboratory for Experiments with Actor-Oriented Design

Concurrency management supporting dynamic model structure.

Director from a library defines component interaction semantics

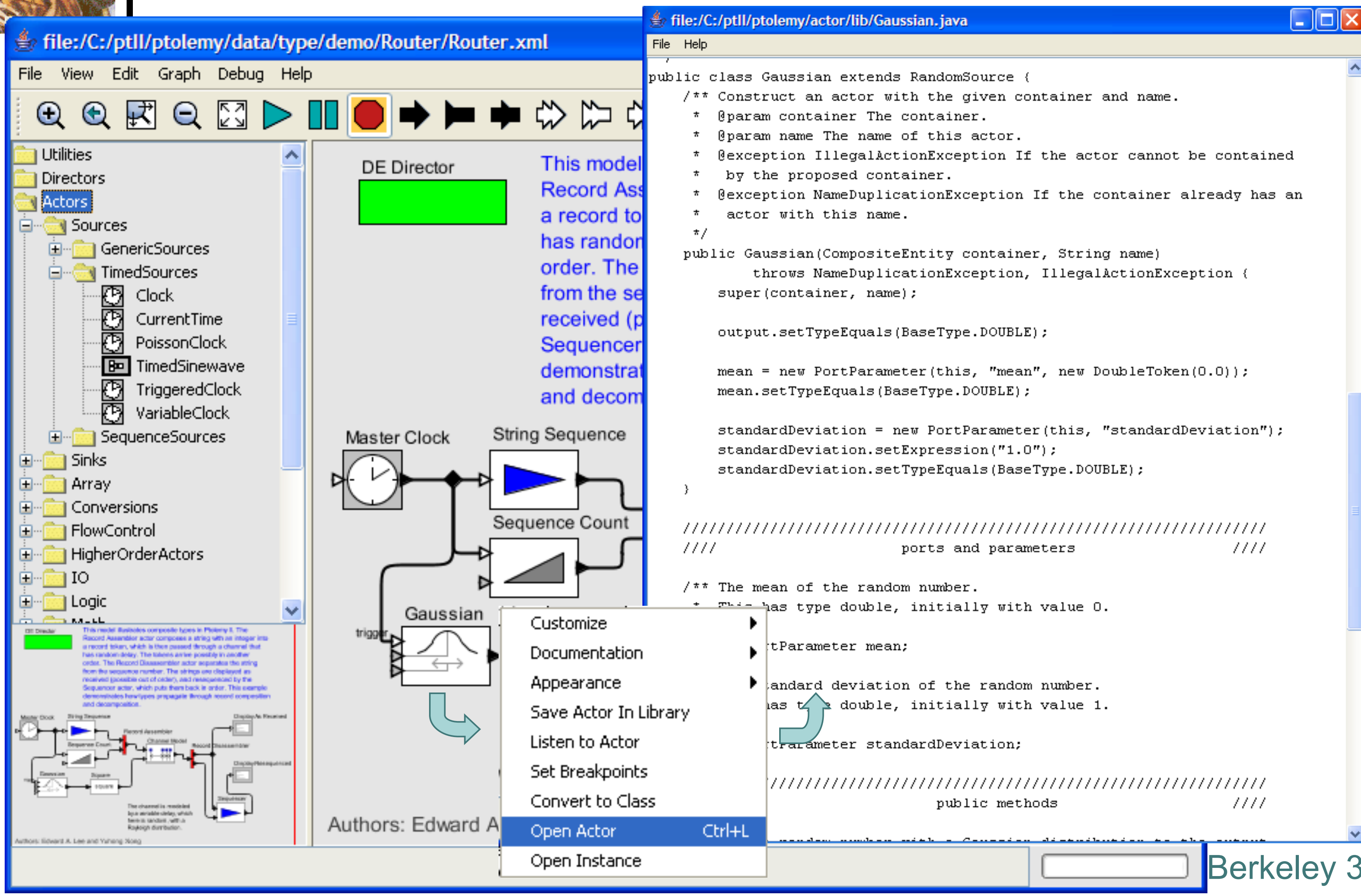


Large, behaviorally-polymorphic component library.

Type system for transported data

Visual editor supporting an abstract syntax

Approach: Concurrent Composition of Software Components, which are themselves designed with Conventional Languages



Our Laboratory Infrastructure

If you want to experimental work in biology, physics, or chemistry, you don't want to start from scratch with a empty room for your lab.

Leverage the work of others!

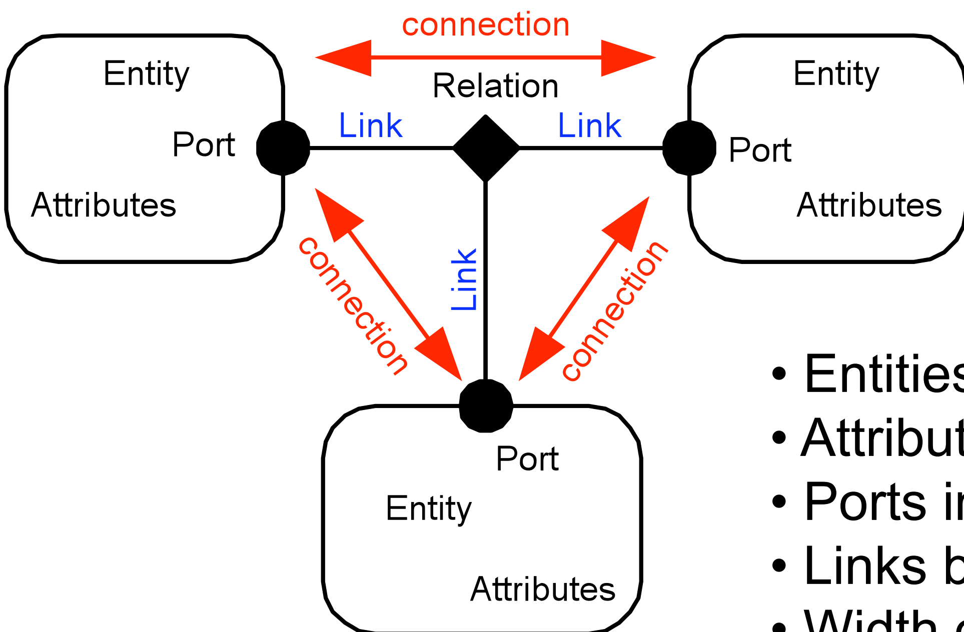
Ohloh (a branch of SourceForge) analysis says that Ptolemy II has 1.8 M lines of code, an estimated effort of 517 person years, worth $28.4 million. (9/7/09)

