Model-Driven Development
From Object-Oriented Design to Actor-Oriented Design

Edward A. Lee
Professor
UC Berkeley

Invited Talk
Workshop on Software Engineering for Embedded Systems
From Requirements to Implementation
a.k.a.: The Monterey Workshop
Chicago, Sept. 24, 2003

Chess:
Center for Hybrid and Embedded Software Systems

Abstract

Most current software engineering is deeply rooted in procedural abstractions. Objects in object-oriented design present interfaces consisting principally of methods with type signatures. A method represents a transfer of the locus of control. Much of the talk of "models" in software engineering is about the static structure of object-oriented designs. However, essential properties of real-time systems, embedded systems, and distributed systems-of-systems are poorly defined by such interfaces and by static structure. These say little about concurrency, temporal properties, and assumptions and guarantees in the face of dynamic invocation.

Actor-oriented design contrasts with (and complements) object-oriented design by emphasizing concurrency and communication between components. Components called actors execute and communicate with other actors. While interfaces in object-oriented design (methods, principally) mediate transfer of the locus of control, interfaces in actor-oriented design (which we call ports) mediate communication. But the communication is not assumed to involve a transfer of control.

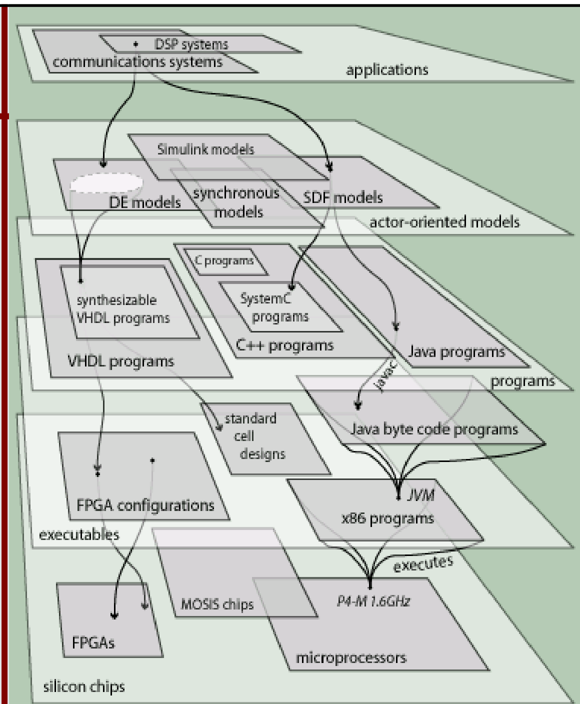
By focusing on the actor-oriented architecture of systems, we can leverage structure that is poorly described and expressed in procedural abstractions. Managing concurrency, for instance, is notoriously difficult using threads, mutexes and semaphores, and yet even these primitive mechanisms are extensions of procedural abstractions. In actor-oriented abstractions, these low-level mechanisms do not even rise to consciousness, forming instead the "assembly-level" mechanisms used to deliver much more sophisticated models of computation.

In this talk, I will outline the models of computation for actor-oriented design that look the most promising for embedded systems.

Platforms

A *platform* is a set of designs (the rectangles at the right, e.g., the set of all x86 binaries).

Model-based design is specification of designs in platforms with useful modeling properties (e.g., Simulink block diagrams for control systems).



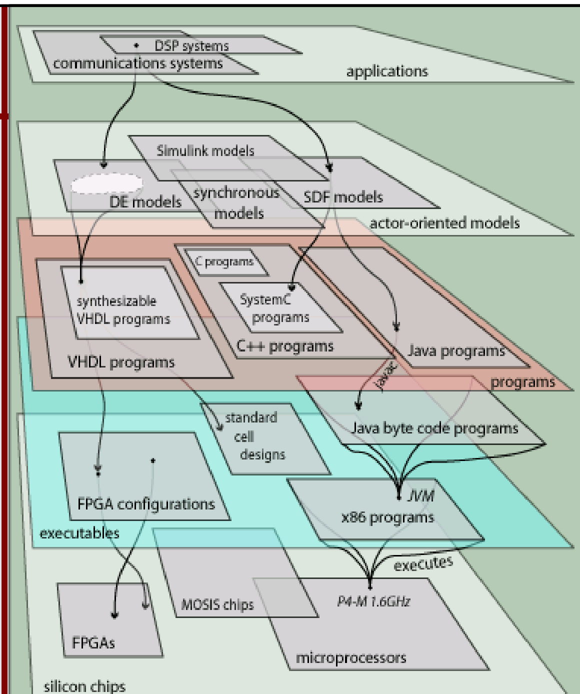
source: Edward A. Lee, UC Berkeley, 2003

Platforms

Where the Action Has Been:

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

Getting from red platforms to blue platforms.



