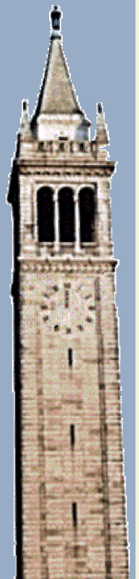
Behavioral Types for Actor-Oriented Design

Edward A. Lee
with special thanks to:
Luca de Alfaro, Tom Henzinger, and Yuhong Xiong

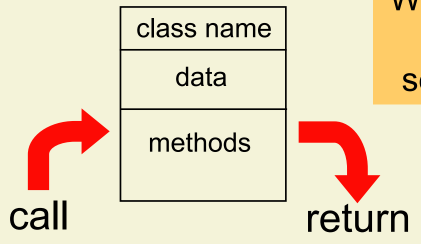
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley

Invited talk, FMCAD, Fourth International Conference on
Formal Methods in Computer-Aided Design, November 6-8, 2002, Portland, Oregon



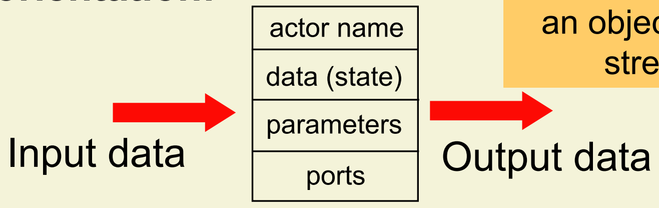
Actor-Oriented Design

- Object orientation:



What flows through an object is sequential control

- Actor orientation:



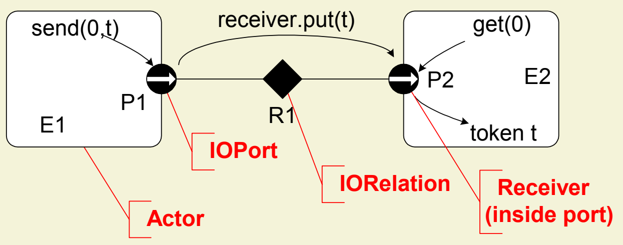
What flows through an object is data streams

Examples of Actor-Oriented Component Frameworks

- Simulink (The MathWorks)
- Labview (National Instruments)
- OCP, open control platform (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:



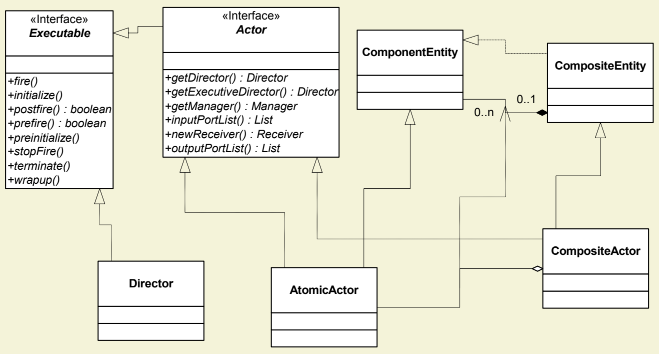
Models of Computation:

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

Key idea: The *model of computation* defines the component interaction patterns and is part of the framework, not part of the components themselves.

Contrast with Object Orientation

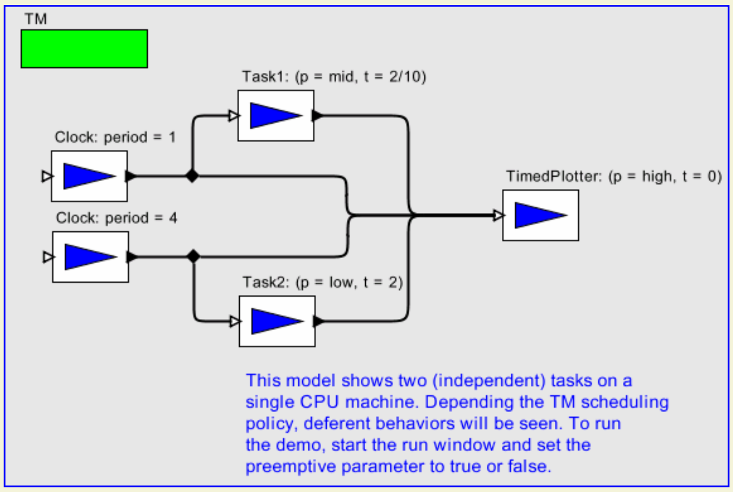
- Call/return imperative semantics
- Concurrency is realized by ad-hoc calling conventions
- Concurrent patterns are supported by futures, proxies, monitors, and semaphores



Object orientation emphasizes inheritance and procedural interfaces.

Actor orientation emphasizes concurrency and communication abstractions.

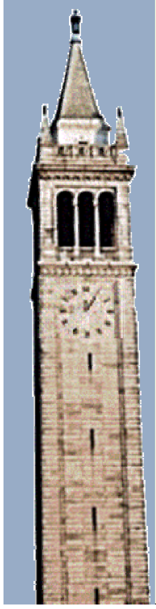
Actor Orientation with a Visual Syntax



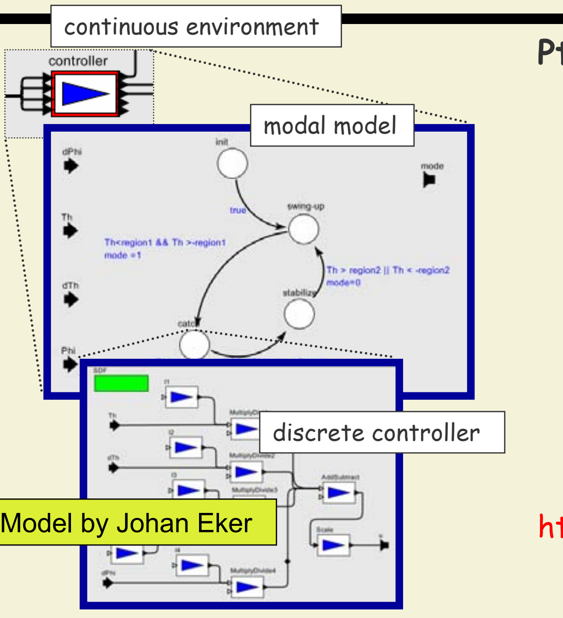
This model shows two (independent) tasks on a single CPU machine. Depending the TM scheduling policy, deferent behaviors will be seen. To run the demo, start the run window and set the preemptive parameter to true or false.