https://www.ohloh.net/p/12005

Separable Tool Architecture

- Abstract Syntax
- Concrete Syntax
- Abstract Semantics
- Concrete Semantics

The Basic Abstract Syntax for Composition

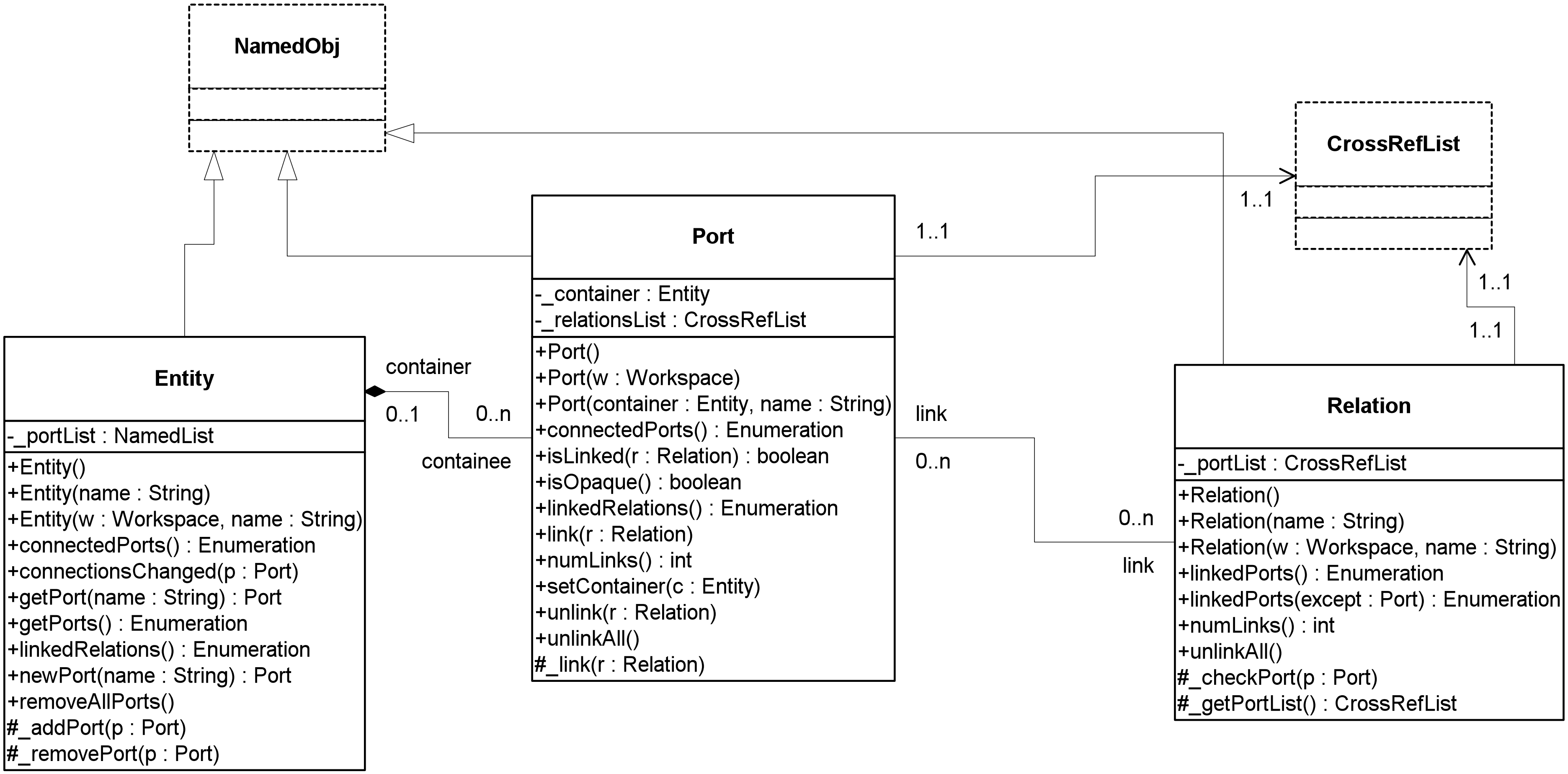


- Entities
- Attributes on entities (parameters)
- Ports in entities
- Links between ports
- Width on links (channels)
- Hierarchy

Concrete syntaxes:

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

Meta Model: Kernel Classes Supporting the Abstract Syntax



These get subclassed for specific purposes.

Separable Tool Archictecture

- Abstract Syntax
- Concrete Syntax
- Abstract Semantics
- Concrete Semantics

MoML
XML Schema for this Abstract Syntax

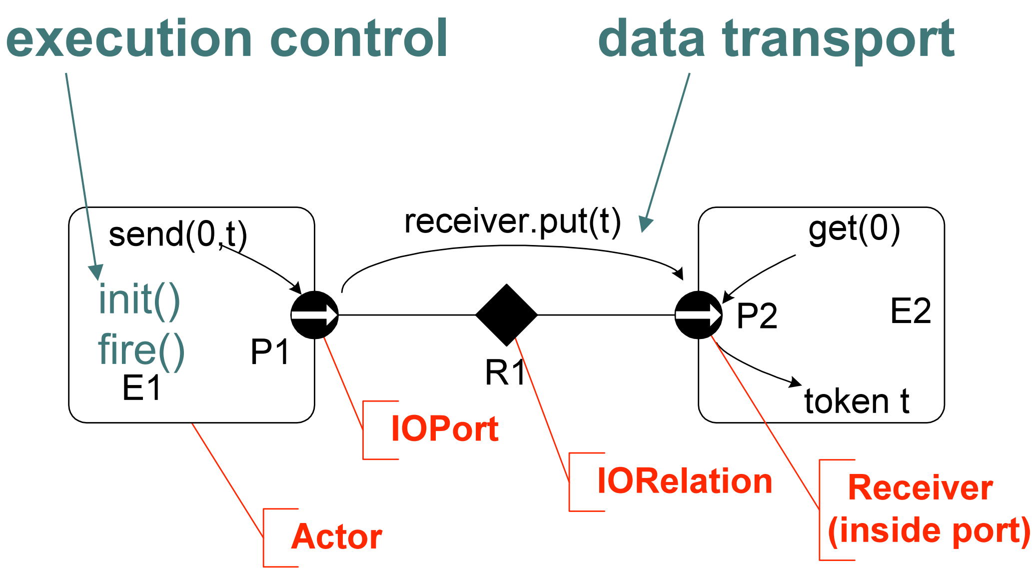
Ptolemy II designs are represented in XML:

```
...
<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"/>
...
```

Separable Tool Archictecture

- Abstract Syntax
- Concrete Syntax
- Abstract Semantics
- Concrete Semantics

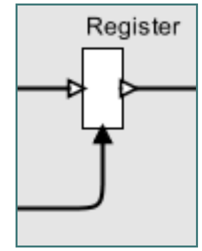
Abstract Semantics (Informally)
of *Actor-Oriented* Models of Computation



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)
- …

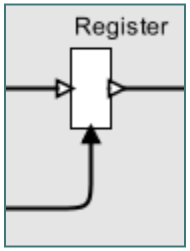
How Does This Work?
Execution of Ptolemy II Actors



Flow of control:

- Initialization
- Execution
- Finalization

How Does This Work?
Execution of Ptolemy II Actors

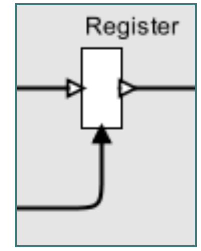


Flow of control:

- Initialization
- Execution
- Finalization

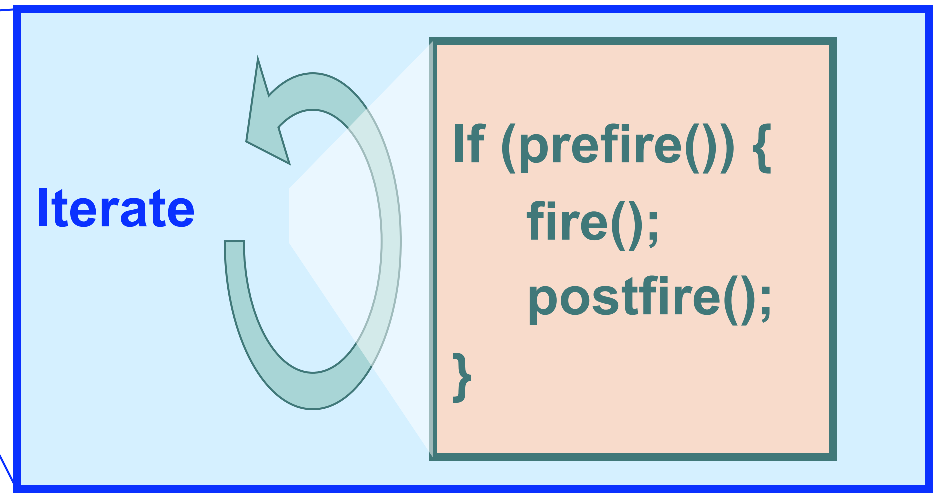
E.g., in DE: Post tags on the event queue corresponding to any initial events the actor wants to produce.

How Does This Work? Execution of Ptolemy II Actors



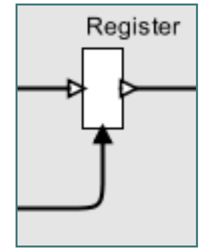
Flow of control:

- Initialization
- Execution
- Finalization



Only the postfire() method should change the state of the actor.

How Does This Work?
Execution of Ptolemy II Actors



Flow of control:

- Initialization
- Execution
- Finalization

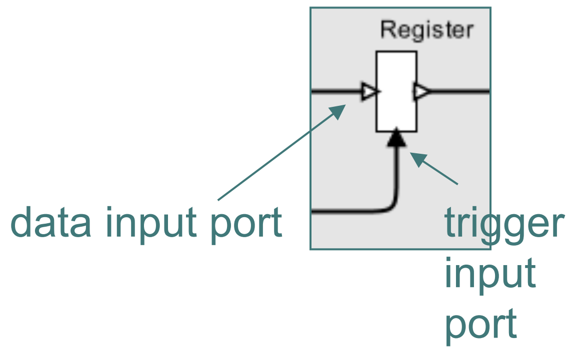
Definition of the Register Actor (Sketch)

```
class Register extends TypedAtomicActor {
  private Object state;
  boolean prefire() {
    if (trigger is known) { return true; }
  }
  void fire() {
    if (trigger is present) {
      send state to output;
    } else {
      assert output is absent;
    }
  }
  void postfire() {
    if (trigger is present) {
      state = value read from data input;
    }
  }
```

Can the actor fire? (prefire)

React to trigger input. (fire)