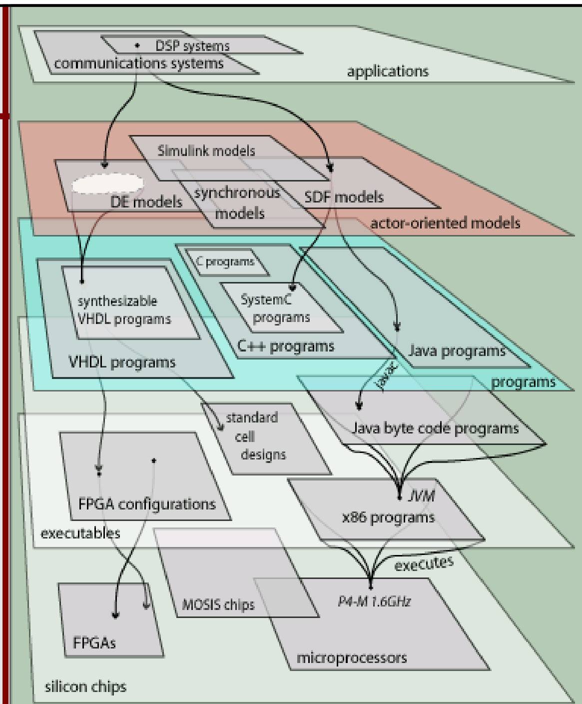
source: Edward A. Lee, UC Berkeley, 2003

Platforms

Where the Action Will Be:

Giving the red platforms useful modeling properties (via models of computation)

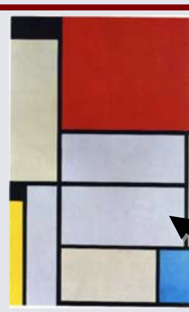
Getting from red platforms to blue platforms.



source: Edward A. Lee, UC Berkeley, 2003

Abstraction

How abstract a design is depends on how many refinement relations separate the design from one that is physically realizable.



Three paintings by Piet Mondrian

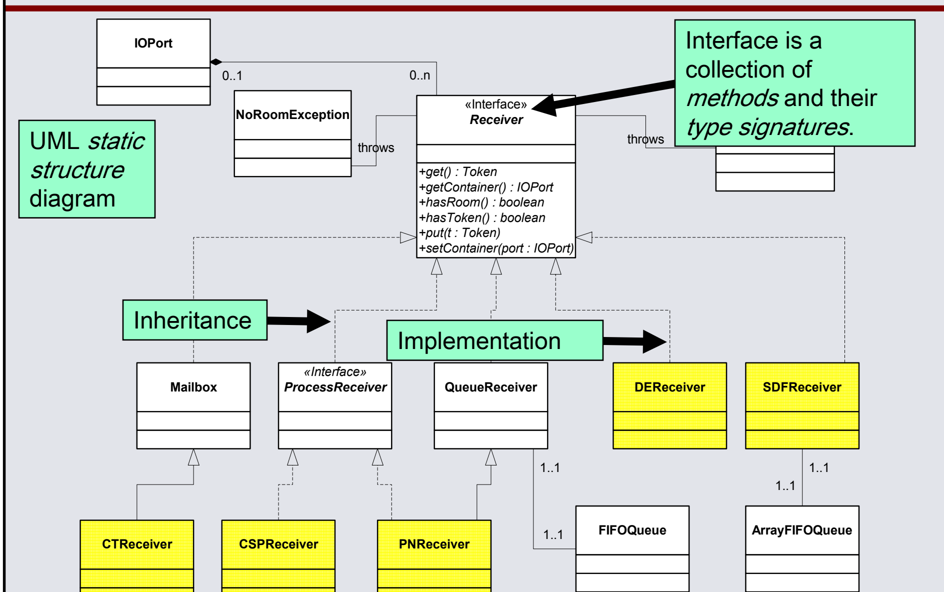
Design Framework

A *design framework* is a collection of platforms and *realizable relations* between platforms where at least one of the platforms is a set of *physically realizable designs*, and for any design in any platform, the transitive closure of the relations from that design includes at least one physically realizable design.

In *model-based design*, a *specification* is a point in a platform with useful modeling properties.

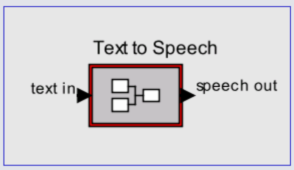
UML and MDA

Trying to Give Useful Modeling Properties to Object-Oriented Designs

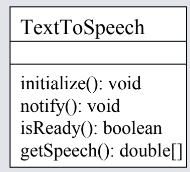


But These Are Fundamentally Rooted in a Procedural Abstraction

- Some Problems:
  - OO says little or nothing about concurrency and time ← Focus on this
  - Components implement low-level communication protocols
  - Distributed components are designed to fixed middleware APIs
  - Re-use potential is disappointing
- Some Partial Solutions
  - Adapter objects (laborious to design and deploy)
  - Model-driven architecture (still fundamentally OO)
  - Executable UML (little or no useful modeling properties)
- Our Solution: *Actor-Oriented Design*

