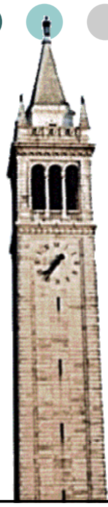


Actor-Oriented Design:
A focus on domain-specific languages for embedded systems

Edward A. Lee
Professor, UC Berkeley
Director, Center for Hybrid and Embedded Software Systems (CHESS)

Formal Methods and Models for Codesign (MEMOCODE'2004)
June 22-25, 2004
San Diego, California

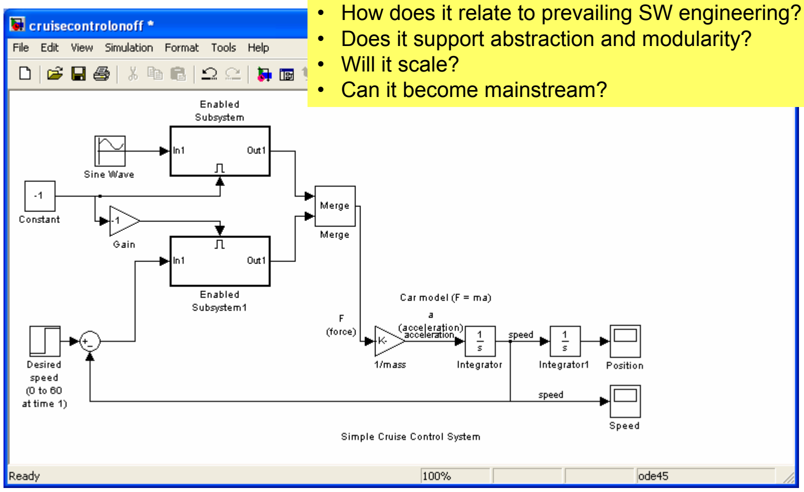
Abstract

Concurrent, domain-specific languages such as Simulink, LabVIEW, Modelica, VHDL, SystemC, and OPNET provide modularization mechanisms that are significantly different from those in prevailing object-oriented languages such as C++ and Java. In these languages, components are concurrent objects that communicate via messaging, rather than abstract data structures that interact via procedure calls. Although the concurrency and communication semantics differ considerably between languages, they share enough common features that we consider them to be a family. We call them *actor-oriented* languages.

Actor-oriented languages, like object-oriented languages, are about modularity of software. I will argue that we can adapt for actor-oriented languages many (if not all) of the innovations of OO design, including concepts such as the separation of interface from implementation, strong typing of interfaces, subtyping, classes, inheritance, and aspects. I will show some preliminary implementations of these mechanisms in a Berkeley system called Ptolemy II.

The Questions

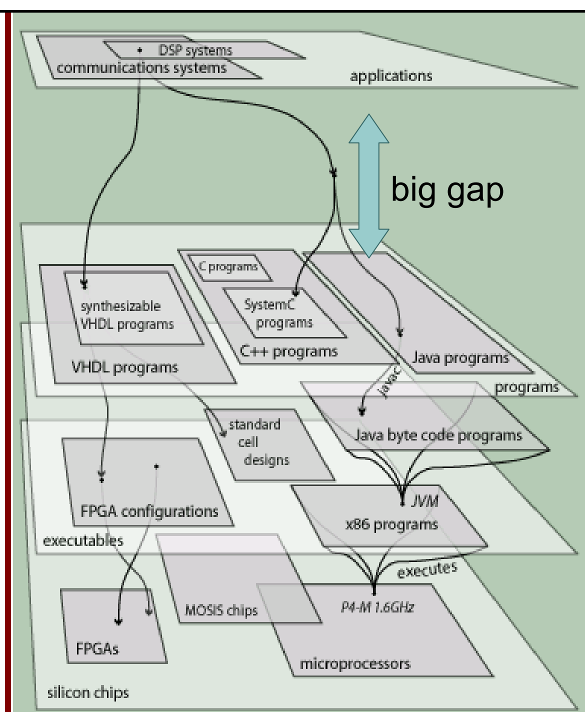
- Is this a good way to do design?
- How does it relate to prevailing SW engineering?
- Does it support abstraction and modularity?
- Will it scale?
- Can it become mainstream?



Platforms

A *platform* is a set of designs.

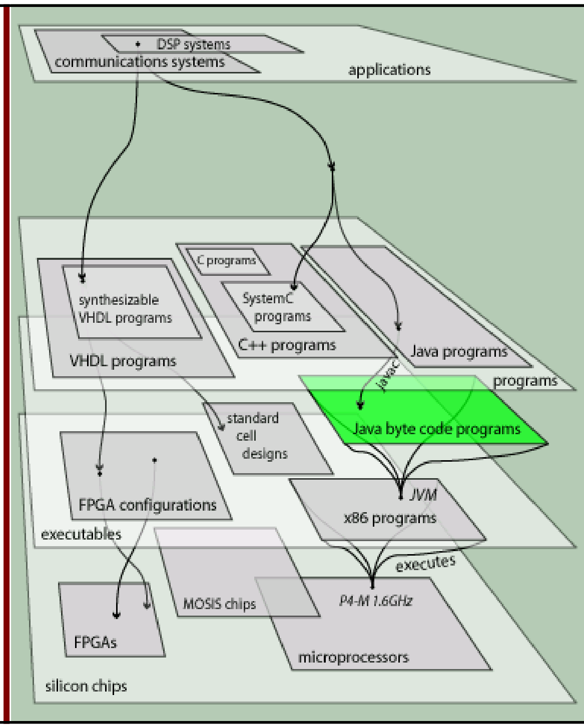
Relations between platforms represent design processes.



