Modeling, Simulation, and Design of Concurrent Real-Time Embedded Systems Using Ptolemy

Edward A. Lee

Robert S. Pepper Distinguished Professor
EECS Department
UC Berkeley

Ptutorial

MODPROD, Workshop on Model-Based Product Development
Linköping, Sweden, February 7, 2012
The Ptolemy Project

The Ptolemy project studies modeling, simulation, and design of concurrent, real-time, embedded systems. The focus is on assembly of concurrent components. The key underlying principle in the project is the use of well-defined models of computation that govern the interaction between components. A major problem area being addressed is the use of heterogeneous mixtures of models of computation. A software system called Ptolemy II is being constructed in Java, and serves as the principal laboratory for experimentation.
The Ptolemy Project
Demographics, 2012

Sponsors:
- Government
  - National Science Foundation
  - Army Research Lab
  - DARPA (MuSyC: Multiscale Systems Center)
  - Air Force Research Lab
- Industry
  - Bosch
  - National Instruments
  - SRC (MuSyC: Multiscale Systems Center)
  - Thales
  - Toyota

History:
The project was started in 1990, though its mission and focus has evolved considerably. An open-source, extensible software framework (Ptolemy II) constitutes the principal experimental laboratory.

Staffing:
- 1 professor
- 9 graduate students
- 3 postdocs
- 2 research staff
- several visitors
# Contributors to Ptolemy II

## Principal Authors
- Christopher Brooks
- Dai Bui
- Chamberlain Fong
- John Davis, II
- Patricia Derler
- Thomas Huining Feng
- Mudit Goel
- Rowland Johnson
- Bilung Lee
- Edward Lee
- Ben Lickly
- Jie Liu
- Xiaojun Liu
- Lukito Muliadi
- Stephen Neuendorffer
- John Reekie
- Neil Smyth
- Jeff Tsay
- Yuhong Xiong
- Haiyang Zheng
- Gang Zhou

## Other Contributors
- Jim Armstrong
- Vincent Arnould
- Kyungmin Bae
- Philip Baldwin
- Chad Berkley
- Frederic Boulanger
- Raymond Cardillo
- Jannette Cardoso
- Adam Cataldo
- Christine Cavanessians
- Chris Chang
- Albert Chen
- Chihong Patrick Cheng
- Elaine Cheong
- Colin Cochran
- Brieuc Desoutter
- Pedro Domecq
- William Douglas
- Johan Eker
- Thomas Huining Feng
- Tobin Fricke
- Teale Fristoe
- Shanna-Shaye Forbes
- Hauke Fuhrmann
- Geroncio Galicia
- Ben Horowitz
- Heloise Hse
- Efrat Jaeger
- Jörn Janneck
- Zoltan Kemenczy
- Bart Kienhuis
- Christoph Meyer Kirsch
- Sanjeev Kohli
- Vinay Krishnan
- Robert Kroeger
- Daniel Lázaro Cuadrado
- David Lee
- Man-kit (Jackie) Leung
- Michael Leung
- John Li
- Isaac Liu
- Andrew Mihal
- Eleftherios Matsikoudis
- Aleksandar Necakov
- Mike Kofi Okyere
- Sarah Packman
- Shankar Rao
- Bert Rodiers
- Rakesh Reddy
- Adriana Ricchiuti
- Sonia Sachs
- Ismael M. Sarmiento
- Michael Shilman
- Sean Simmons
- Mandeep Singh
- Miro Spoenemann
- Peter N. Steinmetz
- Dick Stevens
- Mary Stewart
- Ned Stoffel
- Manda Sutijono
- Stavros Tripakis
- Neil Turner
- Guillaume Vibert
- Kees Vissers
- Brian K. Vogel
- Yuke Wang
- Xavier Warzee
- Scott Weber
- Paul Whitaker
- Winthrop Williams
- Ed Willink
- Michael Wirthlin
- Michael Wetter
- William Wu
- Xiaowen Xin
- Paul Yang
- James Yeh
- Nick Zamora
- Charlie Zhong
References

- Ptolemy project home page:
  http://ptolemy.org

- Tutorial: Building Ptolemy II Models Graphically:
  http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-129.html

- Latest release:
  http://ptolemy.org/ptolemyII/ptIIlatest/

- Latest version in the SVN repository:
  http://chess.eecs.berkeley.edu/ptexternal/
Forthcoming Book

