Overview of Ptolemy II

Principal Investigator:
Edward A. Lee

Technical Staff:
Christopher Hylands
Joern Janneck
John Reekie
Mary P. Stewart
Kees Vissers

Graduate Students:
J. Adam Cataldo
Chris Chang
Elaine Cheong
Xiaojun Liu
Stephen Neuendorffer
James Yeh
Yang Zhao
Haiyang Zheng
Rachel Zhou
Embedded Systems

- Systems that tightly integrate information processing with physical processes, such as:
  - Aircraft
  - Factory robots
  - Cars
  - Trains
  - Appliances

Only 2% of computers today are first and foremost “computers.”
Example: Automotive Engine Control System

- Engine
- Cylinder
- Power Train
- Actuator
- Sensor
- Embedded Controller

Hybrid Automata
Differential Equations
Continuous Time
Timed Multitasking
Discrete Event
Components & Composition

controller

I

C

car model

engine

H

I

C

power

train

E
Actor-Oriented Design

• Components are actors, which are concurrent, parameterized objects
• Actors are composed according to a model of computation (MoC), such as:
  - Communicating sequential processes
  - Continuous time
  - Discrete event
  - Discrete time
  - Process networks
  - Synchronous dataflow
  - Synchronous/reactive
Object-Oriented and Actor-Oriented Design

- **Object orientation**
  - strong typing
  - inheritance
  - procedural interfaces

- **Actor orientation**
  - concurrency
  - communication
  - real time

- These are complementary

**Actor orientation offers:**
- modeling the continuous environment (and hybrid systems)
- understandable concurrency (vs. RPC, semaphores, and mutexes)
- specifications of temporal behavior (vs. “prioritize and pray”)
Ptolemy II - A Software Laboratory

Ptolemy II:
A framework supporting experimentation with
- actor-oriented design
- concurrent semantics
- visual syntaxes

example Ptolemy model: hybrid control system

http://ptolemy.eecs.berkeley.edu
Ptolemy II Infrastructure

• kernel package
  - hierarchical component architecture

• actor package
  - executable interface
  - message-passing interface
  - basic message passing and execution sequencing functionalities that are not specific to a particular model of computation

• support packages
  - data (tokens, type system, expression language)
  - graph (graph algorithms)
  - math (math algorithms)
  - media (audio/video)
  - moml (persistance)
  - plot (signal plotters)
  - vergil (visual editor)
Basic Kernel Classes

```
NamedObj

Entity
- _portList : NamedList
+ connectedPortList() : List
+ connectionsChanged(port : Port)
+ getPort(name : String) : Port
+ linkedRelationList() : List
+ newPort(name : String) : Port
+ portList() : List
+ removeAllPorts()
- _portList : NamedList

Port
- _container : Entity
- _relationsList : CrossRefList
+ connectedPortList() : List
+ insertLink(int : index, relation : Relation)
+ isLinked(r : Relation) : boolean
+ linkedRelationList() : List
+ link(relation : Relation)
+ numLinks() : int
+ unlink(index : int)
+ unlink(relation : Relation)
+ unlinkAll()
  #_checkContainer(container : Entity)
  #_checkLink(relation : Relation)

CrossRefList

Relation
- _portList : CrossRefList
+ linkedPortList() : List
+ linkedPortList(except : Port) : List
+ numLinks() : int
+ unlinkAll()
```
The Ptolemy II kernel defines an abstract syntax - clustered graphs - that is well suited to a wide variety of component-based modeling strategies, ranging from state machines to process networks.
Clustered Graph

Composite entities and ports in Ptolemy II provide a simple and general abstraction mechanism.

The ports deeply connected to the red port are the blue ones.
Mutations

The NamedObj class provides support for

• Processing requests for topology changes
• Delegating change requests to containers
• Registering listeners
• Notifying listeners of topology changes

Thus, models with dynamically changing topologies are cleanly supported. For each model of computation, mutations can be performed in a controlled way.
Actor Package
Object Model for Executable Components

<<Interface>>
Executable
+fire()
+initialize()
+postfire() : boolean
+prefire() : boolean
+preinitialize()
+stopFire()
+terminate()
+wrapup()

<<Interface>>
Actor
+getDirector() : Director
+getExecutiveDirector() : Director
+getManager() : Manager
+inputPortList() : List
+newReceiver() : Receiver
+outputPortList() : List

ComponentEntity

Director

AtomicActor

CompositeEntity

CompositeActor

0..1
0..n
The Executable Interface

execution

- preinitialize()
- initialize()
- iterate()
- wrapup()
- prefire()
- fire()
- postfire()
Hierarchical Heterogeneity:

- Each director implements a model of computation.
- A composite actor with a director becomes opaque.
- The manager provides support for interactive simulation.
Hierarchical vs. Amorphous Heterogeneity

Amorphous

Color is a communication protocol only, which interacts in unpredictable ways with the flow of control.

