System-Level Types for Component-Based Design

Edward A. Lee
Yuhong Xiong

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

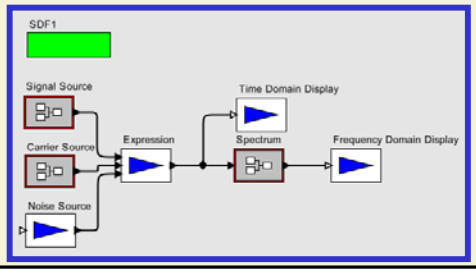
Presented at EMSOFT, Lake Tahoe, October 2001.

Outline

- Component-based design
- System-level types
- Interface Automata
- Interaction Types and Component Behavior
- Type Checking
- Type Order and Polymorphism
- Design Tradeoffs
- Conclusion

Component-Based Design

- Good for designing complex, concurrent, heterogeneous systems
- Two levels of interface:
  - data types and
  - dynamic interaction
- Key aspects of dynamic interaction: communication & execution

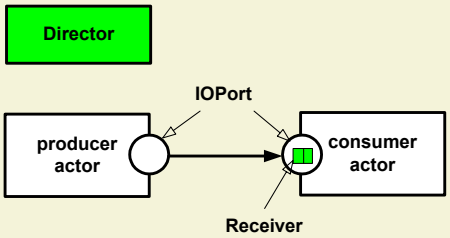


Type Systems

- Type systems are successful
  - Safety through type checking
  - Polymorphism supports reuse (flexible components)
  - Interface documentation, clarification
  - Run-time reflection of component interfaces
- Data types only specify static aspects of interface
- Proposal:
  - Capture the dynamic interaction of components in types
  - Obtain benefits analogous to data typing.
  - Call the result ***system-level types***.

Interaction Semantics

- Flow of control issues ("execution model" - Sifakis)
  - in Ptolemy II, these are defined by a *Director* class
- Communication between components ("interaction model")
  - in Ptolemy II, this is defined by a *Receiver* class

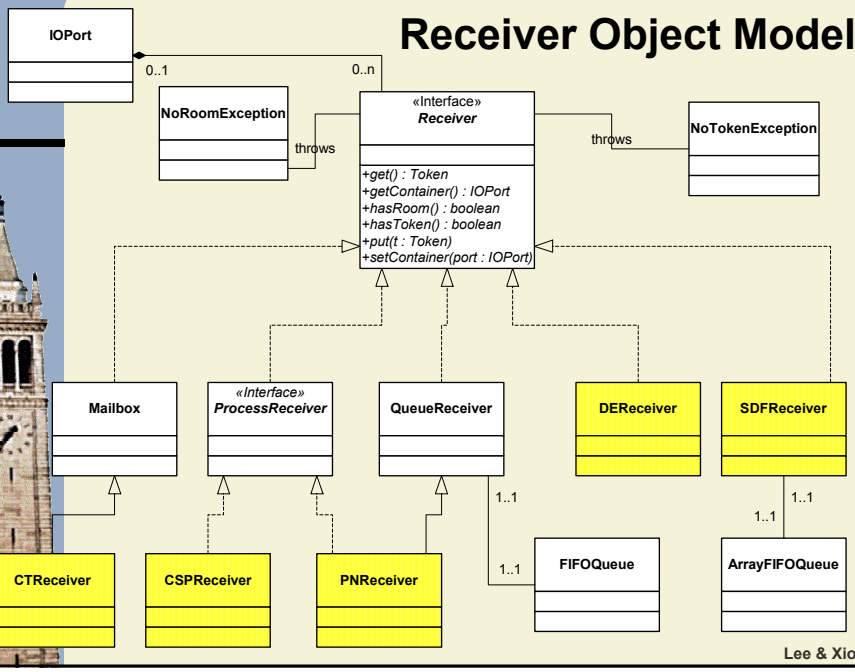


Actor interface for execution: fire
Receiver interface for communication: put, get, hasToken

Models of Computation

- Define the interaction semantics
- Implemented in Ptolemy II by a *domain*
  - Receiver + Director
- Examples:
  - **Communicating Sequential Processes (CSP):** rendezvous-style communication
  - **Process Networks (PN):** asynchronous communication
  - **Synchronous Data Flow (SDF):** stream-based communication, statically scheduled
  - **Discrete Event (DE):** event-based communication
  - **Synchronous/Reactive (SR):** synchronous, fixed point semantics