Chapters
1. Heterogeneous Modeling
2. Building Graphical Models
3. Dataflow
4. Process Networks and Rendezvous
5. Synchronous/Reactive Models
6. Finite State Machines
7. Discrete Event Models
8. Modal Models
9. Continuous Time Models
10. Cyber-Physical Systems

Appendices
A. Expressions
B. Signal Display
C. The Type System
D. Creating Web Pages
Getting More Information: Documentation

PTOLEMY II
HETEROGENEOUS CONCURRENT MODELING AND DESIGN IN JAVA

Volume 1: User-Oriented

Volume 2: Developer-Oriented

Volume 3: Researcher-Oriented

Tutorial information: http://ptolemy/conferences/07/tutorial.htm
The Ptolemy Pteam
Outline

- Building models
  - Models of computation (MoCs)
  - Creating actors
  - Creating directors
  - Software architecture
  - Miscellaneous topics
Building Models – Hello World
Building more interesting models

DE Director specifies that this will be a discrete-event model
Building more interesting models

Model of regularly spaced events (e.g., a clock signal).
Building more interesting models

Model of irregularly spaced events (e.g., a failure event).
Building more interesting models

Model of a subsystem that changes modes at random (event-triggered) times
Building more interesting models

Model of an observer subsystem
Building more interesting models

Events on the two input streams must be seen in time stamp order.
Ptolemy uses \textit{Superdense} Time

\textit{Discrete event signals can have a sequence of distinct events at a time instant.}

Initial segment $I \subseteq \mathbb{R}_+ \times \mathbb{N}$ where the signal is defined

Absent: $s(\tau) = \varepsilon$ for almost all $\tau \in I$. 
This is a Component Technology

Model of a subsystem given as an imperative program.
This is a Component Technology

Model of a subsystem given as a state machine.
This is a Component Technology

Model of a subsystem given as a modal model.

More types of components:
• Modal models
• Functional expressions.
• Submodels in DE
• Submodels in other MoCs
Continuous-Time Example

Hybrid systems are particularly clean with superdense time. The above signal has multiple values at the times of the transitions.
At each tag, the signal has exactly one value. At each time point, the signal has an infinite number of values. The red arrows indicate value changes between tags, which correspond to discontinuities. Signals are piecewise continuous, in a well-defined technical sense.
Contrast with Simulink/Stateflow

In Simulink, a signal can only have one value at a given time. Hence Simulink introduces solver-dependent behavior.
Outline

- Building models
- Models of computation (MoCs)
- Creating actors
- Creating directors
- Software architecture
- Miscellaneous topics
MoC Example 1: Discrete Events (DE)

DE Director implements timed semantics using an event queue.

In DE, actors send time-stamped events to one another, and events are processed in chronological order.

put() method inserts a token into the event queue.
MoC Example 2: Kahn Process Networks (PN)

In PN, every actor runs in a thread, with blocking reads of input ports and non-blocking writes to outputs.

In the PN domain, each actor executes in its own Java thread. That thread iteratively reads inputs, performs computation, and produces outputs.

**Kahn, MacQueen, 1977**

The output is an ordered sequence of integers of the form $2^n \times 3^m \times 5^k$, where $n$, $m$, and $k$ are non-negative integers.
MoC Example 3: Synchronous Dataflow (SDF)

In SDF, actors “fire,” and in each firing, consume a fixed number of tokens from the input streams, and produce a fixed number of tokens on the output streams.

SDF is a special case of PN where deadlock and boundedness are decidable. It is well suited to static scheduling and code generation. It can also be automatically parallelized.

This example illustrates SDF modeling, which is well-suited to signal processing. In SDF, components communicate using streams, but their production and consumption rates are fixed. Because of these fixed rates, extensive static analysis of the model is possible, enabling efficient code generation and optimization.
MoC Example 4: Synchronous/Reactive (SR)

At each tick of a global “clock,” every signal has a value or is absent.

Like SDF, SR is decidable and suitable for code generation. It is harder to parallelize than SDF, however.

SR languages: Esterel, SyncCharts, Lustre, SCADE, Signal.
MoC Example 5: Rendezvous

In Rendezvous, every actor runs in a thread, with blocking reads of input ports and blocking writes to outputs. Every communication is a (possibly multi-way) rendezvous.

CSP (Hoare), SCCS (Milner), Reo (Arbab)
MoC Example 6: Continuous Time (CT)

In CT, actors operate on continuous-time and/or discrete-event signals. An ODE solver governs the execution.

