Process-Based Software Components

Mobies Phase 1, UC Berkeley

Edward A. Lee and Tom Henzinger

PI Meeting, Boca Raton

January 30, 2002

Program Objectives

Our focus is on component-based design using principled *models of computation* and their *runtime environments* for embedded systems. The emphasis of this project is on the dynamics of the components, including the communication protocols that they use to interface with other components, the modeling of their state, and their flow of control. The purpose of the mechanisms we develop is to improve robustness and safety while promoting component-based design.

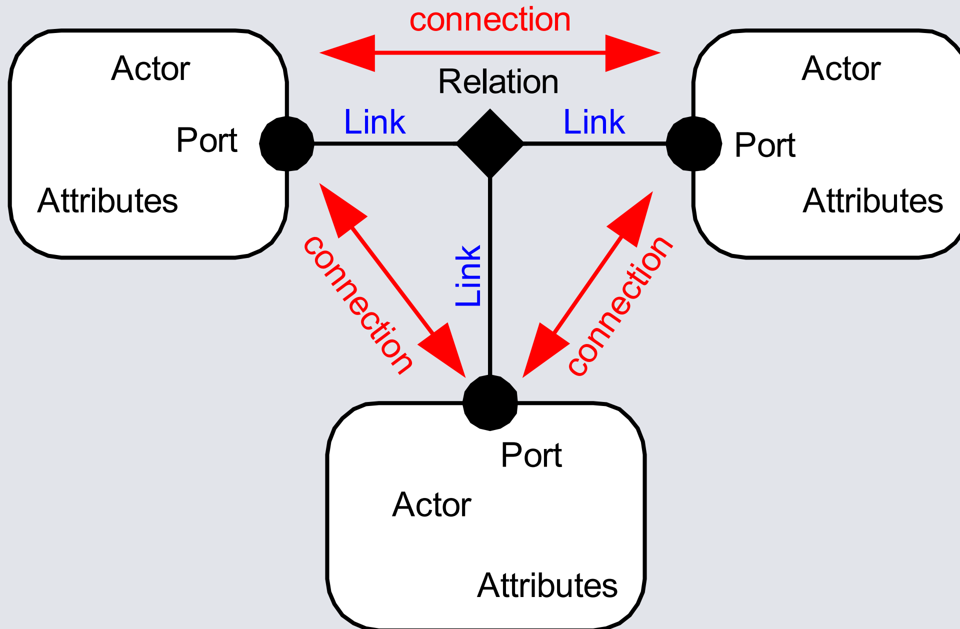
Technical Approach Summary

- Models of computation
  - supporting heterogeneity
  - supporting real-time computation
  - codifications of design patterns
  - definition as *behavioral types*
- Co-compilation
  - joint compilation of components and architecture
  - vs. code generation
  - supporting heterogeneity

Subcontractors and Collaborators

- Subcontractor
  - Univ. of Maryland (C code generation)
- Collaborators
  - UCB Phase II
  - Kestrel
  - Vanderbilt
  - Penn
- Non-Mobies
  - The MathWorks
  - GSRC project (system-level IC design)
  - SEC program (Boeing, etc.)

View of Concurrent Components: *Actors* with *Ports* and *Attributes*



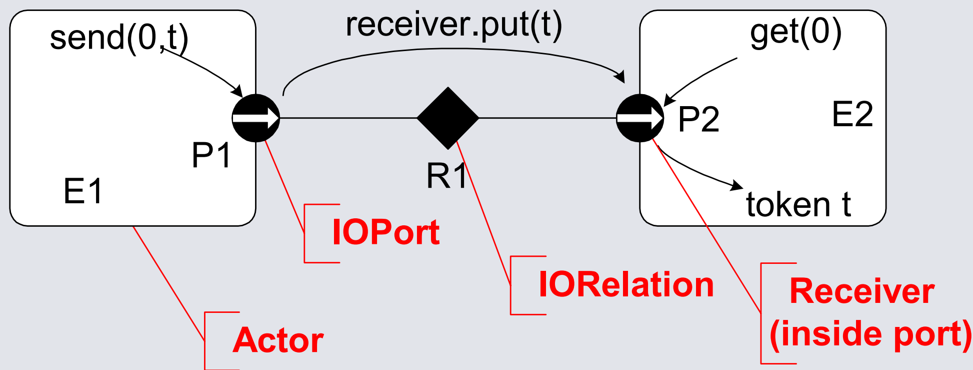
Model of Computation:

- Messaging schema
- Flow of control
- Concurrency

Key idea: The model of computation is part of the framework within which components are embedded not part of the components themselves. It enforces patterns.

Actor View of Producer/Consumer Components

Basic Transport:

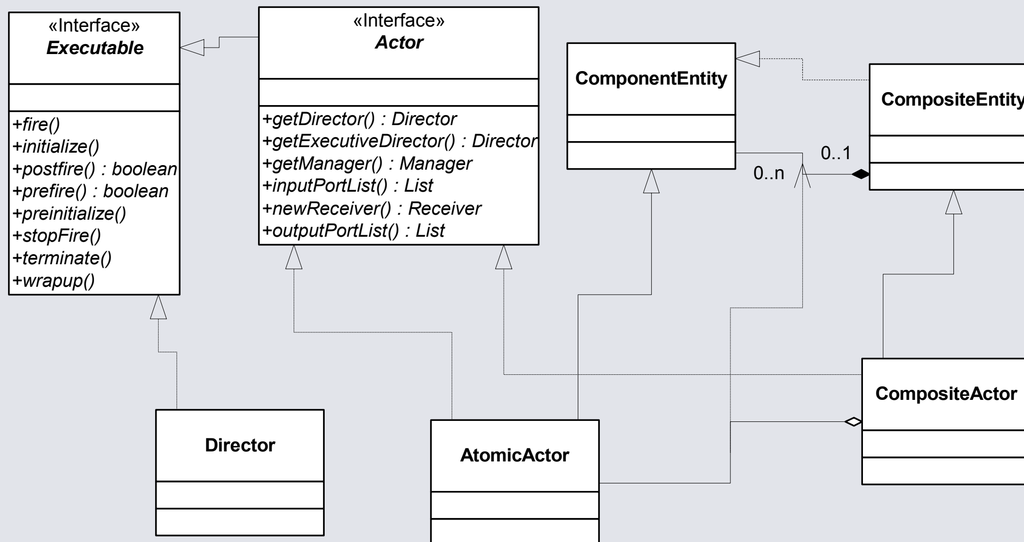


Models of Computation:

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

Contrast with Object Orientation

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



Object orientation emphasizes inheritance and procedural interfaces.

Actor orientation emphasizes concurrency and communication abstractions.