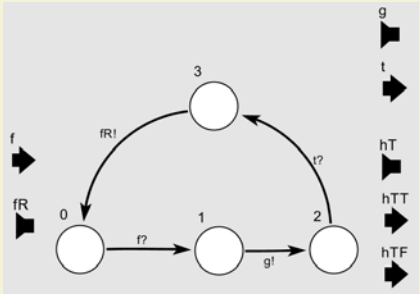
Receiver Object Model



Formal Interaction Semantics: Use *Interface Automata*

- Automata-based formalism
  - Proposed by de Alfaro and Henzinger
  - Optimistic
  - Concise composition
- Compatibility checking
  - Done by automata composition
  - Captures the notion “components can work together”
- 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.
  - Provides the ordering we need for subtyping & polymorphism
- A key theorem about compatibility and alternating simulation

Example: SDF Consumer Actor



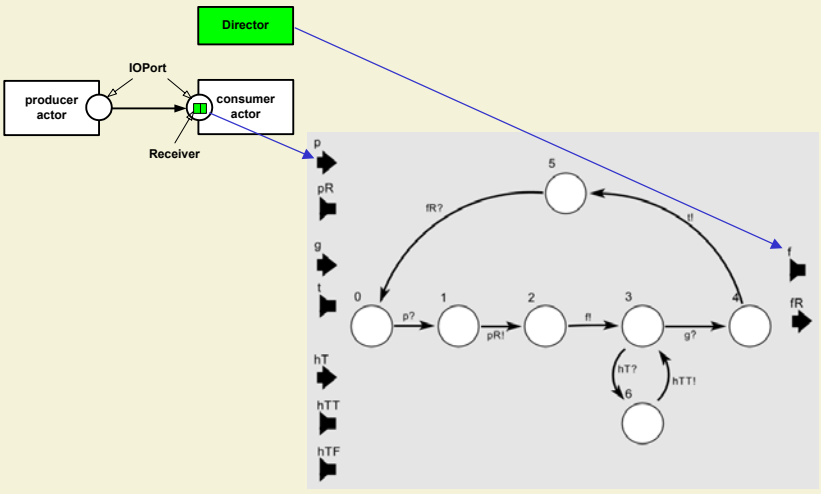
Inputs:

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

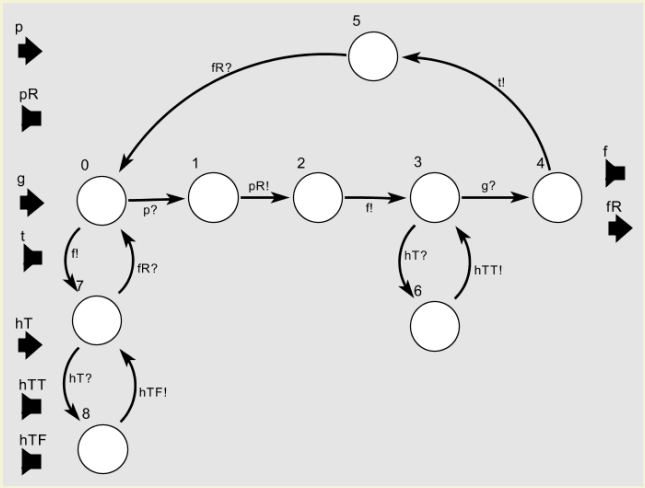
Outputs:

fR	Return from fire
g	get
hT	hasToken

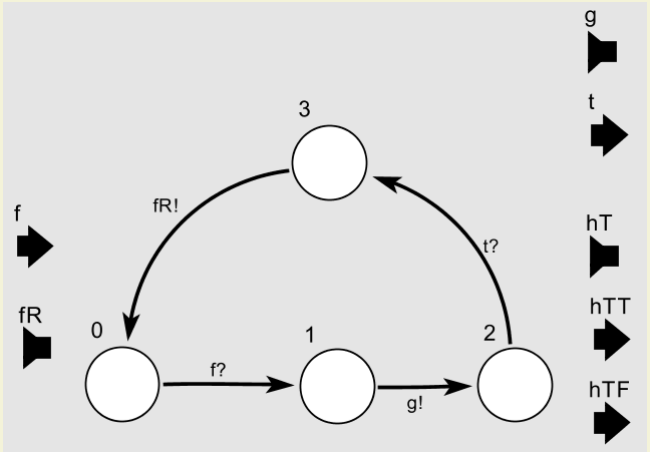
Type Definition - SDFDomain



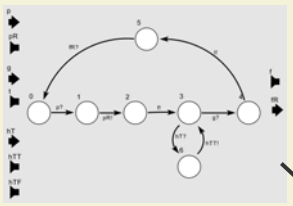
Type Definition - DEDomain



Component Behavior
SDF Consumer Actor

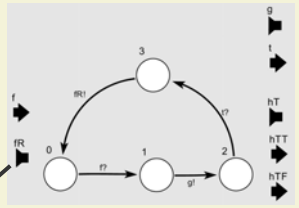


Type Checking
SDF Consumer Actor in SDFDomain

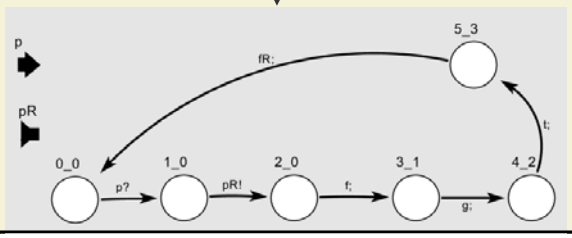


SDFDomain

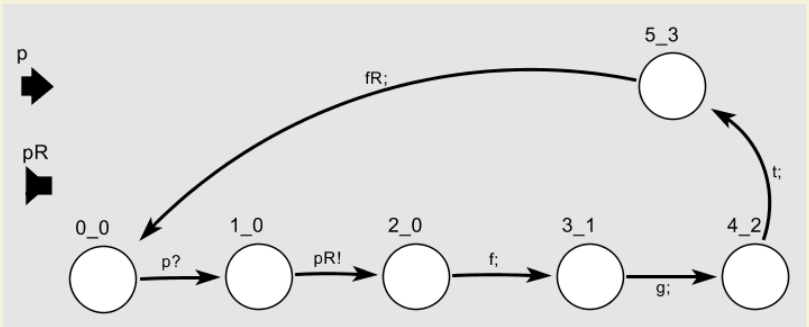
Compose



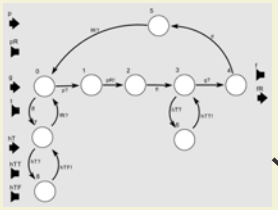
SDF Consumer Actor



Type Checking
SDF Consumer Actor in SDFDomain

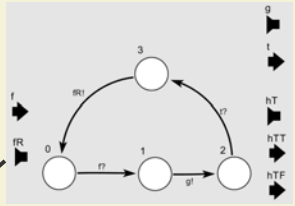


Type Checking
SDFActor in DEDomain



DEDomain

Compose

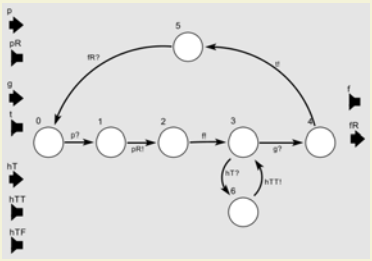


SDF Consumer Actor

- **Empty automata indicating incompatibility**

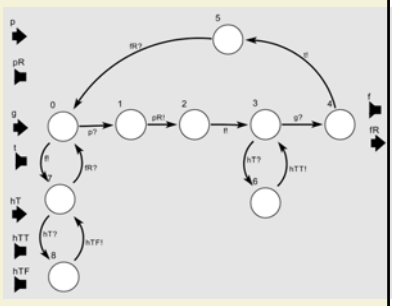
Alternating Simulation
SDF to DE

SDFDomain

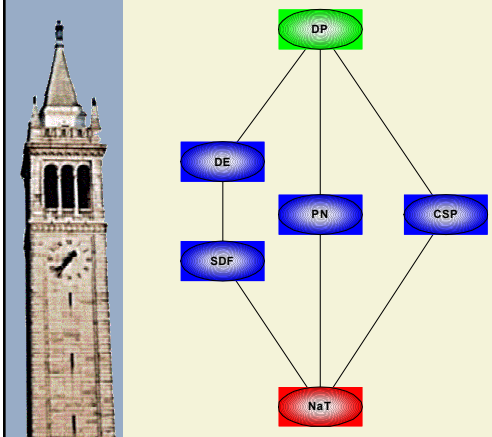


≤

DEDomain

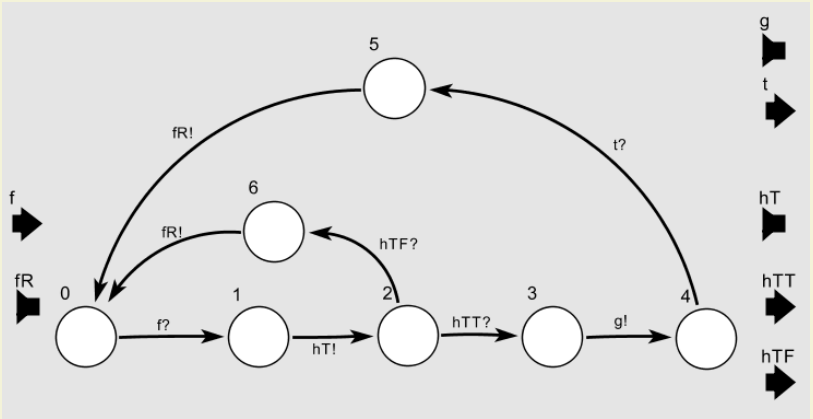


System-Level Type Order Defined by Alternating Simulation

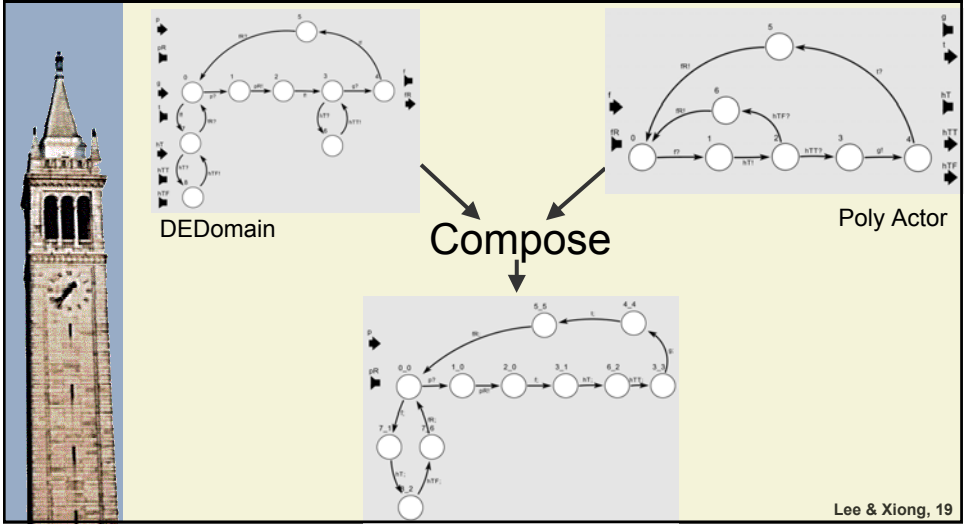