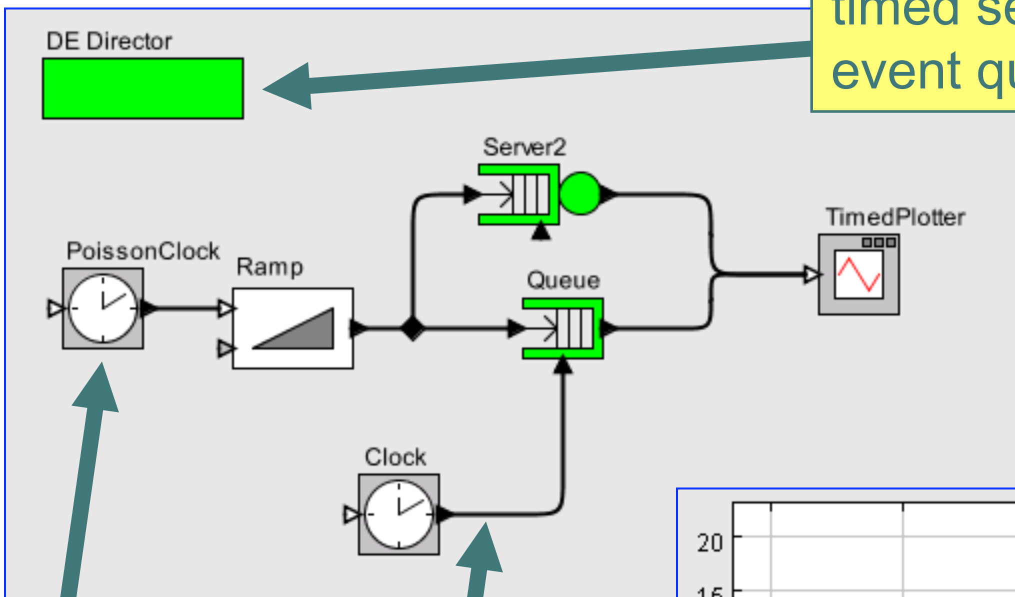
Read the data input and update the state. (postfire)



Separable Tool Archictecture

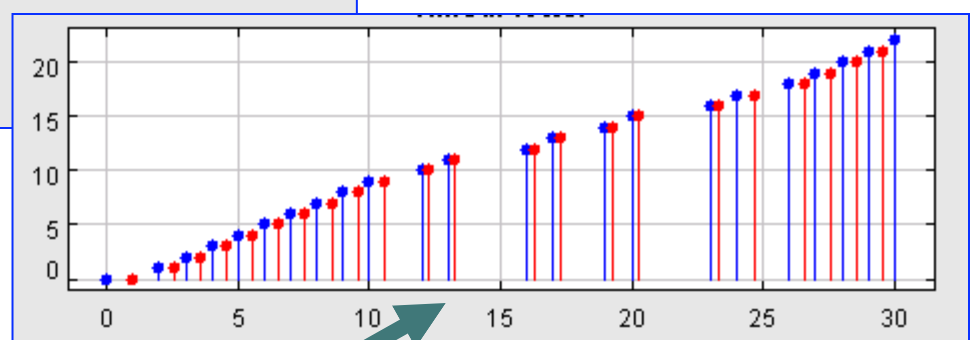
- Abstract Syntax
- Concrete Syntax
- Abstract Semantics
- Concrete Semantics

Concrete Semantics Example 1: Discrete Event (DE) Model of Computation (MoC)



DE Director implements timed semantics using an event queue

In DE, actors send time-stamped events to one another, and events are processed in chronological order.



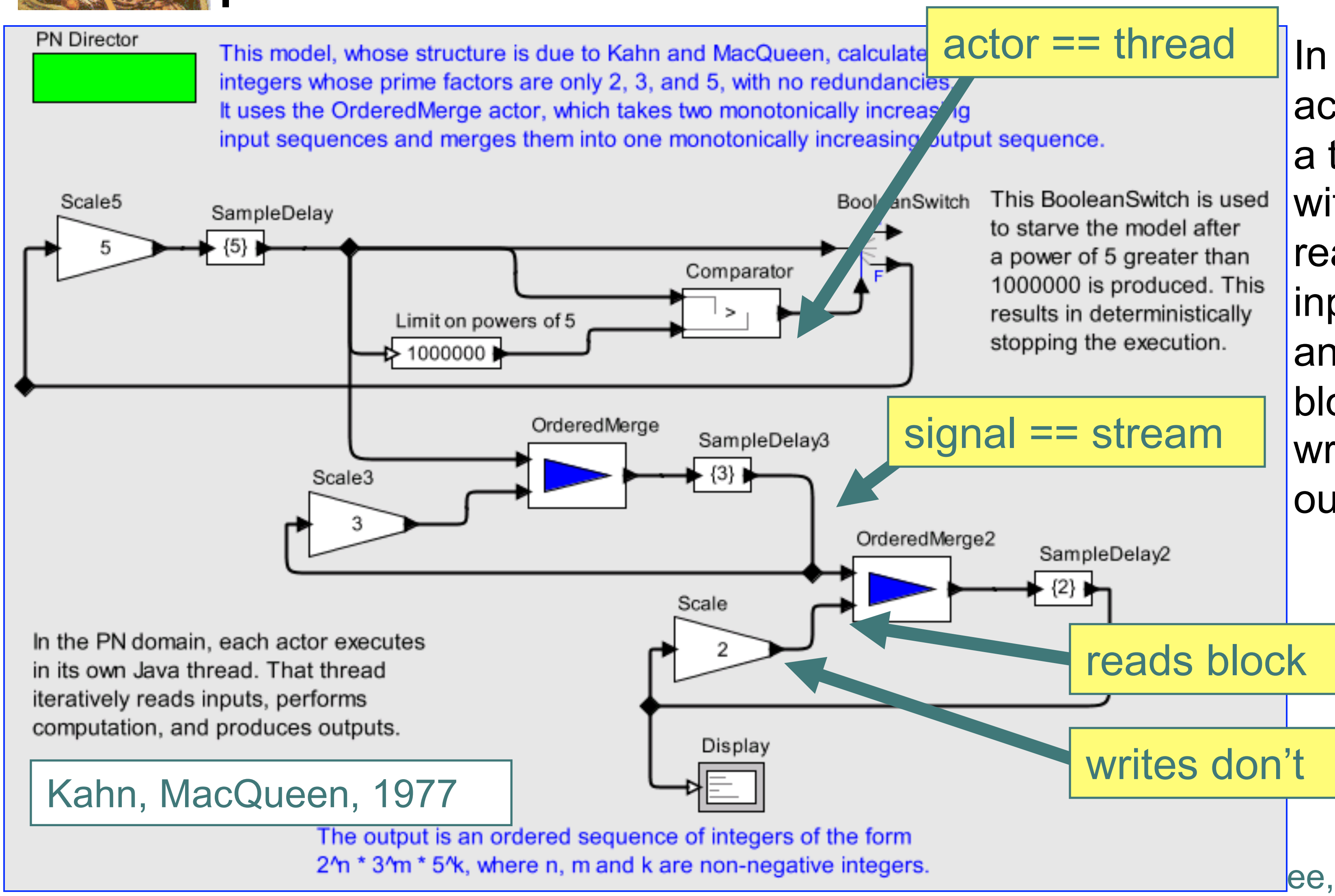
Event source

Signal

Time line

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

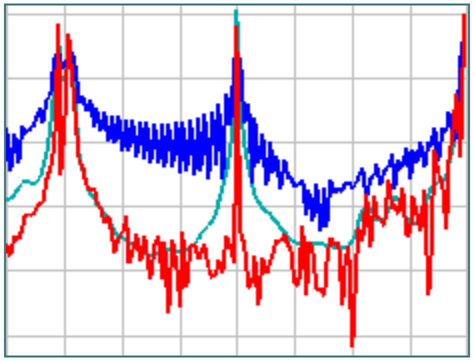
Example 2: Kahn Process Networks (PN) Model of Computation (MoC)

