# An Overview of the Ptolemy Project and Actor-Oriented Design

Edward A. Lee
Professor
UC Berkeley

OMG Technical Meeting
Feb. 4, 2004
Anaheim, CA, USA

Special thanks to the entire Ptolemy Team.

Center for Hybrid and embedded software systems

## Abstract

The Ptolemy Project at UC Berkeley studies modeling, simulation, and design of concurrent, real-time, and embedded systems. The focus is on assembly of concurrent components under "actor-oriented" models of computation, where components are conceptually concurrent and communicate through one of several messaging schemas. This talk describes the principles of actor-oriented design, including common features across models of computation, such as abstract syntax and type systems, and features that differ across models of computation, such concurrent threads of control and messaging schemas. Mechanisms that support the use of heterogeneous mixtures of models of computation are also described. The Ptolemy II system, which is the experimental framework used by the project in its investigations, will be described and used to illustrate key points. The Ptolemy Project at UC Berkeley is part of Chess, the Berkeley Center for Hybrid and Embedded Software Systems.

# Ptolemy Project Participants

Director:
- Edward A. Lee

Staff:
- Christopher Hylands
- Susan Gardner (Chess)
- Nuala Mansard
- Mary P. Stewart
- Neil E. Turner (Chess)
- Lea Turpin (Chess)

Postdocs, Etc.:
- Joern Janneck, Postdoc
- Rowland R. Johnson, Visiting Scholar
- Kees Vissers, Visiting Industrial Fellow
- Daniel Lázaro Cuadrado, Visiting Scholar

Graduate Students:
- J. Adam Cataldo
- Chris Chang
- Elaine Cheong
- Sanjeev Kohli
- Xiaojun Liu
- Eleftherios D. Matsikoudis
- Stephen Neuendorffer
- James Yeh
- Yang Zhao
- Haiyang Zheng
- Rachel Zhou

# Software Legacy of the Project

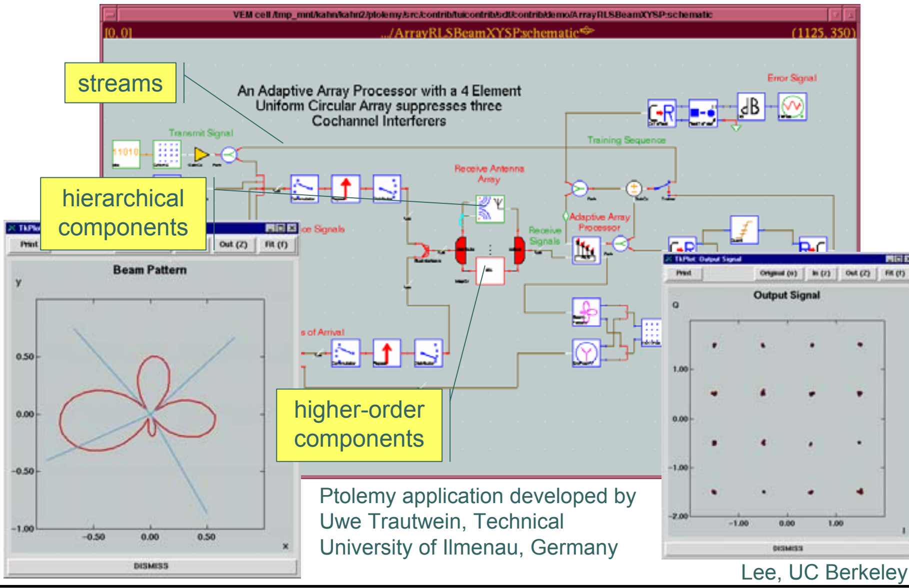
- Gabriel (1986-1991)
  - Written in Lisp
  - Aimed at signal processing
  - Synchronous dataflow (SDF) block diagrams
  - Parallel schedulers
  - Code generators for DSPs
  - Hardware/software co-simulators
- Ptolemy Classic (1990-1997)
  - Written in C++
  - Multiple models of computation
  - Hierarchical heterogeneity
  - Dataflow variants: BDF, DDF, PN
  - C/VHDL/DSP code generators
  - Optimizing SDF schedulers
  - Higher-order components
- Ptolemy II (1996-2022)
  - Written in Java
  - Domain polymorphism
  - Multithreaded
  - Network integrated and distributed
  - Modal models
  - Sophisticated type system
  - CT, HDF, CI, GR, etc.

Each of these served us, first-and-foremost, as a laboratory for investigating design.

- PtPlot (1997-??)
  - Java plotting package
- Tycho (1996-1998)
  - Itcl/Tk GUI framework
- Diva (1998-2000)
  - Java GUI framework

Focus has always been on embedded software.

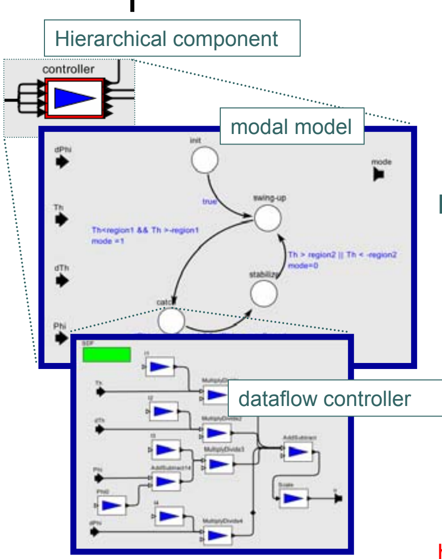
# Ptolemy Classic Example From 1995
(adaptive nulling in an antenna array)




Ptolemy application developed by Uwe Trautwein, Technical University of Ilmenau, Germany



# Ptolemy II




example Ptolemy II model: hybrid control system

**Ptolemy II:**

Our current framework for experimentation with actor-oriented design, concurrent semantics, visual syntaxes, and hierarchical, heterogeneous design.

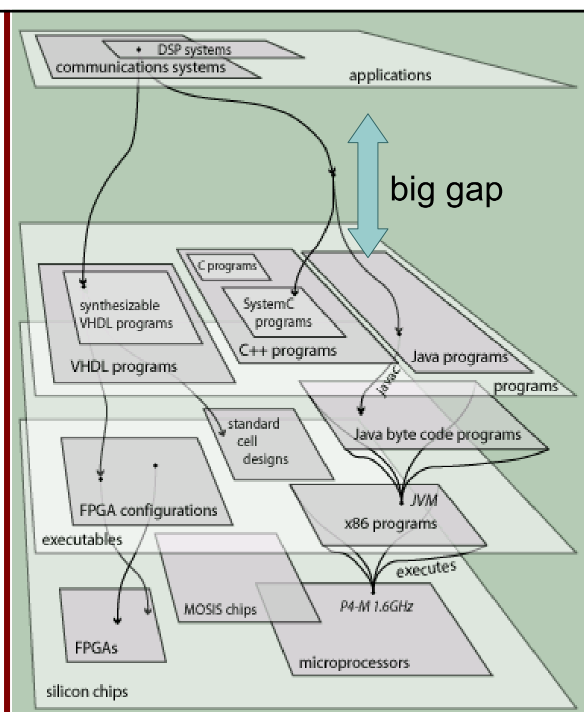
Ptolemy II is truly free software (cf. GPL)

http://ptolemy.eecs.berkeley.edu


## Platforms

A *platform* is a set of designs.