Progress

Many useful technical developments amount to creation of new platforms.

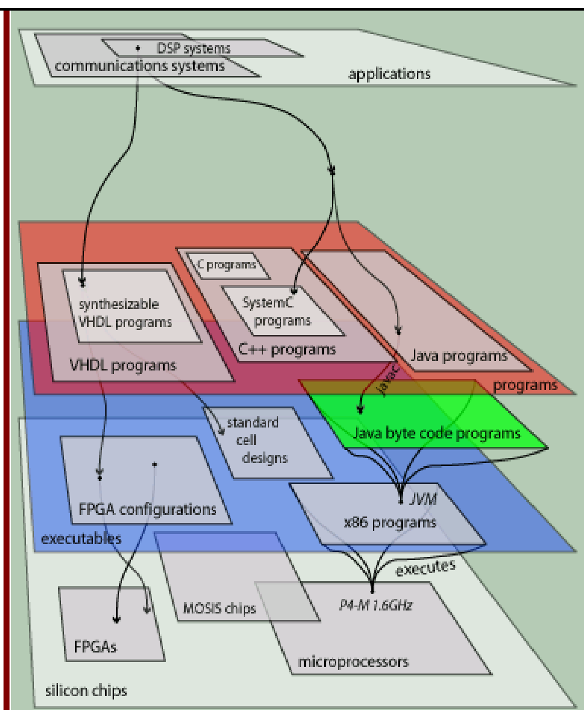
- microarchitectures
- operating systems
- virtual machines
- processor cores
- configurable ISAs



Recent Action

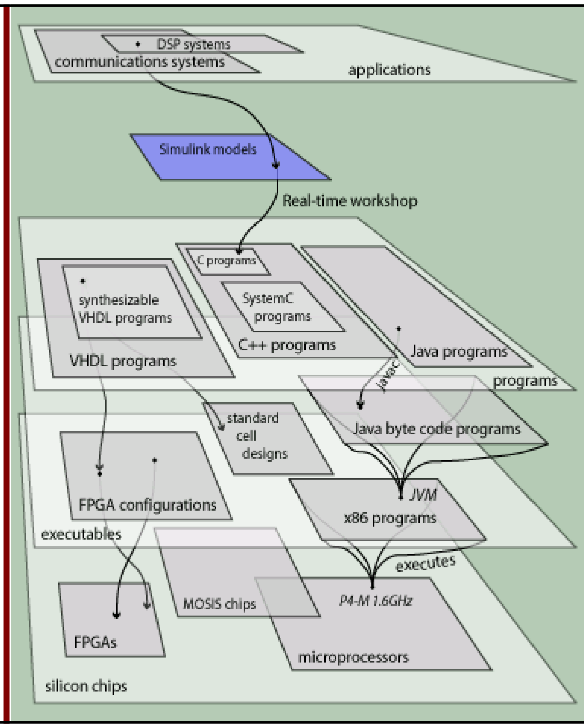
Giving the red platforms useful modeling properties (e.g. verification, SystemC, UML, MDA)

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.

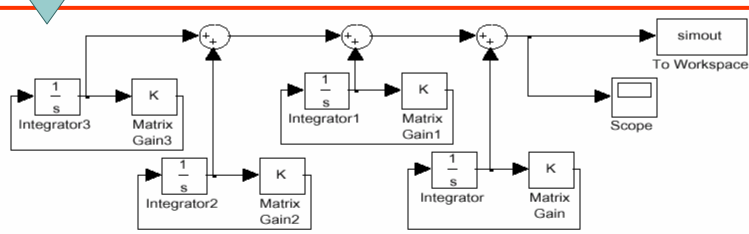


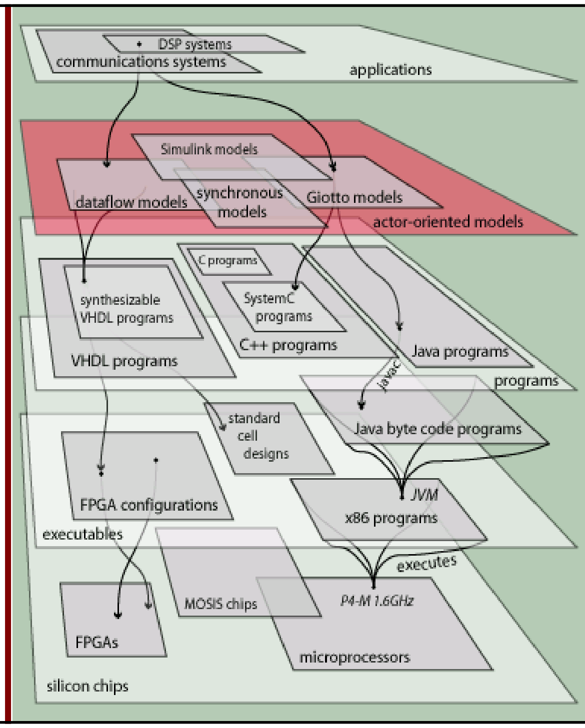
Figure C.12: A block diagram generating a plucked string sound with a fundmental and three harmonics.