This model shows a nonlinear feedback system that exhibits chaotic behavior. It is modeled in continuous time. The CT director uses a sophisticated ordinary differential equation solver to execute the model. This particular model is known as a Lorenz attractor.

Director includes an ODE solver.

Signal is a continuous-time function.
Ptolemy II Hierarchy Supports Heterogeneity

Concurrent actors governed by one model of computation (e.g., Discrete Events).

Modal behavior given in another MoC.

Detailed dynamics given in a third MoC (e.g., Continuous Time)

This requires a composable abstract semantics.
Outline

- Building models
- Models of computation (MoCs)
- Creating actors
- Creating directors
- Software architecture
- Miscellaneous topics
Actors
Ptolemy Components are Actors and Objects

The established: Object-oriented:

- **class name**
- **data**
- **methods**

What flows through an object is sequential control

The alternative: Actor oriented:

- **actor name**
- **data (state)**
- **parameters**
- **ports**

What flows through an object is evolving data

Input data  Output data

Things happen to objects

Actors make things happen
Actors

- Ptolemy has a library of predefined actors
- Java classes that implement the “executable” interface
Actors can be defined in Java, C, Python, Cal, and MATLAB

Cal, developed by Joern Janneck (now at Lund) is a language for defining actors that are analyzable for key behavioral properties.

This model demonstrates the use of function closures inside a CAL actor.

The PrimeSieve actor uses nested function closures to realize the Sieve of Eratosthenes, a method for finding prime numbers. Its state variable, “filter,” contains the current filter function. If it is “false” a new prime number has been found, and a new filter function will be generated.

The PrimeSieve actor expects an ascending sequence of natural numbers, starting from 2, as input.
Approach: Concurrent Composition of Software Components, which are themselves designed with Conventional Languages
public class Ptolemnizer extends TypedAtomicActor {
    public Ptolemnizer(CompositeEntity container, String name)
            throws IllegalActionException, NameDuplicationException {
        super(container, name);
        input = new TypedIOPort(this, "input");
        input.setTypeEquals(BaseType.STRING);
        input.setInput(true);
        output = new TypedIOPort(this, "output");
        output.setTypeEquals(BaseType.STRING);
        output.setOutput(true);
    }
    public TypedIOPort input;
    public TypedIOPort output;
    public void fire() throws IllegalActionException {
        if (input.hasToken(0)) {
            Token token = input.get(0);
            String result = ((StringToken)token).stringValue();
            result = result.replaceAll("t", "pt");
            output.send(0, new StringToken(result));
        }
    }
}
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

CompositeEntity

Director

AtomicActor

CompositeActor

0..n
0..1
Definition of the Register Actor (Sketch)

class Register extends TypedAtomicActor {
    private Object state;
    boolean prefire() {
        if (trigger is known) { return true; }
    }
    void fire() {
        if (trigger is present) {
            send state to output;
        } else {
            assert output is absent;
        }
    }
    void postfire() {
        if (trigger is present) {
            state = value read from data input;
        }
    }
}
Outline

- Building models
- Models of computation (MoCs)
- Creating actors
- Creating directors
- Software architecture
- Miscellaneous topics
Directors
Object Model (Simplified) for Communication Infrastructure

IOPort

NoRoomException

Receiver

'+get(): Token
'+getContainer(): IOPort
'+hasRoom(): boolean
'+hasToken(): boolean
'+put(t: Token)
'+setContainer(port: IOPort)

NoTokenException

 PNReceiver

«Interface»

ProcessReceiver

DEReceiver

SDFReceiver

Mailbox

QueueReceiver

CTReceiver

CSPReceiver

PNReceiver

FIFOQueue

ArrayFIFOQueue

Lee, Berkeley 43
Object-Oriented Approach to Achieving Behavioral Polymorphism

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.

Recall: Behavioral polymorphism is the idea that components can be defined to operate with multiple models of computation and multiple middleware frameworks.
Extension Exercise

Build a director that subclasses PNDirector to allow ports to alter the “blocking read” behavior. In particular, if a port has a parameter named “tellTheTruth” then the receivers that your director creates should “tell the truth” when hasToken() is called. That is, instead of always returning true, they should return true only if there is a token in the receiver.

