An introduction to the CAL actor language

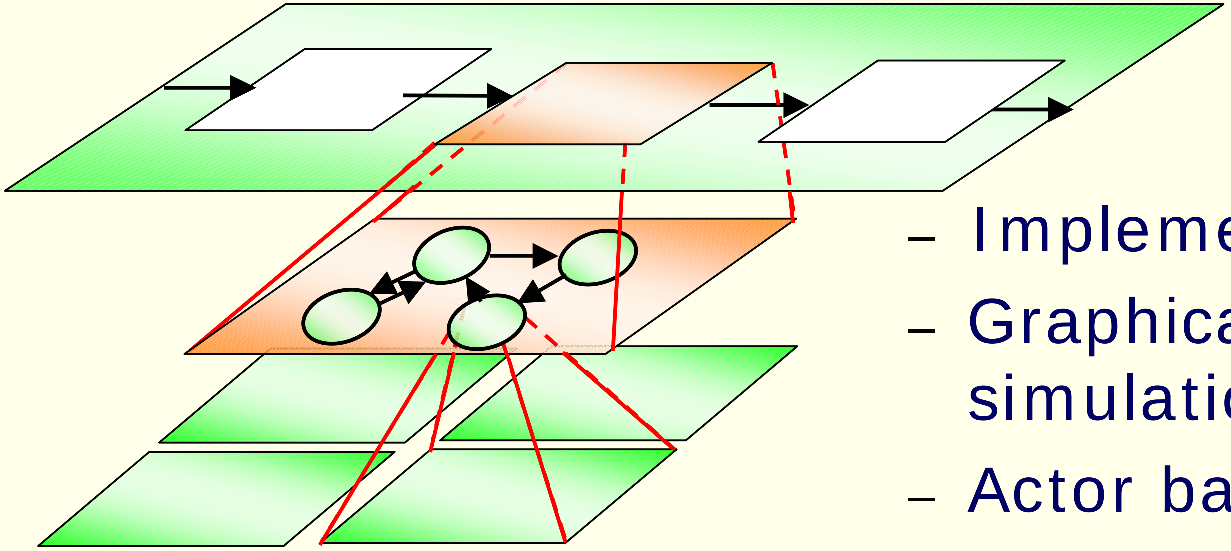
Yang Zhao

May 2, 2002

What is CAL

- A small domain-specific language
- To provide a concise high-level description of an actor
- To be embedded in a host environment or language
- As part of the Ptolemy project
- Under development