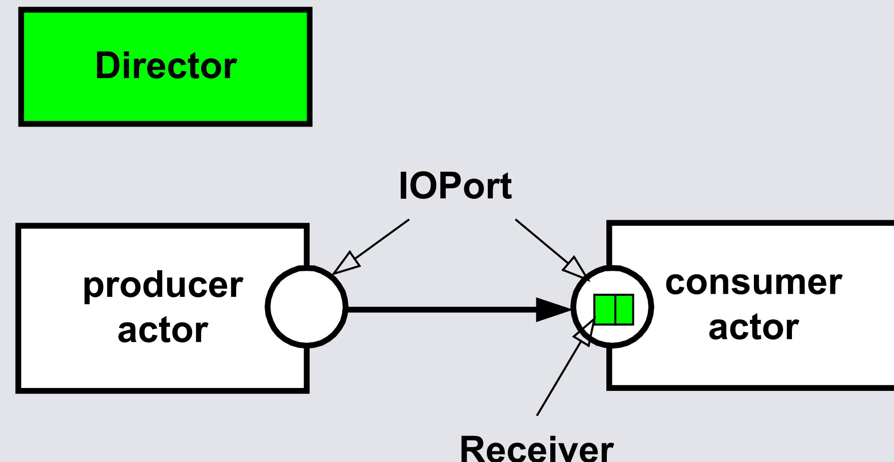
Examples of Actor-Oriented Component Frameworks

- Simulink (The MathWorks)
- Labview (National Instruments)
- 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)
- Port-based objects (U of Maryland)
- I/O automata (MIT)
- VHDL, Verilog, SystemC (Various)
- Polis & Metropolis (UC Berkeley)
- Ptolemy & Ptolemy II (UC Berkeley)
- …

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

Emphasis of Ptolemy II is on methods and infrastructure for designing and building domains, understanding their semantics, and interfacing them heterogeneously.



Example Domains

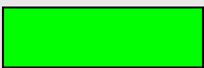
- Time Driven (Giotto):
  - synchronous, time-driven multitasking – *built for Mobies.*
- Synchronous Data Flow (SDF):
  - stream-based communication, statically scheduled
- Discrete Event (DE):
  - event-based communication
- Continuous Time (CT):
  - continuous semantics, ODE solver simulation engine
- Synchronous/Reactive (SR):
  - synchronous, fixed point semantics
- Timed Multitasking (TM):
  - priority-driven multitasking, deterministic communication – *built for SEC.*
- Communicating Sequential Processes (CSP):
  - rendezvous-style communication
- Process Networks (PN):
  - asynchronous communication, determinism

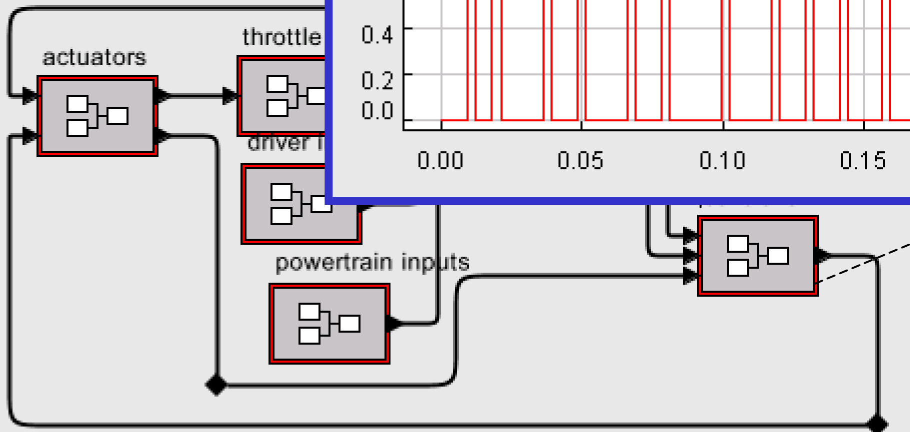
Design Pattern: Periodic/Time-Driven Inside Continuous Time

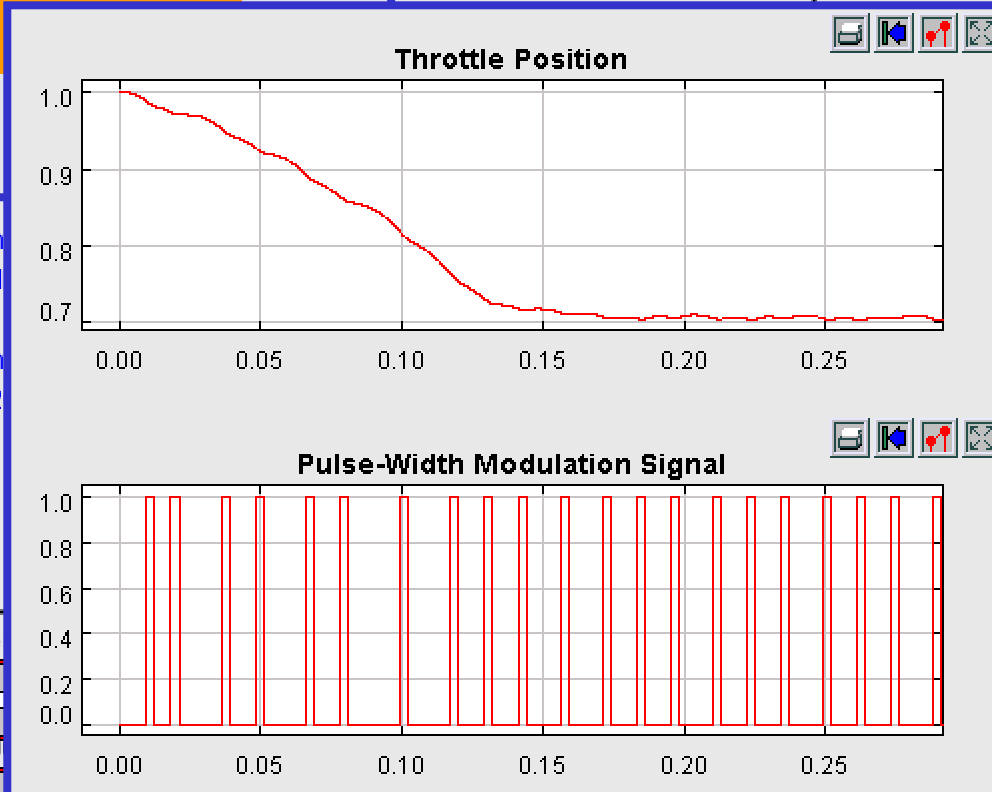
Giotto director indicates a new model of computation.

