Outline

- Last time: Cal language for declaratively specifying actors

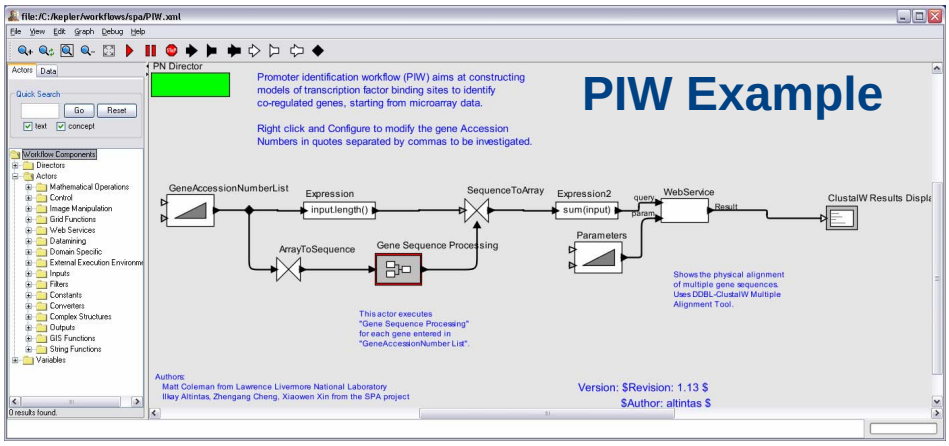
```
actor A (k) Input1, Input2 ==> Output:

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

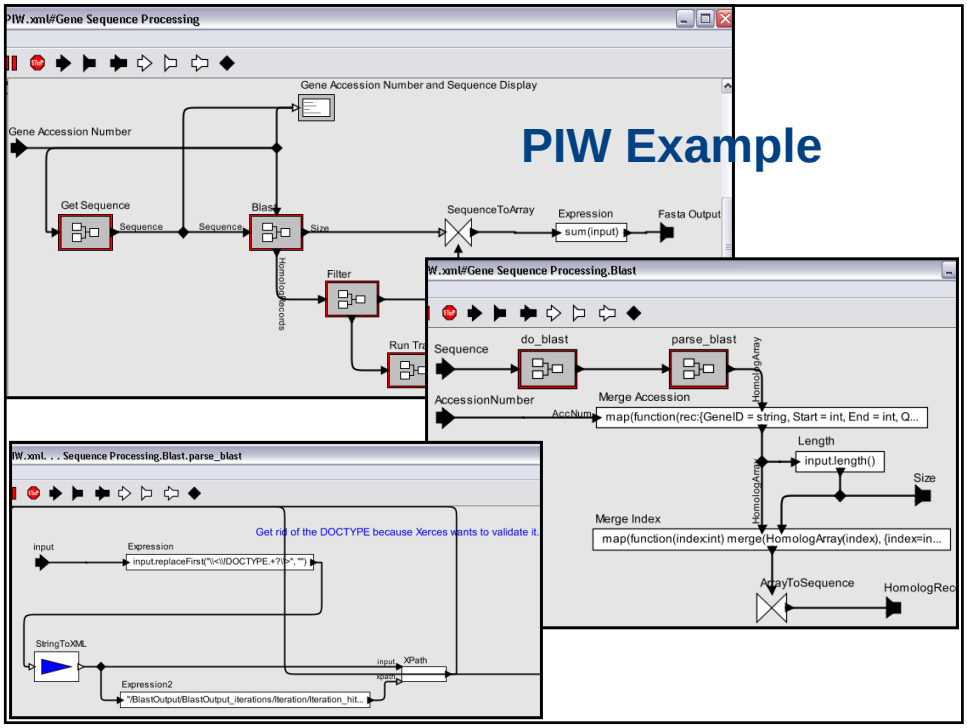
- Today:
  - Ptolemy expression language

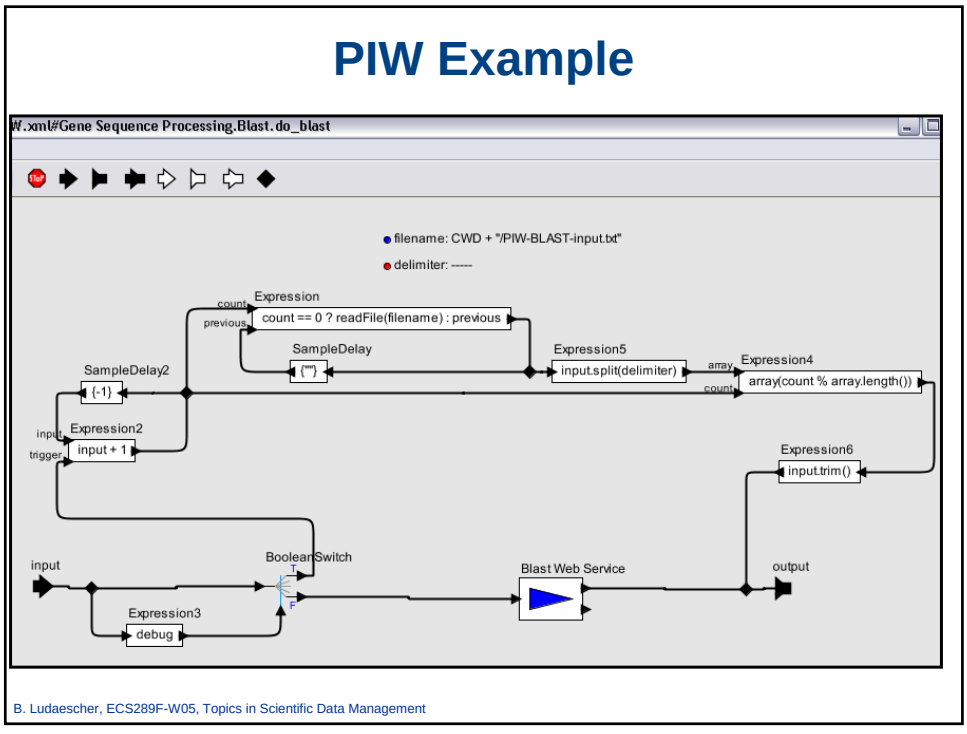
```
map(
    function( rec:{GeneID = string, Start = int, End = int, QueryStart = int, QueryEnd = int} )
       merge(rec, {AccessionNumber=AccNum}),
    HomologArray)
```


Ptolemy II Expression Language

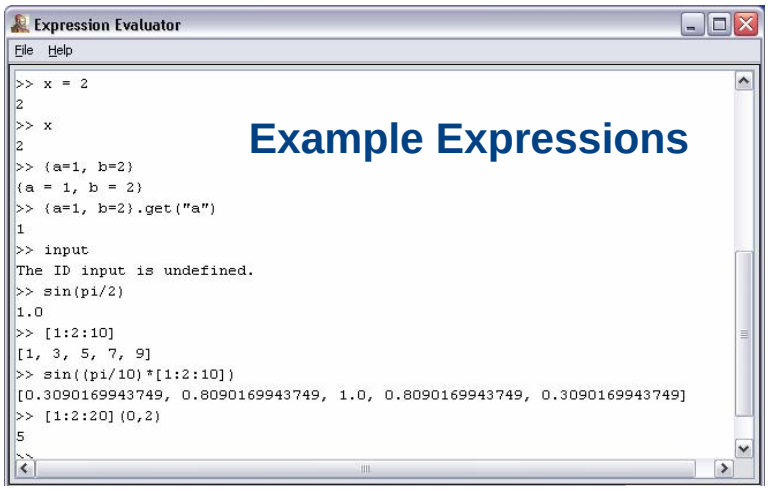
- … can be very useful for "data plumbing"




PIW Example
Gene Sequence Processing
Gene Accession Number and Sequence Display
Gene Accession Number
Get Sequence
Sequence
Blast
Size
SequenceToArray
Expression
sum(input)
Fasta Output
Filter
Gene Sequence Processing.Blast
Sequence
do_blast
parse_blast
HomologArray
AccessionNumber
Merge Accession