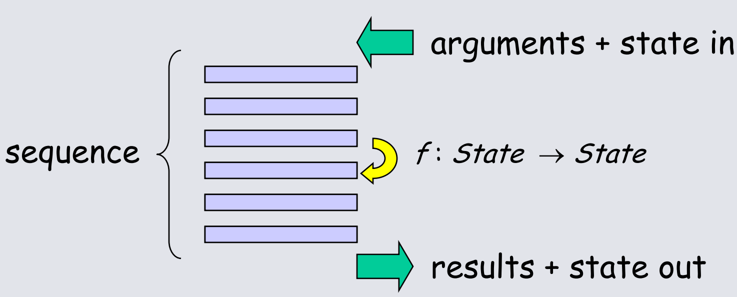


actor-oriented interface definition says "Give me text and I'll give you speech"



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

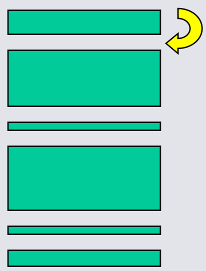
The Turing Abstraction of Computation



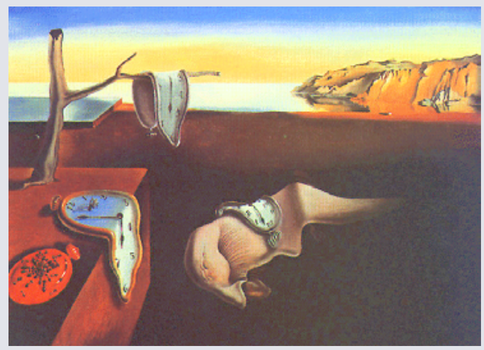
Everything "computable" can be given by a terminating sequential program.

Timing is Irrelevant

All we need is terminating sequences of state transformations! Simple mathematical structure: function composition.



$f : State \rightarrow State$

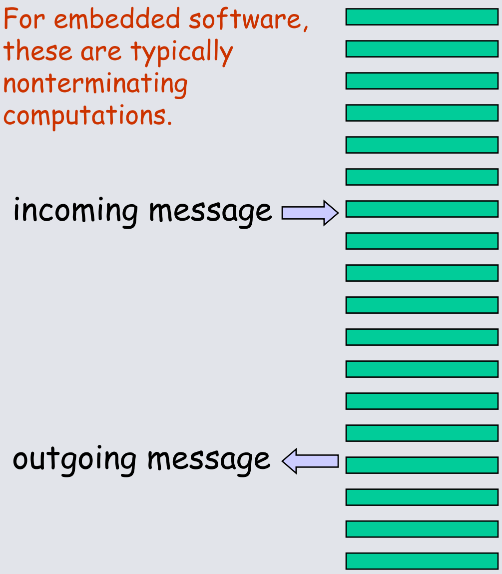


What about "real time"?

Make it faster!

Worse: Processes & Threads are a Terrible Way to Specify Concurrency

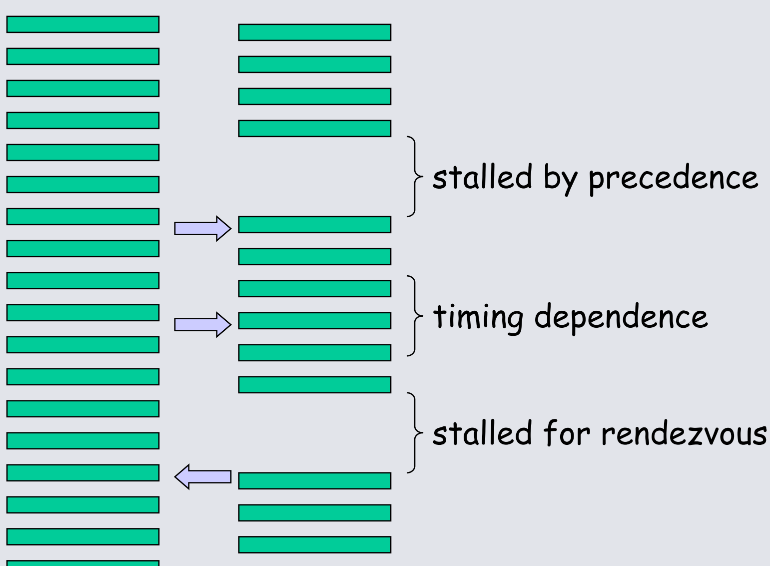
For embedded software, these are typically nonterminating computations.