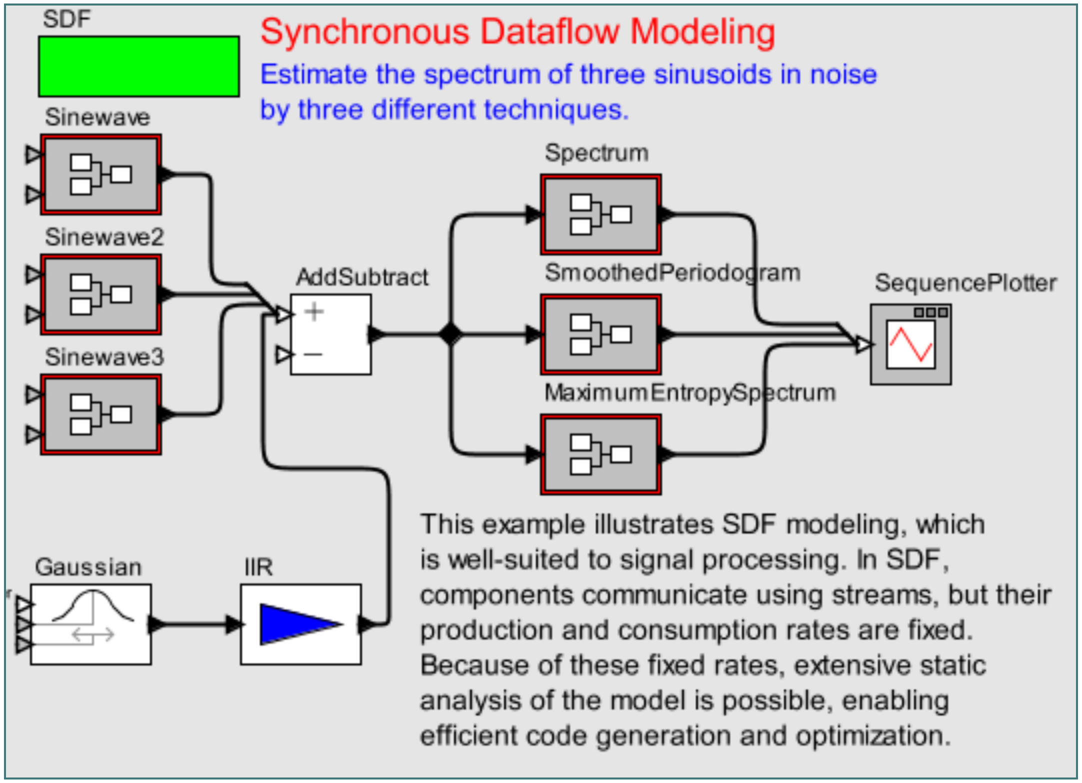


In PN, every actor runs in a thread, with blocking reads of input ports and non-blocking writes to outputs.

Example 3: Synchronous Dataflow (SDF)

In SDF, actors "fire," and in each firing, consume a fixed number of tokens from the input streams, and produce a fixed number of tokens on the output streams.

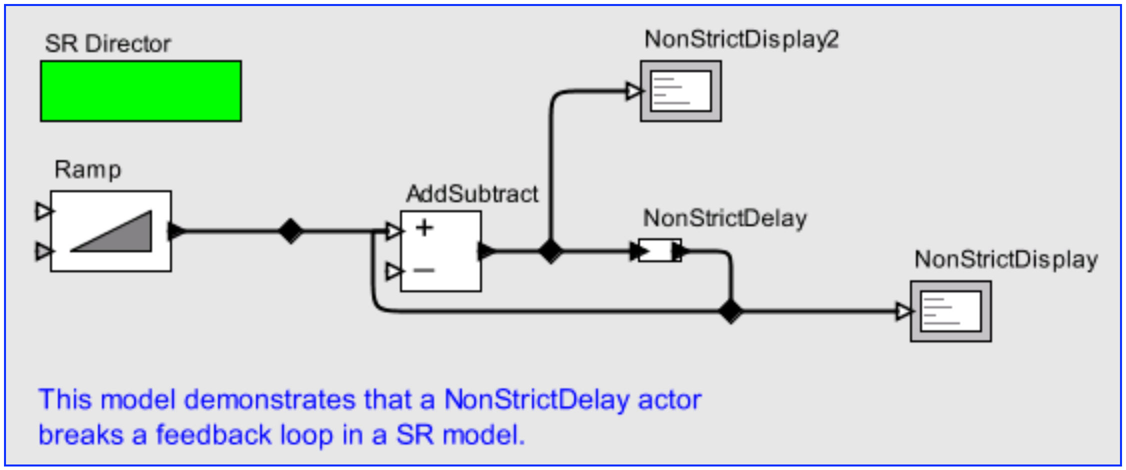


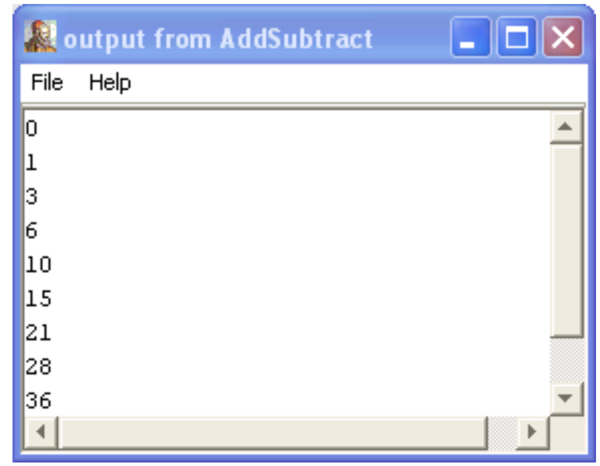


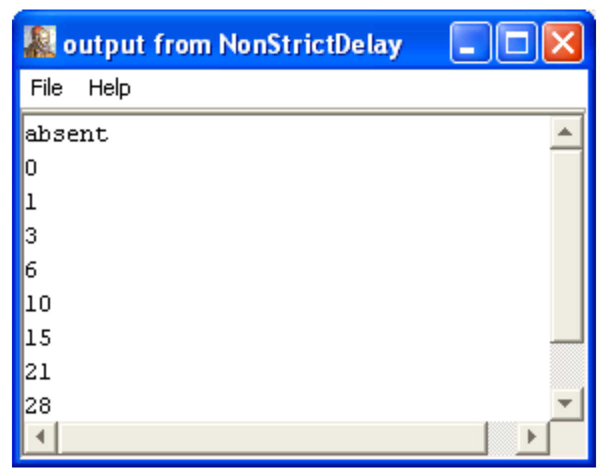
SDF is a special case of PN where deadlock and boundedness are decidable. It is well suited to static scheduling and code generation. It can also be automatically parallelized.

Example 4: Synchronous/Reactive (SR)

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



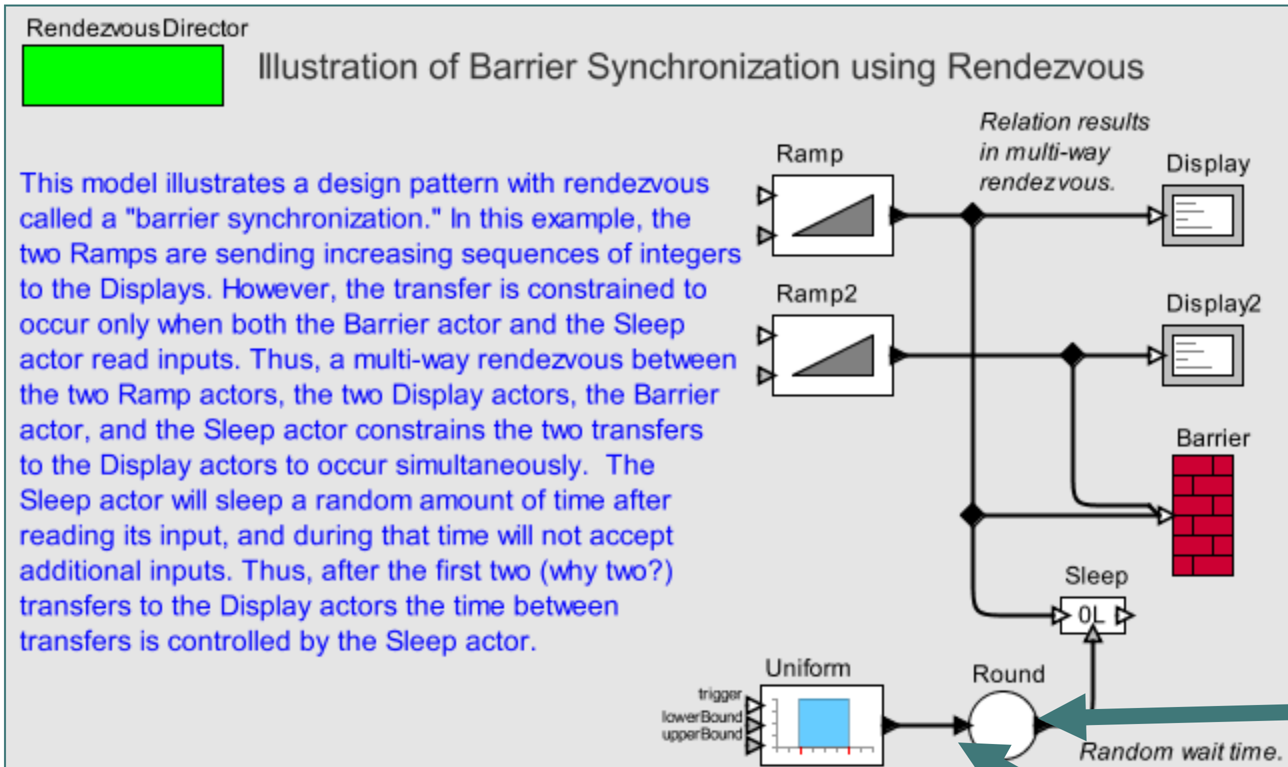




Like SDF, SR is decidable and suitable for code generation. It is harder to parallelize than SDF, however.