Model by Jie Liu

Actor-oriented model of two real-time control systems sharing a single CPU under a priority-driven RTOS scheduler.



Our Evolving Software Laboratory

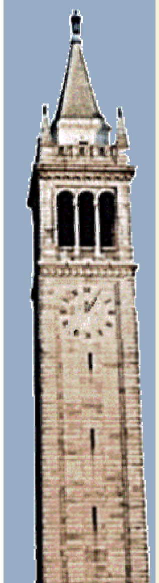


example Ptolemy model: hybrid control system

Ptolemy II:

A framework supporting experimentation with actor-oriented design, concurrent semantics, and visual syntaxes.

http://ptolemy.eecs.berkeley.edu



Realization of a Model of Computation is a "Domain" in Ptolemy II

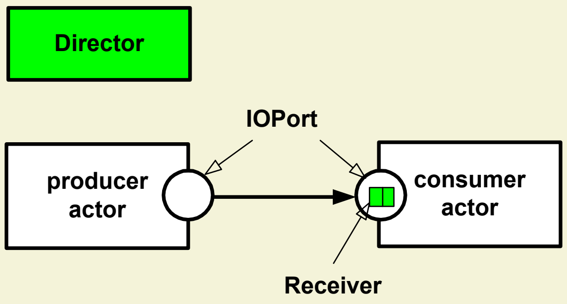
- The "laws of physics" of component interaction
  - communication semantics
  - flow of control constraints

In astrophysics: a "domain" is a region of the universe where a certain set of "laws of physics" applies.

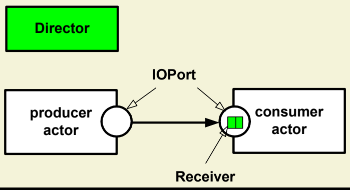
- Multiple domains may be combined hierarchically
  - depends on the concept of "domain polymorphism"

Ptolemy II Domains

- Define the flow(s) of control
  - "execution model"
  - Realized by a *Director* class
- Define communication between components
  - "interaction model"
  - Realized by a *Receiver* class

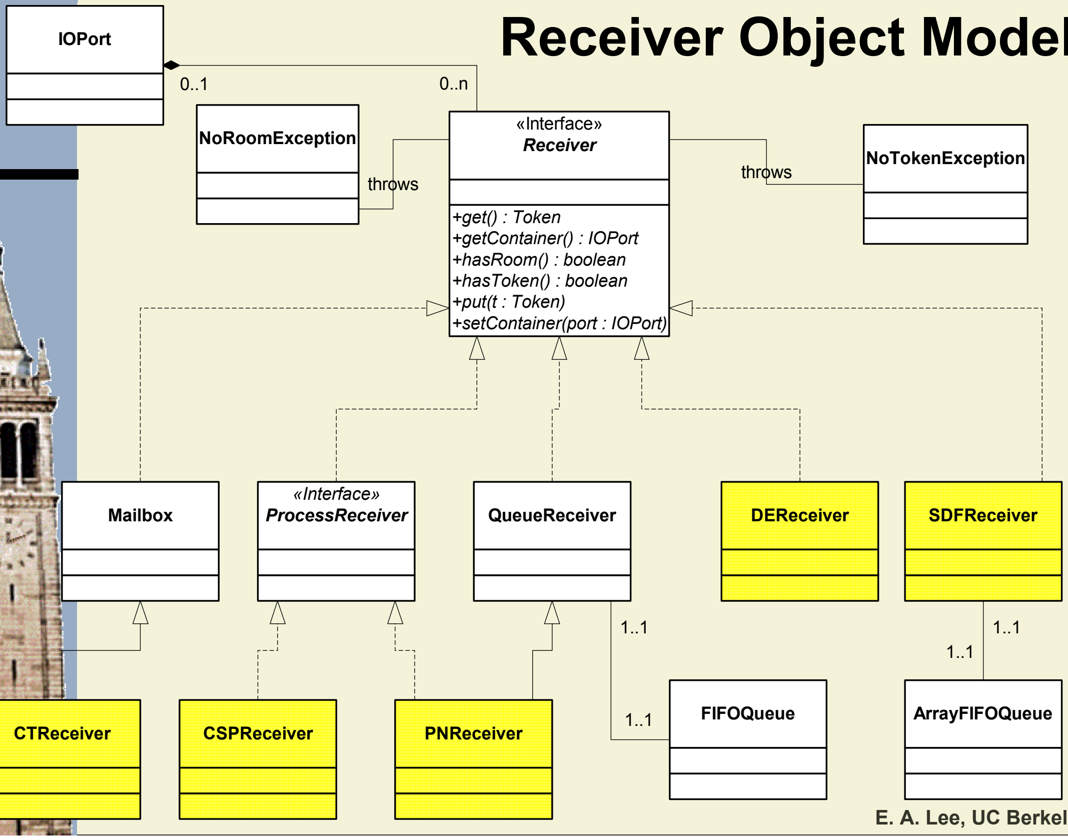


Example Domains



- **Communicating Sequential Processes (CSP):** rendezvous-style communication
- **Process Networks (PN):** asynchronous communication, determinism
- **Synchronous Data Flow (SDF):** stream-based communication, statically scheduled
- **Discrete Event (DE):** event-based communication
- **Synchronous/Reactive (SR):** synchronous, fixed point semantics
- **Time Driven (Giotto):** synchronous, time-driven multitasking
- **Timed Multitasking (TM):** priority-driven multitasking, deterministic communication
- **Continuous Time (CT):** numerical differential equation solver

Receiver Object Model

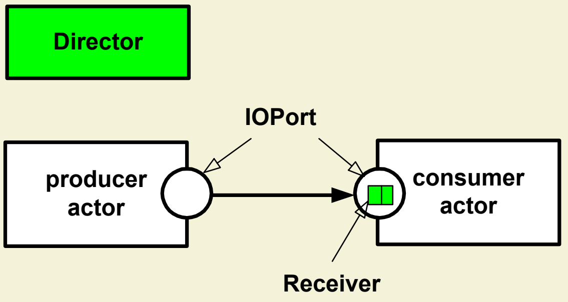


Receiver Interface

«Interface»
Receiver

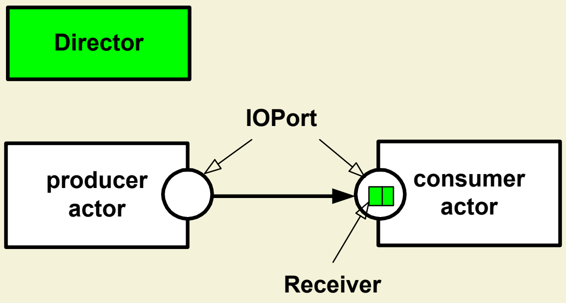
+get() : Token
+getContainer() : IOPort
+hasRoom() : boolean
+hasToken() : boolean
+put(t : Token)
+setContainer(port : IOPort)

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.