Heterogeneous model of th
Electronic Throttle Controll

by Paul Griffiths, Christoph
Last updated January 15, 2



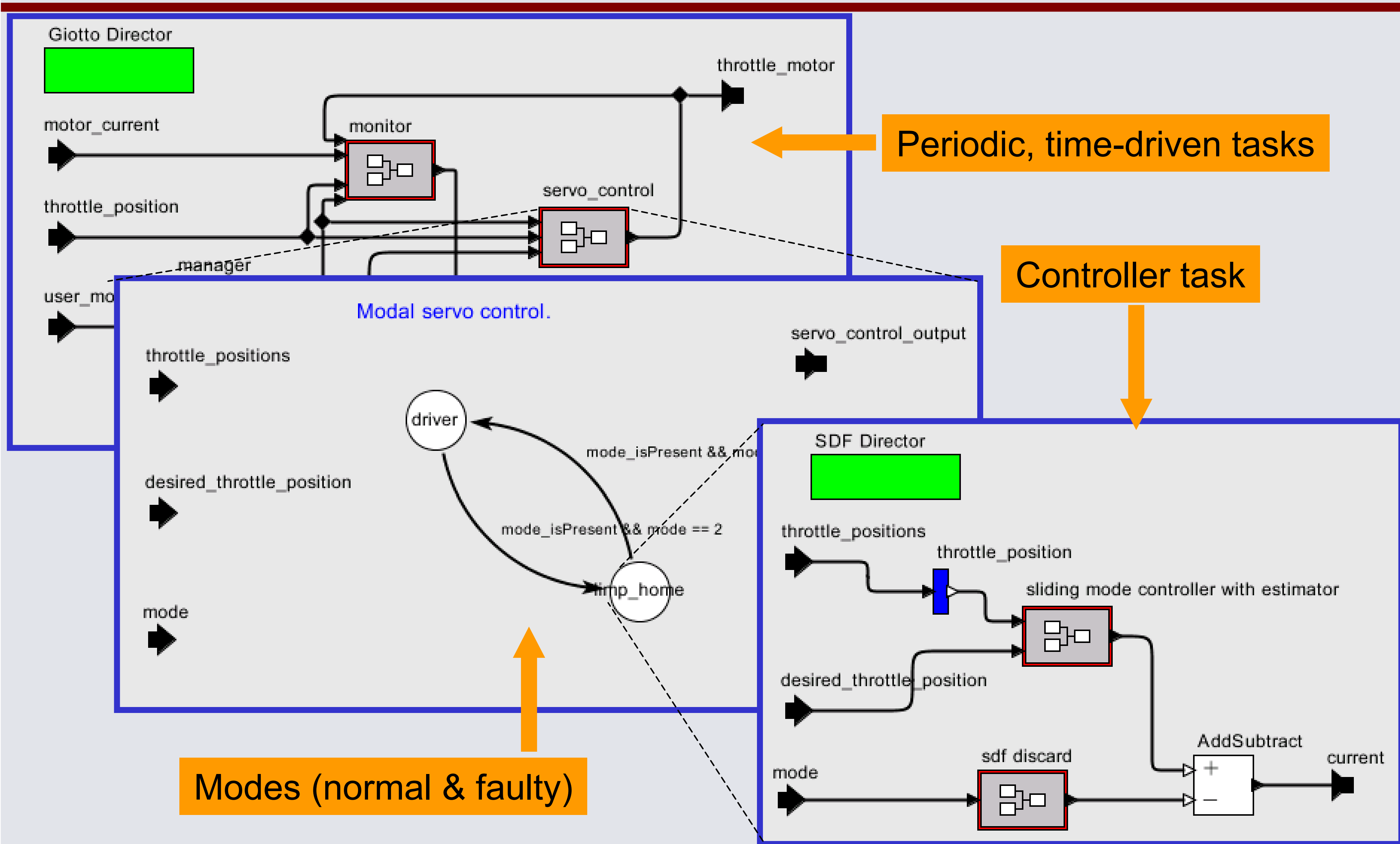




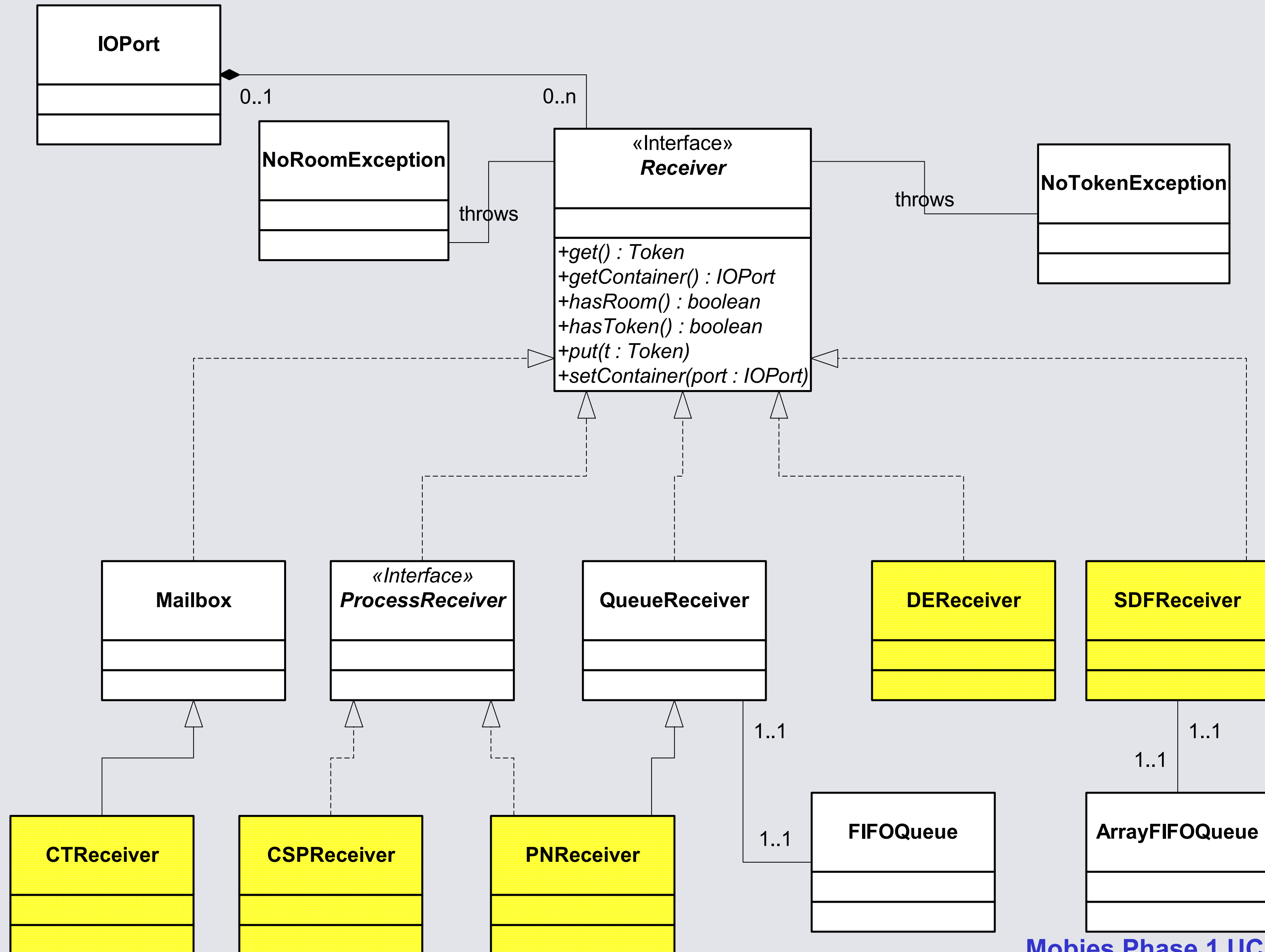
ent.

Domains can be nested and mixed.