SR languages: Esterel, SyncCharts, Lustre, SCADE, Signal.

Example 5: Rendezvous



In Rendezvous, every actor runs in a thread, with blocking reads of input ports and blocking writes to outputs. Every communication is a (possibly multi-way) rendezvous.

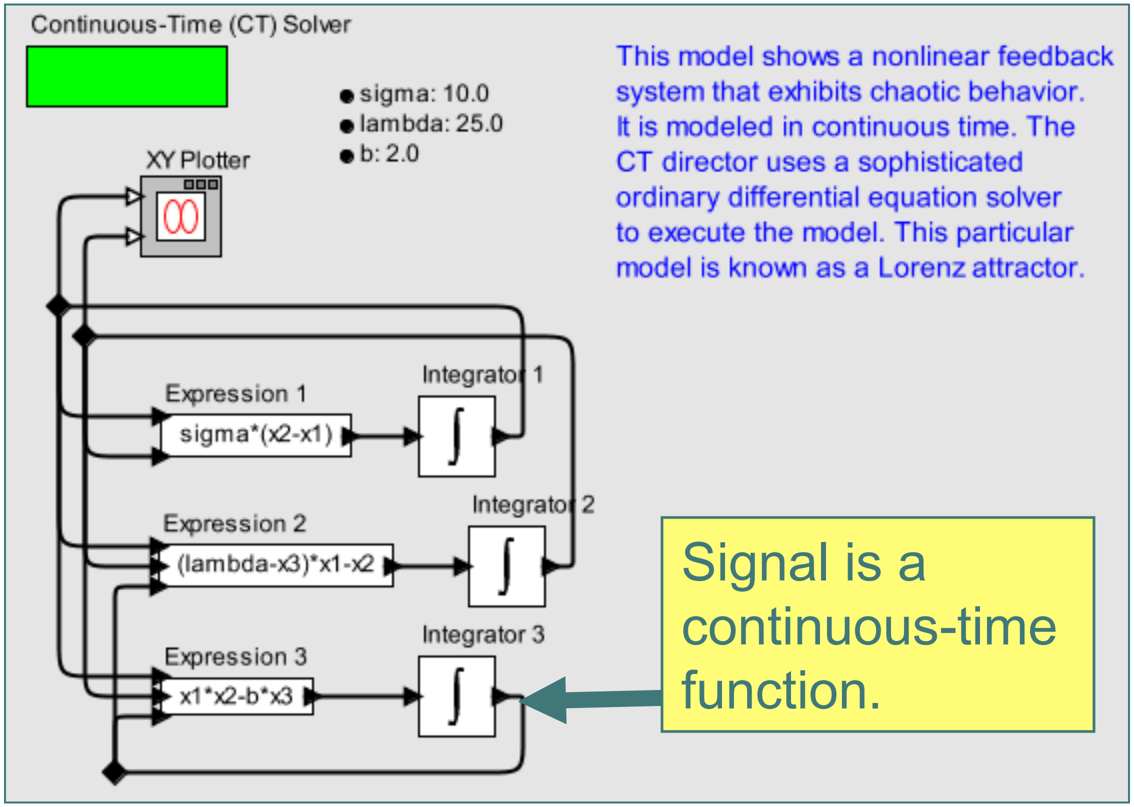
writes block

CSP (Hoare), SCCS (Milner), Reo (Arbab)

actor == thread

reads block

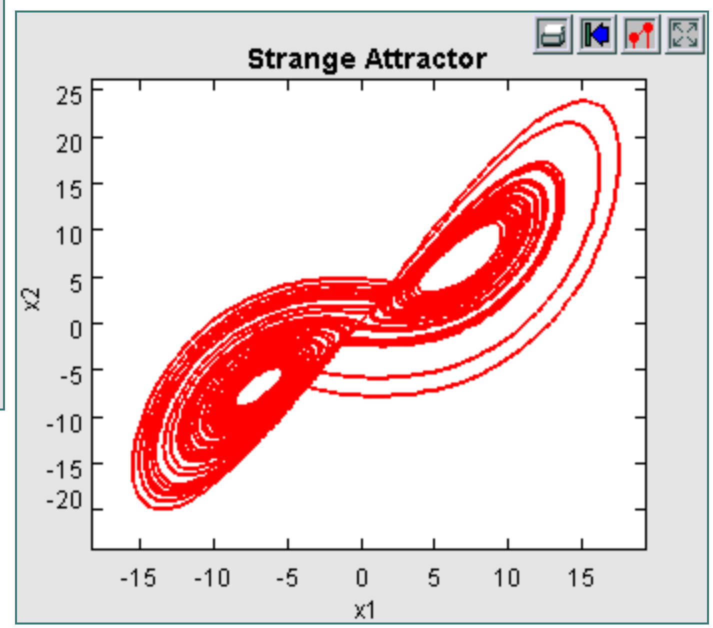
Example 6: Continuous Time (CT)



Signal is a continuous-time function.

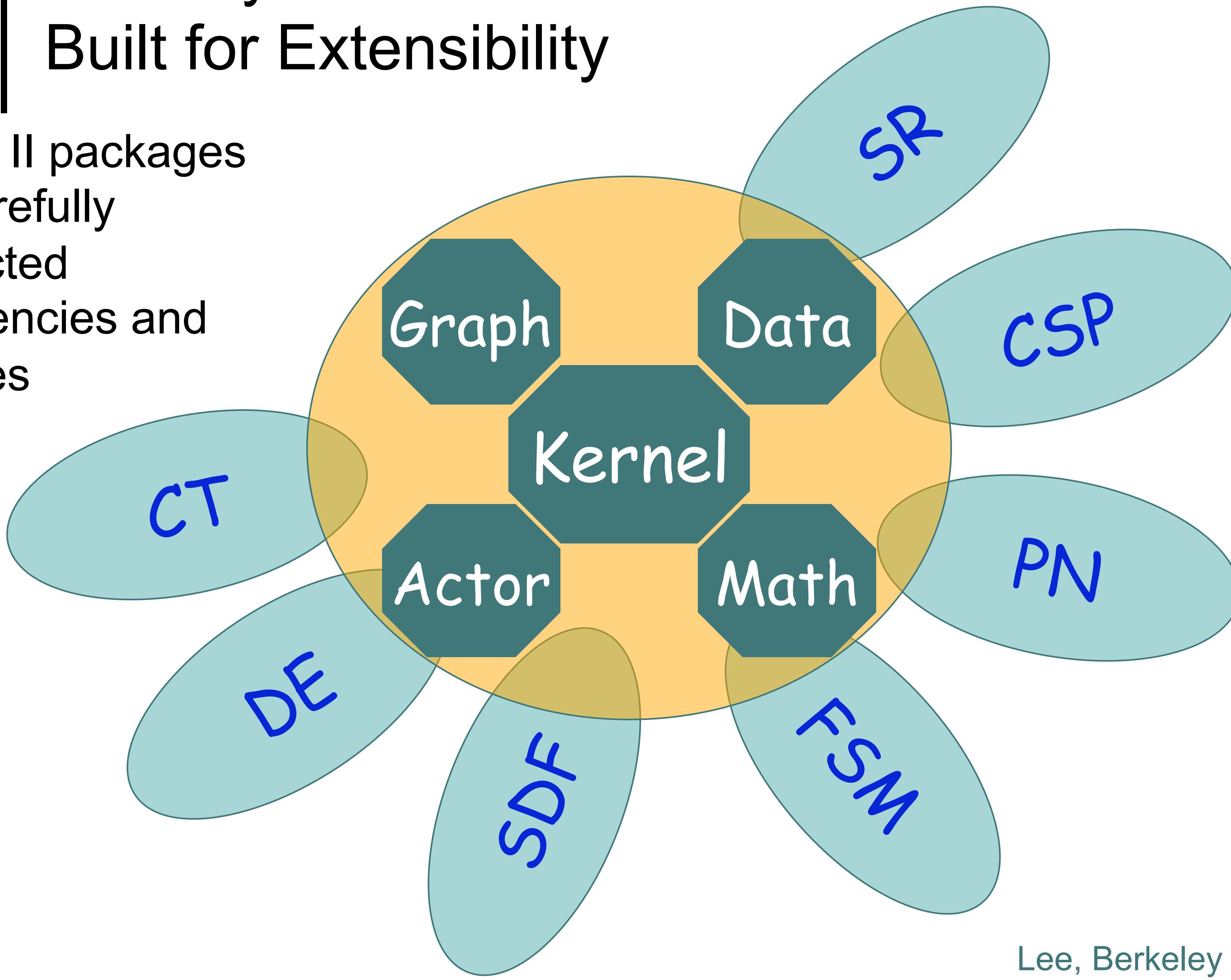
Director includes an ODE solver.