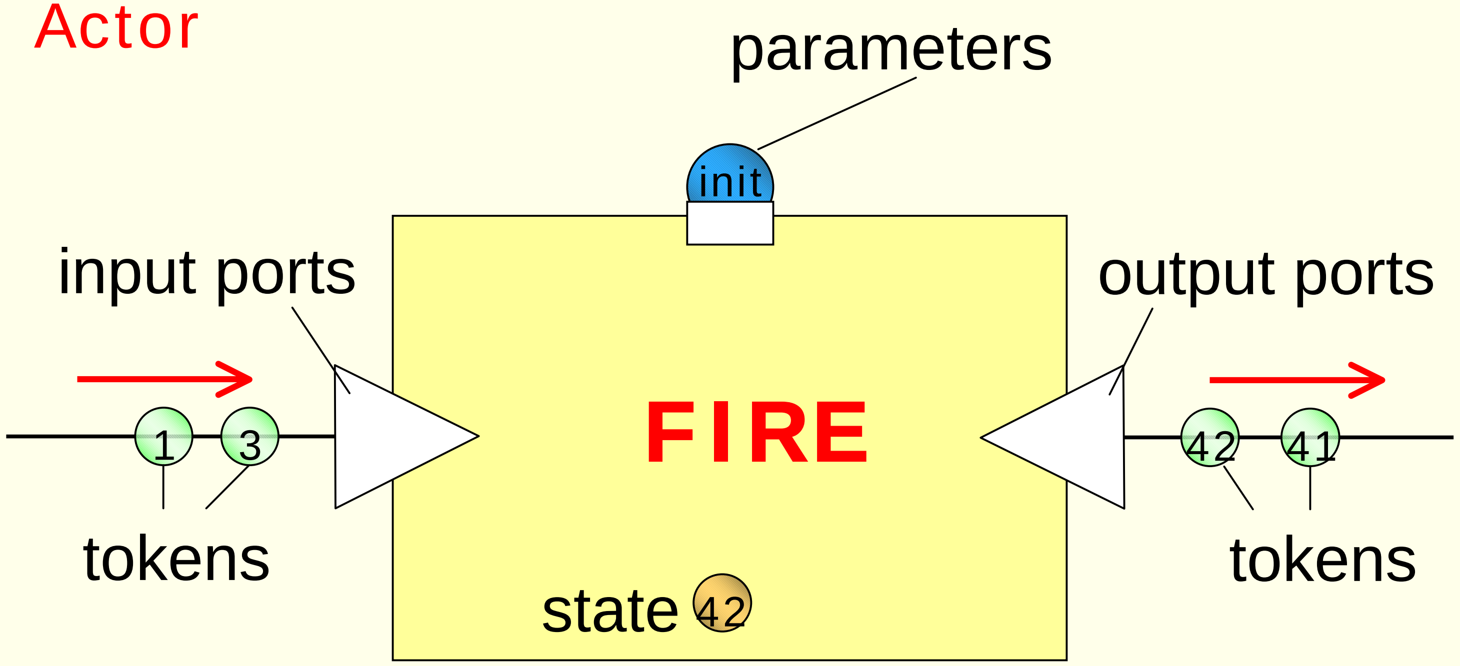
A brief view of Ptolemy II



- Implemented in Java
- Graphical modeling and simulation environment
- Actor based models
- Multiple “models of computation”
- Hierarchical & heterogeneous models