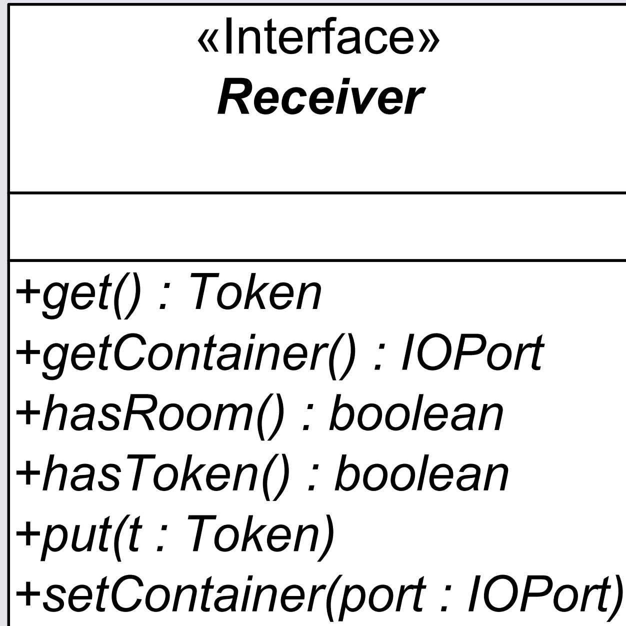
Controller Heterogeneity



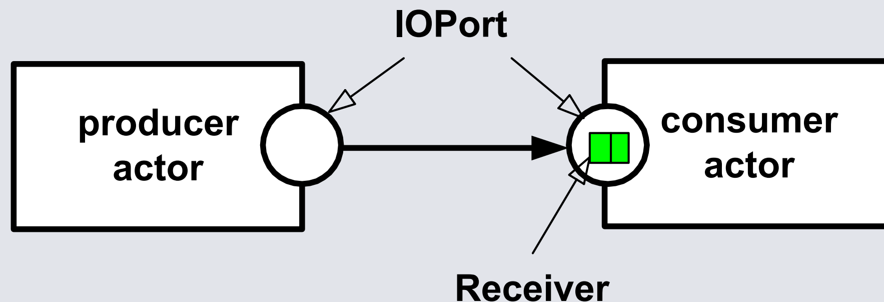
Key to Domain Polymorphism: Receiver Object Model



Receiver Interface

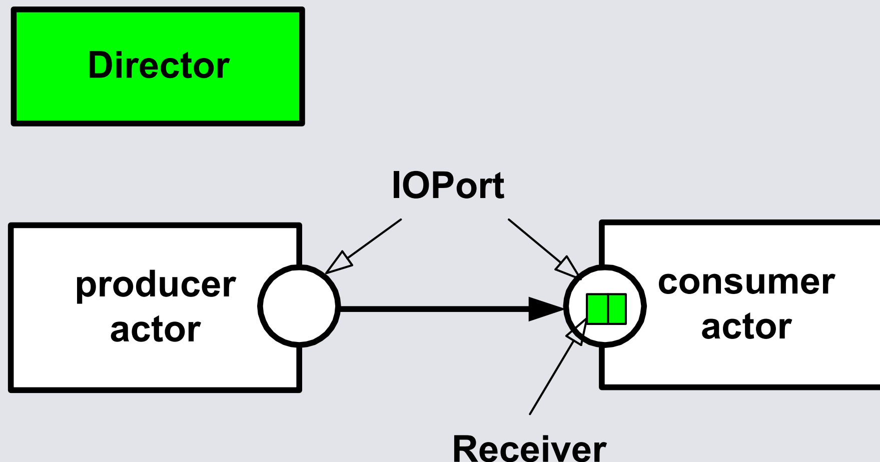


These polymorphic methods implement the communication semantics of a domain in Ptolemy II. The receiver instance used in communication is supplied by the director, not by the component.



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.*

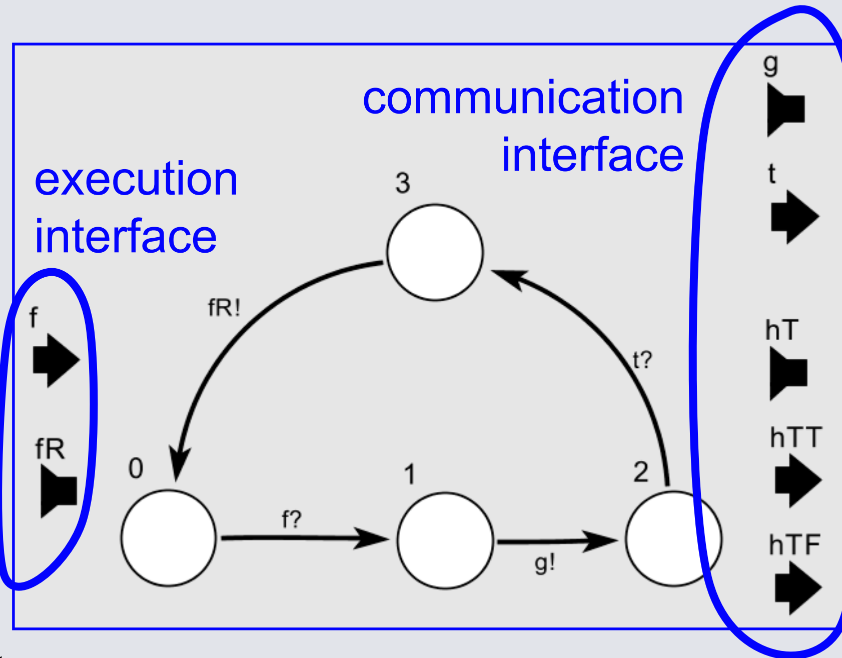


- Communication has
  - data types
  - behavioral types
- Components have
  - data type signatures
  - domain type signatures
- Components are
  - data polymorphic
  - domain polymorphic

Second Version of a Behavioral Type System

- 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"
- 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
- Key theorem about compatibility and alternating simulation