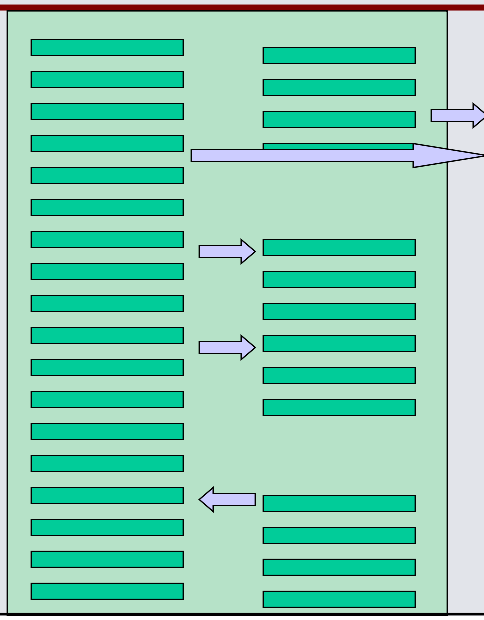
Infinite sequences of state transformations are called "processes" or "threads"

Their "interface" to the outside is a sequence of messages in or out.

Interacting Processes Impose Partial Ordering Constraints on Each Other

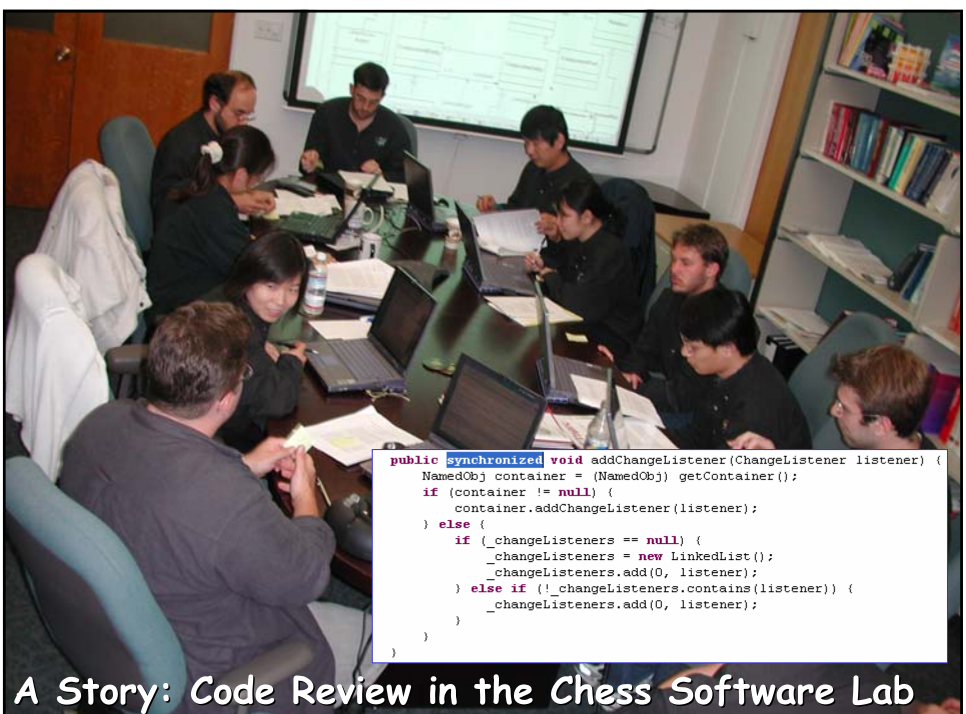


Interacting Processes Impose Partial Ordering Constraints on External Interactions



After composition: External interactions are no longer ordered.

An aggregation of processes is not a process. What is it?



```
public synchronized void addChangeListener(ChangeListener listener) {
    NamedObj container = (NamedObj) getContainer();
    if (container != null) {
        container.addChangeListener(listener);
    } else {
        if (_changeListeners == null) {
            _changeListeners = new LinkedList();
            _changeListeners.add(0, listener);
        } else if (!_changeListeners.contains(listener)) {
            _changeListeners.add(0, listener);
        }
    }
}
```

A Story: Code Review in the Chess Software Lab

Code Review in the Chess Software Lab
A Typical Story

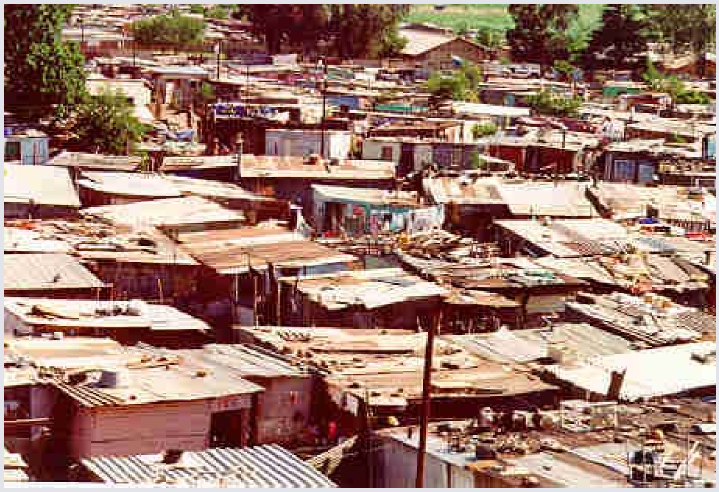
- Code review discovers that a method needs to be synchronized to ensure that multiple threads do not reverse each other's actions.
- No problems had been detected in 4 years of using the code.
- Three days after making the change, users started reporting deadlocks caused by the new mutex.
- Analysis of the deadlock takes weeks, and a correction is difficult.

```
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);
        }
    }
}
```


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.