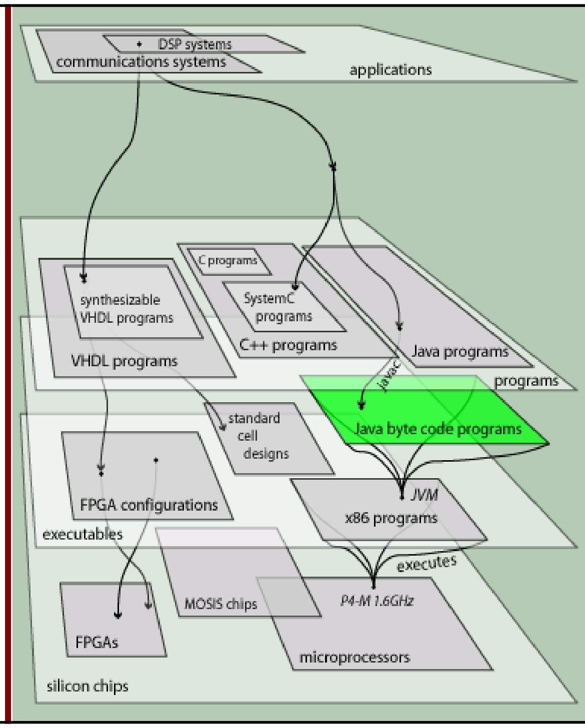
Relations between platforms represent design processes.



## Progress

Many useful technical developments amounted to creation of new platforms.

- microarchitectures
- operating systems
- virtual machines
- processor cores
- configurable ISAs



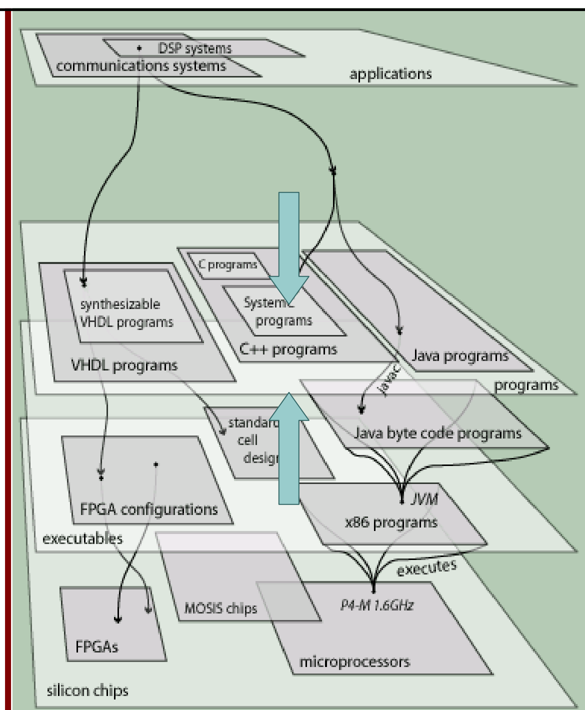

## Desirable Properties

From above:

- modeling
- expressiveness

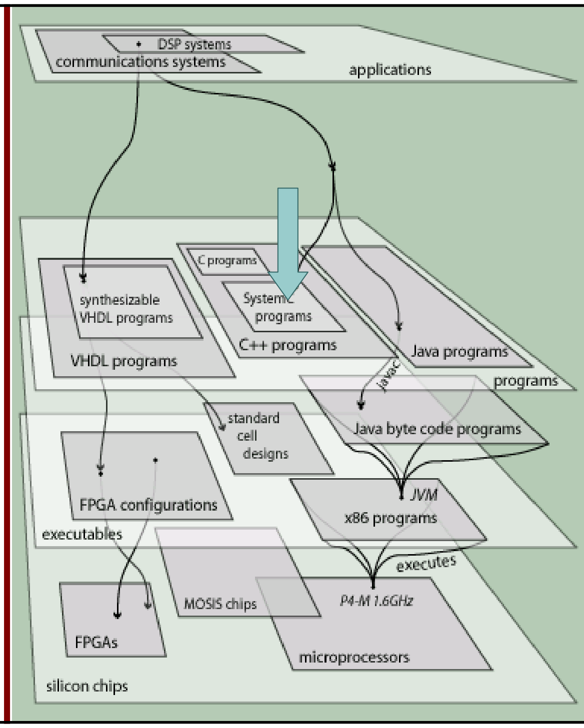
From below:

- correctness
- efficiency



# Model-Based Design

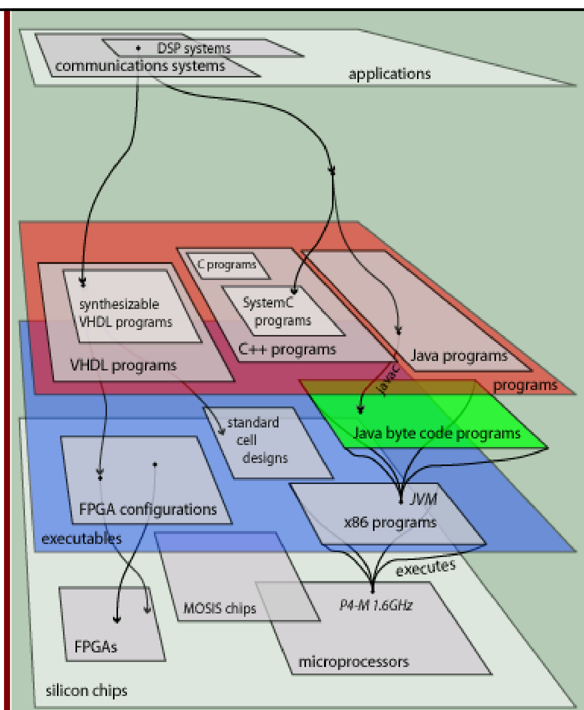
*Model-based design* is specification of designs in platforms with “useful modeling properties.”