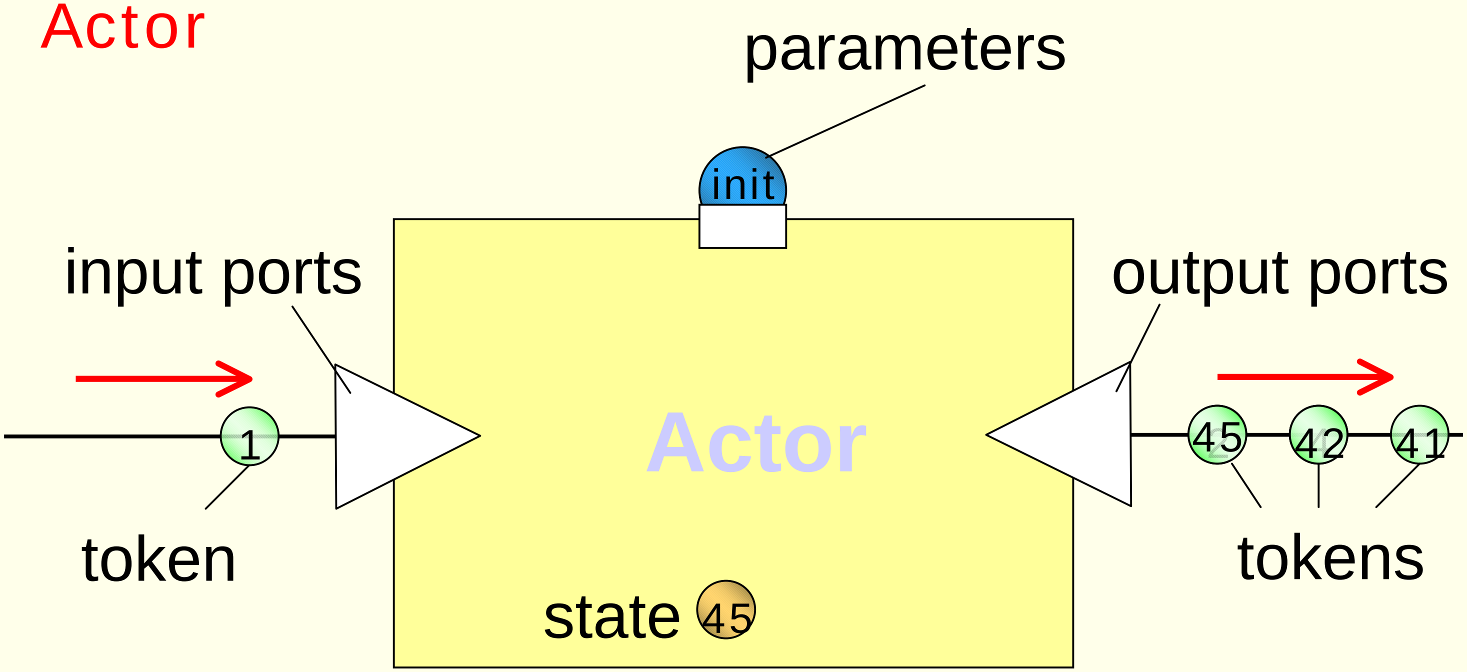
Ptolemy II Basics

- A model is a a set of interconnected *actors* and one *director*
- Actor



Ptolemy II Basics

- A model is a a set of interconnected *actors* and one *director*
- Actor

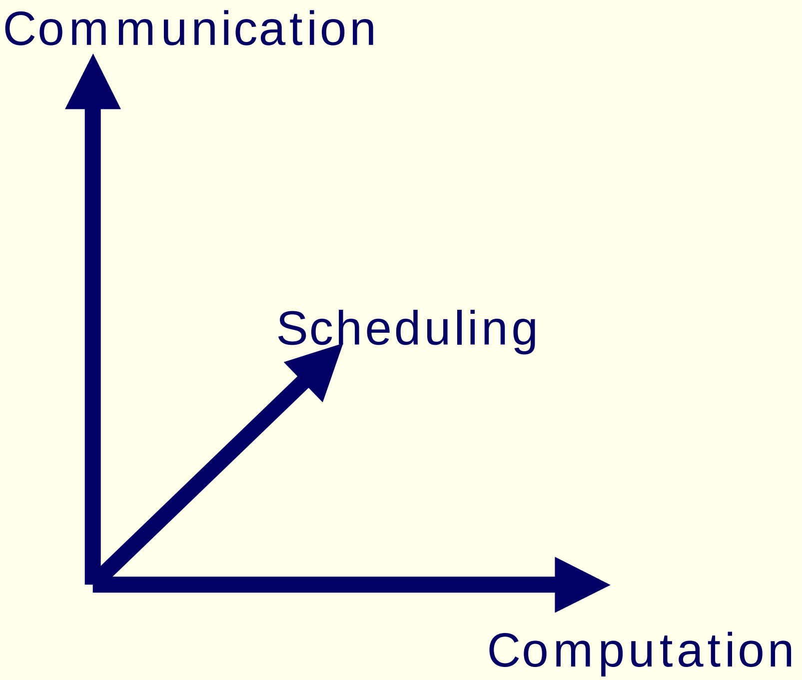


Ptolemy II Basics

- Director
  - – Manages the data flow and the scheduling of the actors
  - – The director fires the actors
- Receiver
  - – Defines the semantics of the port buffers
- Models of Computation
  - – Define the interaction semantics
  - – Implemented in Ptolemy II by a *domain*
    - Director + Receiver

Key of Ptolemy II

- Orthogonalize the concerns



Writing a Ptolemy actor

```
/* An actor that outputs the sum of the inputs so far.*/

package ptolemy.actor.lib;

import ptolemy.actor.TypedIOPort;
import ptolemy.kernel.CompositeEntity;
Import ......

public class Sum extends Transformer {
    public Sum(CompositeEntity container, String name)
            throws NameDuplicationException, IllegalActionException  {
        super(container, name);
        input.setMultiport(true);
        init = new Parameter(this, "init", new IntToken(0));
        output.setTypeAtLeast(init);
        output.setTypeAtLeast(input);
    }
    /////////////        ports and parameters        ///////////////
    public Parameter init;

    /////////////           public methods           ///////////////

    public Object clone(Workspace workspace)
              throws CloneNotSupportedException {
        Accumulator newObject = (Accumulator)super.clone(workspace);
```

Continued:

```
          newObject.output.setTypeAtLeast(newObject.init);
          newObject.output.setTypeAtLeast(newObject.input);
        return newObject;
    }

    public void fire() throws IllegalActionException {
        _latestSum = _sum;
        for (int i = 0; i < input.getWidth(); i++) {
            if (input.hasToken(i)) {
                Token in = input.get(i);
                _latestSum = _latestSum.add(in);
            }
        }
        output.broadcast(_latestSum);
    }

    public void initialize() throws IllegalActionException {
        super.initialize();
        _latestSum = _sum = init.getToken();
    }

    public boolean postfire() throws IllegalActionException {
        _sum = _latestSum;
        return super.postfire();
    }

    ///////////////////////////////////////////////////////////////////
    ////                         private variables                   ////

    private Token _sum;
    private Token _latestSum;
}
```

Writing Ptolemy actors in Java..