From below: Synchronous concurrent composition of components

Actor-Oriented Platforms

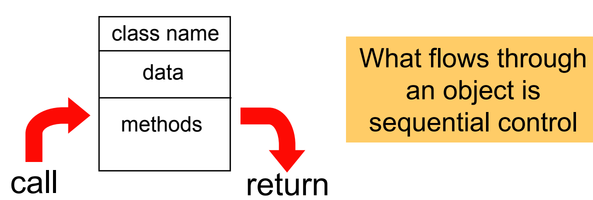
Actor oriented models compose concurrent components according to a model of computation.

Time and concurrency become key parts of the programming model.

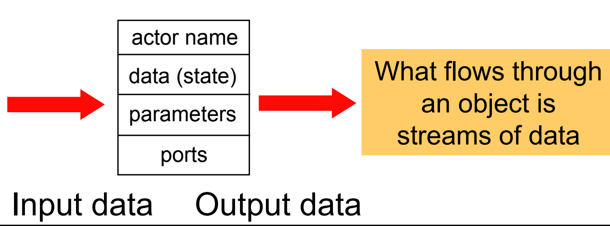


Actor-Oriented Design

Object orientation:



Actor orientation:



Actor Orientation vs. Object Orientation

Object oriented

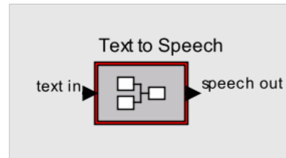
TextToSpeech

---

initialize(): void
notify(): void
isReady(): boolean
getSpeech(): double[]

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

Actor oriented



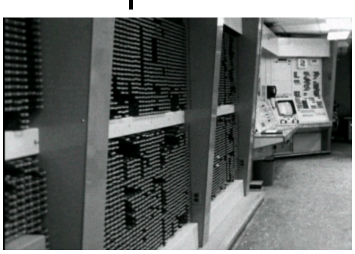
actor-oriented interface definition says "Give me text and I'll give you speech"

- Identified limitations of 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

The First (?) Actor-Oriented Programming Language

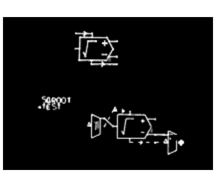
The On-Line Graphical Specification of Computer Procedures

W. R. Sutherland, Ph.D. Thesis, MIT, 1966



MIT Lincoln Labs TX-2 Computer

Bert Sutherland with a light pen



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

Bert Sutherland used the first acknowledged object-oriented framework (Sketchpad, created by his brother, Ivan Sutherland) to create the first actor-oriented programming framework.

Your Speaker in 1966

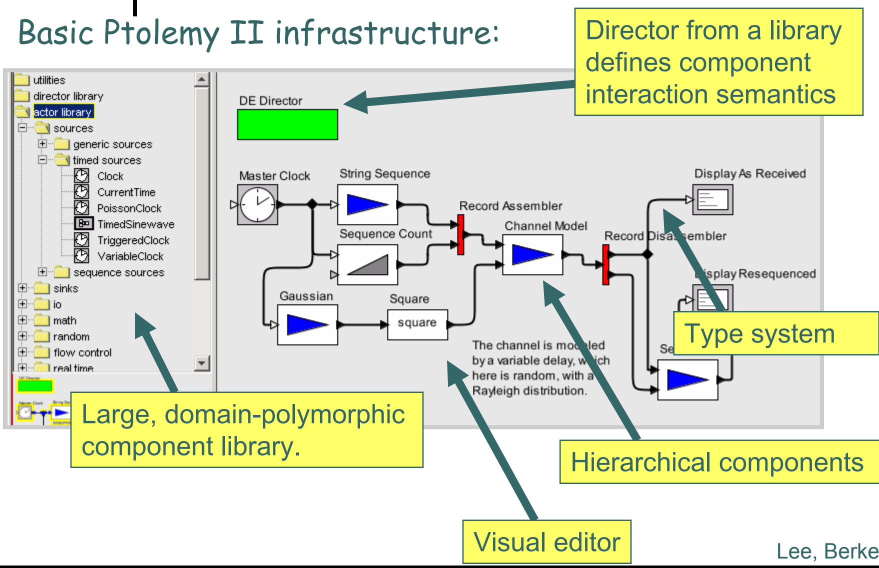
Modern Examples of Actor-Oriented Component Frameworks

- Simulink (The MathWorks)
- Labview (National Instruments)
- Modelica (Linkoping)
- OPNET (Opnet Technologies)
- Polis & Metropolis (UC Berkeley)
- Gabriel, Ptolemy, and Ptolemy II (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)
- …

Except Ptolemy, all of these define a fixed model of computation.