- Analogous to subtyping
- If an actor is compatible with a certain type, it is also compatible with the subtypes

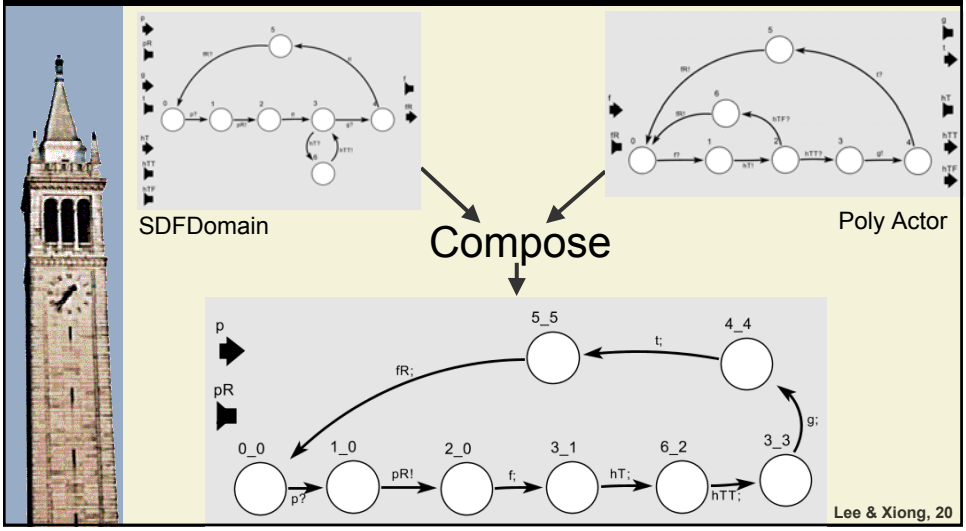
Component Behavior DomainPolymorphicActor



DomainPolymorphicActor is Compatible with DEDomain



So it is also Compatible with SDFDomain



Trade-offs in Type System Design

- Amount of property checked vs. cost of checking
- Static vs. run-time checking
- Example of more static checking: deadlock detection in Dining Philosopher model
- Bottom line: static checking of communication protocols a good starting point

Conclusion and Future Work

- We capture dynamic property of component interaction in a type system framework: *system-level types*
- **We describe interaction types and component behavior using interface automata.**
- We do type checking through automata composition.
- **Subtyping order is given by the alternating simulation relation, supporting polymorphism.**
- We can reflect component state in a run-time environment, providing *system-level reflection*.