In CT, actors operate on continuous-time and/or discrete-event signals. An ODE solver governs the execution.



Ptolemy II Software Architecture Built for Extensibility

Ptolemy II packages have carefully constructed dependencies and interfaces

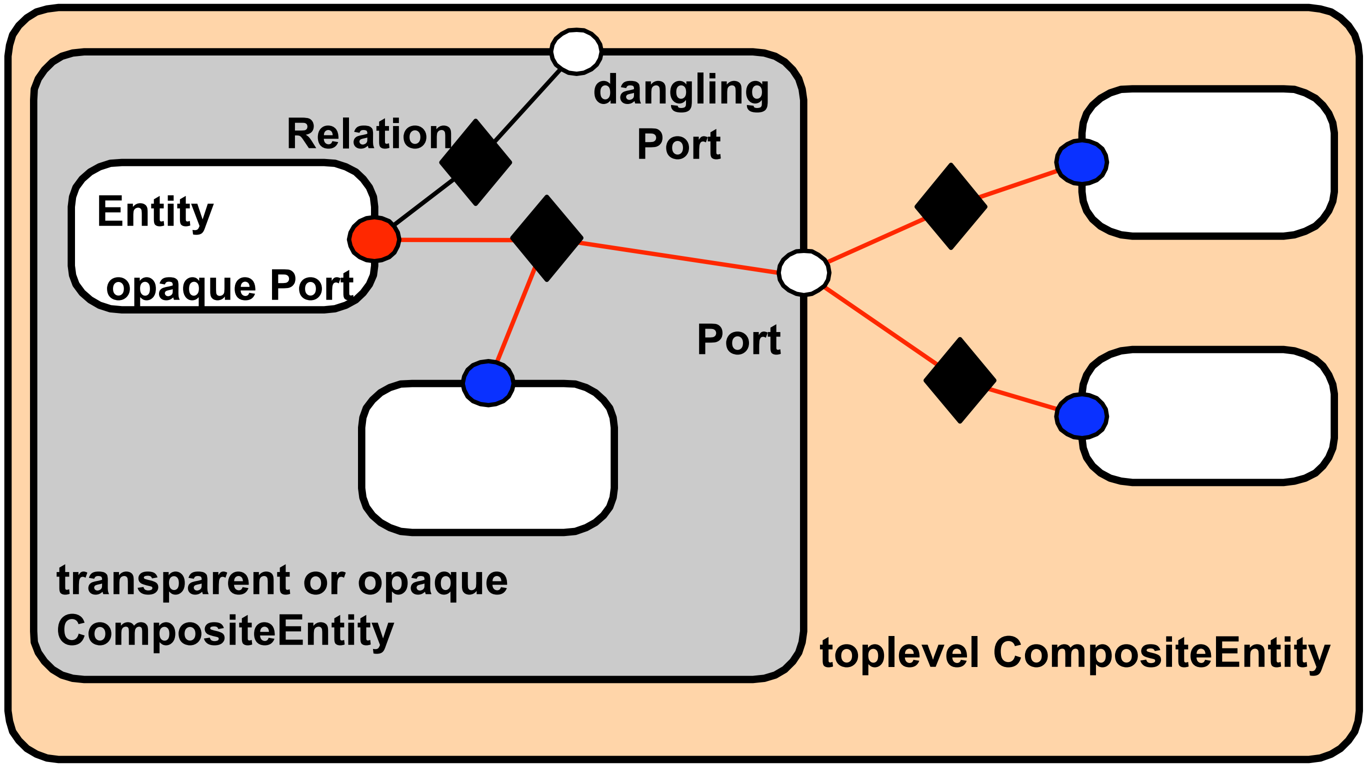


Models of Computation Implemented in Ptolemy II

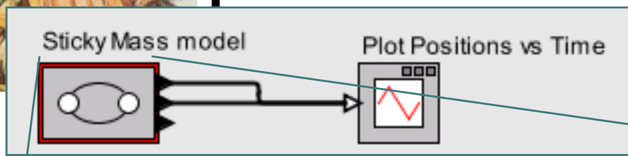
- CI – Push/pull component interaction
- Click – Push/pull with method invocation
- CSP – concurrent threads with rendezvous
- Continuous – continuous-time modeling with fixed-point semantics
- CT – continuous-time modeling
- DDF – Dynamic dataflow
- DE – discrete-event systems
- DDE – distributed discrete 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

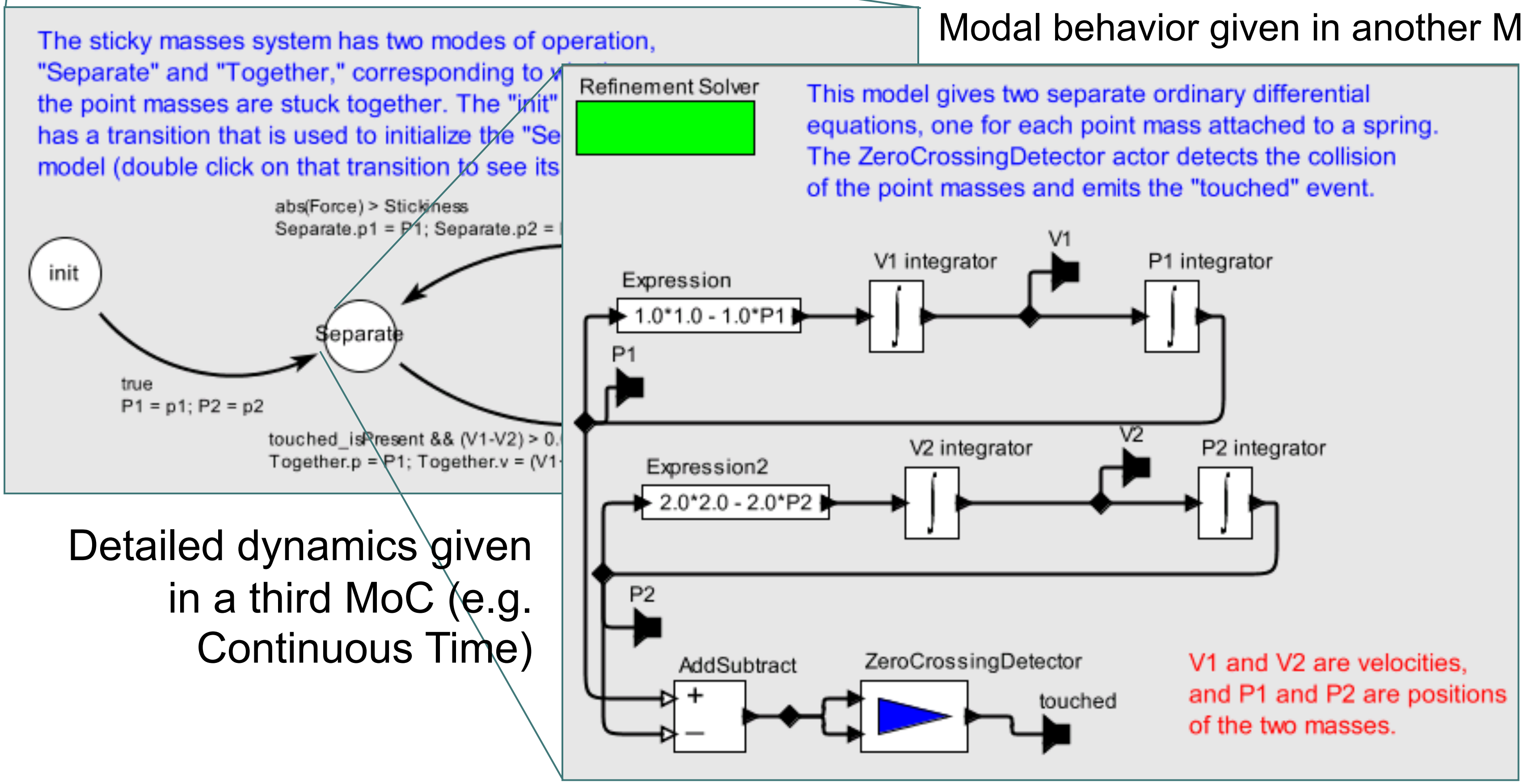


Ptolemy II Hierarchy Supports Heterogeneity



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

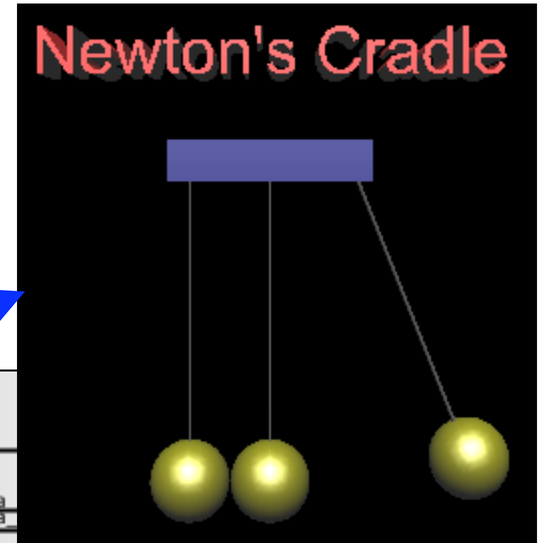
Modal behavior given in another MoC.