Ptolemy II Framework for Experimenting with AO Design

Basic Ptolemy II infrastructure:



Actors in 2004: “Capsules” (UML-RT) and “Composite Structures” (UML-2)

- UML-RT borrowed from Selic’s ROOM the notion of “capsules,” which structurally look like actors.
- UML-2 is introducing the notion of “composite structures,” which also look like actors.
- UML capsules and composite structures specify abstract syntax (and a concrete syntax), but no semantics.
- What this says is that there is huge potential for actor-oriented design to be done wrong…

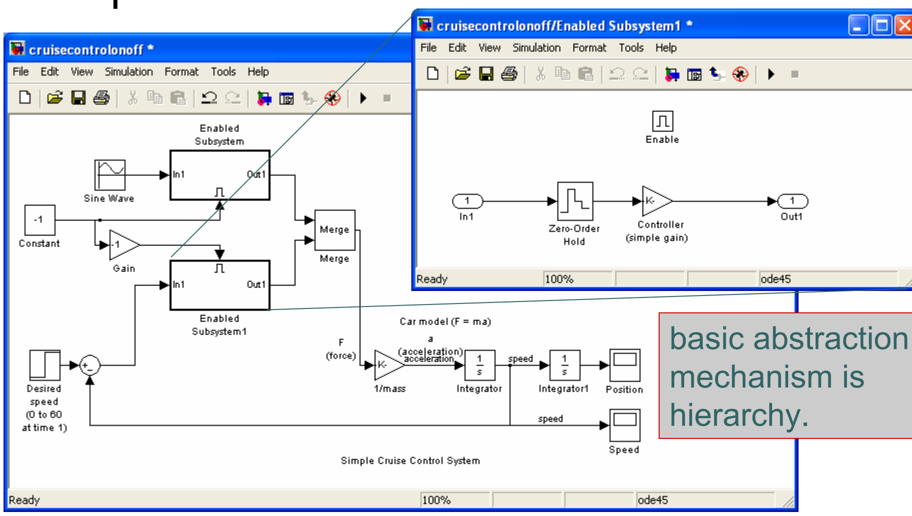
Why Use the Term "Actors"

- 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.
- But UML uses the term "actor" in its use cases.

Does Actor-Oriented Design Offer Best-Of-Class SW Engineering Methods?

- Abstraction
  - procedures/methods
  - classes
- Modularity
  - subclasses
  - inheritance
  - interfaces
  - polymorphism
  - aspects
- Correctness
  - type systems

Example of an Actor-Oriented Framework: Simulink



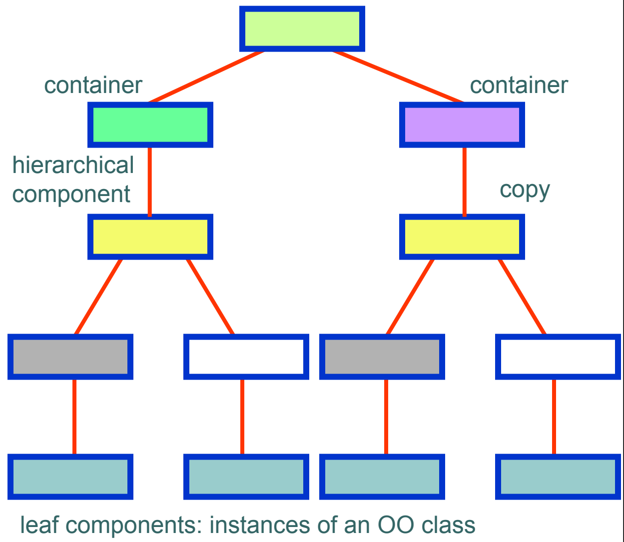
basic abstraction mechanism is hierarchy.

Observation

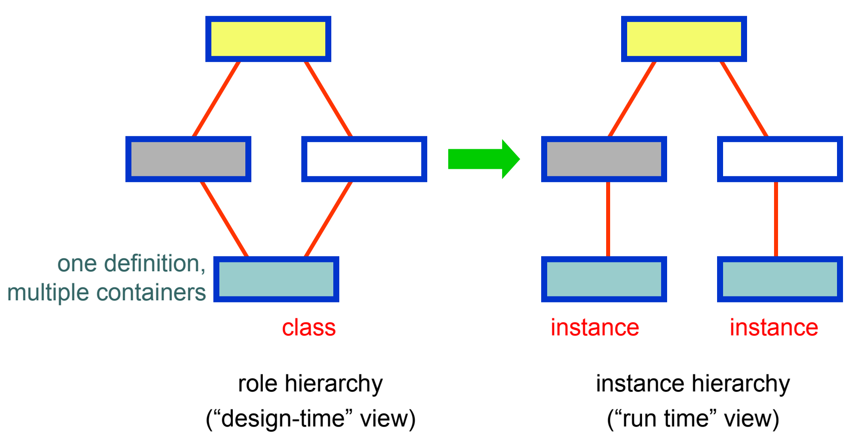
By itself, hierarchy is a very weak abstraction mechanism.

Tree Structured Hierarchy

- Does not represent common *class* definitions. Only instances.
- Multiple instances of the same hierarchical component are *copies*.