Example: Synchronous Dataflow (SDF) Consumer Actor Type Definition



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

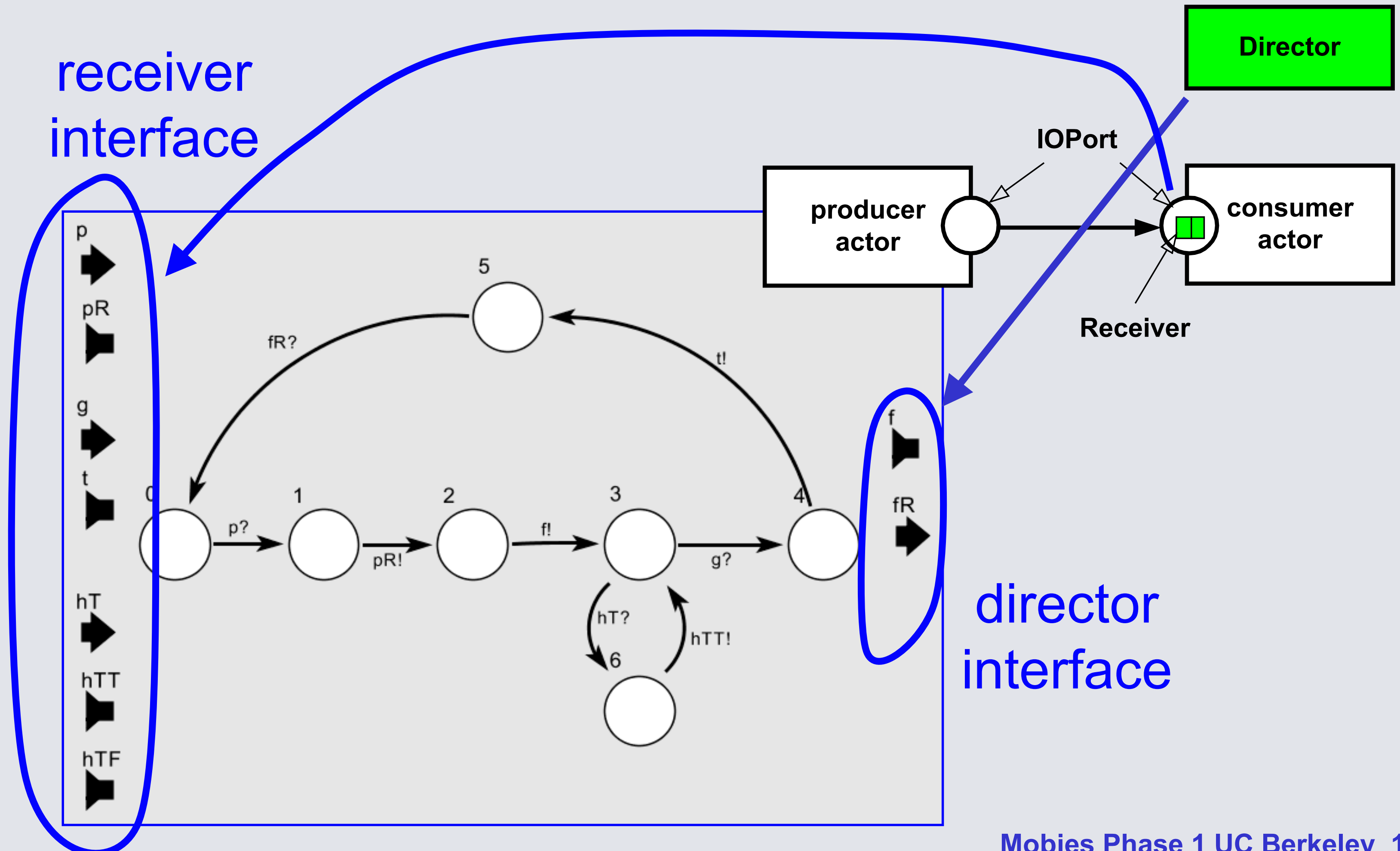
Inputs:

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

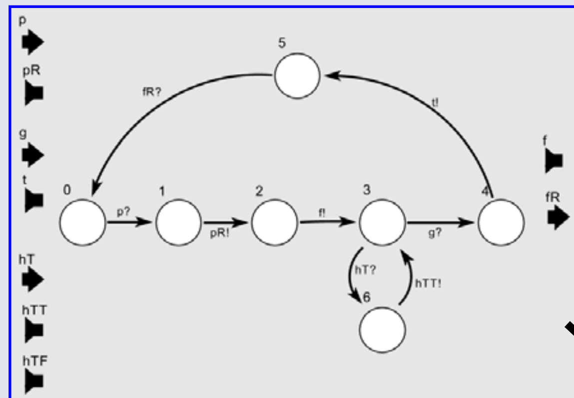
Outputs:

fR	Return from fire
g	get
hT	hasToken

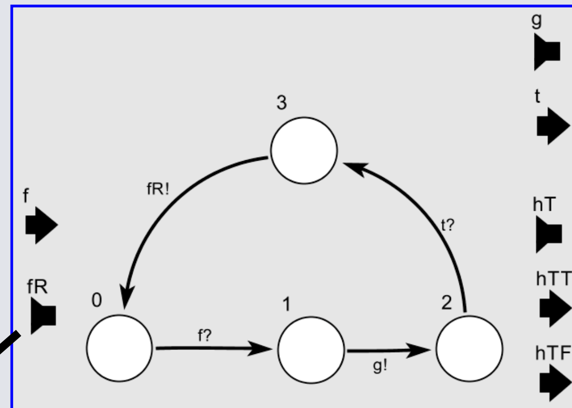
Type Definition – Synchronous Dataflow (SDF) Domain