# Recent Action

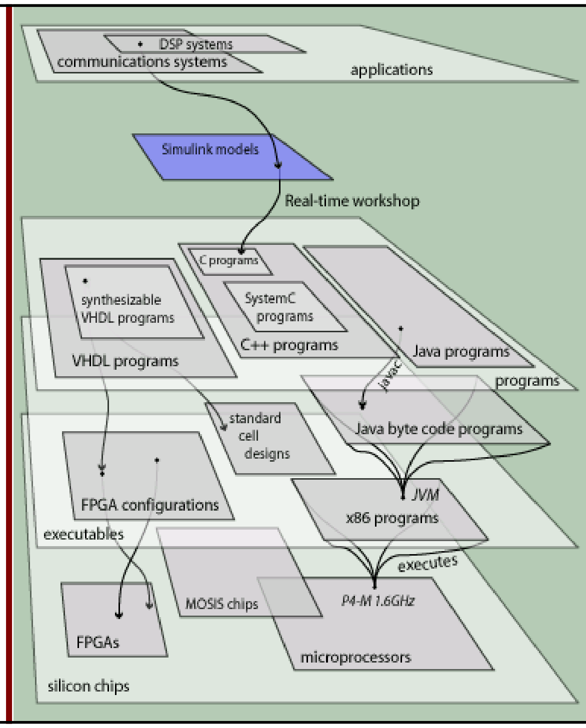
Giving the red platforms useful modeling properties (e.g. verification, SystemC, UML, MDA)

Getting from red platforms to blue platforms (e.g. correctness, efficiency, synthesis of tools)



## Better Platforms

Platforms with modeling properties that reflect requirements of the application, not accidental properties of the implementation.




## How to View This Design

From above: Signal flow graph with linear, time-invariant components.

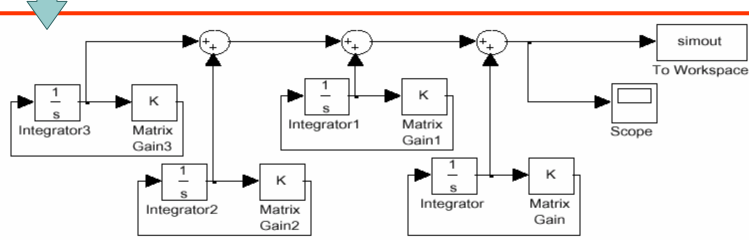
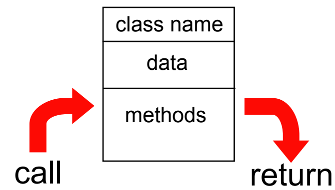



Figure C.12: A block diagram generating a plucked string sound with a fundmental and three harmonics.

From below: Synchronous concurrent composition of components

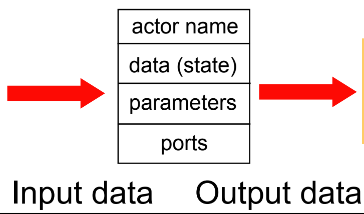
# Actor-Oriented Design

## Object orientation:




What flows through an object is sequential control

## Actor orientation:

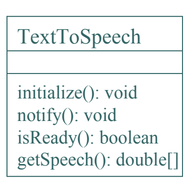



What flows through an object is streams of data



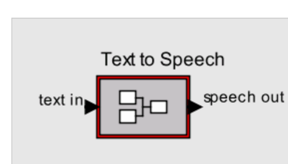
# Actor Orientation vs. Object Orientation

Object oriented




OO interface definition gives procedures that have to be invoked in an order not specified as part of the interface definition.

Actor oriented




actor-oriented interface definition says "Give me text and I'll give you speech"

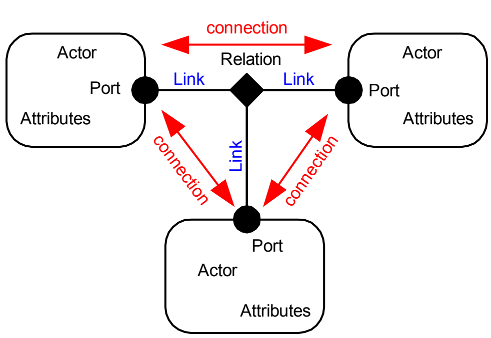
- Identified problems with object orientation:
  - Says little or nothing about concurrency and time
  - Concurrency typically expressed with threads, monitors, semaphores
  - Components tend to implement low-level communication protocols
  - Re-use potential is disappointing
- Actor orientation offers more potential for useful modeling properties, and hence for model-based design.

# "Actors" vs. "Capsules"

- Actors are more like UML capsules than like UML actors
- The term "actors" was introduced in the 1970's by Carl Hewitt of MIT to describe autonomous reasoning agents.
- The term evolved through the work of Gul Agha and others to refer to a family of concurrent models of computation, irrespective of whether they were being used to realize autonomous reasoning agents.
- The term "actor" has also been used since 1974 in the dataflow community in the same way, to represent a concurrent model of computation.



# Abstract Syntax: Hierarchical Entities, Ports, Connections and Attributes




Our abstract syntax choices:

- Hierarchy is tree structured (like XML).
- A *relation* mediates connections.
- Ports can link multiple relations and relations can link multiple ports.
- Ports mediate connections across levels of the hierarchy (no statecharts-style level-crossing links)
- …

Abstract syntax defines the structure of a model, but says little about what it means.

# MoML – An XML Concrete Syntax (Modeling Markup Language)

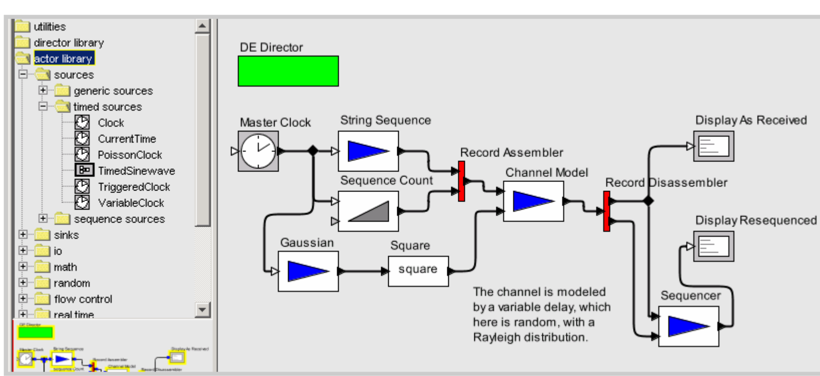
```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE model PUBLIC "…" "http://…">
<model name="top" class="path name">
  <entity name="source" class="path name">
    <port name="output"/>
  </entity>
  <entity name="sink" class="path name">
    <port name="input"/>
  </entity>
  <relation name="r1" class="path name"/>
  <link port="source.output" relation="r1"/>
  <link port="sink.input" relation="r1"/>
</model>
```

MoML is the persistent file format of Ptolemy II.



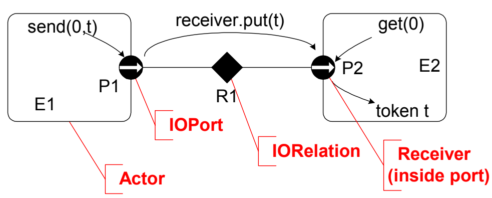
# Visual Renditions of Models

Ptolemy II model rendered in Vergil, a visual editor:



## Semantics of Producer/Consumer Components

**Basic Transport:**




Models of Computation:

- continuous-time
- dataflow
- rendezvous
- discrete events
- synchronous
- time-driven
- publish/subscribe
- …

This abstract syntax is compatible with many semantic interpretations. The concurrency and communication model together is what we call the *model of computation* (MoC).



