Applications of Rewriting Logic in Biology
III
Transformation to Petri Nets and Interactive Visualization

Carolyn Talcott
SRI International
July 2007

Questions???

PLAN

- Petri Net representation
- The IOP-IMaude Interaction Framework
- Intro to the Pathway Logic Assistant (PLA)

Why Petri Nets?

- Simple, easy to visualize representation as graphs
  - Directly represents concurrency and dependencies
- Efficient analysis, especially for 1-safe nets (at most on mark on any place -- conservation of matter)
- Graph-based algorithms for analyzing structure, finding `modules'
- Also represented in Maude

Main Insight

- Cells and Dishes can be represented as sets of occurrences (aka places, things tagged with location)
  - Egf on the outside < Egf, out >
  - Egfr in the membrane < EgfR, CLm >
  - Activated Egfr in the membrane < [EgfR - act], CLm >
- Rules are then transitions
  - < Egf, out > < EgfR, CLm > => < [Egf - bound], CLo > < [EgfR-act], CLm >

Realizing the Insight

- Objective: a Petri net representation Pn(R,D) of a model with rules R and dish (initial state) D, that *gives* the right answers to queries.
- Problem
  - A Petri net has a finite set of places and transitions
  - Maude rules have variables with unbounded range.
- Solution
  - Consider only rule instances possible using values declared in components.maude
  - Restrict occurrences to those appearing in some rule instance

Petri Net Representation: Overview

- Specification of Petri Nets
  - occurrences and transitions
  - functions for manipulating Petri Nets
- Converting Maude models to Petri Nets
  - Rules --> Transition Schemes
  - Components + Transition Schemes --> Transition list (knowledge base)
- Computing with Petri net models

Petri Nets:
Occurrences and Transitions

(See pl-aux.maude modules DISH-OPS, PETRI)

Occurrences 1

Sorts and constructors:

```
sort Loc . subsort LocName < Loc .
op Out : -> Loc .
sorts Occ Occs *** multiset of Occ, id: none
op <_,_> : Thing Loc -> Occ [ctor] .
```

Converting dishes (and soups) to occurrence sets.

```
pl2occs(th-1...th-k [ ct | { loc | lt-1 .. lt-m } ...]) =>
  < th-1,Out > ... < th-k,Out >
  < lt-1,loc > ... < lt-m,loc > ...
```

Example:

```
rasDish := PD(Egf [HMEC | {CLo | empty }
    {CLm | EgfR  PIP2 }
    {CLi | [Hras - GDP]  Src }
    {CLc | Gab1 Grb2 Pi3k Plcg Sos1 } ]) .

rasOccs = pl2occs(rasDish) =
 < Egf,Out > < EgfR,CLm > < PIP2,CLm >
 < Src,CLi > < [Hras - GDP],CLi >
 < Gab1,CLc > < Grb2,CLc > < Pi3k,CLc > < Plcg,CLc > < Sos1,CLc >
```

OCCURRENCES II

Set operations on occurrences

`member(occ, occs)`

returns `occ` if `occ` is present in occs and **`none`** ow

`Odiff(occs0,occs1)`

returns the elements of `occs0` not in `occs1`

`Osame(occs0,occs1)`

returns the intersection of `occs0` and `occs1`

PNets I

Sorts and Constructors

```
sort PNTrans .
op pnTrans : Qid Occs Occs Occs -> PNTrans [ctor] .
****  pnTrans(rid,inOccs,outOccs,bothOccs)
```

```
rl[1.EgfR.on]: ?ErbB1L:ErbB1L
  [CellType:CellType | ct
  {CLo | clo                    }
  {CLm | clm EgfR               } ]
  =>
  [CellType:CellType | ct
  {CLo | clo [?ErbB1L:ErbB1L - bound] }
  {CLm | clm [EgfR - act]       } ] .
```

```
rl[5.Grb2.reloc]:
  {CLm | clm [EgfR - act]       }
  {CLi | cli                    }
  {CLc | clc Grb2               }
  =>
  {CLm | clm [EgfR - act]       }
  {CLi | cli [Grb2 - reloc]     }
  {CLc | clc                    } .
```

```
Rule1 Transition:
pnTrans('1.EgfR.act,
        < Egf, Out > < EgfR,CLm >,
        <[Egf - bound],CLo >
           <[EgfR - ct],CLm >,
       none)
```

```
Rule5 Transition:
pnTrans('5.Grb2.reloc,
        < Grb2,CLc >,
        <[Grb2 - reloc],CLi >,
        <[EgfR - act],CLm >)
```

PNets II

More Sorts and Constructors

```
sort PNTransList .
subsort  PNTrans < PNTransList .
op nil : -> PNTransList [ctor] .
op  __ : PNTransList PNTransList -> PNTransList
          [ctor assoc id: nil] .

sort PNet .
op pnet : PNTransList Occs -> PNet [ctor] .

rasNet = pnet(rasPntl,rasOccs)
```