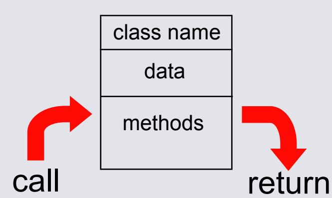
Threads, Mutexes, and Semaphores are a *Terrible* Basis for Concurrent Software Architectures



Ad hoc composition.

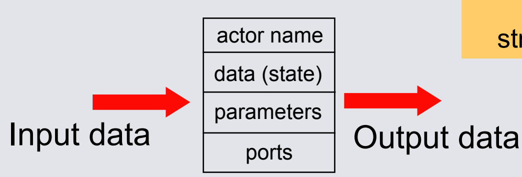
Focus on Actor-Oriented Design

- Object orientation:



What flows through an object is sequential control

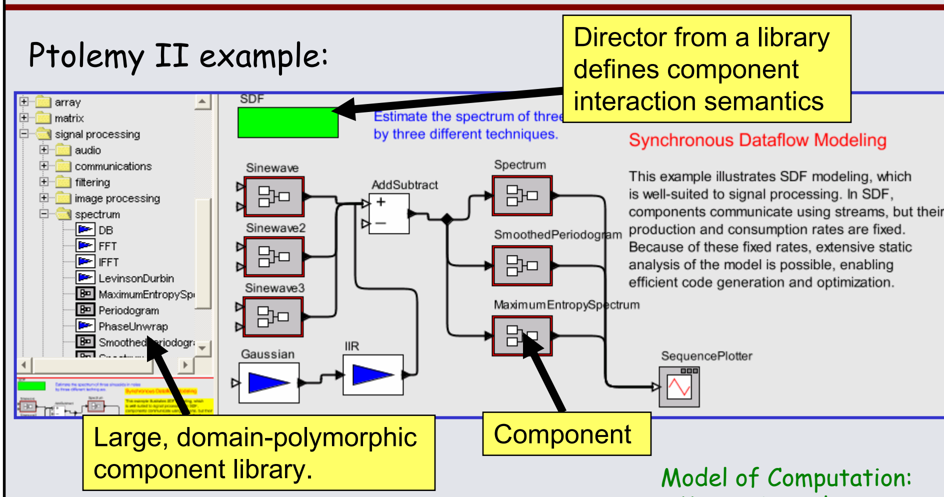
- Actor orientation:



What flows through an object is streams of data

Example of Actor-Oriented Design
(in this case, with a visual syntax)

Ptolemy II example:



Key idea: The model of computation is part of the framework within which components are embedded rather than part of the components themselves. Thus, components need to declare behavioral properties.

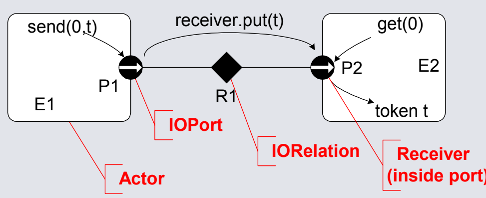
Model of Computation:
- Messaging schema
- Flow of control
- Concurrency

Examples of Actor-Oriented Component Frameworks

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

Actor View of Producer/Consumer Components

Basic Transport:



Models of Computation:

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

Many actor-oriented frameworks assume a producer/consumer metaphor for component interaction.

Actor Orientation vs. Object Orientation

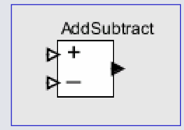
- Object Orientation
  - procedural interfaces
  - a class is a type (static structure)
  - type checking for composition
  - separation of interface from implementation
  - subtyping
  - polymorphism
- Actor Orientation
  - concurrent interfaces
  - a behavior is a type
  - type checking for composition of behaviors
  - separation of behavioral interface from implementation
  - behavioral subtyping
  - behavioral polymorphism

Focus on this

This is a vision of the future: Few actor-oriented frameworks fully offer this view. Eventually, all will.

Polymorphism

- Data polymorphism:
  - Add numbers (int, float, double, Complex)
  - Add strings (concatenation)
  - Add composite types (arrays, records, matrices)
  - Add user-defined types



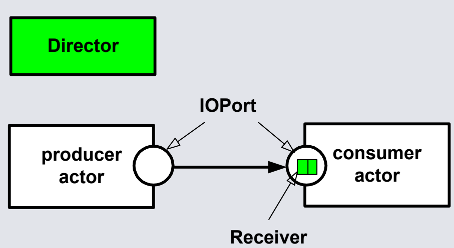
- Behavioral polymorphism:
  - In dataflow, add when all connected inputs have data
  - In a time-triggered model, add when the clock ticks
  - In discrete-event, add when any connected input has data, and add in zero time
  - In process networks, execute an infinite loop in a thread that blocks when reading empty inputs
  - In CSP, execute an infinite loop that performs rendezvous on input or output
  - In push/pull, ports are push or pull (declared or inferred) and behave accordingly
  - In real-time CORBA, priorities are associated with ports and a dispatcher determines when to add

By not choosing among these when defining the component, we get a huge increment in component re-usability. But how do we ensure that the component will work in all these circumstances?