- requires certain knowledge about the Ptolemy II API
- results in platform specific classes
- is error-prone
- is often redundant
- makes it hard to extract information from the actors

→ Specifying actors in Java is problematic

A better approach: CAL

We should be able to **generate** actors from a more abstract description.

- CAL provides a concise high-level description of an actor.
- CAL is a textual language for writing down the functionality of actors.
  - input ports, output ports
  - parameters, states
  - typing constraints
  - firing rules

Introduction to CAL

The sum actor written in CAL:

```
actor sum (Integer init) Integer A ==> Integer B:
  Integer sum := init;

  action [a] ==> [sum]  do
    sum := sum + a;
  endaction
endactor
```

Introduction to CAL

actor name:

```
actor sum (Integer init) Integer A ==> Integer B:
  Integer sum := init;

  action [a] ==> [sum]  do
    sum := sum + a;
  endaction
endactor
```

Introduction to CAL

parameters:

```
actor sum (Integer init) Integer A ==> Integer B:
  Integer sum := init;

  action [a] ==> [sum]  do
    sum := sum + a;
  endaction
endactor
```

Introduction to CAL

input ports:

```
actor sum (Integer init) Integer A ==> Integer B:
  Integer sum := init;

  action [a] ==> [sum]  do
    sum := sum + a;
  endaction
endactor
```

Introduction to CAL

output ports:

```
actor sum (Integer init) Integer A ==> Integer B:
  Integer sum := init;

  action [a] ==> [sum]  do
    sum := sum + a;
  endaction
endactor
```

Introduction to CAL

states:

```
actor sum (Integer init) Integer A ==> Integer B:
  Integer sum := init;

  action [a] ==> [sum]  do
    sum := sum + a;
  endaction
endactor
```

Introduction to CAL

actions:

```
actor sum (Integer init) Integer A ==> Integer B:
  Integer sum := init;

  action [a] ==> [sum]  do
    sum := sum + a;
  endaction
endactor
```

Introduction to CAL

input pattern:

```
actor sum (Integer init) Integer A ==> Integer B:
  Integer sum := init;

  action [a] ==> [sum]  do
    sum := sum + a;
  endaction
endactor
```

Introduction to CAL

type:

```
actor sum (Integer init) Integer A ==> Integer B:
  Integer sum := init;

  action [a] ==> [sum]  do
    sum := sum + a;
  endaction
endactor
```

Introduction to CAL

The sum actor with type variable T:

```
actor sum[T] (T init) T A ==> T B:
  Integer sum := init;

  action [a] ==> [sum]  do
    sum := sum + a;
  endaction
endactor
```

Structure of a CAL actor

```
actor actorname[type parameters] (actor parameters)
      type input ports ==> type output ports:
State
Initialization statements

action inputpatterns ==> [outputexpressions]
  guard guard conditions do
    action statements
endaction
……

Action……
  do
    ……
endaction

endactor
```