## Examples of Actor-Oriented Component Frameworks

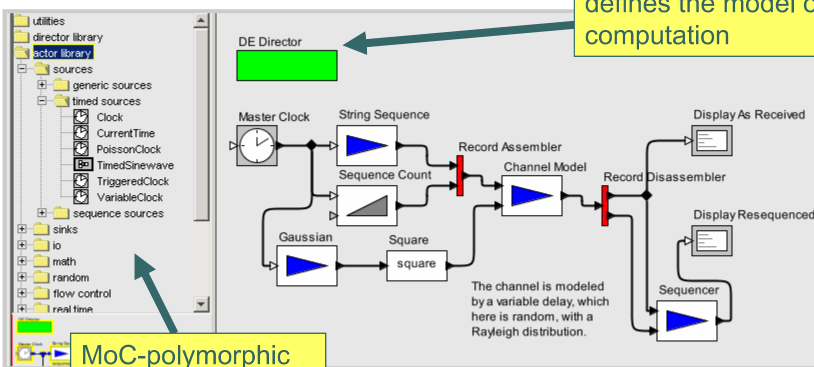
- Simulink (The MathWorks)
- Labview (National Instruments)
- Modelica (Linkoping)
- Polis & Metropolis (UC Berkeley)
- OCP, open control platform (Boeing)
- GME, actor-oriented meta-modeling (Vanderbilt)
- SPW, signal processing worksystem (Cadence)
- System studio (Synopsys)
- ROOM, real-time object-oriented modeling (Rational)
- Easy5 (Boeing)
- Port-based objects (U of Maryland)
- I/O automata (MIT)
- VHDL, Verilog, SystemC (Various)
- …

Unlike Ptolemy II, most of these define a fixed model of computation.

## Ptolemy Project Principle

The model of computation is not built in to the software framework.

Director from a library defines the model of computation

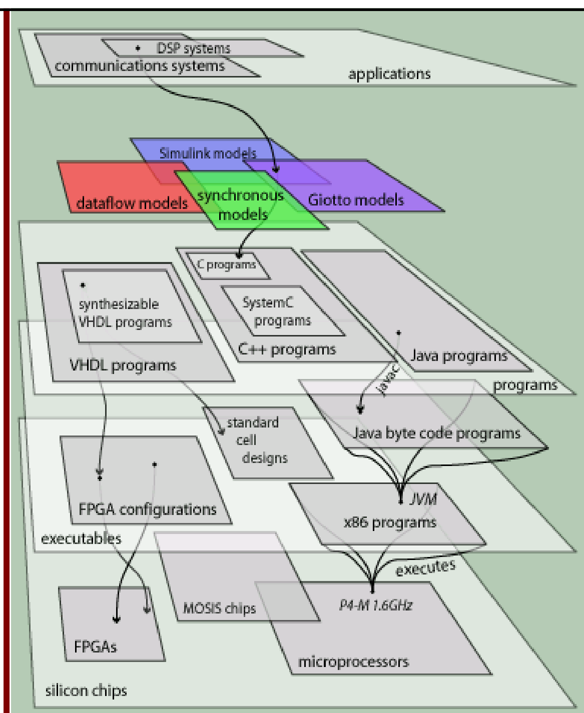



MoC-polymorphic component library.



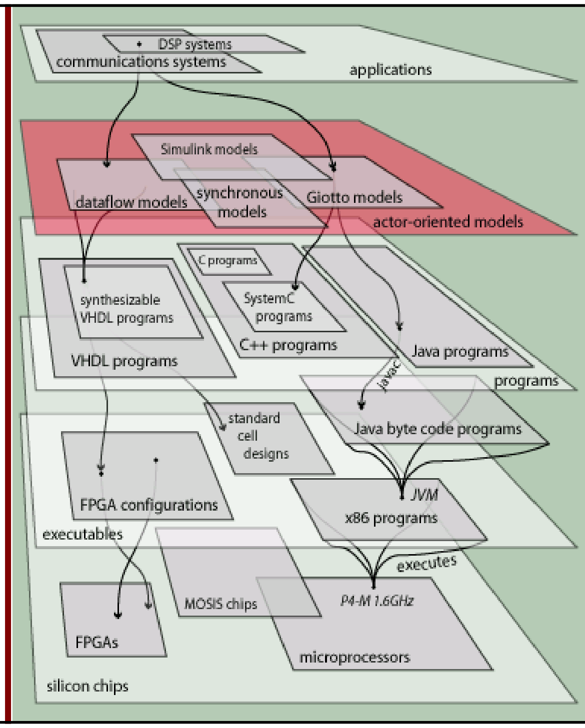
## Actor-Oriented Design is not One But Many Techniques

A rich set of possible semantic and syntactic approaches, each with useful modeling and implementation properties.



## Actor-Oriented Platforms

*Actor oriented* models compose concurrent components according to a model of computation.




## Examples of Models of Computation

- Dataflow
- Discrete events
- Continuous time
- Finite state machines
- Synchronous reactive
- Time driven
- Publish and subscribe
- Communicating sequential processes
- Process networks
- …

Each of these has several competing variants

# Start With Dataflow

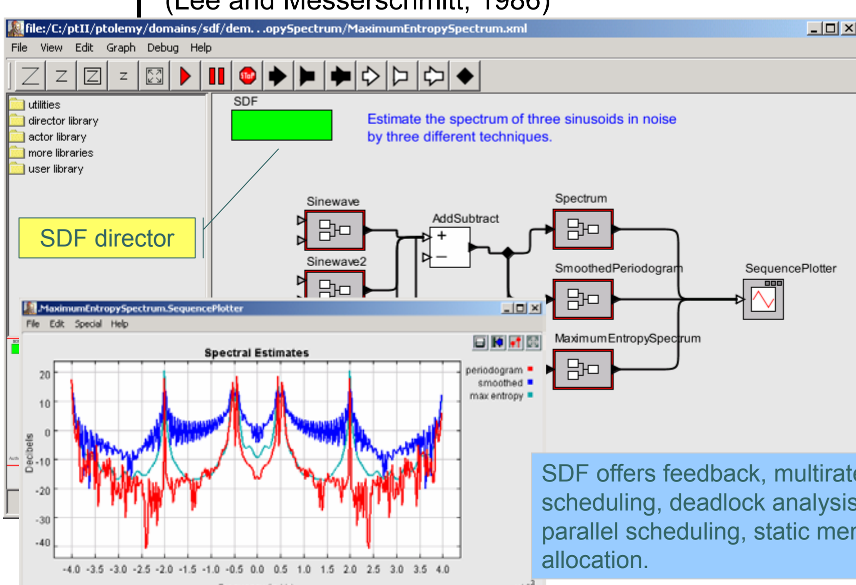
- Computation graphs [Karp & Miller - 1966]
- Visual programs [Sutherland – 1966]
- Process networks [Kahn - 1974]
- Static dataflow [Dennis - 1974]
- Dynamic dataflow [Arvind, 1981]
- Structured dataflow [Matwin & Pietrzykowski 1985]
- K-bounded loops [Culler, 1986]
- Synchronous dataflow [Lee & Messerschmitt, 1986]
- Structured dataflow [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]
- …

Many tools, software frameworks, and hardware architectures have been built to support one or more of these.