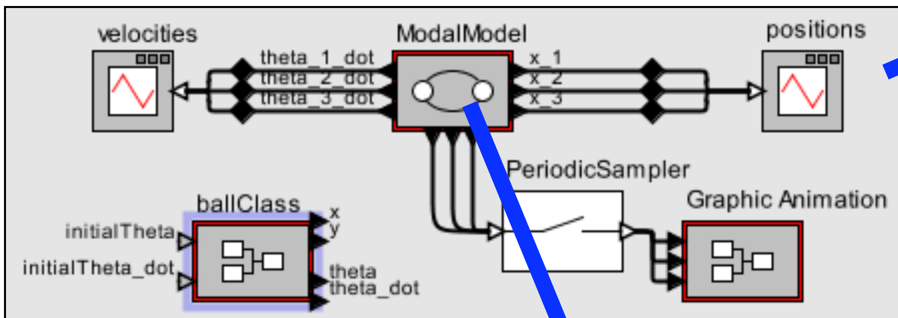
Detailed dynamics given in a third MoC (e.g. Continuous Time)

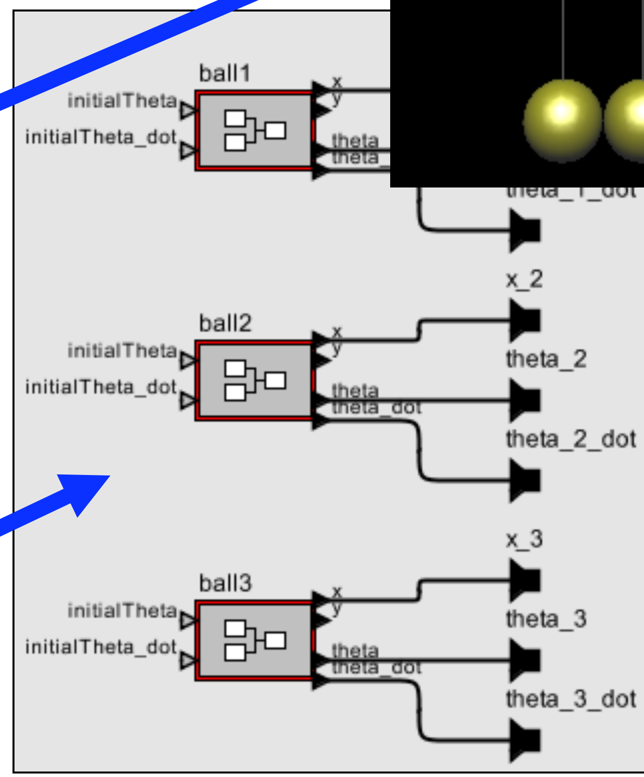
This requires a composable abstract semantics.

Hierarchical Heterogeneity (HH) Supports Hybrid Systems

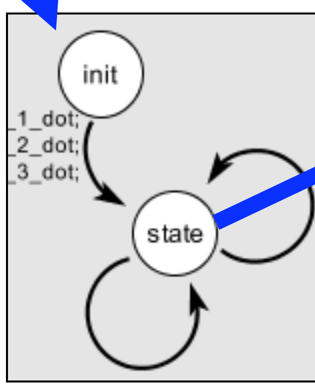


Combinations of synchronous/reactive, discrete-event, and continuous-time semantics offer a powerful way to represent and execute hybrid systems.

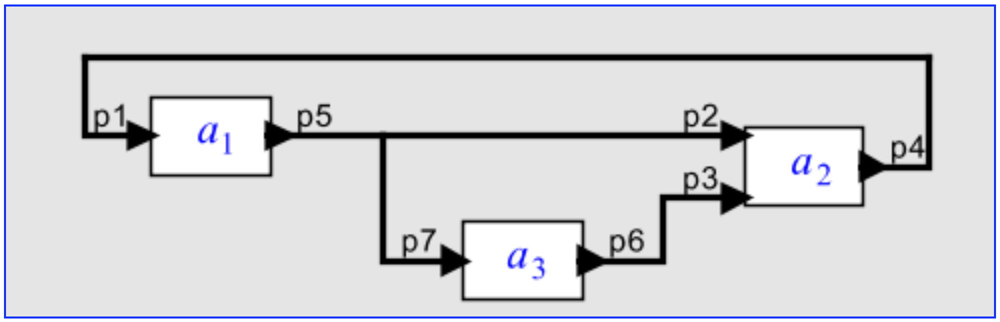




HyVisual is a specialization of the meta framework Ptolemy II.



In All Cases: Composition Semantics



Each actor is a function:

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

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.

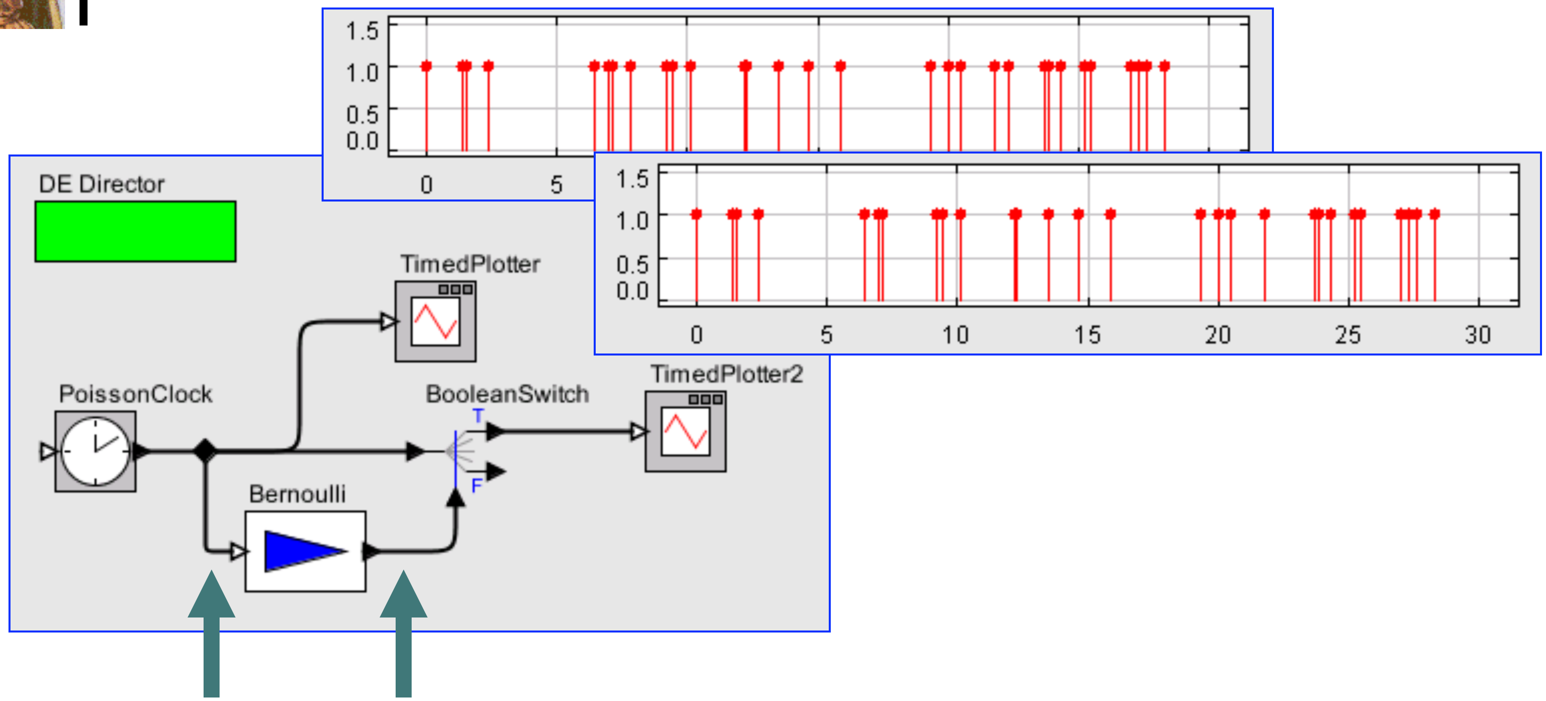
The concurrency model is called the "model of computation" (MoC).