Parameterizing the behavior of a receiver is a simple form of communication refinement, a key principle in, for example, Metropolis.
package doc.tutorial;
import ...

public class NondogmaticPNDirector extends PNDirector {
    public NondogmaticPNDirector(CompositeEntity container, String name)
        throws IllegalActionException, NameDuplicationException {
        super(container, name);
    }

    public Receiver newReceiver() {
        return new FlexibleReceiver();
    }

    public class FlexibleReceiver extends PNQueueReceiver {
        public boolean hasToken() {
            IOPort port = getContainer();
            Attribute attribute = port.getAttribute("tellTheTruth");
            if (attribute == null) {
                return super.hasToken();
            }

            // Tell the truth...
            return _queue.size() > 0;
        }
    }
}
Using It

Model of a sensor sensing a sinusoidal signal with the specified frequency and phase at the specified sampling frequency. This composite actor simulates real-time behavior by sleeping the amount of time given by the samplingPeriod (in seconds) before producing an output.
Designing a Sensible MoC is not so easy! Consider Kahn Process Networks (PN)

- A set of components called *actors*.
- Each representing a sequential procedure.
- Where steps in these procedures receive or send messages to other actors (or perform local operations).
- Messages are communicated asynchronously with unbounded buffers.
- A procedure can always send a message. It does not need to wait for the recipient to be ready to receive.
- Messages are delivered reliably and in order.
- When a procedure attempts to receive a message, that attempt blocks the procedure until a message is available.
Coarse History

- Semantics given by Gilles Kahn in 1974.
  - Fixed points of continuous and monotonic functions
- More limited form given by Kahn and MacQueen in 1977.
  - Blocking reads and nonblocking writes.
- Generalizations to nondeterministic systems
  - Kosinski [1978], Stark [1980s], …
- Bounded memory execution given by Parks in 1995.
  - Solves an undecidable problem.
- Debate over validity of this policy, Geilen and Basten 2003.
  - Relationship between denotational and operational semantics.
- Many related models intertwined.
  - Actors (Hewitt, Agha), CSP (Hoare), CCS (Milner), Interaction (Wegner), Streams (Broy, …), Dataflow (Dennis, Arvind, …)…
Dataflow models are similar to PN models except that actor behavior is given in terms of discrete “firings” rather than processes. A firing occurs in response to inputs.
A few variants of dataflow MoCs

- Computation graphs [Karp and Miller, 1966]
- Static dataflow [Dennis, 1974]
- Dynamic dataflow [Arvind, 1981]
- Structured dataflow [Matwin & Pietrzykowski 1985]
- K-bounded loops [Culler, 1986]
- Synchronous dataflow [Lee & Messerschmitt, 1986]
- Structured dataflow and LabVIEW [Kodosky, 1986]
- PGM: Processing Graph Method [Kaplan, 1987]
- Synchronous languages [Lustre, Signal, 1980’s]
- Well-behaved dataflow [Gao, 1992]
- Boolean dataflow [Buck and Lee, 1993]
- Multidimensional SDF [Lee, 1993]
- Cyclo-static dataflow [Lauwereins, 1994]
- Integer dataflow [Buck, 1994]
- Bounded dynamic dataflow [Lee & Parks, 1995]
- Scenarios [Geilen & Stuijk, 2010]
- …
Some Subtleties

- Termination, deadlock, and livelock (halting)
- Bounding the buffers.
- Fairness
- Parallelism
- Data structures and shared data
- Determinism
- Real-time constraints
- Syntax
Question 1:
Is “Fair” Scheduling a Good Idea?

In the following model, what happens if every actor is given an equal opportunity to run?
Question 2: Is “Data-Driven” Execution a Good Idea?

In the following model, if actors are allowed to run when they have input data on connected inputs, what will happen?
Question 3:
When are Outputs Required?

Is the execution shown for the following model the “right” execution?
Question 4: Is “Demand-Driven” Execution a Good Idea?

In the following model, if actors are allowed to run when another actor requires their outputs, what will happen?
Question 5: What is the “Correct” Execution of This Program?
Question 6: What is the Correct Behavior of this Program?
Naïve Schedulers Fail

- Fair
- Demand driven
- Data driven
- Most mixtures of demand and data driven

*If people insist on building their own MoCs from scratch, what will keep them from repeating the mistakes that have been made by top experts in the field?*
Programmers should not have to figure out how to solve these problems!