# Synchronous Dataflow (SDF)

(Lee and Messerschmitt, 1986)




SDF offers feedback, multirate, static scheduling, deadlock analysis, parallel scheduling, static memory allocation.

# Synchronous Dataflow (SDF)

## Fixed Production/Consumption Rates

- Balance equations (one for each channel):

$$f_A N = f_B M$$

number of tokens consumed

number of firings per "iteration"

number of tokens produced

- Schedulable statically
- Get a well-defined "iteration"
- Decidable:
  - buffer memory requirements
  - deadlock

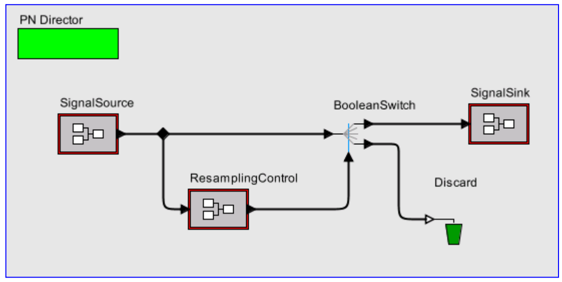



# Dynamic Dataflow (DDF)

- Actors have *firing rules*
  - Set of finite prefixes on input sequences
  - For determinism: No two such prefixes are joinable under a prefix order
  - Firing function applied to finite prefixes yield finite outputs
- Scheduling objectives:
  - Do not stop if there are executable actors
  - Execute in bounded memory if this is possible
  - Maintain determinacy if possible
- Policies that fail:
  - Data-driven execution
  - Demand-driven execution
  - Fair execution
  - Many balanced data/demand-driven strategies
- Policy that succeeds (Parks 1995):
  - Execute with bounded buffers
  - Increase bounds only when deadlock occurs

key properties of DDF models are undecidable (deadlock, bounded memory, schedule)

## Application of Dynamic Dataflow: Resampling of Streaming Media



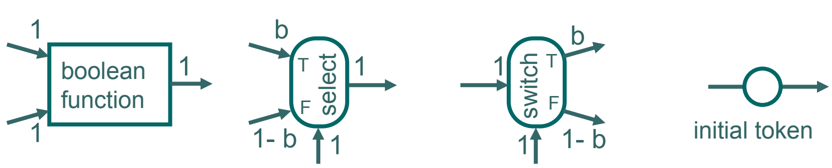

- This pattern requires the use of a semantically richer dataflow model than SDF because the BooleanSwitch is not an SDF actor.
- This has a performance cost and reduces the static analyzability of the model.



## Undecidability: What SDF Avoids

(Buck '93)

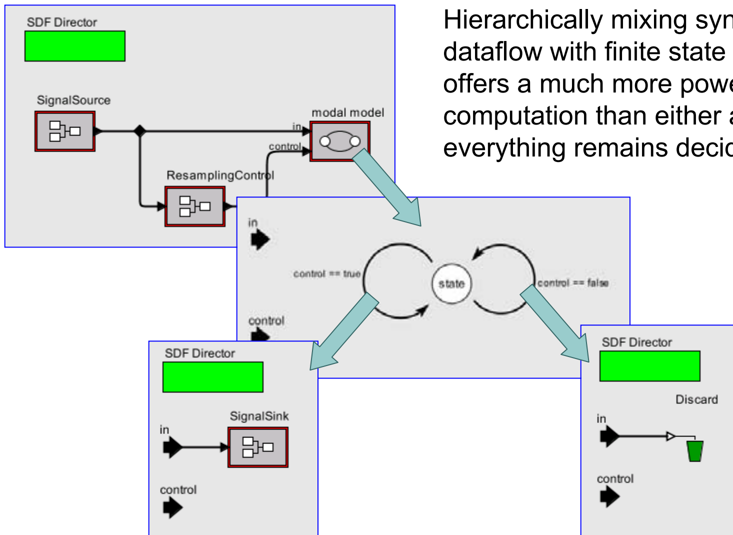
- Sufficient set of actors for undecidability:
  - boolean functions on boolean tokens
  - switch and select
  - initial tokens on arcs




- Undecidable:
  - deadlock
  - bounded buffer memory
  - existence of an annotated schedule

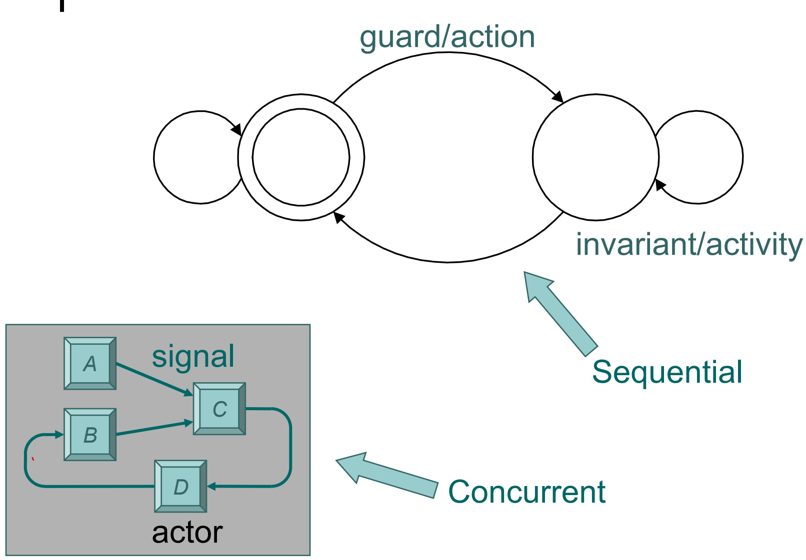
## Resampling Design Pattern using Hierarchical Heterogeneity

Hierarchically mixing synchronous dataflow with finite state machines offers a much more powerful model of computation than either alone. And everything remains decidable!