Behavioral Types – Codification of Domain Semantics

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

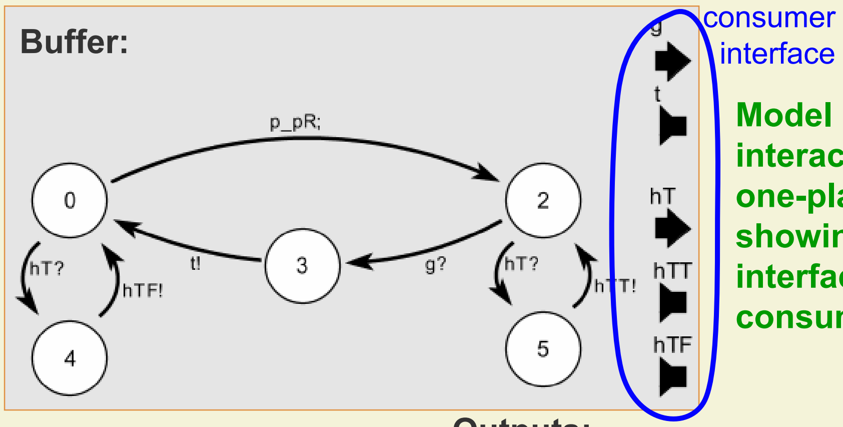


- Communication has
  - data types
  - behavioral types
- Components have
  - data type signatures
  - behavioral type signatures
- Components are
  - data polymorphic
  - domain polymorphic

A Behavioral Type System With Contravariant Inputs and Outputs

- Based on *Interface automata*
  - Proposed by de Alfaro and Henzinger
  - Concise composition (vs. standard automata)
  - *Alternating simulation* provides contravariance
- Compatibility checking
  - Done by automata composition
  - Captures the notion “components can work together”
- Subtyping & polymorphism
  - Alternating simulation (from Q to P)
  - All input steps of P can be simulated by Q, and
  - All output steps of Q can be simulated by P.
  - Used to build a partial order among types

Simple Example: One Place Buffer Showing Consumer Interface Only



Model of the interaction of a one-place buffer, showing the interface to a consumer actor.

Inputs:

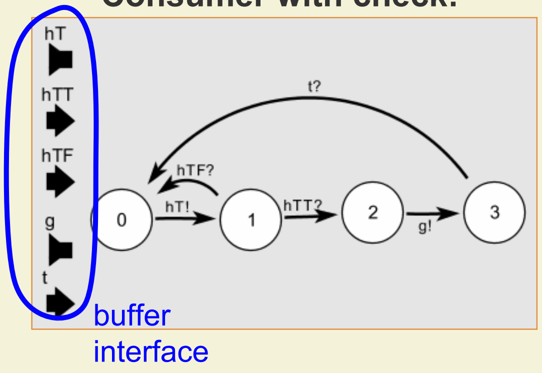
g	get
hT	hasToken

Outputs:

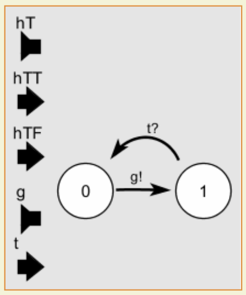
t	Token
hTT	Return True from hasToken
hTF	Return False from hasToken

Two Candidate Consumer Actors

Consumer with check:



Consumer without check:

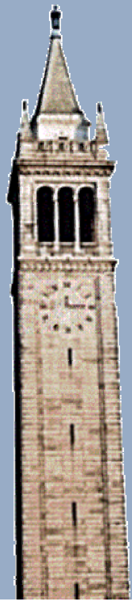


Inputs:

t	Token
hTT	Return True from hasToken
hTF	Return False from hasToken

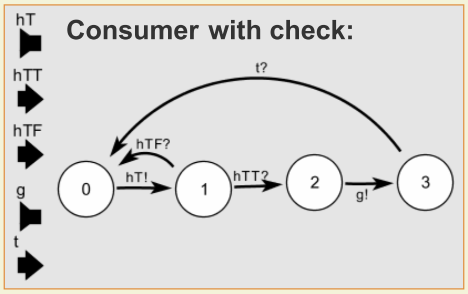
Outputs:

g	get
hT	hasToken

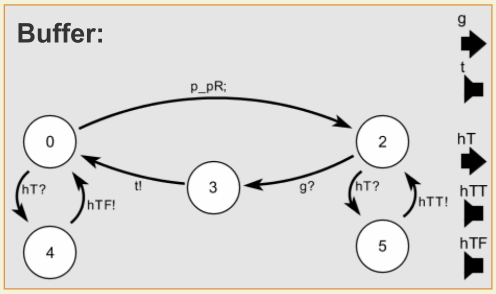


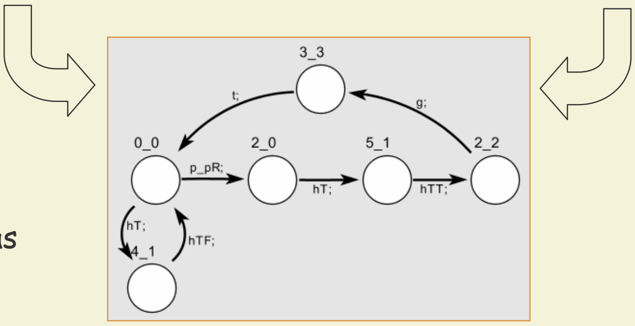
Composition: Behavioral Type Check

Consumer with check:

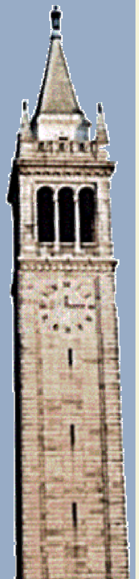


Buffer:



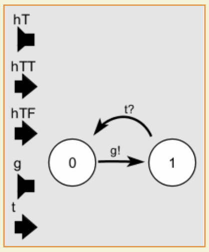


Illegal states are pruned out of the composition. A composite state is illegal if an output produced by one has no corresponding input in the other.