Type Checking – Compose SDF Consumer Actor with SDF Domain

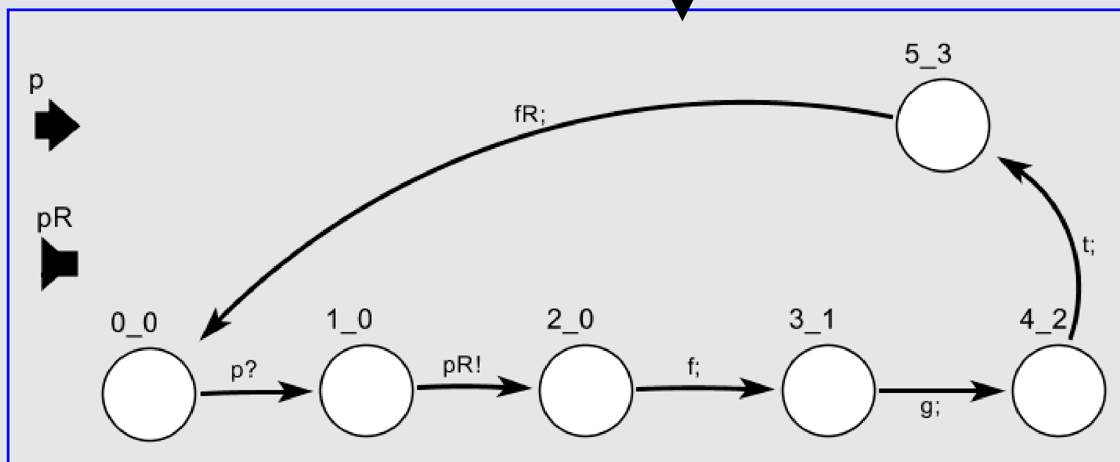


SDF Domain



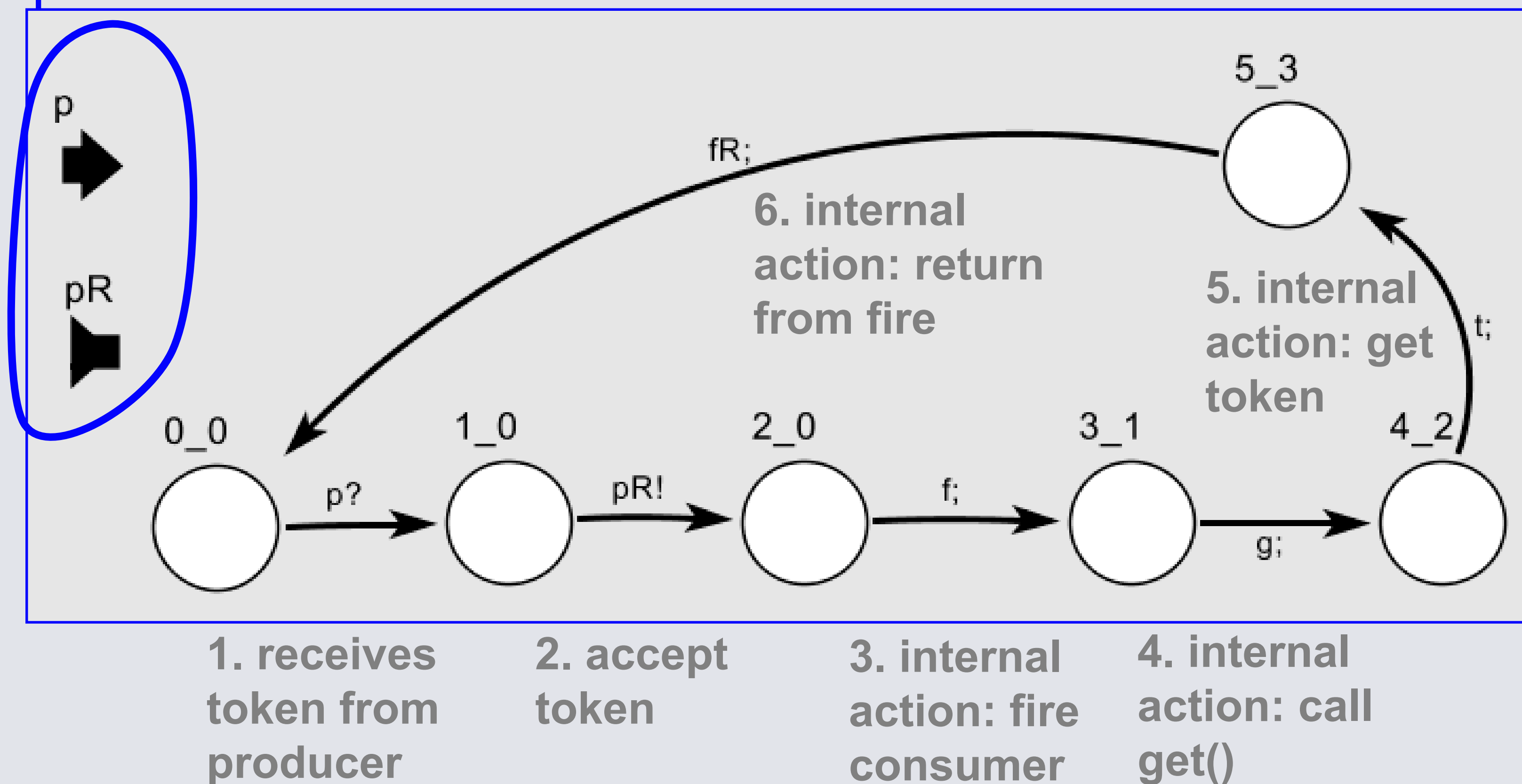
SDF Consumer Actor

Compose



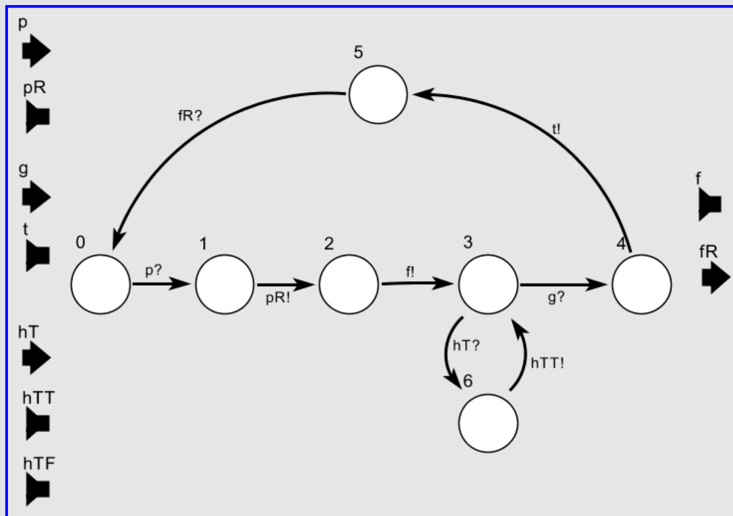
Interface automaton (IA) domain (by Yuhong Xiong) is used for experimentation.

Type Definition for Composition – SDF Consumer Actor in SDF Domain

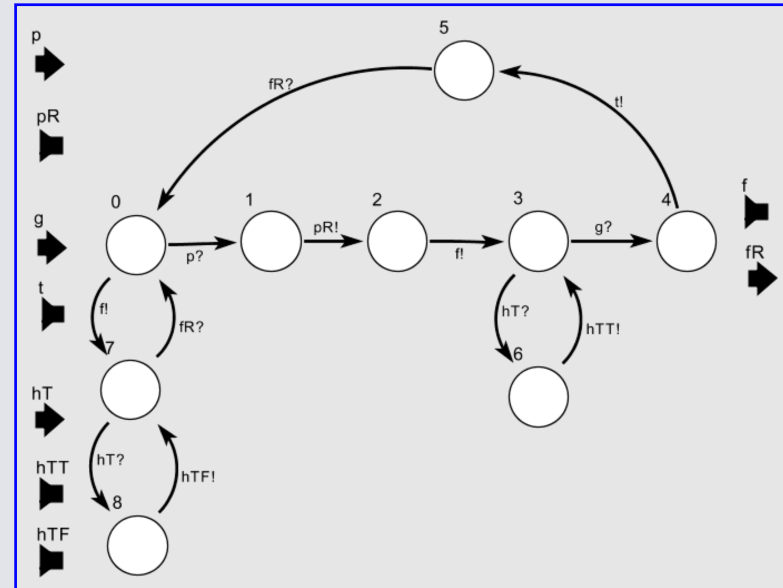


Subtyping Relation
Alternating Simulation: SDF $\leq$ DE

SDF Domain



DE Domain



Partial order relation between behavioral types makes this a type system.

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*.