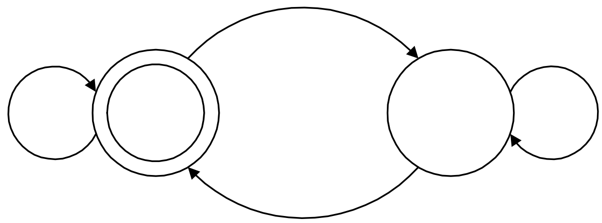



## State Machines & Block Diagrams



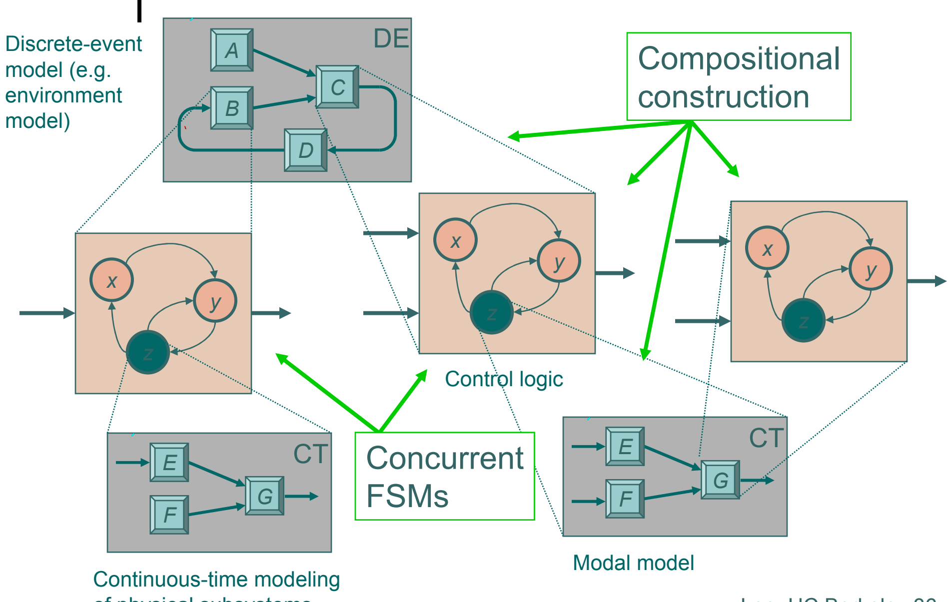
## Useful State Machine Models

- Von-Neumann computers
- Imperative programming languages
- Finite state machines (FSMs)





## Concurrency + Control Logic

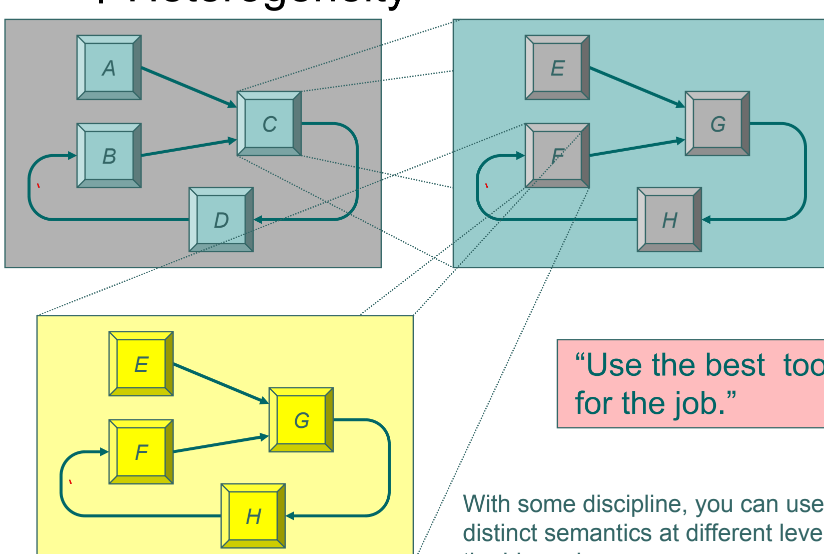


# Contrast With Statecharts

- Statecharts bundle orthogonal semantic issues
  - state machines
  - concurrency
- Statecharts chooses synchronous semantics for the concurrency model
  - what if I want an asynchronous model?
  - what if I want continuous time (to get hybrid systems)?
  - what if I want time-stamped discrete events?



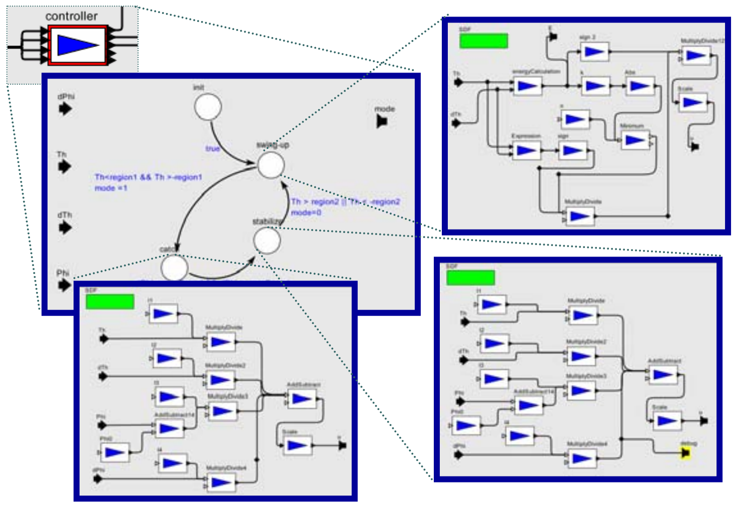
# The Principle of Hierarchical Heterogeneity




"Use the best tool for the job."

With some discipline, you can use distinct semantics at different levels of the hierarchy.

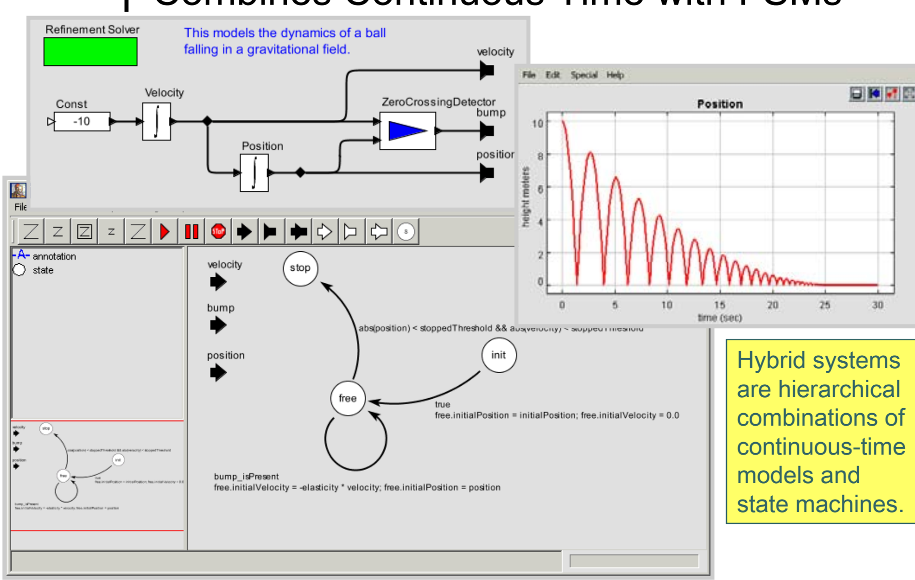
## Example: Heterochronous Dataflow (HDF) Combines Dataflow with FSMs




We can keep everything decidable, but greatly improve expressiveness.