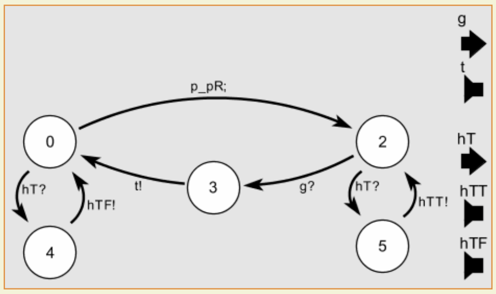


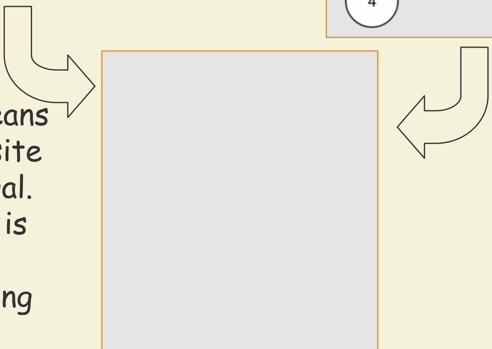
Composition: Behavioral Type Check

Consumer without check:



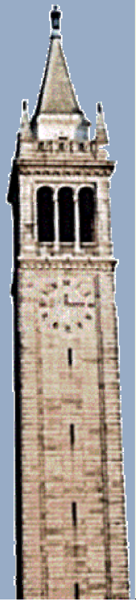
Buffer:



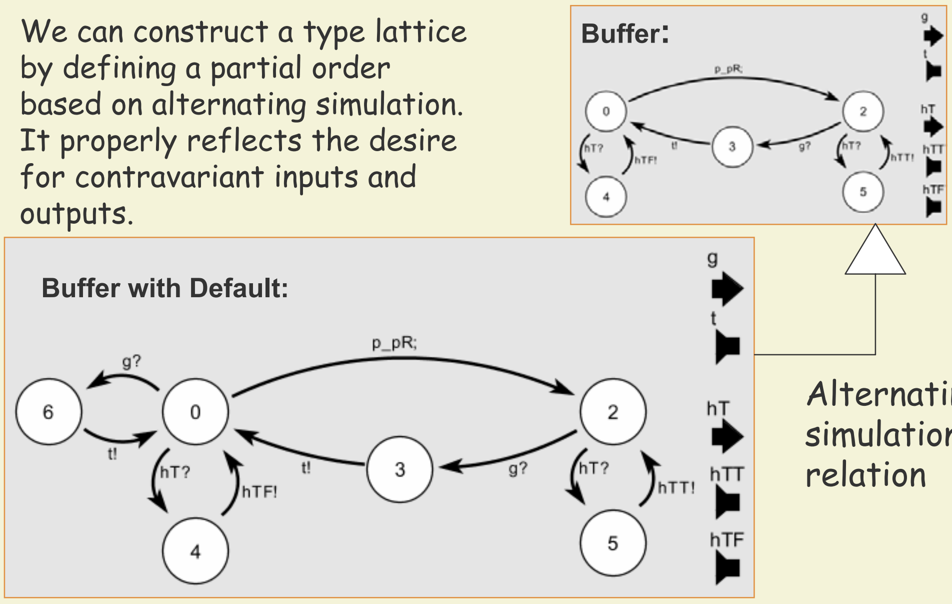


An empty composition means that all composite states are illegal. E.g., here, 0_0 is illegal, which results in pruning all states.

Subclassing and Polymorphism

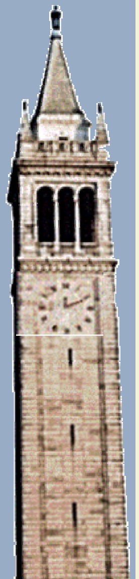


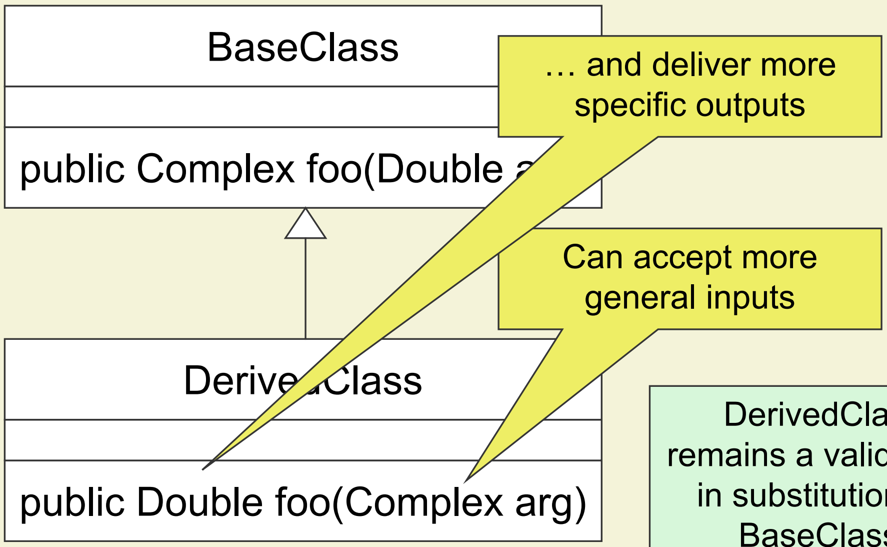
We can construct a type lattice by defining a partial order based on alternating simulation. It properly reflects the desire for contravariant inputs and outputs.