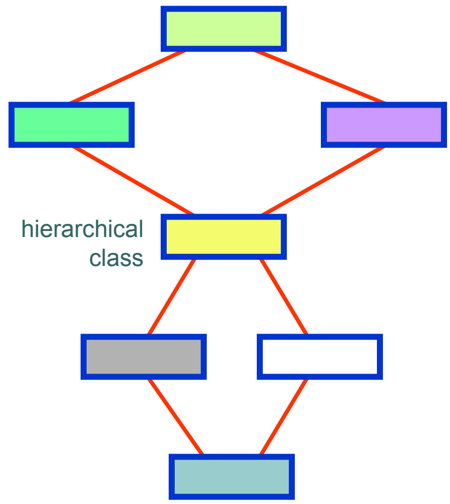


Alternative Hierarchy: Roles and Instances

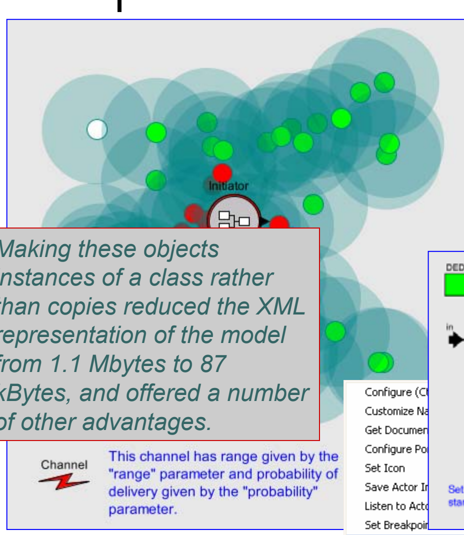


Role Hierarchy

- Multiple instances of the same hierarchical component are represented by *classes* with multiple containers.
- This makes hierarchical components more like leaf components.



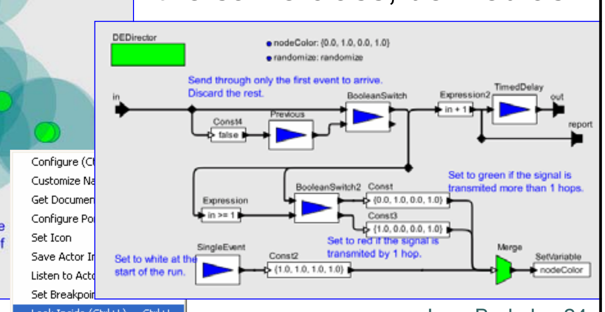
A Motivating Application: Modeling Sensor Networks



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.

Model of Massimo Franceschetti's "small world" phenomenon with 49 sensor nodes.

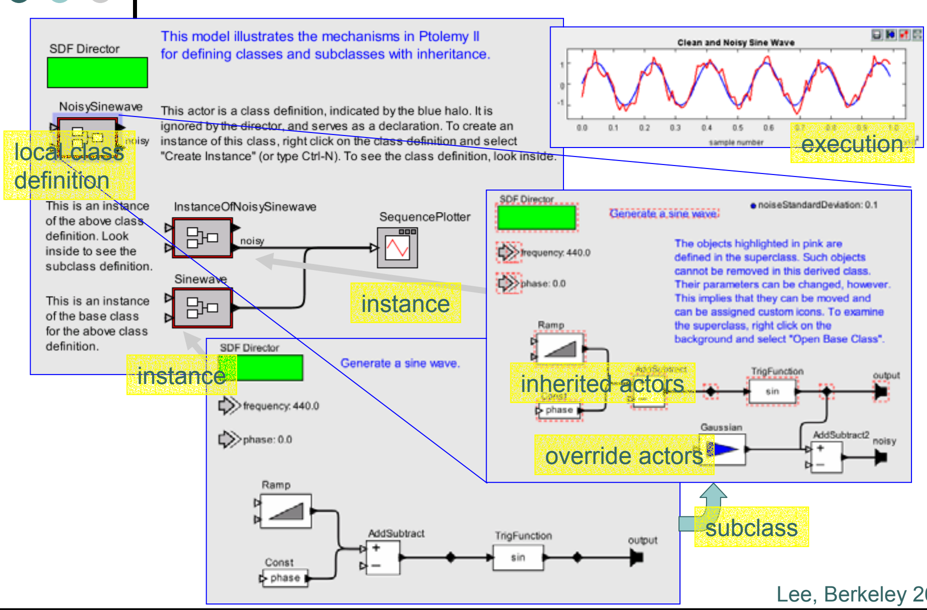
These 49 sensor nodes are actors that are instances of the same class, defined as:



Subclasses, Inheritance? Interfaces, Subtypes? Aspects?

- Now that we have classes, can we bring in more of the modern programming world?
  - subclasses?
  - inheritance?
  - interfaces?
  - subtypes?
  - aspects?