## Another Example: Hybrid Systems Combines Continuous Time with FSMs




Hybrid systems are hierarchical combinations of continuous-time models and state machines.

## Heterogeneous Models

We refer to models that combine FSMs hierarchically with concurrent models of computation as *modal models.*

Modal models are one example of a family of hierarchically heterogeneous models, where diverse models of computation are combined in a hierarchy.



## How Does This Work? Abstract Semantics is the Key

flow of control

- Initialization
- Execution
- Finalization

- **preinitialize()**
  - **declare static information, like type constraints, scheduling properties, temporal properties, structural elaboration**
- **initialize()**
  - **initialize variables**

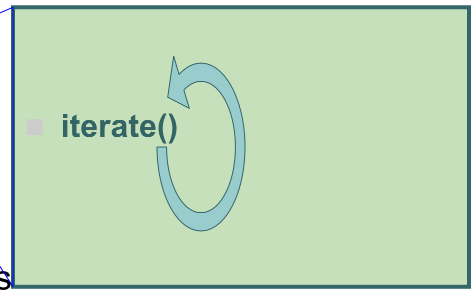
communication

- Structure of signals
- Send/receive protocols

# Abstract Semantics – The Key To Hierarchical Heterogeneity

flow of control
- Initialization
- Execution
- Finalization

communication
- Structure of signals
- Send/receive protocols



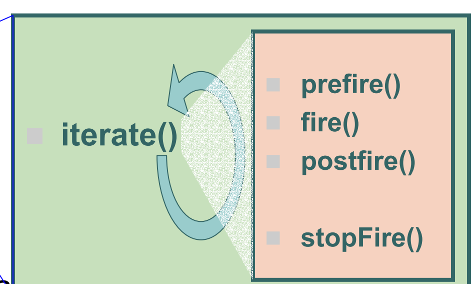



# Abstract Semantics – The Key To Hierarchical Heterogeneity

**The order in which component methods prefire(), fire(), postfire(), depends on the model of computation.**

flow of control
- Initialization
- Execution
- Finalization

communication
- Structure of signals
- Send/receive protocols




**In hierarchical heterogeneity, the fire() method iterates a submodel, but according to its model of computation.**

# Lifecycle Management

- It is possible to hierarchically compose the Ptolemy II abstract semantics.
- Actors providing common patterns:
  - *RunCompositeActor* is a composite actor that, instead of firing the contained model, executes a complete lifecycle of the contained model.
  - *ModelReference* is an atomic actor whose function is provided by a complete execution of a referenced model in another file or URL.
- Provides systematic approach to building systems of systems.



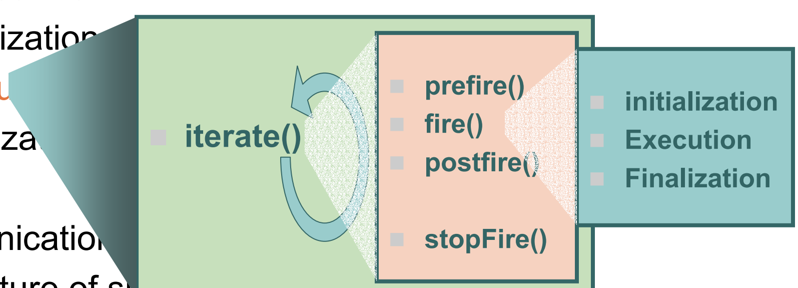
# Hierarchical Composition of the Ptolemy II Abstract Semantics

flow of control

- Initializatio
- Execu
- Finaliza

communicatio

- Structure of si
- Send/receive protocols



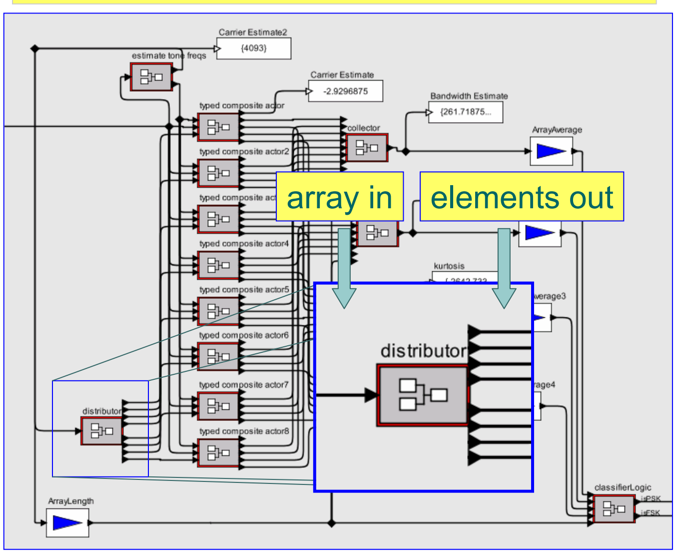
# Other Stream-Like Models of Computation Compatible with this Abstract Semantics

- Discrete events (e.g. NS)
  - data tokens have time stamps
- Synchronous languages (e.g. Esterel)
  - sequence of values, one per clock tick
  - fixed-point semantics
- Time triggered (e.g. Giotto)
  - similar, but no fixed-point semantics
- Process networks
  - separate thread per actor
  - asynchronous communication
- Communicating sequential processes
  - separate thread per actor
  - synchronous communication
- Push/Pull (e.g. Click)
  - dataflow with disciplined nondeterminism



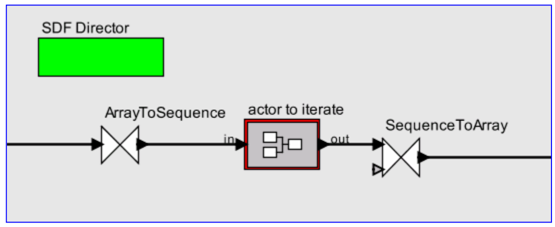
# Is Using Visual Syntaxes a Good Idea?

Example: Need to separately process elements of an array




- naïve approach:
  - 8 elements
  - 8 signal paths
- hard to build
- hardwired scale
- distributor:
  - converts an array of dimension 8 to a sequence of 8 tokens.