Alternating simulation relation

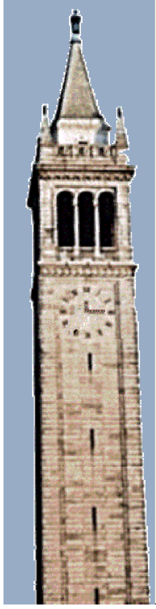
Contravariance of Inputs and Outputs in a Classical Type System

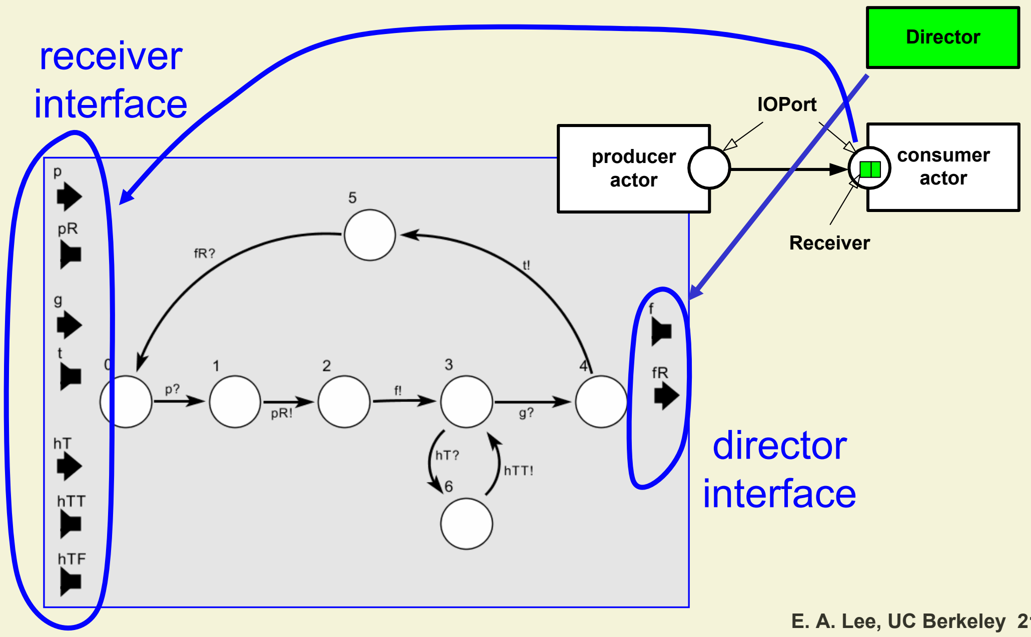



BaseClass
public Complex foo(Double arg)
DerivedClass
public Double foo(Complex arg)
… and deliver more specific outputs
Can accept more general inputs

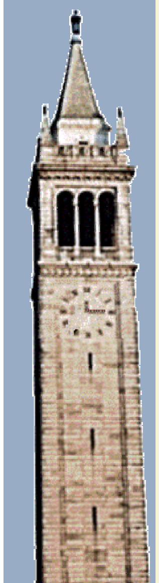
DerivedClass remains a valid drop-in substitution for BaseClass.

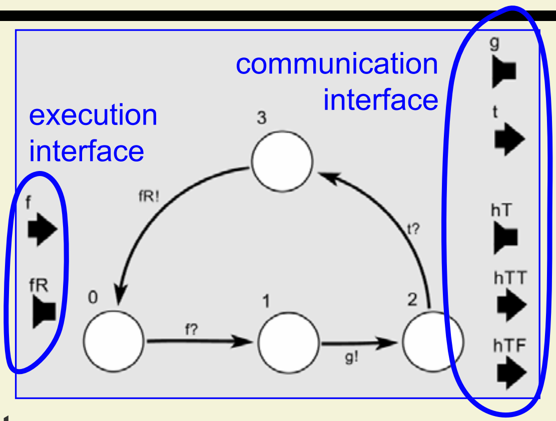
Representing Models of Computation Synchronous Dataflow (SDF) Domain





Consumer Actor With Firing-Type Definition





Such actors are passive, and assume that input is available when they fire.

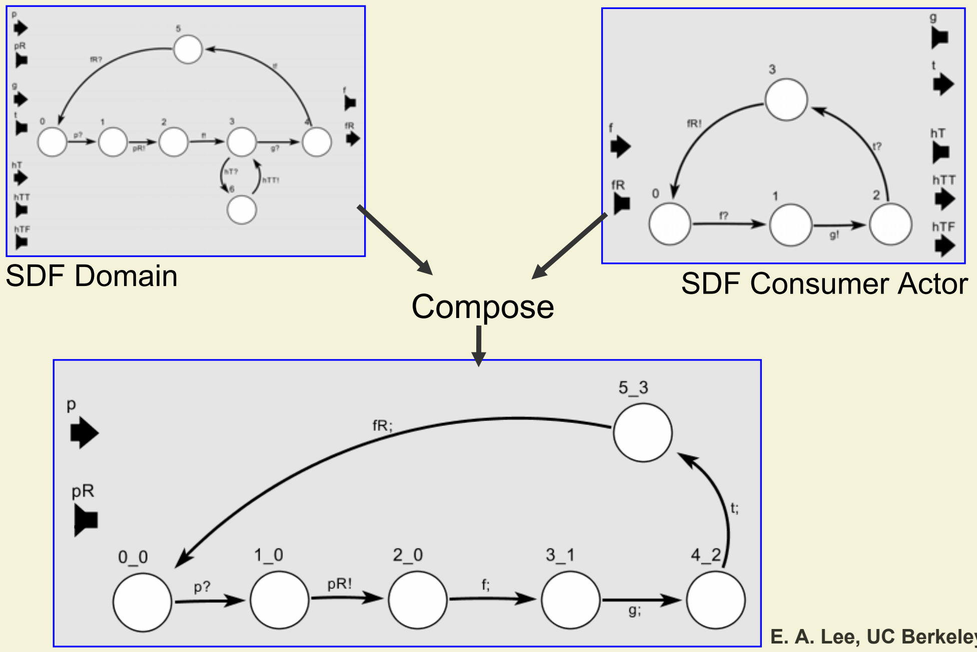
Inputs:

f	fire
t	Token
hTT	Return True from hasToken
hTF	Return False from hasToken

Outputs:

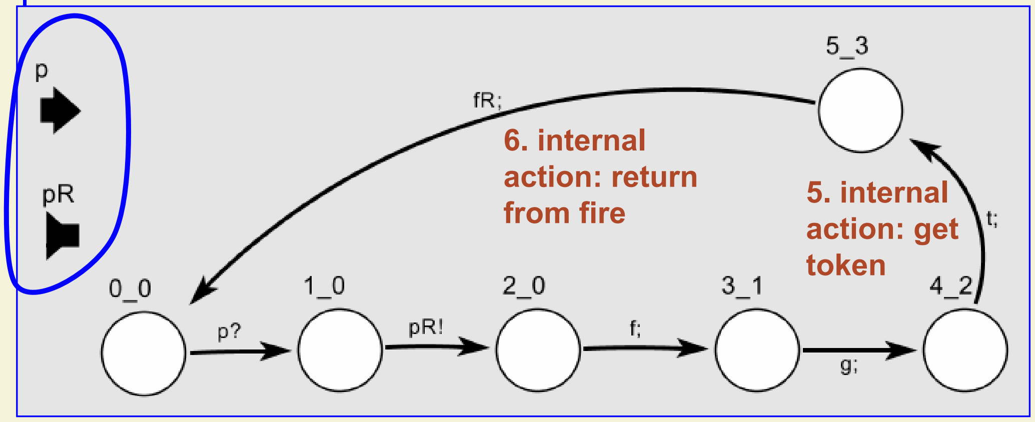
fR	Return from fire
g	get
hT	hasToken

Type Checking – Compose SDF Consumer Actor with SDF Domain



Type Definition – SDF Consumer Actor in SDF Domain

interface to producer actor



6. internal action: return from fire

5. internal action: get token

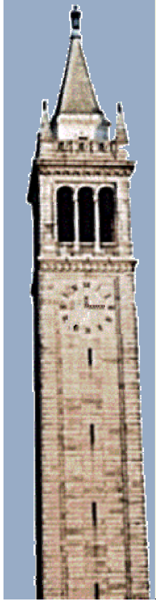
1. receives token from producer

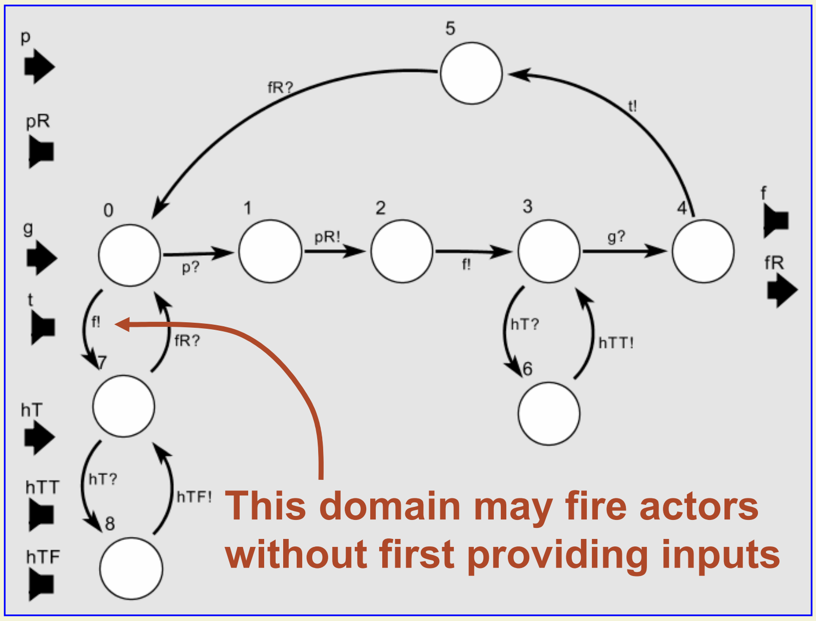
2. accept token

3. internal action: fire consumer

4. internal action: call get()

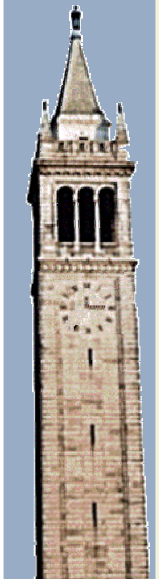
Representing Models of Computation – Discrete Event (DE) Domain