Object-Oriented Approach to Achieving Behavioral Polymorphism

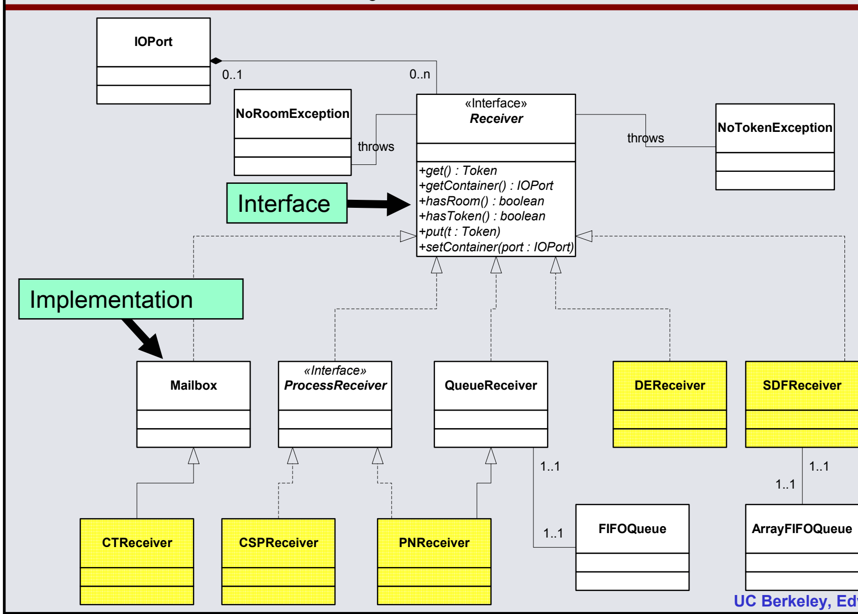
«Interface» ***Receiver***
+get() : Token *+getContainer() : IOPort* *+hasRoom() : boolean* *+hasToken() : boolean* *+put(t : Token)* *+setContainer(port : IOPort)*

These polymorphic methods implement the communication semantics of a domain in Ptolemy II. The receiver instance used in communication is supplied by the director, not by the component.



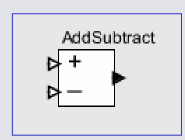
Recall: Behavioral polymorphism is the idea that components can be defined to operate with multiple models of computation and multiple middleware frameworks.

Behavioral Polymorphism
The Object Oriented View



But What If…

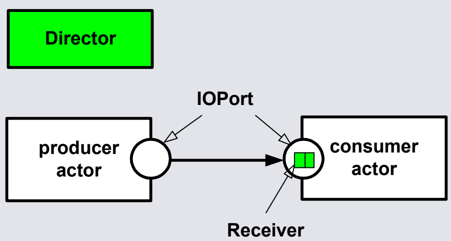
- The component requires data at all connected input ports?
- The component can only perform meaningful operations on two successive inputs?
- The component can produce meaningful output before the input is known (enabling it to break potential deadlocks)?
- The component has a mutex monitor with another component (e.g. to access a common hardware resource)?



None of these is expressed in the object-oriented interface definition, yet each can interfere with behavioral polymorphism.

Behavioral Types – A Practical Approach

- Capture the dynamic interaction of components in *types*
- Obtain benefits analogous to data typing.
- Call the result *behavioral types*.

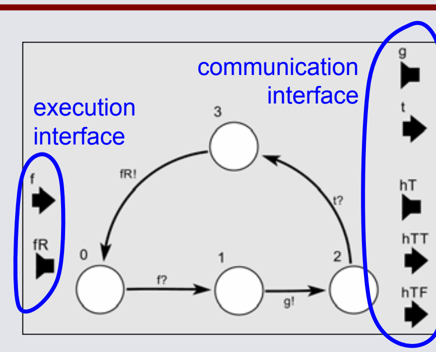


- Communication has
  - data types
  - behavioral types
- Components have
  - data type signatures
  - behavioral type signatures
- Components are
  - data polymorphic
  - behaviorally polymorphic

See Liskov & Wing, ACM, 1994 for an intro to behavioral types.

Behavioral Type System

- We capture patterns of component interaction in a type system framework.
- We describe interaction types and component behavior using extended *interface automata* (de Alfaro & Henzinger)
- We do type checking through *automata composition* (detect component incompatibilities)
- Subtyping order is given by the alternating simulation relation, supporting *behavioral polymorphism*.



