A Practical Introduction to Kepler

Guest Lecturer: Ilkay Altintas
altintas@sdsc.edu
San Diego Supercomputer Center, UCSD

Scientific Workflows

• Goals:
  – automate a scientist's repetitive data management and analysis tasks
• Typical phases:
  – data access, scheduling, generation, transformation, aggregation, analysis, visualization
  – design, test, share, deploy, execute, reuse, … SWFs

SWF Systems Requirements

USER REQUIREMENTS:
  – Design tools— especially for non-expert users
    • Need to look into how scientist's define their processes
  – Ease of use— fairly simple user interface having more complex features hidden in the background
  – Reusable generic features
  – Generic enough to serve to different communities but specific enough to serve one domain (e.g. geosciences, molecular biology)
  – Extensibility for the expert user— almost a visual programming interface
  – Registration and publication of data products and "process products" (=workflows); provenance

TECHNICAL REQUIREMENTS:
  – Error detection and recovery from failure
  – Logging information for each workflow
  – Allow data-intensive and compute-intensive tasks (Maybe at the same time)
  – HPC + Data management/integration
  – Allow status checks and on the fly updates
  – Visualization
  – Semantics and metadata based dataset access
  – Certification, trust, security…
Kepler based on Ptolemy II

- A set of Java packages for heterogeneous, concurrent modeling, design and execution.
- Strengths include:
  - Precisely defined models of computation and component interaction
  - E.g. Process Networks (PN) – data-flow oriented
  - An intuitive GUI that lets rapid workflow composition
  - A modular, reusable and extendable object-oriented environment
  - An XML based workflow definition – MoML
    - Workflows defined in Ptolemy II MoML XML schema
    - Easily exchangeable

Abstract Syntax of PTII Models

- Hierarchical Entities, Ports, Connections and Attributes
  Abstract syntax choices:
  - Hierarchy is tree structured (like XML).
  - A relation mediates connections.
  - Ports can link multiple relations and relations can link multiple ports.
  - Ports mediate connections across levels of the hierarchy (no statecharts-style level-crossing links)

The GUI -- Vergil

The rest... outline

- How to get and install Kepler
- Designing a Kepler workflow
  - HOW TO best do it?
- Some demos
- Building actors
Installing Kepler

- Kepler website: http://kepler-project.org
- Latest alpha release at:
  http://kepler-project.org/Wiki.jsp?page=Downloads
  Installers for Windows, MacOSX and Linux
  Install and run Kepler's .exe file in the Kepler directory it was installed.
- Issues:
  - Some workflows don't run on the fly
  - Local files dependencies, username/password requirements, broken
- Alternative installation for a more recent version:
  - Contact the Kepler cvs admin
  - Get a read-only account
  - Build it from scratch
    - Eclipse instructions at:
      http://kepler-project.org/Wiki.jsp?page=usingEclipseForKeplerDevelopment
    - Command line building using Ant
      Tutorial: http://kepler-project.org/Wiki.jsp?page=presentations

Designing Your Workflows in Kepler

- Write down the problem
- Generate a conceptual design of the workflow
  - Data flow: Task1 -> Task2->..->Taskn
  - Data requirements: Types of data, I/O for each task
- Look for existing Kepler actors for each task
- If there are related tasks, think how to use them; If not, design the stub actors
- Design the workflow using existing and stub actors
- Specify parts that can be sub-workflows and create hierarchies (composite actors)
- Implement the missing actors
- Run tests on the wf for a set of inputs and many times;
  - to check if it executes correctly, and if it produces the same results
  - for the same inputs
- Annotate your workflow and improve usability

Building Kepler Extensions

FOCUS: How to build actors?

Ingredients:
- Java 1.4.2
- Cygwin for Developers (for Windows users)
- Ant 1.5 or higher

Adding Actors

- Domain and/or data polymorphic actors
- Use object-oriented inheritance to avoid code duplication
  - 3 base classes: Source, Sink, Transformer
- To use the actors in Vergil
  - Add them to one of the actor libraries
  - Most libraries are under $PTII/ptolemy/actor/lib
  - Libraries are XML files
  - In Kepler, this needs to be done through the actor ontology!!!
- The basic structure of an actor:
  - See http://www.sdsc.edu/~altintas/KeplerTutorial/ActorStructure.txt
Actor Interfaces: Ports & Parameters

- Parameters:
  - $a_1 = \text{value}$
  - $a_2 = \text{value}$

Example:

Anatomy of an Actor: Ports

- Used for “message transport”, can be an input, an output, or both.
- Key class: IOPort (Can be connected to other IOPort instances via IORelations.)
- Use TypedIOPort in order to benefit from the type system! (Domain specific: DEIOPort)
- Receiver and Sender interfaces depending on the usage of the port.
- Public members of the actors!!!