Example Using AO Classes

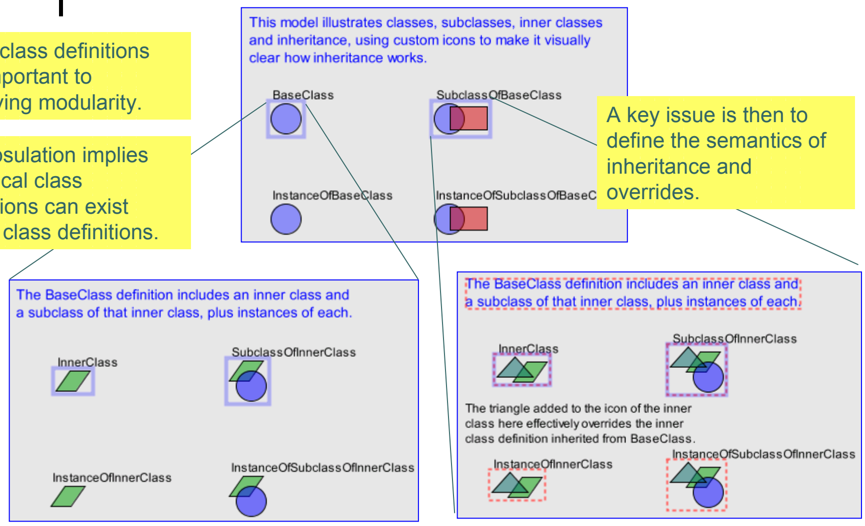


Inner Classes

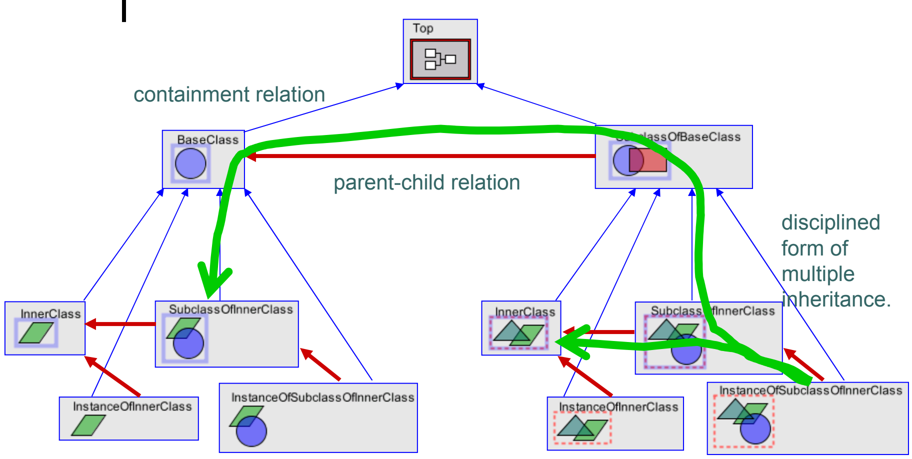
Local class definitions are important to achieving modularity.

Encapsulation implies that local class definitions can exist within class definitions.

A key issue is then to define the semantics of inheritance and overrides.

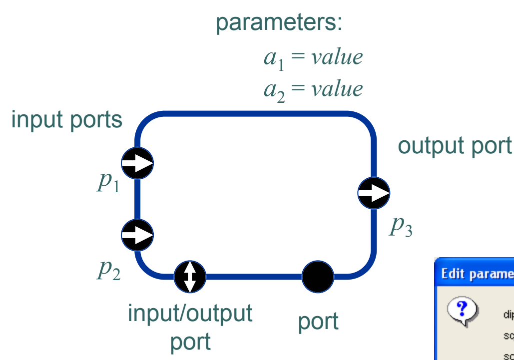


Ordering Relations

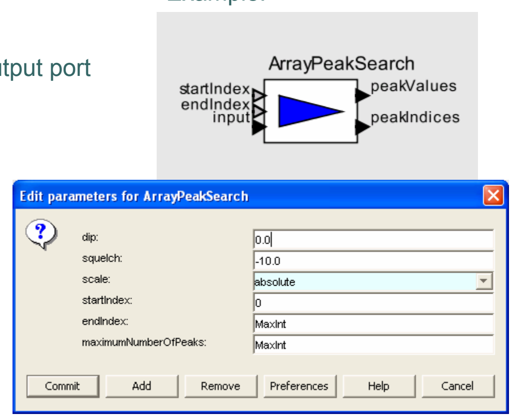


Mathematically, this structure is a *doubly-nested diposet*, the formal properties of which help to define a clean inheritance semantics. The principle we follow is that *local* changes override *global* changes.

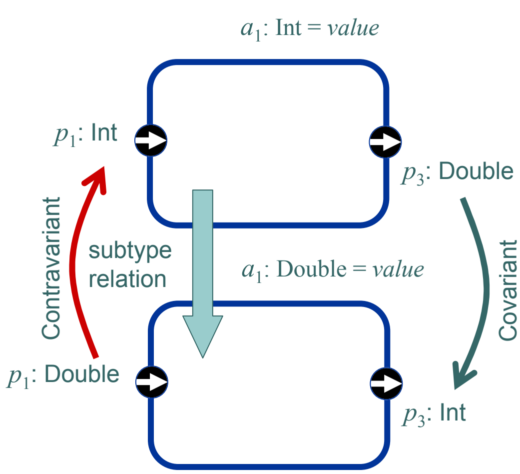
Defining Actor Interfaces: Ports and Parameters



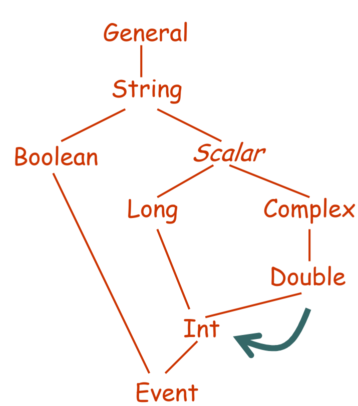
Example:



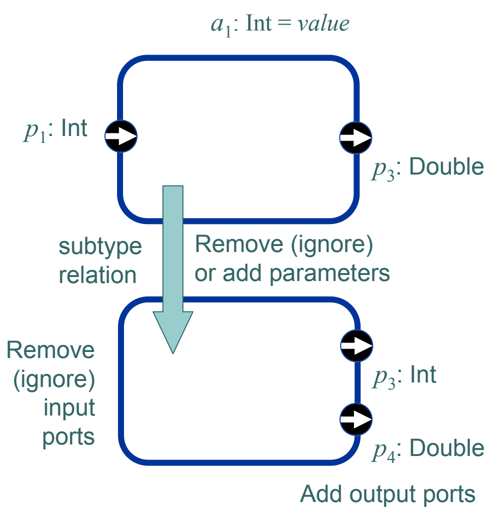
Actor Subtypes



Example of a simple type lattice:



Actor Subtypes (cont)



Subtypes can have:

- Fewer input ports
- More output ports

Of course, the types of these can have co/contravariant relationships with the supertype.