*Undecidability and Turing Completeness* [Buck 93]

Given the following four actors and Boolean streams, you can construct a universal Turing machine:

![Diagram of four actors: Boolean function, SampleDelay, BooleanSelect, BooleanSwitch]

Hence, the following questions are undecidable:

- Will a model deadlock (terminate)?
- Can a model be executed with bounded buffers?
Question 7: How to support nondeterminism?

Merging of streams is needed for some applications. Does this require fairness? What does fairness mean?
These problems have been solved! Let’s not make programmers re-solve them for every program.

Library of directors

Program using actor-oriented components and a PN MoC

Directors should be designed by experts in languages and concurrency, not by application model builders.
The PN Director solves the above problems by implementing a “useful execution”

Define a **correct execution** to be any execution for which after any finite time every signal is a prefix of the signal given by the (Kahn) least-fixed-point semantics.

Define a **useful execution** to be a correct execution that satisfies the following criteria:

1. For every non-terminating model, after any finite time, a useful execution will extend at least one stream in finite (additional) time.
2. If a correct execution satisfying criterion (1) exists that executes with bounded buffers, then a useful execution will execute with bounded buffers.
Our solution:
Parks’ Strategy [Parks 95]

This “solves” the undecidable problems:

- Start with an arbitrary bound on the capacity of all buffers.
- Execute as much as possible.
- If deadlock occurs and at least one actor is blocked on a write, increase the capacity of at least one buffer to unblock at least one write.
- Continue executing, repeatedly checking for deadlock.

This delivers a useful execution (possibly taking infinite time to tell you whether a model deadlocks and how much buffer memory it requires).
There are many more subtleties!

*We need disciplined concurrent models of computation, not arbitrarily flexible libraries.*

Some principles:

- Do not use nondeterministic programming models to accomplish deterministic ends.

- Use concurrency models that have analogies in the physical world (actors, not threads).

- Provide these in the form of models of computation (MoCs) with well-developed semantics and tools.

- Use specialized MoCs to exploit semantic properties (avoid excess generality).

- Leave the choice of shared memory or message passing to the compiler.
Extension Exercise 2

Build a director that subclasses Director and allows different receiver classes to be used on different connections. This is a form of what we call “amorphous heterogeneity.”

We will not do this today.
See $PTII/doc/tutorial/domains
Extension Exercise 3

Build a director that fires actors in left-to-right order, as they are laid out on the screen.

We will not do this today.
See $PTII/doc/tutorial/domains
Outline

- Building models
- Models of computation (MoCs)
- Creating actors
- Creating directors
- Software architecture
- Miscellaneous topics
Ptolemy II Software Architecture
Built for Extensibility

Ptolemy II packages have carefully constructed dependencies and interfaces.
Hierarchy - Composite Components

- toplevel CompositeEntity
  - transparent or opaque CompositeEntity
  - Relation
  - dangling Port
  - Port
  - opaque Port

Lee, Berkeley 75
Separable Tool Architecture

- Abstract Syntax
- Concrete Syntax
- Abstract Semantics
- Concrete Semantics
The Basic Abstract Syntax for Composition

- Entities
- Attributes on entities (parameters)
- Ports in entities
- Links between ports
- Width on links (channels)
- Hierarchy

Concrete syntaxes:
- XML
- Visual pictures
- Actor languages (Cal, StreamIT, …)
Meta Model: Kernel Classes
Supporting the Abstract Syntax