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

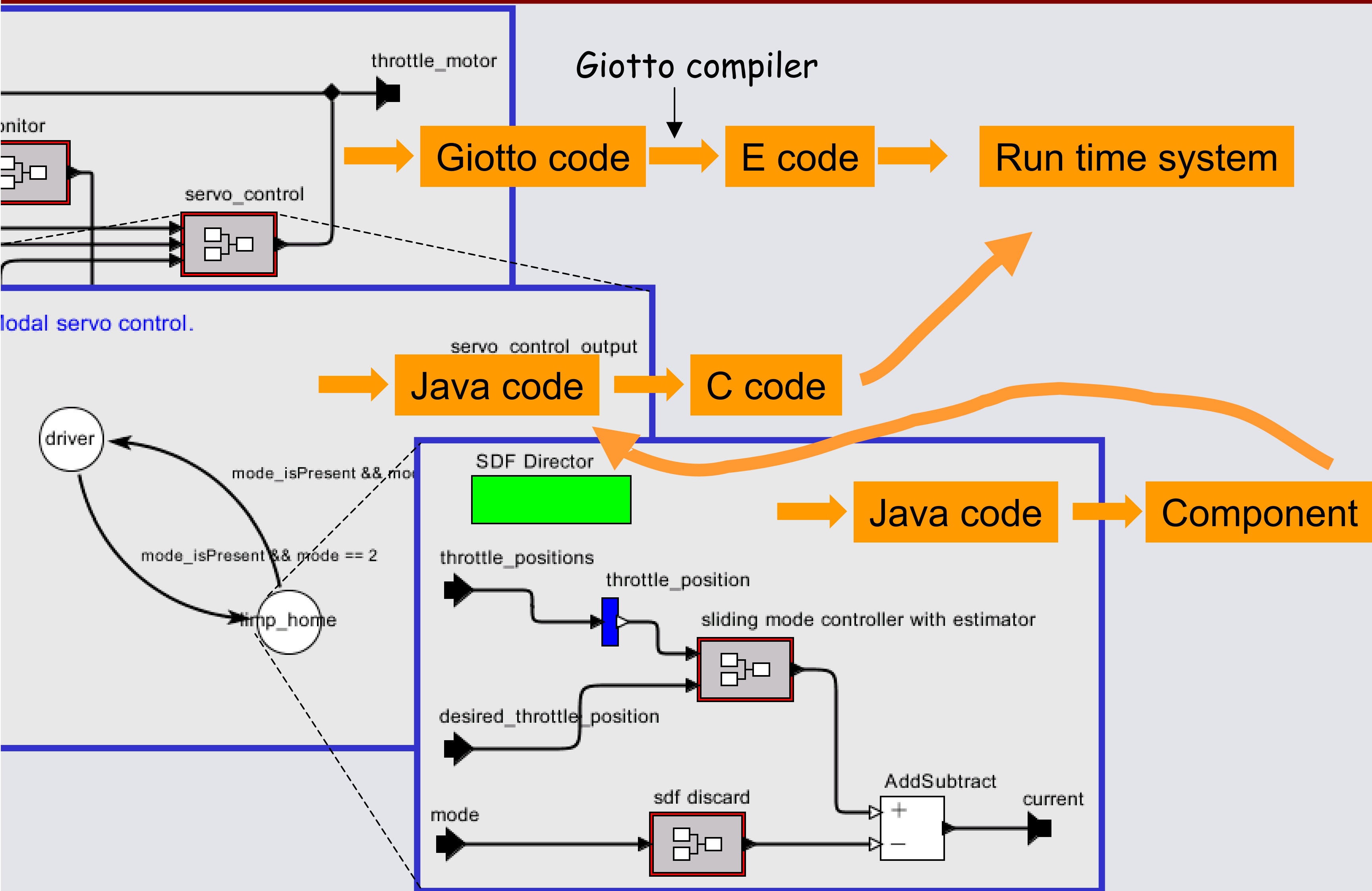
Code Generation

- MoC semantics defines
  - flow of control across actors
  - communication protocols between actors
- Actors define:
  - functionality of components
- Actors are compiled by a MoC-aware compiler
  - generate specialized code for actors in context
- Hierarchy & heterogeneity:
  - Code generation at a level of the hierarchy produces a new actor definition

We call this *co-compilation.*

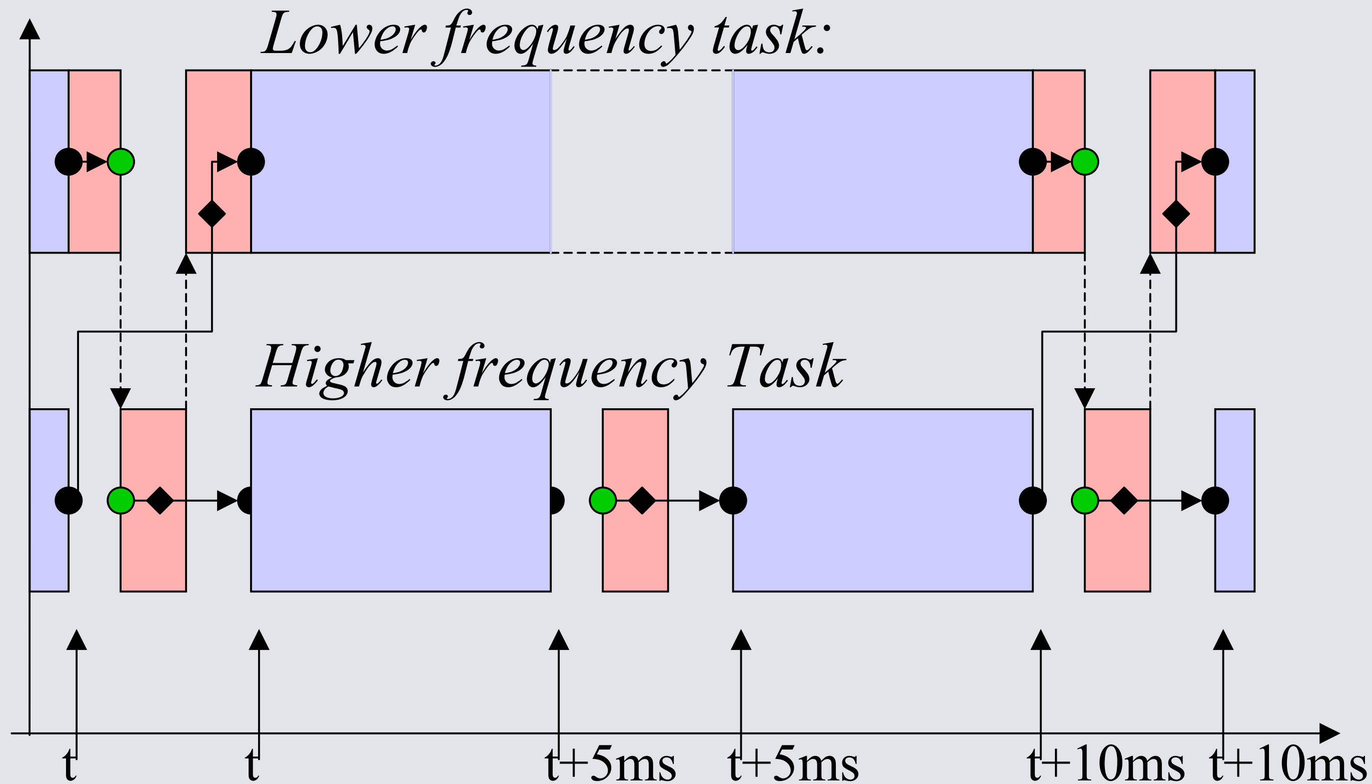
Multiple domains may be used in the same model

Integrated Code Generation



Giotto – Periodic Hard-Real-Time Tasks with Precise Mode Changes

Domain was built for Mobies. Major part of the experiment was to interface this domain to others: CT above, FSM below for modal modeling, and SDF for task definition.