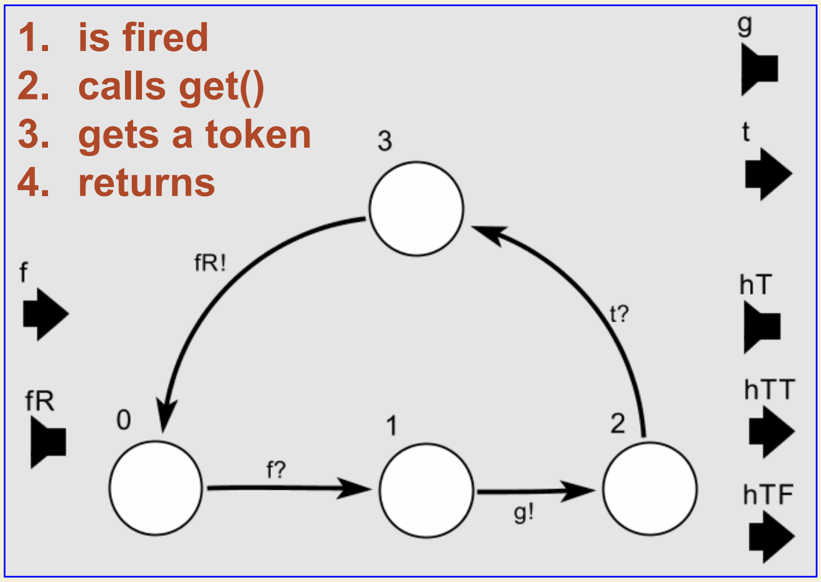


This domain may fire actors without first providing inputs

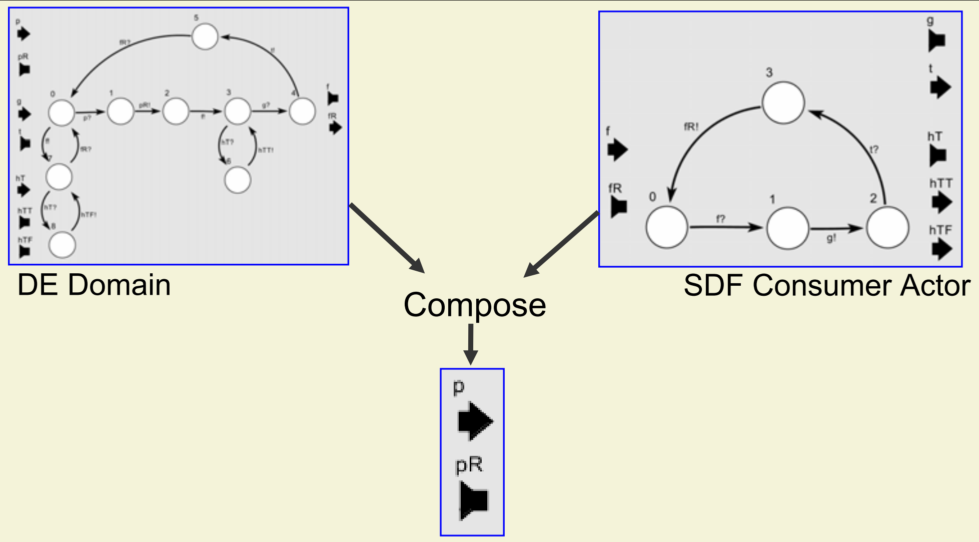
Recall Component Behavior SDF Consumer Actor