PNets III

Operations on transition lists

`len(pntl)` is the length of `pntl`

`getPre(n,pntl)` is the prefix of `pntl` of length `n`

`getPost(n,pntl)` is the suffix of `pntl` after the first `n`

`pntl = getPre(n,pntl) getPost(n,pntl)`

`unionTrans(pntl0,pntl1)`

concatenates `pntl1` to `pntl0` removing duplicates

`intersectTrans(pntl0,pntl1)` -- the transitions in both lists

Petri Nets:
Functions for Transforming

(See pl-aux.maude modules RELEVANT

PNETS III

Auxiliary sort for tupling results

```
sort  PNTL3 .   **** Transitionlist plus 3 Occ sets
op `{_,_,_,_`} : PNTransList Occs Occs Occs -> PNTL3 [ctor] .
```

Selecting tuple components

```
op pntls-0 : PNTL3 -> PNTransList .
ops pntls-1 pntls-2 pntls-3 : PNTL3 -> Occs .
```

Forward Collection

`fwdCollect(pntl,initOccs) = {pntl',ioccs',unrch,rch}`

where

`pntl'` is the sublist of transitions in `pntl` reachable from `initOccs`

`pnTrans(id,ioccs,ooccs,boccs)` reachable if `Odiff(ioccs boccs,rch)`

`initOccs` are contained in `rch`

`ooccs` are contained in `rch` if `pnTrans(id,ioccs,ooccs,boccs)` reachable

```
ioccs' = Osame(initOccs,rch)
unrch' = Odiff(initOccs,rch)
```

PNets IV

Backward Collection:

`bwdCollect(pntl,goals) = pntl'`

where

`pntl'` is the sublist of transitions in `pntl` that might contribute to `goals`

`pnTrans(id,ioccs,ooccs,boccs)` might contribute
    if `Osame(ooccs,goccs)=/= none`

`goals` are contained in `goccs`
`ioccs boccs` are contained in goccs
    if `pnTrans(id,ioccs,ooccs,boccs)` might contribute

Pruning a net:

`omitRules(pntl,rids)` removes transitions from `pntl` with identifier in `rids`
`avoidOccs(pntl, aviods)` removes `pnTrans(id,ioccs,ooccs,boccs)`
    if `Osame(ioccs ooccs boccs, avoids) =/= none`

Converting Maude Models to Petri Nets

Transformation Idea

- Make the transition knowledge base TKB(R) for rules R (this is a meta-level operation)
  - convert each rule to occurrence form
  - make a transition for each substitution for the component variables
- For Rules R and dish D, P(R,D) is the transition list computed by forward collection from TKB(R) together with the occurrence form of D.

Transformation Example

- convert each rule to occurrence form

```
rl[1.EgfR.on]: ?ErbB1L:ErbB1L
  [CellType:CellType | ct {CLo | clo} {CLm | clm EgfR}]
  =>
  [CellType:CellType | ct
    {CLo | clo [?ErbB1L:ErbB1L - bound]}
    {CLm | clm [EgfR - act]}] .
```

becomes

```
rl[PN1.EgfR.on]:
  < ?ErbB1L:ErbB1L, out > < EgfR, CLm >
  =>
  < [?ErbB1L:ErbB1L - bound], CLo > < [EgfR-act], CLm > .
```