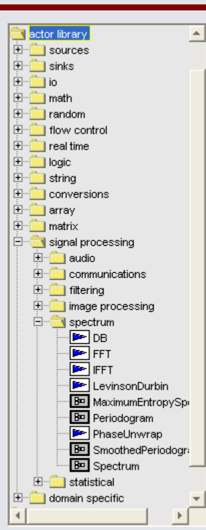
A type signature for a consumer actor.

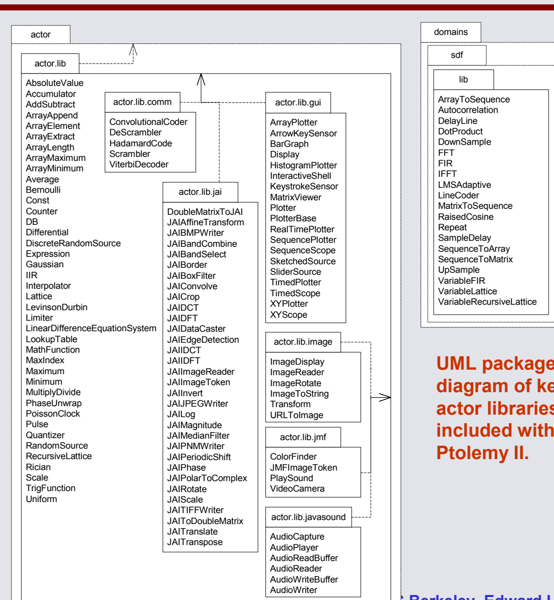
An alternative representation of behavioral types would be pre/post conditions, as in Liskov & Wing.

Enabled by a Behavioral Type System

- Checking behavioral compatibility of components that are composed.
- Checking behavioral compatibility of components and their frameworks.
- Behavioral subclassing enables interface/implementation separation.
- Helps with the definition of behaviorally-polymorphic components.

Enabled by Behavioral Polymorphism (1): More Re-Usable Component Libraries



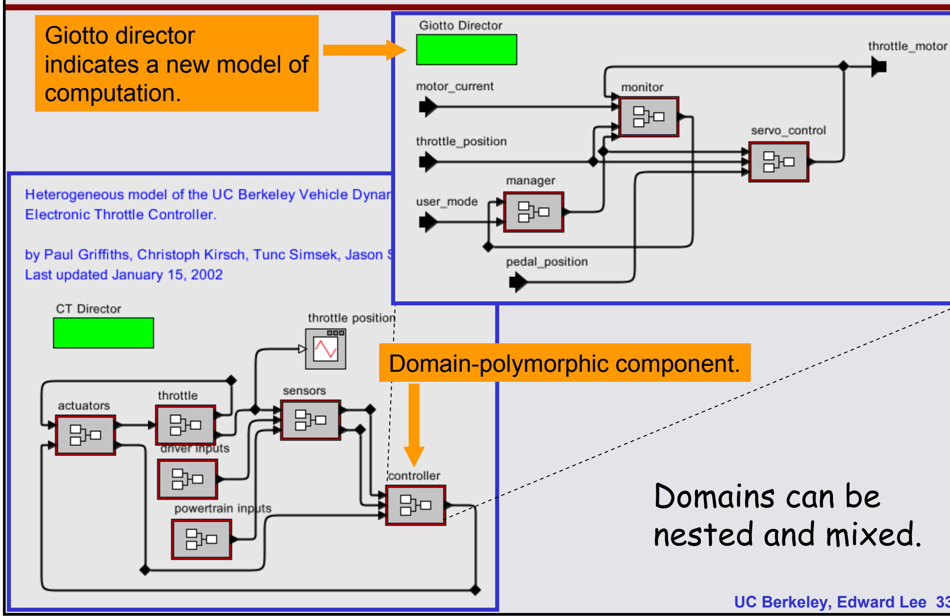


UML package diagram of key actor libraries included with Ptolemy II.

- **Data polymorphic components**
- **Domain polymorphic components**

Enabled by Behavioral Polymorphism (2): Hierarchical Heterogeneity

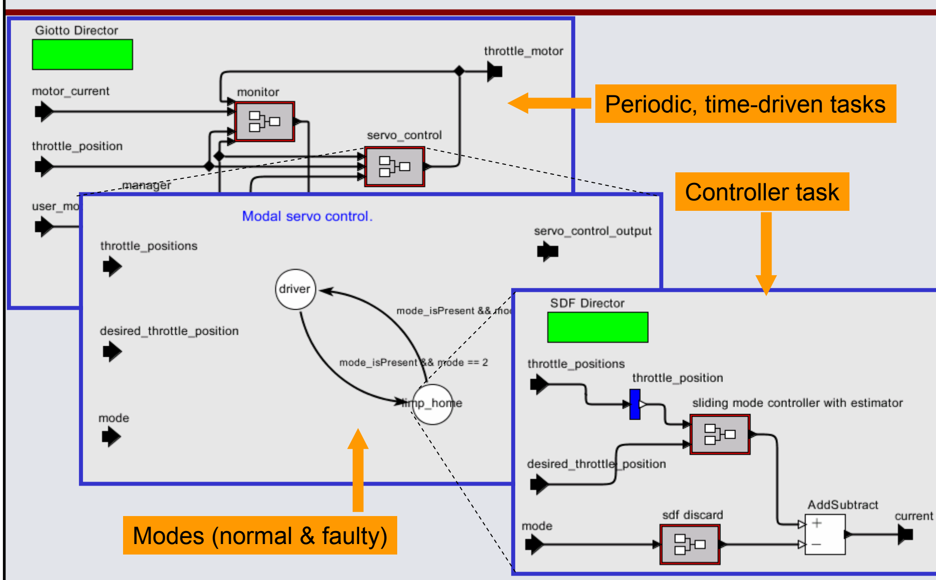
Giotto director indicates a new model of computation.