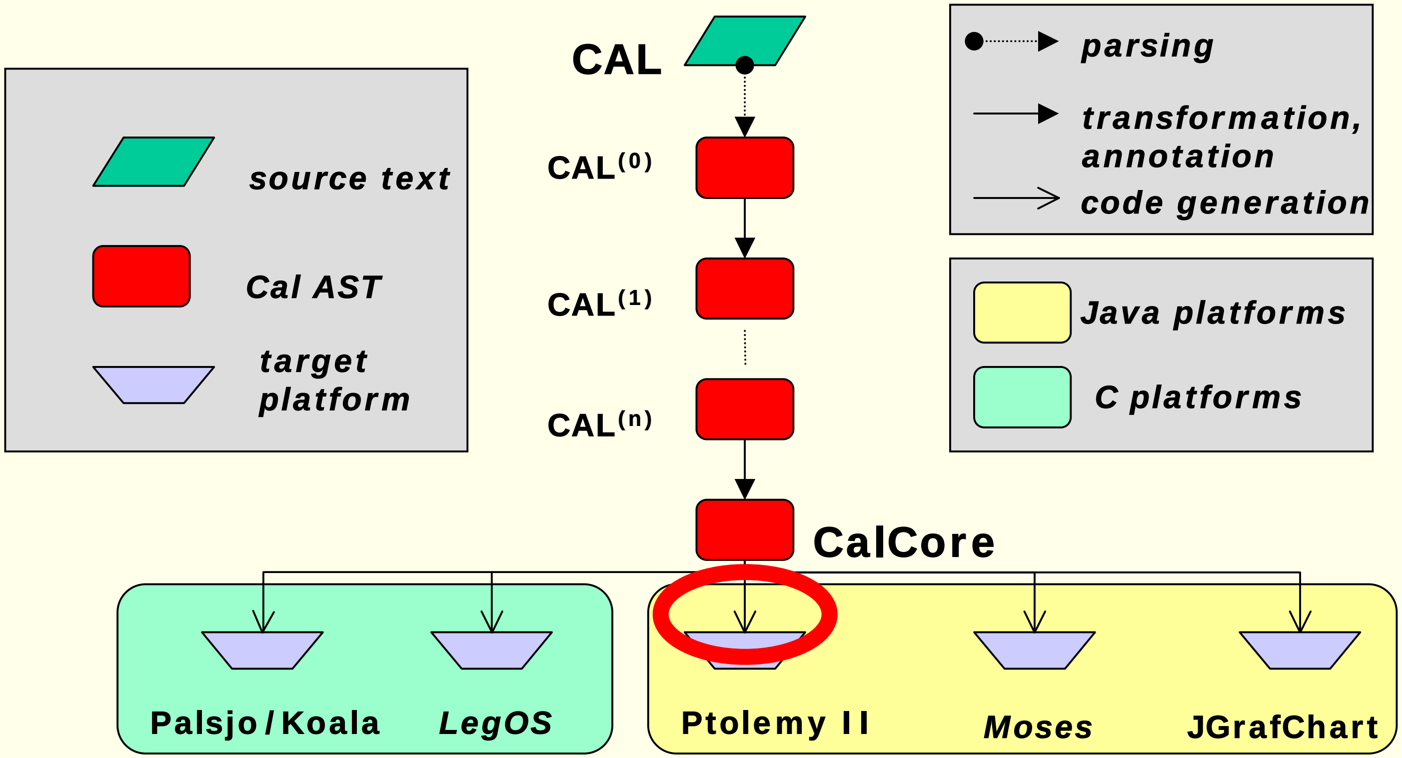
Advantages of using CAL

- It reduces amount of code to be written.
- It insulate the actor behavior from the specificities of the APIs.
  - makes writing actors more accessible.
  - improves the portability and reusability of actors.
- It reduces error probability.
- Actors gets more readable.

Advanced benefits of CAL

- **to describe actors at an abstract level**
- **Actors get more analyzable**
  - token type, token rates can be easily extracted from a CAL actor, and can be used for type checking, data flow analysis, behavioral analysis
- **A new notion of actors**
  - ports, states, parameters
  - **Actions**: atomic; unsequencial;

Target CAL actor to platforms



source text

Cal AST

target platform

CAL

CAL$^{(0)}$

CAL$^{(1)}$

CAL$^{(n)}$

CalCore

parsing

transformation, annotation

code generation

Java platforms

C platforms

Palsjo / Koala

LegOS

Ptolemy II

Moses

JGrafChart

The CAL group

Joern Janneck

- language design, actor composition, …

Johan Eker, Lund University

- language design, C code generation

Chris Chang, UCB

- type system, program transformations

Yang Zhao, UCB

- action level scheduling, actor composition

Lars Wernli, ETH Zurich

- Java code generation, generic code generation infrastructure

www.gigascale.org/caltrop