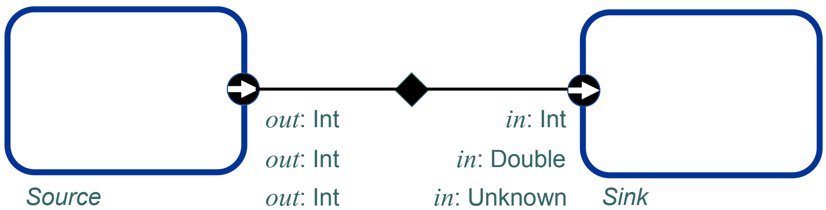
Observations

- Subtypes can remove (or ignore) parameters and also add new parameters because parameters always have a default value (unlike inputs, which a subtype cannot add)
- Subtypes cannot modify the types of parameters (unlike ports). Co/contravariant at the same time.
- PortParameters are ports with default values. They can be removed or added just like parameters because they provide default values.
- Are there similar exceptions to co/contravariance in OO languages?

Composing Actors

A connection implies a type constraint. Can:

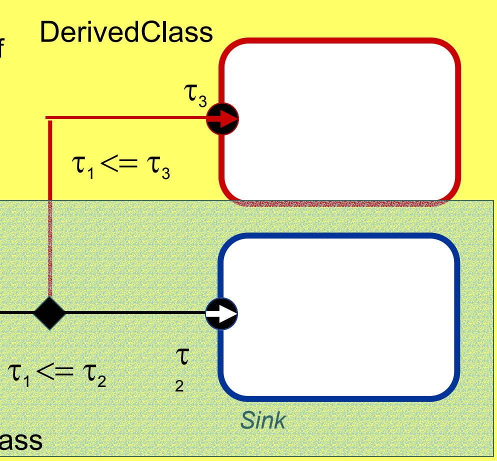
- check compatibility
- perform conversions
- infer types



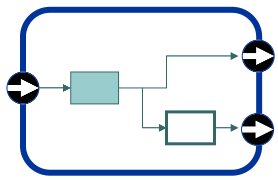
The Ptolemy II type system does all three.

What Happens to Type Constraints When a Subclass Adds Connections?

Type resolution results may be different in different subclasses of the same base class (connection with let-bound variables in a Hindley-Milner type system?)



Abstract Actors?



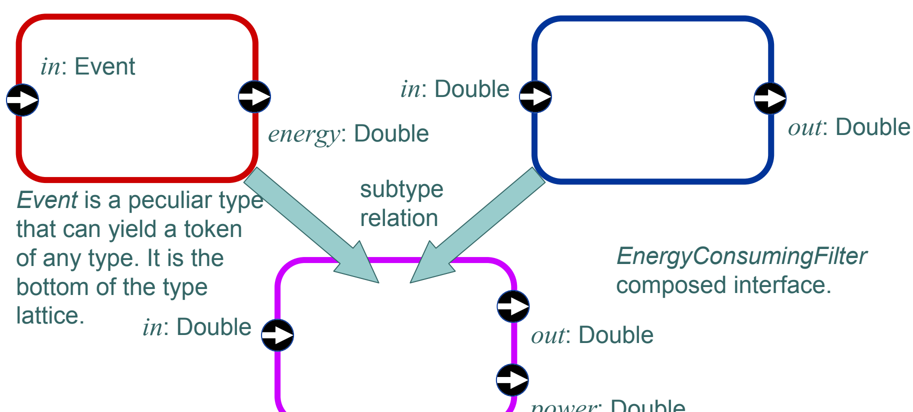
Suppose one of the contained actors is an interface only. Such a class definition cannot be instantiated (it is abstract). Concrete subclasses would have to provide implementations for the interface.

Is this useful?

Implementing Multiple Interfaces An Example

EnergyConsumer interface has a single output port that produces a Double representing the energy consumed by a firing.

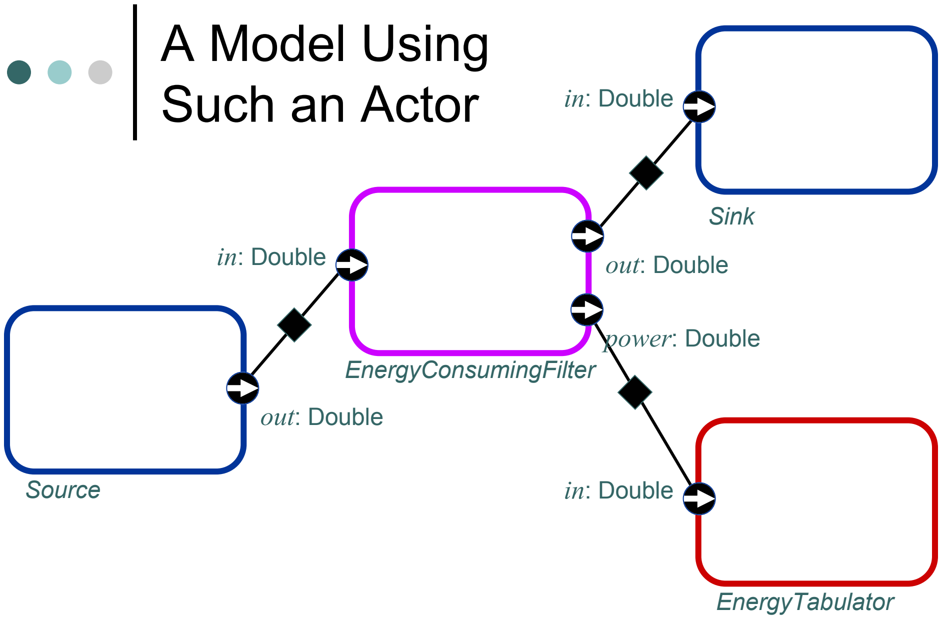
Filter interface for a stream transformer component.