These get subclassed for specific purposes.
Separable Tool Architecture

- Abstract Syntax
- Concrete Syntax
- Abstract Semantics
- Concrete Semantics
Ptolemy II designs are represented in XML:

```xml
...<entity name="FFT" class="ptolemy.domains.sdf.lib.FFT">
  <property name="order" class="ptolemy.data.expr.Parameter" value="order">
    </property>
  <port name="input" class="ptolemy.domains.sdf.kernel.SDFIOPort">
    ...
  </port>
  ...
</entity>
...
<brink port="FFT.input" relation="relation"/>
<link port="AbsoluteValue2.output" relation="relation"/>
...
Separable Tool Architecture

- Abstract Syntax
- Concrete Syntax
- Abstract Semantics
- Concrete Semantics
Abstract Semantics (Informally) of Actor-Oriented Models of Computation

Actor-Oriented Models of Computation that we have implemented:

- dataflow (several variants)
- process networks
- distributed process networks
- Click (push/pull)
- continuous-time
- CSP (rendezvous)
- discrete events
- distributed discrete events
- synchronous/reactive
- time-driven (several variants)
- …
Implemented as a Java interface

**Interface “Executable”**

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>void</strong> fire()</td>
</tr>
<tr>
<td>Fire the actor.</td>
</tr>
<tr>
<td><strong>boolean</strong> isFireFunctional()</td>
</tr>
<tr>
<td>Return true if this executable does not change state in either the prefire() or the fire() method.</td>
</tr>
<tr>
<td><strong>boolean</strong> isStrict()</td>
</tr>
<tr>
<td>Return true if this executable is strict, meaning all inputs must be known before iteration.</td>
</tr>
<tr>
<td><strong>int</strong> iterate(int count)</td>
</tr>
<tr>
<td>Invoke a specified number of iterations of the actor.</td>
</tr>
<tr>
<td><strong>boolean</strong> postfire()</td>
</tr>
<tr>
<td>This method should be invoked once per iteration, after the last invocation of fire() in that iteration.</td>
</tr>
<tr>
<td><strong>boolean</strong> prefire()</td>
</tr>
<tr>
<td>This method should be invoked prior to each invocation of fire().</td>
</tr>
<tr>
<td><strong>void</strong> stop()</td>
</tr>
<tr>
<td>Request that execution of this Executable stop as soon as possible.</td>
</tr>
<tr>
<td><strong>void</strong> stopFire()</td>
</tr>
<tr>
<td>Request that execution of the current iteration complete.</td>
</tr>
<tr>
<td><strong>void</strong> terminate()</td>
</tr>
<tr>
<td>Terminate any currently executing model with extreme prejudice.</td>
</tr>
</tbody>
</table>
Example execution sequence

FIGURE 2.14. Example execution sequence implemented by run() method of the Director class.
How Does This Work?
Execution of Ptolemy II Actors

Flow of control:
- Initialization
- Execution
- Finalization
How Does This Work?
Execution of Ptolemy II Actors

Flow of control:
- Initialization
- Execution
- Finalization

E.g., in DE: Post tags on the event queue corresponding to any initial events the actor wants to produce.
How Does This Work? Execution of Ptolemy II Actors

Flow of control:
- Initialization
- Execution
- Finalization

Iterate

If (prefire()) {
  fire();
  postfire();
}

Only the postfire() method should change the state of the actor.
How Does This Work?
Execution of Ptolemy II Actors

Flow of control:
- Initialization
- Execution
- Finalization
Definition of the Register Actor (Sketch)

class Register extends TypedAtomicActor {
    private Object state;
    boolean prefire() {
        if (trigger is known) { return true; }
    }
    void fire() {
        if (trigger is present) {
            send state to output;
        } else {
            assert output is absent;
        }
    }
    void postfire() {
        if (trigger is present) {
            state = value read from data input;
        }
    }
}

Can the actor fire?
React to trigger input.
Read the data input and update the state.
Separable Tool Architecture

- Abstract Syntax
- Concrete Syntax
- Abstract Semantics
- Concrete Semantics
Models of Computation Implemented in Ptolemy II

- CI – Push/pull component interaction
- Click – Push/pull with method invocation
- CSP – concurrent threads with rendezvous
- Continuous – continuous-time modeling with fixed-point semantics
- CT – continuous-time modeling
- DDF – Dynamic dataflow
- DE – discrete-event systems
- DDE – distributed discrete events
- DPN – distributed process networks
- FSM – finite state machines
- DT – discrete time (cycle driven)
- Giotto – synchronous periodic
- GR – 3-D graphics
- PN – process networks
- Rendezvous – extension of CSP
- SDF – synchronous dataflow
- SR – synchronous/reactive
- TM – timed multitasking

Most of these are actor oriented.
Outline

- Building models
- Models of computation (MoCs)
- Creating actors
- Creating directors
- Software architecture
- Miscellaneous topics
Example Extensions
Using Models to Control Models

This model illustrates the use of a "run composite actor" component. That component contains another Ptolemy II model. Each time it fires, it performs a complete execution of that other Ptolemy II model, rather than just one firing as would be typical of a composite actor.