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

Useful MoCs:

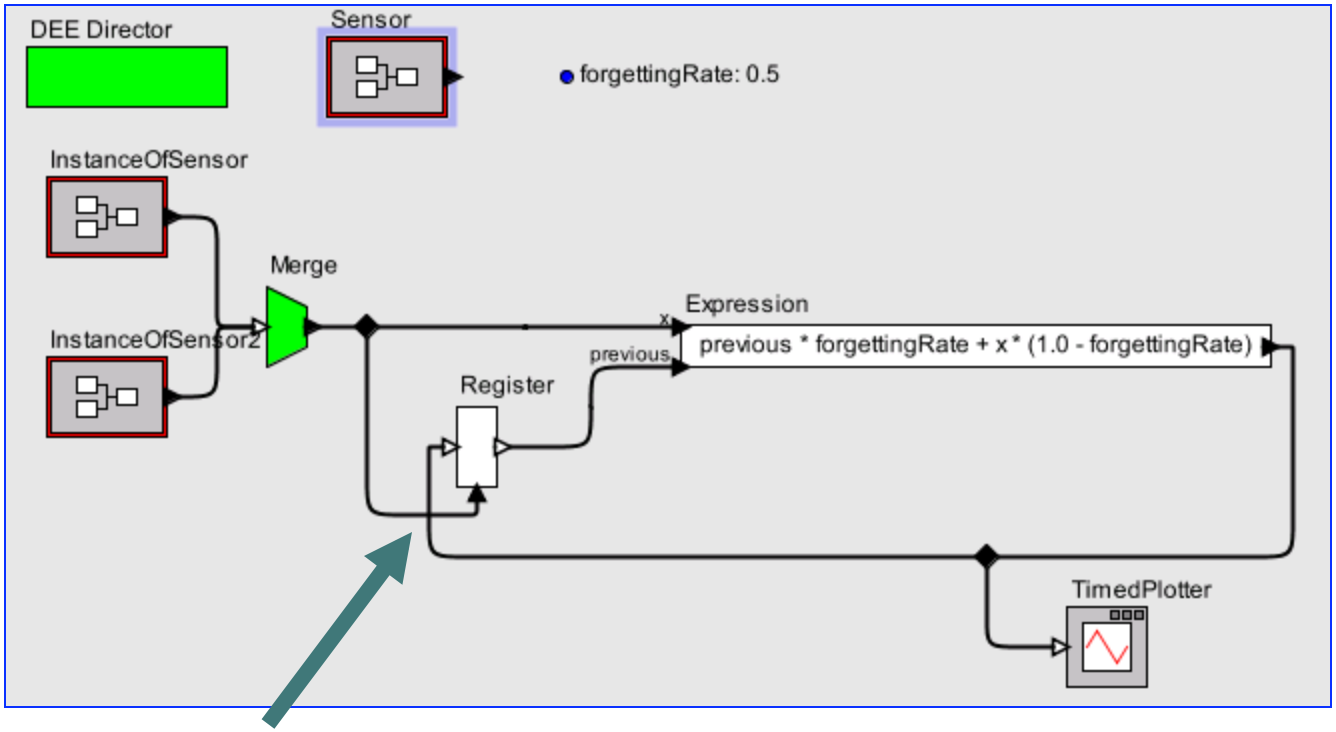
- *Process Networks*
- *Synchronous/Reactive*
- *Time-Triggered*
- *Discrete Events*
- *Dataflow*
- *Rendezvous*
- *Continuous Time*
- *…*

Semantics Clears Up Subtleties: E.g. Simultaneous Events



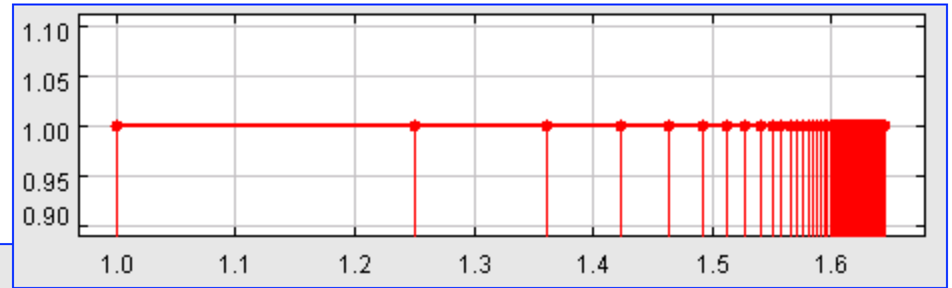
By default, an actor produces events with the same time as the input event. But in this example, we expect (and need) for the BooleanSwitch to "see" the output of the Bernoulli in the same "firing" where it sees the event from the PoissonClock. Events with identical time stamps are also ordered, and reactions to such events follow data precedence order.

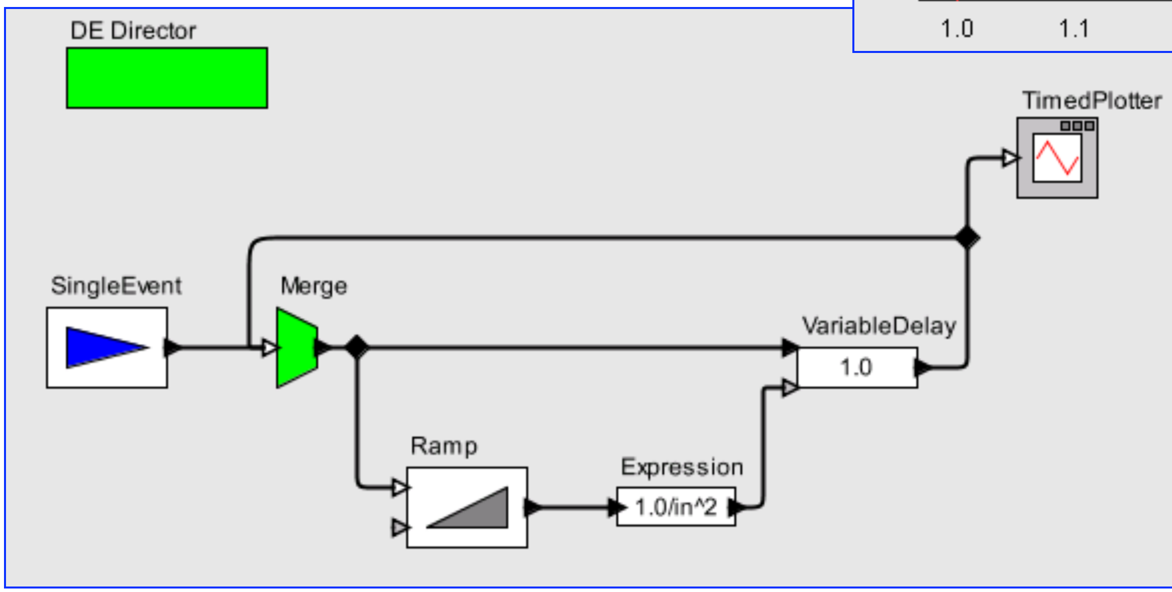
Semantics Clears Up Subtleties: E.g. Feedback



Data precedence analysis has to take into account the non-strictness of this actor (that an output can be produced despite the lack of an input).

Zeno Systems





Theorem: If every directed cycle contains a delta-causal component, then the system is non-Zeno.

Some Research Thrusts in the Ptolemy Project

- **Systems of systems**: Modeling and design of large scale systems, those that include networking, database, grid computing, and information subsystems.
- **Understandable concurrency**: This effort focuses on models of concurrency in software that are more understandable and analyzable than the prevailing abstractions based on threads.
- **Multicore and parallelism in embedded systems**: Code generation for parallel machines, scalable parallelism, model engineering, model transformation.
- **Precision-timed (PRET) machines**: Introduce timing into the core abstractions of computing, beginning with instruction set architectures, using configurable hardware as an experimental platform.
- **Real-time software**: Models of computation with time and concurrency, metaprogramming techniques, code generation and optimization, domain-specific languages, schedulability analysis, programming of sensor networks.
- **Distributed computing**: Models of computation based on distributed discrete events, backtracking techniques, lifecycle management, unreliable networks, modeling of sensor networks.
- **Hybrid systems**: Blended continuous and discrete dynamics, models of time, operational semantics, language design.

The Ptolemy Pteam