Domain-polymorphic component.

Domains can be nested and mixed.

Enabled by Behavioral Polymorphism (3): Modal Models



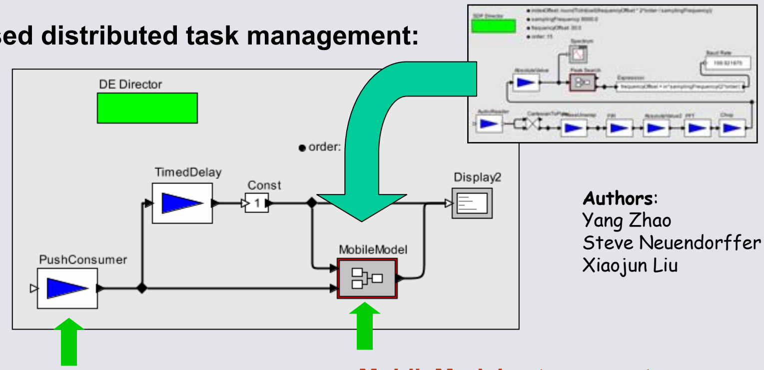
Periodic, time-driven tasks

Controller task

Modes (normal & faulty)

Enabled by Behavioral Polymorphism (4): Mobile Models

Model-based distributed task management:



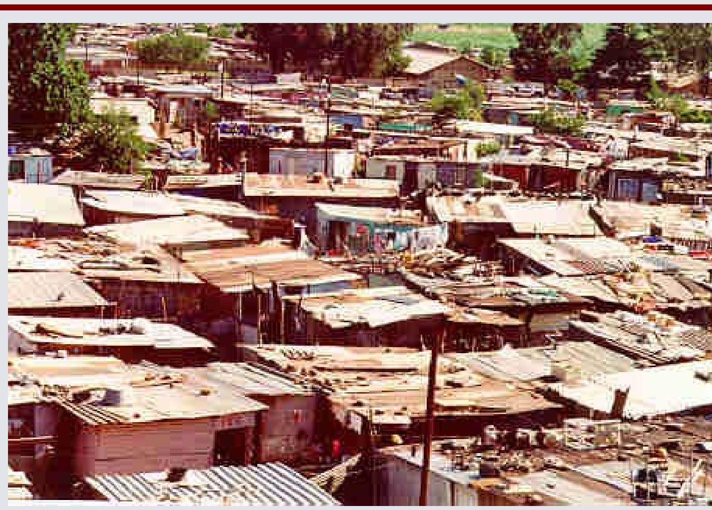
Authors:
Yang Zhao
Steve Neuendorffer
Xiaojun Liu

PushConsumer actor receives pushed data provided via CORBA, where the data is an XML model of a signal analysis algorithm.

MobileModel actor accepts a StringToken containing an XML description of a model. It then executes that model on a stream of input data.

Data *and* domain type safety will help make such models secure

Will Model-Based Design Yield Better Designs?



What we are trying to replace: Today's software architecture.

Will Model-Based Design Yield Better Designs?

"Why isn't the answer XML, or UML, or IP, or something like that?"

Direct quote for a high-ranking decision maker at a large embedded systems company with global reach.

The Box, Eric Owen Moss

Source: *Contemporary California Architects*, P. Jodidio, Taschen, 1995

Mandating use of the wrong platform is far worse than tolerating the use of multiple platforms.

Better Architecture is Enabled but not Guaranteed by Actor-Oriented Design

Two Rodeo Drive, Kaplan, McLaughlin, Diaz

Source: *Kaplan McLaughlin Diaz*, R. Rappaport, Rockport, 1998

- Understandable concurrency
- Systematic heterogeneity (enabled by behavioral polymorphism)
- More re-usable component libraries

Conclusion – What to Remember

- Actor-oriented design
  - concurrent components interacting via ports
- Models of computation
  - principles of component interaction
- Understandable concurrency
  - compositional models
- Behavioral types
  - a practical approach to verification and interface definition
- Behavioral polymorphism
  - defining components for use in multiple contexts

http://ptolemy.eecs.berkeley.edu
http://chess.eecs.berkeley.edu