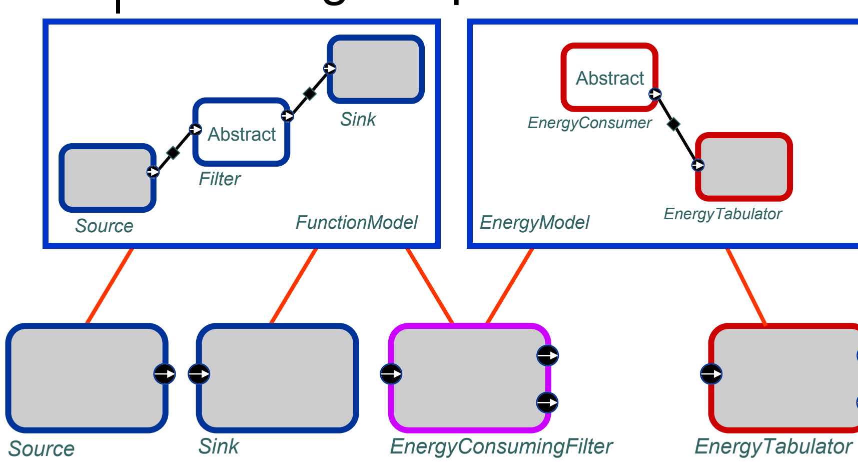
Event is a peculiar type that can yield a token of any type. It is the bottom of the type lattice.

EnergyConsumingFilter composed interface.

A Model Using Such an Actor



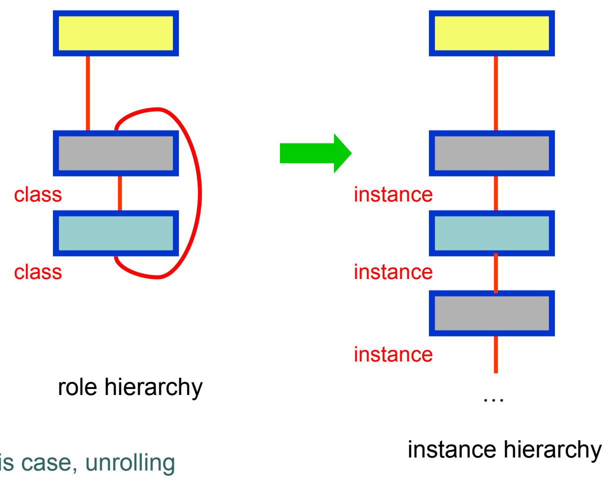
Heterarchy? Multi-View Modeling? Aspects?



This is *multi-view modeling*, similar to what GME (Vanderbilt) can do.

Is this an *actor-oriented* version of *aspect-oriented* programming?

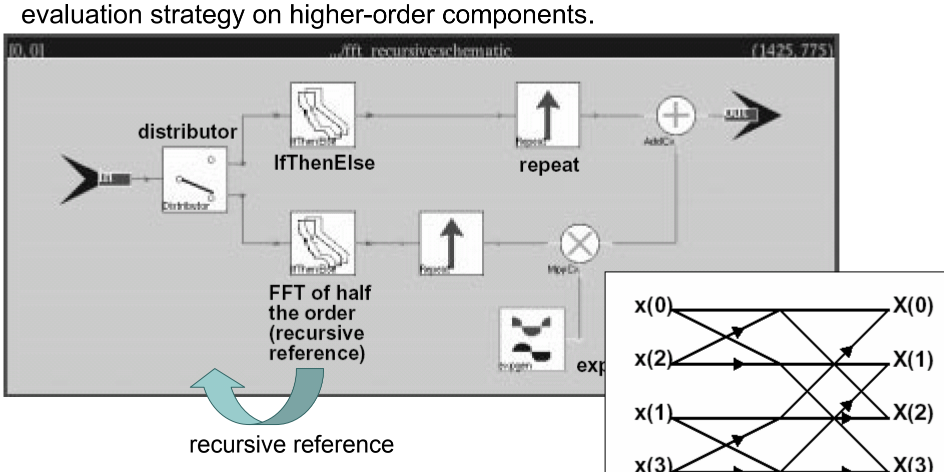
Recursive Containment
Can Hierarchical Classes Contain Instances of Themselves?



Note that in this case, unrolling cannot occur at "compile time".

Early Realization of this in Ptolemy Classic

FFT implementation in Ptolemy Classic (1995) used a partial evaluation strategy on higher-order components.



Conclusion

- Actor-oriented design remains a relatively immature area, but one that is progressing rapidly.
- It has huge potential.
- Many questions remain…