map(function(rec:{GeneID = string, Start = int, End = int, Q...
Length
input.length()
Size
Merge Index
map(function(index:int) merge(HomologArray(index), {index=in...
ArrayToSequence
HomologRec
Sequence Processing.Blast.parse_blast
Get rid of the DOCTYPE because Xerces wants to validate it.
input
Expression
input.replaceFirst("\\<\\!DOCTYPE.+?\\>", "")
StringToXML
XPath
Expression2
"/BlastOutput/BlastOutput_iterations/Iteration/Iteration_hit...


PIW Example
Gene Sequence Processing.Blast.do_blast
filename: CWD + "/PIW-BLAST-input.txt"
delimiter: -----
Expression
count
previous
count == 0 ? readFile(filename) : previous
SampleDelay
{""}
Expression5
input.split(delimiter)
Expression4
array
count
array(count % array.length())
SampleDelay2
{-1}
Expression2
input
trigger
input + 1
Expression6
input.trim()
input
BooleanSwitch
T
F
Blast Web Service
output
Expression3
debug

Example Expressions



- constants (pi, e, true, i, …), literals (2.0, 2, 2+3i, “a string”, ...), variables (x, …),
- 1==1.0 (value comparison→ true)
- 1.equals(1.0) (also type comparison → false)
- BooleanExpr ? TrueExpr, FalseExpr
- …

Use of Expressions

- in actor **parameters** (beware of string parameters though)
- in **port parameters**
  - a parameter that is also a port
  - provides a default value, which can be overridden with the value provided at the port

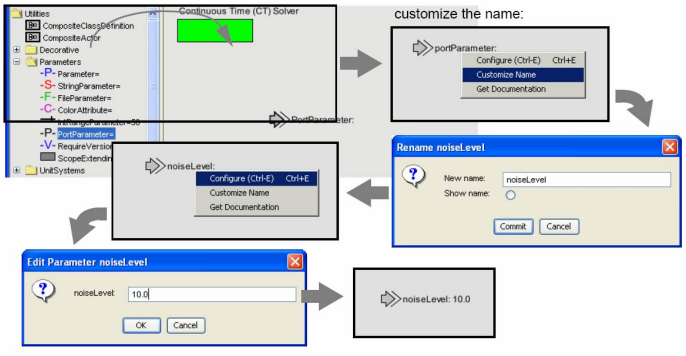


FIGURE 3.2. A *portParameter* is both a port and a parameter. To use it in a composite actor, drag it into the actor, change its name to something meaningful, and set its default value.

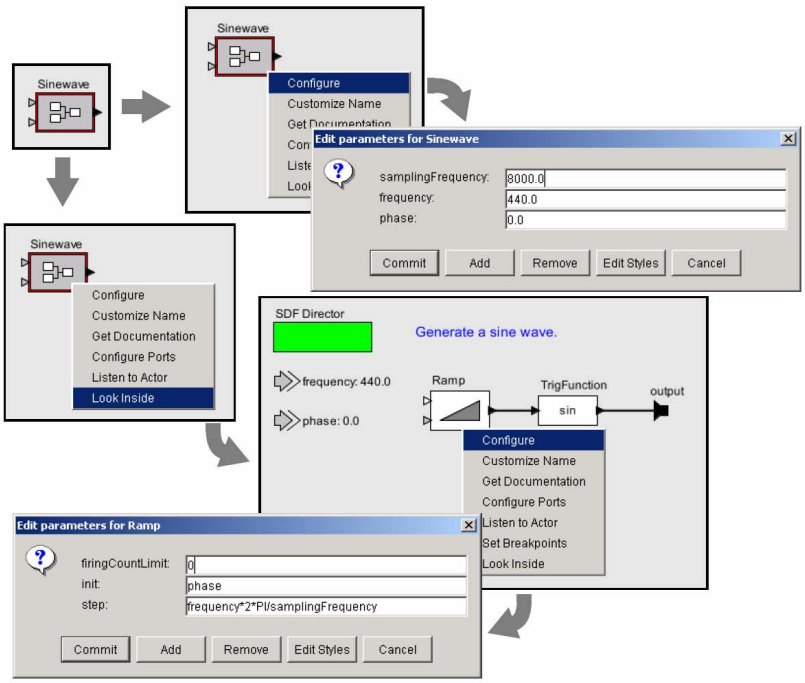


FIGURE 3.3. Sinewave actor, showing its port parameters, and their use at the lower level of the hierarchy.

Expression Actor

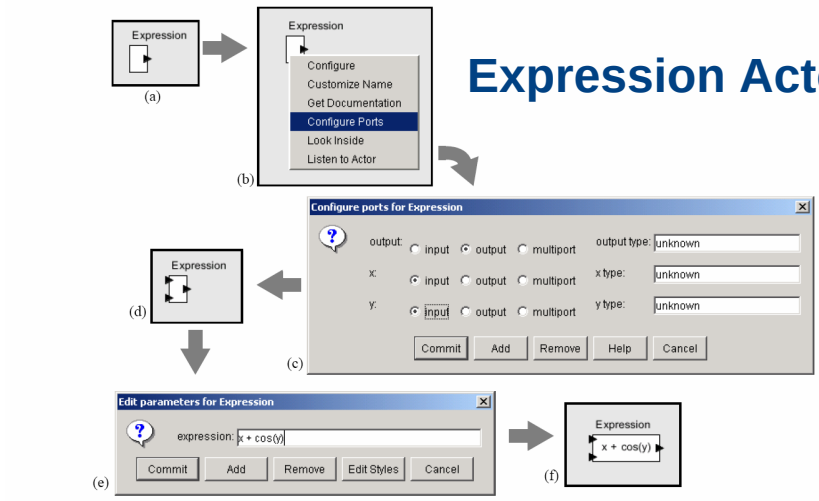


FIGURE 3.5. Illustration of the *Expression* actor.

- The Expression actor by default has one output and no inputs (a). The first step in using it is to add ports, as shown in (b) and (c), resulting in a new icon as shown in (d).
- In (c) when you click on Add, you will be prompted for a Name (pick one) and a Class. Leave the Class entry blank and click OK. You then specify an expression using the port names, as shown in (e), resulting in the icon shown in (f).

Composite Data Types

- Arrays
  - … are ordered sets of tokens, e.g.:
  - {1, 2.3}
  - {{1, 2}, {3, 4, 5}}
  - element access:
    - >> {1.0, 2.3}(1)
    - 2.3
- Matrices
  - … are multi-dimensional arrays, but of for some (mostly numeric) types only
- Array and matrix operations available
  - E.g. matrix multiplication, multiplication w/ a scalar, etc.

Composite Data Types: Records …

- … like tuples with named attributes
- e.g. {a=1, b="foo"} (note: type is Integer x String)
- parts are accessed as expected:
  - {a=1, b="foo"}.a() (or just ".a") yields … 1
- operators can be applied as well:

```
>> {foodCost=40, hotelCost=100}
   +{foodCost=20, taxiCost=20}
{foodCost=60}
```

- works like an intersection

```
>> intersect({a=1, c=2}, {a=3, b=4})
{a=1}
```

(this is really "intersect on attributes and pick first")

- record merge:

```
>> merge({a=1, b=2}, {a=3, c=3})
{a=1, b=2, c=3}
```

Defining Functions

The expression language supports definition of functions. The syntax is:

```
function(arg1:Type, arg2:Type...)
    function body
```

where "function" is the keyword for defining a function. The type of an argument can be left unspecified, in which case the expression language will attempt to infer it. The function body gives an expression that defines the return value of the function. The return type is always inferred based on the argument type and the expression. For example:

```
function(x:double) x*5.0
```

defines a function that takes a *double* argument, multiplies it by 5.0, and returns a double. The return value of the above expression is the function itself. Thus, for example, the expression evaluator yields:

```
>> function(x:double) x*5.0
(function(x:double) (x*5.0))
>>
```

To apply the function to an argument, simply do

```
>> (function(x:double) x*5.0) (10.0)
50.0
```

Higher Order Functions; Passing Functions

>> iterate(function(x:int) x+3, 5, 0)

{0, 3, 6, 9, 12}

>> map(function(x:int) x+3, {0, 2, 3})

{3, 5, 6}

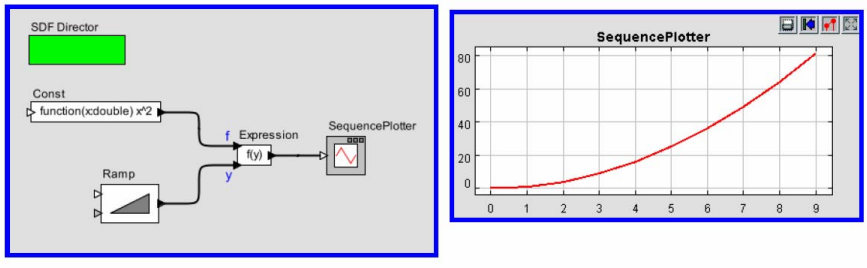


FIGURE 3.6. Example of a function being passed from one actor to another.

More on those: Haskell Prelude

iterate

type:
```
iterate :: (a -> a) -> a -> [a]
```

description: iterate f x returns the infinite list [x,f(x),f(f(x)),...].

definition:
```
iterate f x = x : iterate f (f x)
```

usage:
```
Prelude> iterate (+1) 1
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, .....
```

map

type:
```
map :: (a -> b) -> [a] -> [b]
```

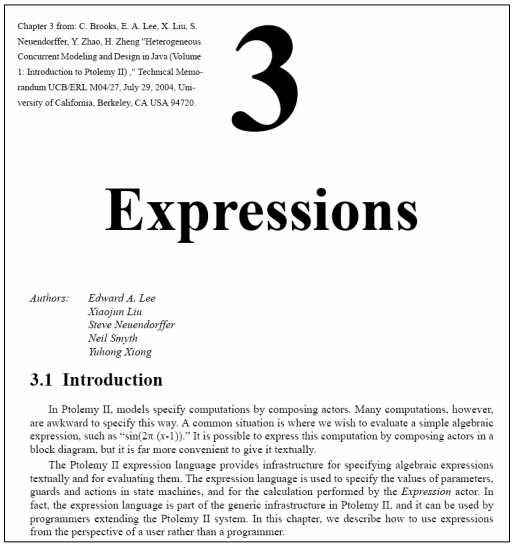
description: given a function, and a list of any type, returns a list where each element is the result of applying the function to the corresponding element in the input list.

definition:
```
map f xs = [f x | x <- xs]
```

usage:
```
Prelude> map sqrt [1..5]
[1.0, 1.41421, 1.73205, 2.0, 2.23607]
```


References & Further Reading

- Chapter 3 from: C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng "Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II) ," Technical Memorandum UCB/ERL M04/27, July 29, 2004, University of California, Berkeley, CA USA 94720.



Chapter 3 from: C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng "Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II) ," Technical Memorandum UCB/ERL M04/27, July 29, 2004, University of California, Berkeley, CA USA 94720.

3

Expressions

Authors: *Edward A. Lee*
Xiaojun Liu
Steve Neuendorffer
Neil Smyth
Yuhong Xiong

3.1 Introduction

In Ptolemy II, models specify computations by composing actors. Many computations, however, are awkward to specify this way. A common situation is where we wish to evaluate a simple algebraic expression, such as "sin(2π (x-1))." It is possible to express this computation by composing actors in a block diagram, but it is far more convenient to give it textually.

The Ptolemy II expression language provides infrastructure for specifying algebraic expressions textually and for evaluating them. The expression language is used to specify the values of parameters, guards and actions in state machines, and for the calculation performed by the *Expression* actor. In fact, the expression language is part of the generic infrastructure in Ptolemy II, and it can be used by programmers extending the Ptolemy II system. In this chapter, we describe how to use expressions from the perspective of a user rather than a programmer.