Hierarchical

Color is a domain, which defines both the flow of control and the communication protocol.
Execution Sequencing

**Execution Sequence: Manager.run()**

<table>
<thead>
<tr>
<th>Initialization</th>
<th>Execution</th>
<th>Wrapup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manager</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- initialize top level</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- process mutations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- check types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- prefire top level</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- ready? yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- fire top level</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- postfire top level</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- done? yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- wrapup top level</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Top level Composite Actor</th>
</tr>
</thead>
<tbody>
<tr>
<td>- initialize director</td>
</tr>
<tr>
<td>- prefire director</td>
</tr>
<tr>
<td>- fire director</td>
</tr>
<tr>
<td>- postfire director</td>
</tr>
<tr>
<td>- wrapup director</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Director</th>
</tr>
</thead>
<tbody>
<tr>
<td>- initialize actors</td>
</tr>
<tr>
<td>- prefire</td>
</tr>
<tr>
<td>- select an actor</td>
</tr>
<tr>
<td>- fire actor</td>
</tr>
<tr>
<td>- postfire actor</td>
</tr>
<tr>
<td>- done?</td>
</tr>
<tr>
<td>- transfer outputs</td>
</tr>
<tr>
<td>- postfire</td>
</tr>
<tr>
<td>- wrapup actors</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opague Composite Actor</th>
</tr>
</thead>
<tbody>
<tr>
<td>- initialize director</td>
</tr>
<tr>
<td>- prefire director</td>
</tr>
<tr>
<td>- request transfer inputs</td>
</tr>
<tr>
<td>- fire director</td>
</tr>
<tr>
<td>- request transfer outputs</td>
</tr>
<tr>
<td>- postfire director</td>
</tr>
<tr>
<td>- wrapup director</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Director</th>
</tr>
</thead>
<tbody>
<tr>
<td>- initialize actors</td>
</tr>
<tr>
<td>- prefire</td>
</tr>
<tr>
<td>- transfer inputs</td>
</tr>
<tr>
<td>- prefire actor</td>
</tr>
<tr>
<td>- fire actor</td>
</tr>
<tr>
<td>- postfire actor</td>
</tr>
<tr>
<td>- done?</td>
</tr>
<tr>
<td>- postfire</td>
</tr>
<tr>
<td>- wrapup actors</td>
</tr>
</tbody>
</table>

**Notes:**
- not ready
- no more actors
- not ready
- no
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 actors.
Actor View of Producer/Consumer Components

Basic Transport:

Supported models of computation:
- push/pull
- continuous time
- dataflow
- rendezvous
- discrete event
- synchronous
- time-driven
- publish/subscribe
- ...

Many actor-oriented frameworks assume a producer/consumer metaphor for component interaction.
Object Model for Actor Messaging
Ptolemy II Domains

- CSP - concurrent threads with rendezvous
- CT - continuous-time modeling
- DDE - distributed discrete-event modeling
- DE - discrete-event systems
- DT - discrete time (cycle driven)
- FSM - finite state machines, modal models
- Giotto - time-triggered real-time systems
- GR - 3D graphics
- PN - process networks
- SDF - synchronous dataflow
- SR - synchronous/reactive
- TM - timed multitasking
Example: 3D Graphics

This model shows a simple pendulum system to illustrate the 3-D graphics and ODE capabilities of Ptolemy II.
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.
Example: Co-Compilation

Giotto code → E code → Run time system

Java code → C code

Java code → Component
Software Practice

- Design and code reviews
- Object models in UML
- Design patterns
- Layered software architecture
- Design document
- Nightly build
- Regression tests
- Code rating
Active Projects

• Actor definition languages and strategies
• Actor composition semantics
• Code generation (co-compilation)
• Behavior types
• Visual syntaxes
• Software engineering
• Domain-specific modeling