Ports: Introduction to the API

```java
public TypedIOPort portName; //Definition
//Create the port
portName = new TypedIOPort(this, "portName", true, false);
portName.setMultiport(true); //Can support more than one link
int width = portName.getWidth(); //0 or 1 if single port
//Reading and Writing
portName.send(channelNumber, token);
Token token = portName.get(channelNumber);
//Setting the type of the port
portName.setTypeEquals(BaseType…a type in the type system…);
portName.setTypeAtLeast(…must be a port or parameter…);
```

Anatomy of an Actor: Parameters

- Public members of the actors!
- Similar API with ports...

```java
public Parameter parameterName; //Definition
//Creation and setting the initial value: 2-ways
parameterName = new Parameter (this, "parameterName", new Token(…token value…));
OR
parameterName = new Parameter (this, "parameterName");
parameterName.setExpression(…tokenValue…);
```
Anatomy of an Actor: Constructors

• The major task:
  – To create and configure ports and parameters
• `super(container, name);`
  – Carries the NameDuplication and IllegalAction exceptions from the super class.
• The icon for the actor can be set here.

Execution of an actor

```
initialize()  
\downarrow
prefire

Iterations
\downarrow
fire()

\downarrow
postfire()

wrapUp()
```

Action Methods

• `initialize()`: Initialize the state variables of an actor.
• `prefire()`: Returns a boolean which indicates if the actor wants to fire.
  – Can also be used to perform an operation that will happen exactly once per iteration.
• `fire()`: The main point of execution.
  – For reading inputs, producing outputs, read the current parameter values.

Action Methods (cont.)

• `postfire()`: Has two tasks:
  – Updating persistent state
  – Determining whether the execution of an actor is complete.
• `wrapUp()`: For displaying final results.
Things to remember when implementing a fire() method

- Use the methods of the Token class for arithmetic whenever possible (to get data polymorphism)
- If data-polymorphism is not necessary, set the type to a base type then cast the token to that type.
- Cannot assume that there is be data available at all the input ports (for domain-polymorphism)
- Do not update the persistent state in fire() (use postfire())

Implementing Polymorphism

- Class: PortParameterFunction
  - A PortParameterFunction object will be returned as a function of two objects.
  - Set the type of the output equal to the type of this object.
  - Type system will compute the type of the PortParameterFunction object and use it as the type of the output when necessary.

The manager

- Controls the overall execution of a model.
- Interacts only with the "top-level composite actor"
- startRun() -> run() -> execute()
- ExecutionListener interface provides the manager with info on the events generated during execution.

Exceptions

- A uniform mechanism for reporting errors
- Base class: KernelException
- Exception chaining re-implemented since Java versions < 1.4 doesn’t support it.
  - The detail message includes the detail message from the cause argument.
  - A protected _setCause() method is implemented, but not the public initCause() method that JDK1.4 has.
- Non-severe exceptions: IllegalActionException, NameDuplicationException, NameDuplicationException.
- Severe-exceptions: KernelRuntimeError, InvalidStateException, InternalError

*Ptolemy design doc... show how Ptolemy II handles mixing models of computations hierarchically.*
Actor-Oriented Design

Focus on Actor-Oriented Design

Object orientation:
- Class name
- Data
- Methods
- Call
- Return

What flows through an object is sequential control

Actor orientation:
- Actor name
- Data (state)
- Parameters
- Input data
- Output data

What flows through an object is streams of data

Layered Software Architecture

Ptolemy II packages have carefully constructed dependencies and interfaces

Ptolemy II Architecture

- Core packages (actor, data, kernel, math, util)
  - The data model (abstract syntax) of the models
  - Abstract semantics
- User Interface (UI) packages
  - MoML and visual interface support
- Library packages
  - Domain polymorphic actors
- Domain packages
  - Implementations of different models of computation
Overview of the Key Classes

Models of Computation

- Semantic interpretations of the abstract syntax
  Different models ↔ Different semantics ↔ Different execution

Models of Computation:
- continuous-time
- dataflow
- rendezvous
- synchronous
- time-driven
- publish/subscribe
  ...

One Class of Semantic Models: Producer / Consumer

process {
  write();
  ...
}

process {
  read();
  ...
}

channel
port
receiver

Director

- Governs the execution of a composite entity.
  - Scheduling, dispatching threads, generate code, etc.
- A composite entity is called opaque if it doesn’t have a local director.
  - An opaque composite entity inherits the director of its container as its executive director.

MoML

- Modeling Markup Language
- A primary persistent XML file format for Ptolemy II.

- ptolemy filename.xml
- ptexecute filename.xml
- vergil filename.xml
- moml configuration.xml filename.xml
An Example MoML File

```xml
<?xml version="1.0" standalone="no"/>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">
<entity name="test" class="ptolemy.actor.TypedCompositeActor">
  <property name="director" class="ptolemy.domains.sdf.kernel.SDFDirector"/>
  <entity name="ramp" class="ptolemy.actor.lib.Ramp"/>
  <entity name="plot" class="ptolemy.actor.lib.gui.SequencePlotter"/>
  <relation name="r" class="ptolemy.actor.TypedIORelation"/>
  <link port="ramp.output" relation="r"/>
  <link port="plot.input" relation="r"/>
</entity>
```

Type System

Reference: Ptolemy II Design Document-1, Chapter 6