1. is fired
2. calls get()
3. gets a token
4. returns

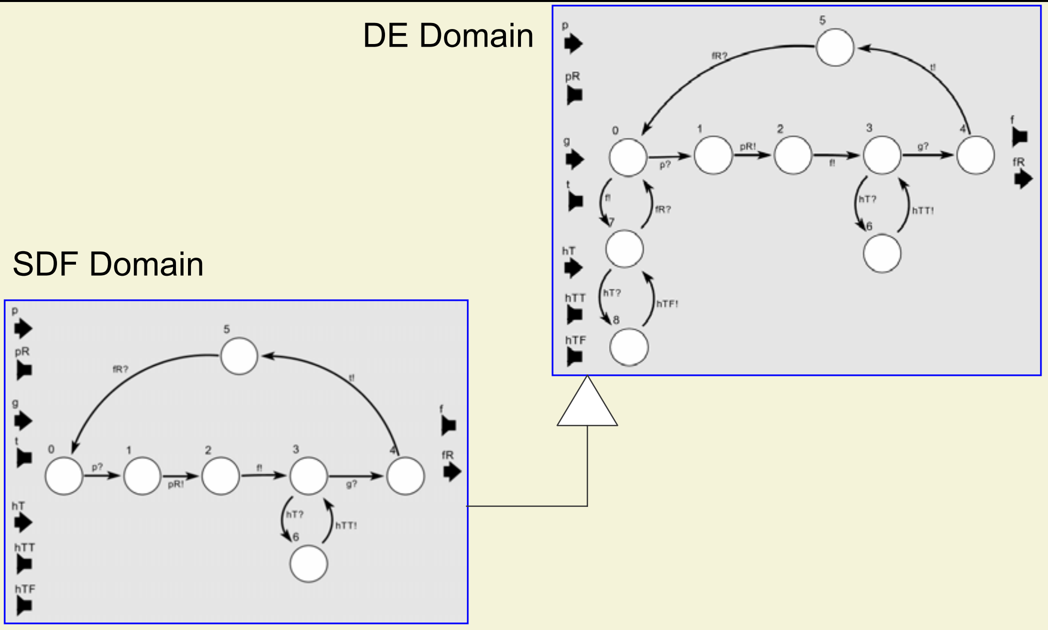


Type Checking – Compose
SDF Consumer Actor with DE Domain

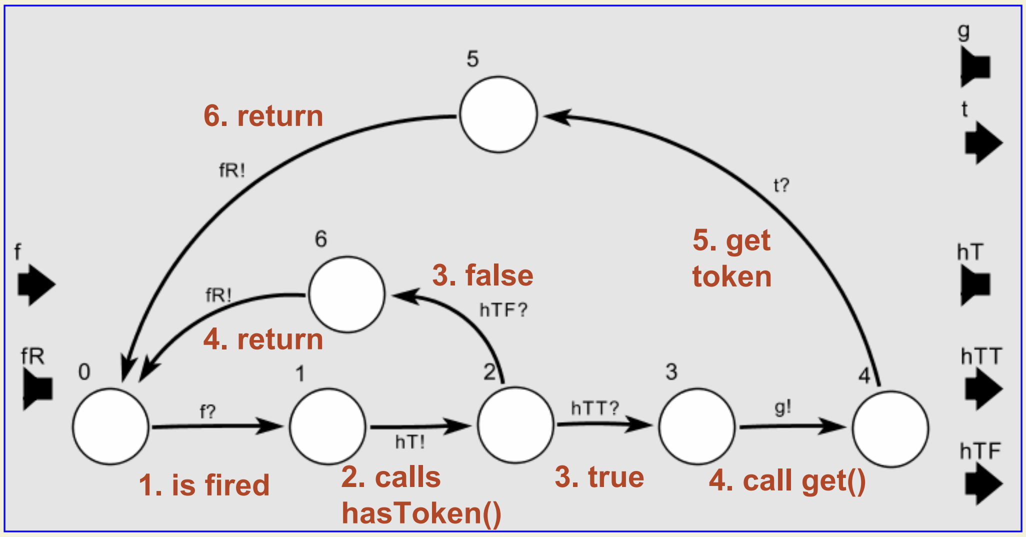


- Empty automaton indicates incompatibility
- Composition type has no behaviors

Subtyping Relation
Alternating Simulation: SDF ≤ DE

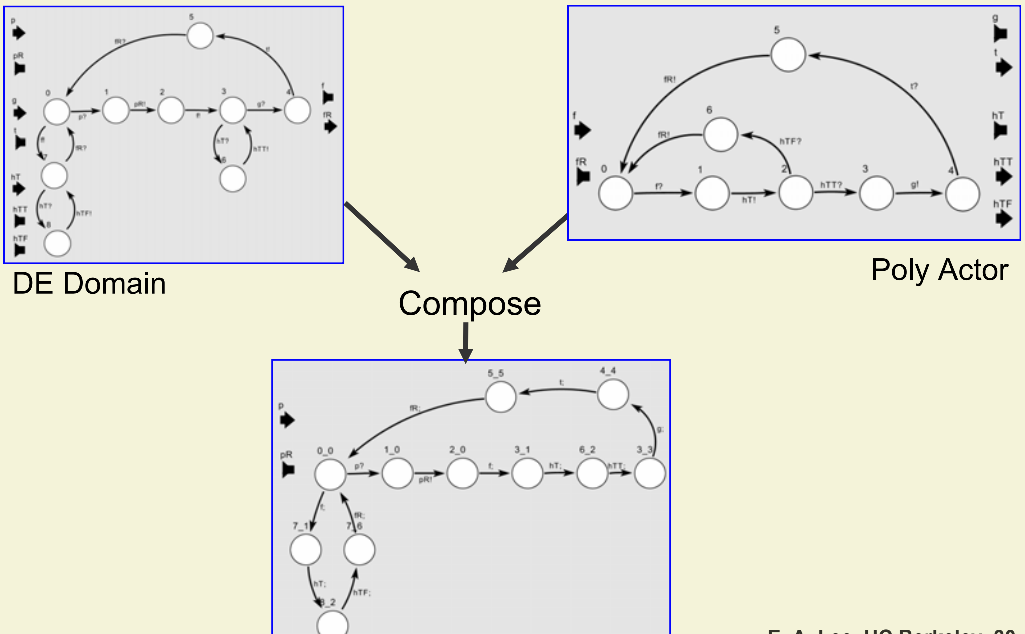


Domain Polymorphic Type Definition – Consumer Actor with Firing

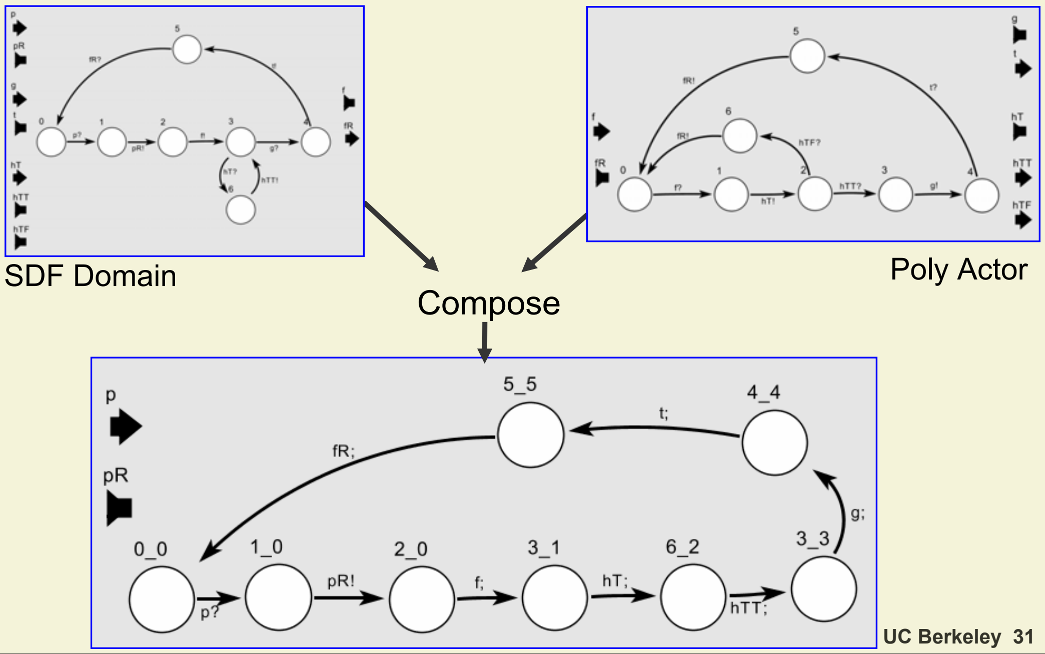


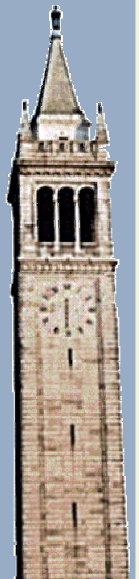
This actor checks for token availability before attempting to get the token.

Domain Polymorphic Actor Composes with the DE Domain



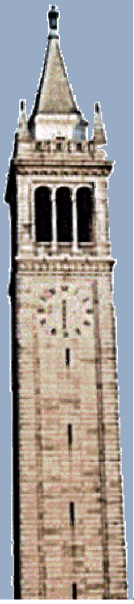
Domain Polymorphic Actor Also Composes with the SDF Domain





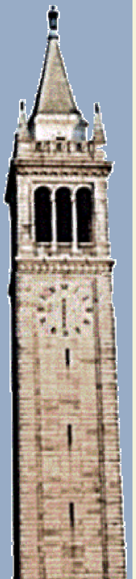
Summary of Behavioral Types Results

- We capture patterns of component interaction in a type system framework: *behavioral types*
- We describe interaction types and component behavior using *interface automata*.
- We do type checking through *automata composition* (detect component incompatibilities)
- Subtyping order is given by the alternating simulation relation, supporting *polymorphism*.
- A *behavioral type system* is a set of automata that form a lattice under alternating simulation.



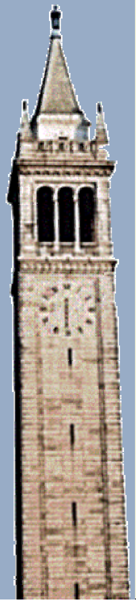
Scalability

- Automata represent behavioral types
  - Not arbitrary program behavior
  - Descriptions are small
  - Compositions are small
  - Scalability is probably not an issue
- Type system design becomes an issue
  - What to express and what to not express
  - Restraint!
    - Will lead to efficient type check and type inference algorithms



Issues and Ideas

- Composition by name-matching
  - awkward, limiting.
  - use ports in hierarchical models?
- Rich subtyping:
  - extra ports interfere with alternating simulation.
  - projection automata?
  - use ports in hierarchical models?
- Synchronous composition:
  - composed automata react synchronously.
  - modeling mutual exclusion is awkward
  - use transient states?
  - hierarchy with transition refinements?



More Speculative

- We can reflect component dynamics in a run-time environment, providing *behavioral reflection*.
  - admission control
  - run-time type checking
  - fault detection, isolation, and recovery (FDIR)
- Timed interface automata may be able to model *real-time* requirements and constraints.
  - checking consistency becomes a type check
  - generalized schedulability analysis
- Need a *language* with a behavioral type system
  - Visual syntax given here is meta modeling
  - Use this to build domain-specific languages