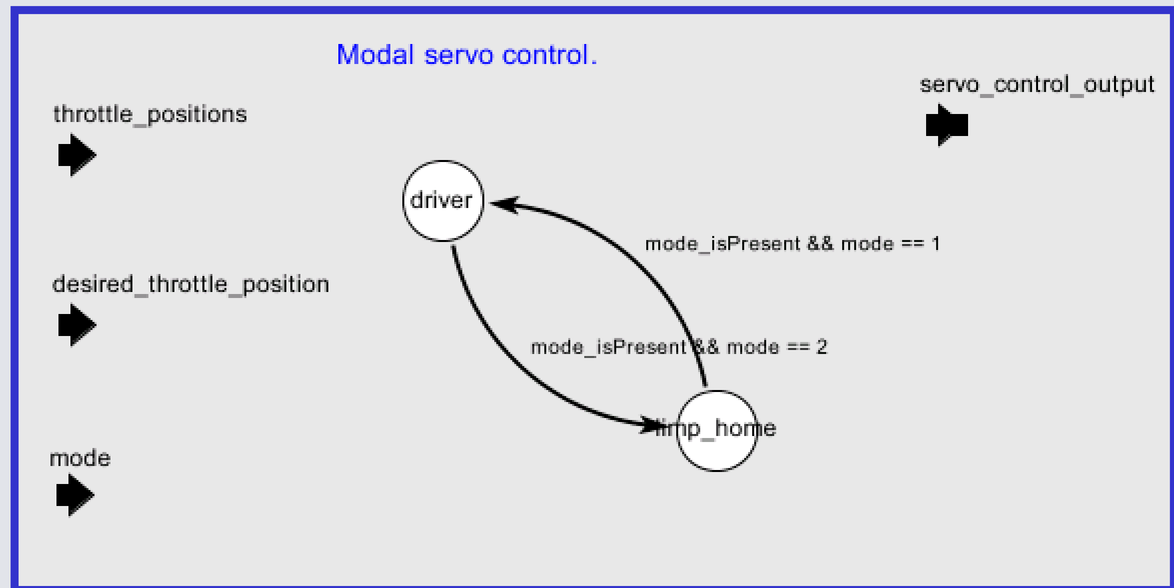


- Giotto compiler targets the E Machine
- First version Ptolemy II Giotto code generator is implemented

Modal Models – The FSM Domain

- Refines components in *any* domain
  - with CT, get hybrid systems
  - with Giotto, get on-line schedule customization
  - with SR, get statecharts semantics
  - with PN, get SDL-style semantics

Design of Giotto domain was greatly simplified by leveraging the FSM domain. We improved the Giotto semantics by introducing modes with limited scope. We learned how to integrate Giotto with other MoCs.

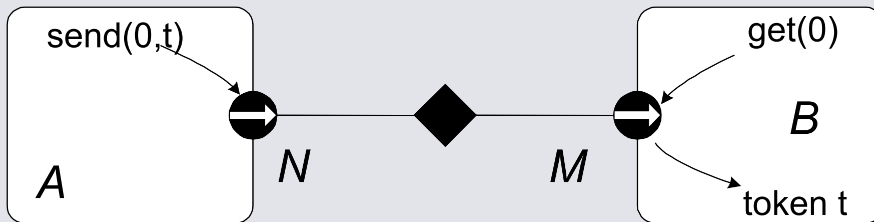


Synchronous Dataflow (SDF) Preferred Domain for Task Definition

- Balance equations (one for each channel):

  $$F_A N = F_B M$$

- Scheduled statically
- Decidable resource requirements



Domains like Giotto, TM, orchestrate large-grain components. The components themselves need not be designed at the low level in C. They can be designed using other Ptolemy II domains.

Available optimizations:

- eliminate checks for input data
- statically allocate communication buffers
- statically sequence actor invocations (and inline)

Code Generation Objective

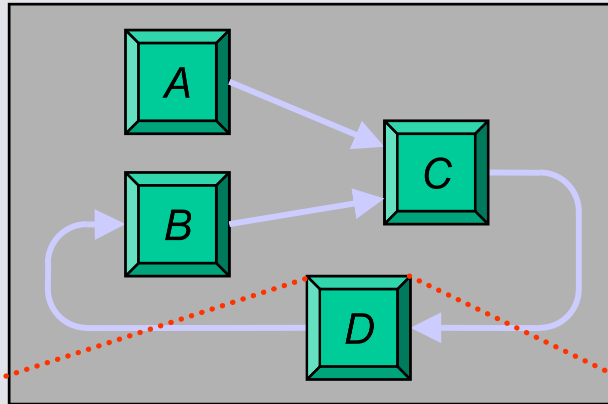
- It is not sufficient to build a mechanism for generating code from one, fixed, modeling environment.
- Modeling strategies must be nested hierarchically.
- Code generators have to be heterogeneously composable.

We aren't there yet, but we have a plan…

Code Generation Status

- Giotto code generator from Giotto domain
  - still need code generation from FSM to get modal models
- Java code generator from SDF domain
  - based on Soot compiler infrastructure (McGill)
  - 80% of SDF test suite passes
  - type specialization
  - static scheduling, buffering
  - code substitution using model of computation semantics
- C code generation from Java
  - University of Maryland subcontract
  - based on Soot compiler infrastructure (McGill)
  - preliminary concept demonstration built
- Configurable hardware synthesis
  - targeted Wildcard as a concept demonstration
  - collaborative with BYU (funded by another program)

Actor Code is the Component Spec



Code generate a domain-polymorophic component definition.

```
public TypedIOPort input;
public TypedIOPort output;
public Parameter constant;
public void fire() {
        Token t = input.get(0);
        Token sum = t.add(constant.getToken());
        output.send(0, t2);
}
```

Actor Definition: Caltrop

- Java is not the ideal actor definition language. Key meta-data is hard to extract:
  - token production/consumption patterns
  - firing rules (preconditions)
  - state management (e.g. recognize stateless actors)
  - type constraints must be explicitly given
  - modal behavior
- Defining an actor definition format (Caltrop):
  - enforce coding patterns
  - make meta-data available for code generation
  - infer behavioral types automatically
  - analyze domain compatibility
  - support multiple back-ends (C, C++, Java, Matlab)

Summary of Accomplishments to Date

- Heterogeneous modeling
  - Domain polymorphism concept & realization
  - Behavioral type system
  - Giotto semantics & integration with other MoCs
  - Component definition principles (Caltrop)
- Code generation
  - Co-compilation concept
  - Giotto program generation
  - Java code generation from SDF
    - 80% of SDF test suite passes
  - C code generation from Java
    - Early phase, concept demonstration

Plans

- Midterm experiment
  - ETC and V2V models and code generators
- Complete actor definition framework
  - define the meta-semantics for domain-polymorphic actors
- Behavioral types
  - reflection
  - real-time properties as dependent types
- Complete SDF code generation
  - token unboxing
  - elimination of memory management
  - 100% of test suite must pass
- Code generate Ptolemy II expressions
  - use of expression actor simplifies models
- Implement FSM code generation
  - support modal models
- Complete C code generation
  - support key subset of Java libraries
- Integrate heterogeneous code generators
  - systematize hierarchy support
  - define Java subset that generates well to C