This model generates Lissajous figures, which are plots of one sinusoid vs. another. On each execution, it generates one figure.

This is an example of a “higher-order component,” or an actor that references one or more other actors.

MultipleRuns SDF demo
Examples of Extensions
Mobile Models

Model-based distributed task management:

**PushConsumer actor** receives pushed data provided via CORBA, where the data is an XML model of a signal analysis algorithm.

**MobileModel actor** accepts a StringTokenizer containing an XML description of a model. It then executes that model on a stream of input data.

*Authors:*
Yang Zhao
Steve Neuendorffer
Xiaojun Liu
Ptolemy II Extension Points

- Define actors
- Interface to foreign tools (e.g. Python, MATLAB)
- Interface to verification tools (e.g. Chic)
- Define actor definition languages
- Define directors (and models of computation)
- Define visual editors
- Define textual syntaxes and editors
- Packaged, branded configurations

All of our “domains” are extensions built on a core infrastructure.
Extension of Discrete-Event Modeling for Wireless Sensor Nets

VisualSense extends the Ptolemy II discrete-event domain with communication between actors representing sensor nodes being mediated by a channel, which is another actor.

The example at the left shows a grid of nodes that relay messages from an initiator (center) via a channel that models a low (but non-zero) probability of long range links being viable.
Viptos: Extension of VisualSense with Programming of TinyOS nodes

Viptos demo: Multihop routing (Surge)

Physical environment Simulation (with visualization of routing tree)

Hardware

Software Code generation: Models to nesC.

Another Extension: HyVisual – Hybrid System Modeling Tool Based on Ptolemy II

HyVisual was first released in January 2003.
Another Extension: Kepler: Aimed at Scientific Workflows

Key capabilities added by Kepler:
- Database interfaces
- Data and actor ontologies
- Web service wrappers
- Grid service wrappers
- Semantic types
- Provenance tracking
- Authentication framework

This example shows the use of data ontologies and database wrappers.
CPES Fusion Simulation Workflow

- **Fusion Simulation Codes:** (a) GTC; (b) XGC with M3D
  - e.g. (a) currently 4,800 (soon: 9,600) nodes Cray XT3; 9.6TB RAM; 1.5TB simulation data/run
- **GOAL:**
  - automate remote simulation job submission
  - continuous file movement to analysis cluster for dynamic visualization & simulation control
  - … with runtime-configurable observables

**Select JobMgr**
Submit Simulation Job
Submit FileMover Job
WaitForSimFinish
WaitForMoveFinish
Execution Log (=> Data Provenance)

Overall architect (& prototypical user): Scott Klasky (ORNL)
WF design & implementation: Norbert Podhorszki (UC Davis)
Leverage: Kepler is a Team Effort

Contributor names and funding info are at the Kepler website: http://kepler-project.org

Other contributors:
- Chesire (UK Text Mining Center)
- DART (Great Barrier Reef, Australia)
- National Digital Archives + UCSD-TV (US)
- ...
Graph Transformation

Model transformation workflow specifies iterative graph rewriting to transform the top-right model into the bottom-left model.

Executing the model at the left transforms the top model into the bottom model.

This model demonstrates how one can possibly optimize a model. The original input is the model in BaseModel.xml, which the FileReader actor reads in. The contents of this model are then converted into an ActorToken by the ModelGenerator. OptimizeOnce is a transformation rule that gets repeatedly applied to this model until no further optimization is possible (i.e., a fixpoint is reached). In each application, two_consts that are wired to an AddSubtract actor, a MultiplyDivide actor, or a Maximum actor are replaced by a single Const with the statically computed value.

Author: Thomas Huining Feng (Inspired by Thomas Mandl)
Workflows

Here we have used Event-Relationship graphs [Schruben 83] to specify the workflow logic.
Some Current Research Thrusts in the Ptolemy Project

- **Precision-timed (PRET) machines**: Introduce timing into the core abstractions of computing, beginning with instruction set architectures, using configurable hardware as an experimental platform.

- **Distributed real-time computing (PTIDES)**: Models of computation based on distributed discrete events, embedded OS (PtidyOS), analysis and synthesis techniques.

- **Model engineering**: Modeling and design of large scale systems, those that include networking, database, grid computing, and information subsystems.

- **Semantics of concurrent and real-time systems**: Mathematical models of programs in conjunction with models of their physical environment.
Forthcoming Book

Chapters
1. Heterogeneous Modeling
2. Building Graphical Models
3. Dataflow
4. Process Networks and Rendezvous
5. Synchronous/Reactive Models
6. Finite State Machines
7. Discrete Event Models
8. Modal Models
9. Continuous Time Models
10. Cyber-Physical Systems

Appendices
A. Expressions
B. Signal Display
C. The Type System
D. Creating Web Pages

http://Ptolemy.org