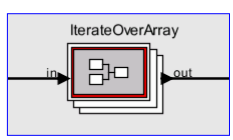
# Scalability of Visual Syntaxes Iteration by Dataflow




- Although sometimes useful, this design pattern has limitations:
  - array size must be statically fixed
  - actor to iterate must be stateless, or
  - desired semantics must be to carry state across array elements

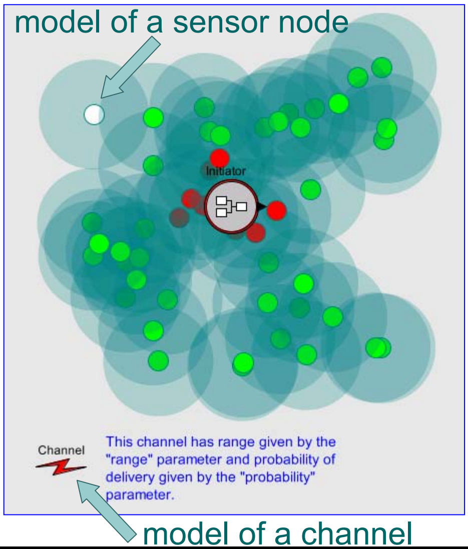


# Analogy to Structured Programming in Actor-Oriented Models




- A library of actors that encapsulate common design patterns:
  - IterateOverArray: Serialize an array input and provide it sequentially to the contained actor.
  - MapOverArray: Provide elements of an array input to distinct instances of the contained actor.
  - Zip, Scan, Case, …
- Like the *higher-order functions* of functional languages, but unlike functions, actors can have state.
- The implementation leverages the abstract semantics of Ptolemy II.

## What About All Those Wires? If You Don't Want Them, Don't Use Them

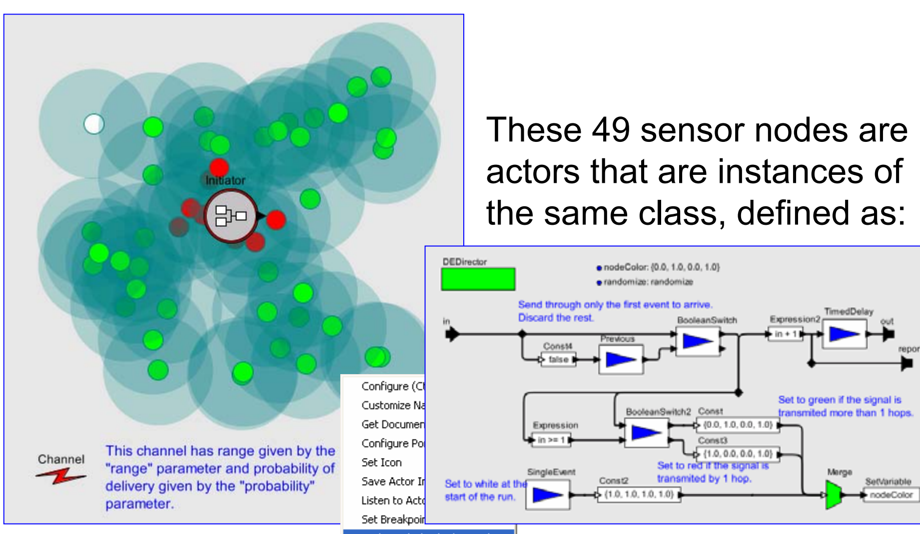



- Ptolemy II framework for modeling wireless sensor networks
- Connectivity is wireless
- Customized visualization
- Location-aware models
- Channel models include:
  - packets losses
  - power attenuation
  - distance limitations
  - collisions
- Component models include:
  - Antenna gains
  - Terrain models
  - Jamming

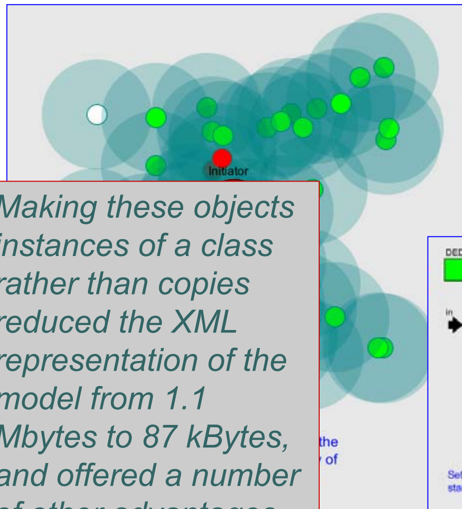


## What About Abstraction?

These 49 sensor nodes are actors that are instances of the same class, defined as:

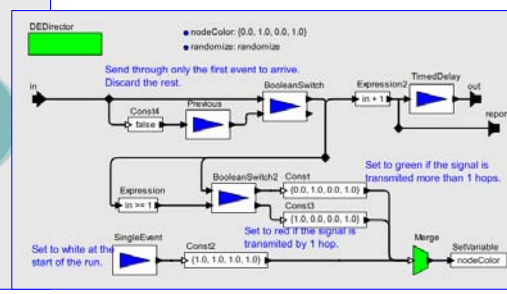


# What About Modularity?




*Making these objects instances of a class rather than copies reduced the XML representation of the model from 1.1 Mbytes to 87 kBytes, and offered a number of other advantages.*

The definition below is a *class* and objects at the left are *instances*, not copies.

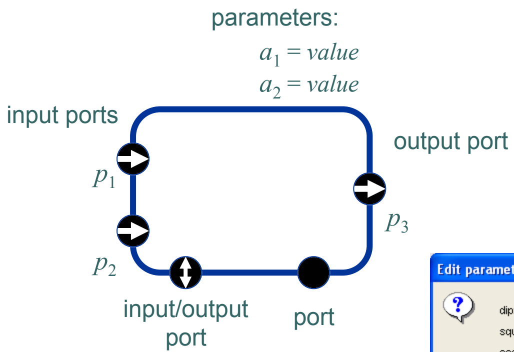




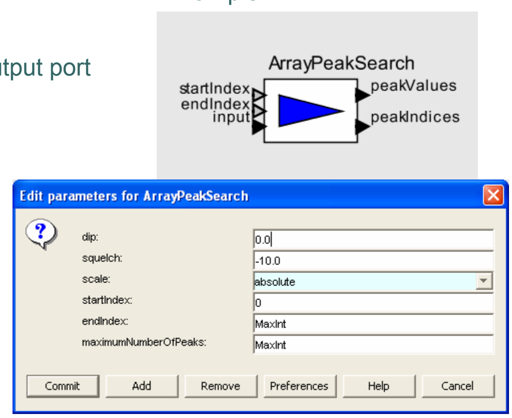

# Now that we have *classes*, can we bring in more of the modern programming world?

- subclasses?
- inheritance?
- interfaces?
- subtypes?
- aspects?

# Actor Interfaces: Ports and Parameters




Example:






# Subclasses? Inheritance? Interfaces? Subtypes? Aspects?

These are a part of what the Berkeley Center for Hybrid and Embedded Software Systems (Chess) is doing.

## Yes We Can!

- subclasses and inheritance
  - hierarchical models that inherit structure from a base class
- interfaces and subtypes
  - ports and parameters of actors form their interface
- aspects
  - heterarchical models interweave multiple hierarchies, providing true multi-view modeling.

*All* of these operate at the abstract syntax level, and are independent of the model of computation, and therefore can be used with any model of computation! Thus, they become available in domain-specific actor-oriented languages.

# Conclusion

- Actor-oriented design remains a relatively immature area, but one that is progressing rapidly.
- Ptolemy II is free and open software for experimenting with actor-oriented design techniques.
- see http://ptolemy.eecs.berkeley.edu