CAL - An actor language

Jörn W. Janneck
The Ptolemy Group
University of California, Berkeley

5th Biennial Ptolemy Miniconference
Berkeley, CA, May 9, 2003

CAL people

- Chris Chang
- Johan Eker (now Ericsson Mobile Platforms, Research)
- Ernesto Wandeler (ETH Zurich)
- Lars Wernli (then ETH Zurich)
- Ed Willink (Thales Research)
- Yang Zhao
Why another language?

- Writing simple actors should be simple.
  - Ptolemy II API very rich
  - actor writing requires considerable skill
  - BUT: Actors have a lot of common structure.
- Models should allow efficient code generation.
  - actor descriptions contain a lot of "admin" code
- local precedent:
  - ptlang in Ptolemy Classic (J. Buck)

Why another language?

We should generate actors from a more abstract description.

- reduces amount of code to be written
- makes writing actors more accessible
- reduces error probability
- makes code more versatile
  - retargeting (other platforms, new versions of the Ptolemy API)
  - analysis & composition
**Simple actors**

actor ID (X) In ==> Out:

  action [a] ==> [a] end
end

actor A (k) Input1, Input2 ==> Output:

  action [a], [b] ==> [k*(a + b)] end
end

actor Merge ()

  Input1, Input2 ==> Output:

    action Input1: [x] ==> [x] end
    action Input2: [x] ==> [x] end
end

actor firing ≡ execution of one enabled action

**An actor with state**

actor Sum () Input ==> Output:

  sum := 0;
  action [a] ==> [sum]
    do
      sum := sum + a;
    end
end
Action guards

**actor** FairMerge ()

Input1, Input2 ==> Output:

s := 0;

**action** Input1: [x] ==> [x]

**guard** s = 0
do
s := 1;
end

**action** Input2: [x] ==> [x]

**guard** s = 1
do
s := 0;
end
end

Action schedules

**actor** FairMerge ()

Input1, Input2 ==> Output:

A: **action** Input1: [x] ==> [x] end
B: **action** Input2: [x] ==> [x] end

**schedule** fsm State0:
State0 (A) --> State1;
State1 (B) --> State0;
end
end

**actor** FairMerge ()

Input1, Input2 ==> Output:

A: **action** Input1: [x] ==> [x] end
B: **action** Input2: [x] ==> [x] end

**schedule** regexp
(A B)*
end
end
First-class functions

actor Sieve (predicate) Input ==> Output:

filter := lambda (a) : false end;

action [a] ==> []
guard filter(a) end

action [a] ==> [a]
guard not filter(a)
var f = filter
do
  filter := lambda(b) :
    f(b) or predicate(b,a)
  end;
end
end

Programming language features

• optionally typed
  - generic polymorphic type system
• full functional sub-language
• everything first-class citizen (well, almost)
  - functions
  - procedures
  - NOT actors or actions (yet)
• lexically scoped
• no aliasing of stateful structures
  - useful for handling concurrency
Executing CAL: Interpreter

- **Ptolemy actor**
  - configured by CAL script
  - smooth embedding into Ptolemy II
  - first version in current release
- **domain-dependent interpretation**
  (Chris Chang)
  - interpreter adapts to domain
  - making actors more domain-polymorphic
  - What’s a model of computation?

Executing CAL: Translators

XML for representing actors (CALML)
- persistent format
- infrastructure for checking, transformation
- XSLT as implementation language
  - analysis
  - program transformation
  - code generation

Ptolemy Miniconference 11

Ptolemy Miniconference 12
Executing CAL: Composer/Translator

a Ptolemy II model

model of computation
CALML composition

Generating CALML code

Executing CAL: Discovering concurrency

actor B (a, b => x, y):
  s := <something>;
  action a: [v] => x: [f(v, s)] end
  action b: [v] => y: [g(v)]
    do
      s := h(v, s);
    end
end
Conclusion

- **$\text{CAL}$** is a Ptolemy scripting language
  - simple, portable description of actors
  - can be analyzed, interpreted, compiled, composed

- new research directions
  - composers as models of computation
    - composer languages?
  - infrastructure for executing actors
    - component models, execution environments
  - transformations/analyses of actor networks
    - distribution
    - efficient translation

Thank you.

resources:  www.gigascale.org/caltrop
contact:  janneck@eecs.berkeley.edu