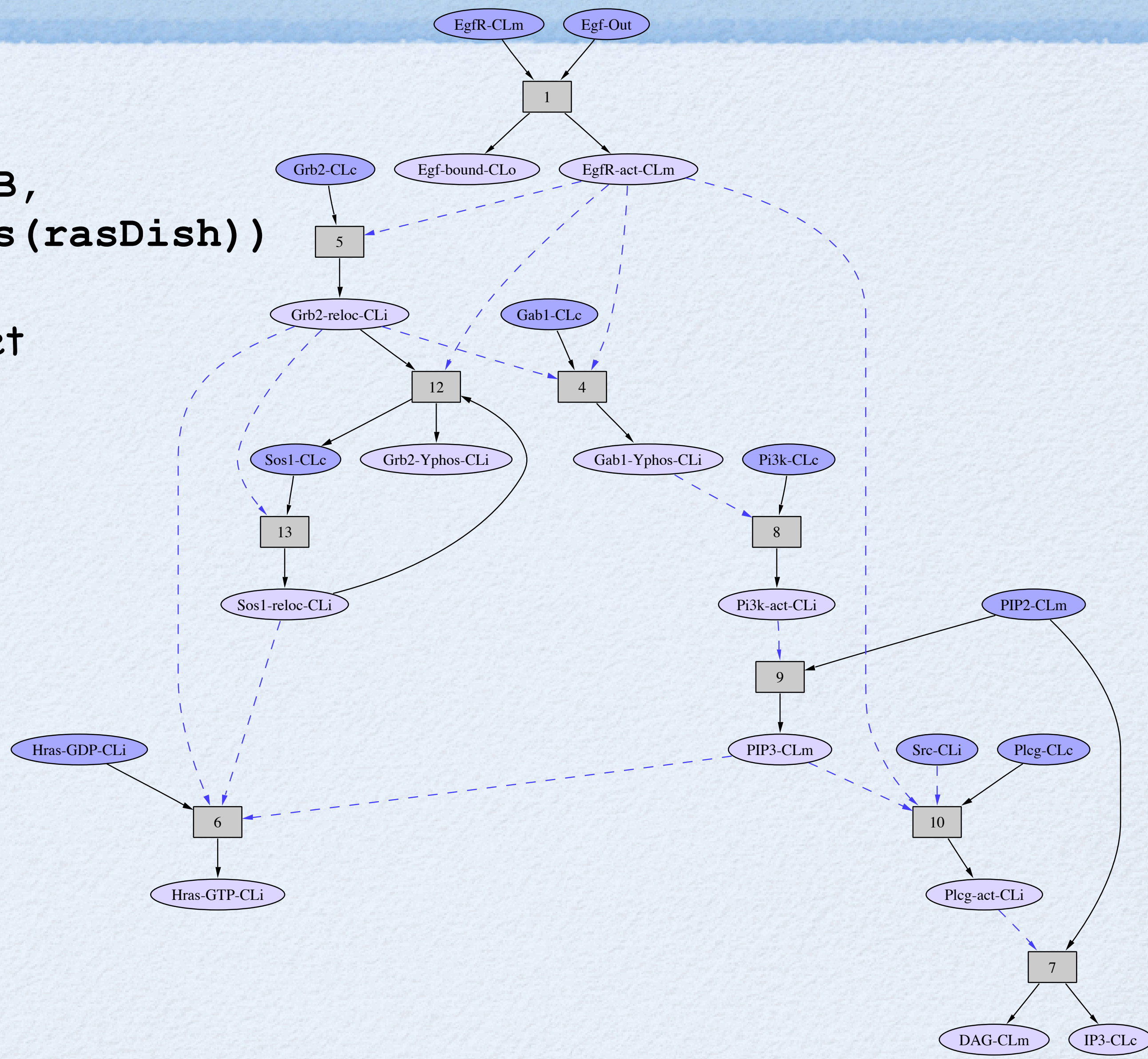
- there are two substitutions binding `?ErbB1L:ErbB1L` to `Egf` or `Tgfa` giving two PNTransitions

```
pnTrans('1.EgfR.act,< Egf, Out > < EgfR,CLm >,
         <[Egf - bound],CLo > <[EgfR - act],CLm >,none)
pnTrans('1.EgfR.act#1,< EgfR,CLm > < Tgfa,Out >,
        <[EgfR - act],CLm > <[Tgfa - bound],CLo >,none)
```

RasNet Model as Petri Net

```
fwdCollect(smallKB,
           pl2occs(rasDish))
```

- R -the SmallKB rule set
- smallKB is TKB(R)



Executing PNet Transitions in Maude

A pnet state carries along its transition list

```
op ps : PNTransList Occs -> State [ctor] .
op initPs : PNet -> State .
eq initPs(pnet(pntl:PNTransList,i:Occs)) = ps(pntl:PNTransList,i:Occs) .
crl[psStep]:
  ps(pntl:PNTransList, i:Occs b:Occs occs:Occs) =>
  ps(pntl:PNTransList, o:Occs b:Occs occs:Occs)
if pntl:PNTransList :=
     pntl0:PNTransList pnTrans(rid:Qid,i:Occs,o:Occs,b:Occs)
     pntl1:PNTransList  .
```

It may also carries along a history of rules fired

```
op psp : PNTransList Occs QidList -> State [ctor] .
op initPsp : PNet -> State .
eq initPsp(pnet(pntl:PNTransList,i:Occs)) =
      psp(pntl:PNTransList,i:Occs,nil) .
crl[psStep]:
  psp(pntl:PNTransList, i:Occs b:Occs occs:Occs, rids:QidList) =>
  ps-(pntl:PNTransList, o:Occs b:Occs occs:Occs, rids:QidList rid:Qid)
if pntl:PNTransList :=
     pntl0:PNTransList pnTrans(rid:Qid,i:Occs,o:Occs,b:Occs)
     pntl1:PNTransList  .
```

Computations

For a set of rules R, a sequence

$R \mid\!- D_0 \text{ -}rl_1\text{-> } \ldots \text{ -}rl_k\text{-> } D_k$

is computation from dish D_0 to dish D_k via rules rl_1 ... rl_k if D_{i-1} rewrites to D_i by an application of rule rl_i

For a PNTransList P, a sequence

$P \mid\!- O_0 \text{ -}pnt_0\text{-> } \ldots\text{-}pnt_k\text{-> } O_k$

is computation from occurrences O_0 to O_k via transitions pnt_1 ... pnt_k if $ps(P,O_{i-1})$ rewrites to $ps(P,O_i)$ by a step using transition pnt_i

Petri Net Correctness

Theorem: If P = TKB(R,C), D_0 is a dish over C, and O_0 is the corresponding occurrence set then there is a 1-1 correspondence between computations from D_0 and those from O_0

- R |- D_0 -rl_1-> ... -rl_k-> D_k <-> P |- O_0 -pnt_0-> ...-pnt_k-> O_k

where O_0 = pl2occs(D_0), and pnt_i is an instance of the occurrences form of rl_i

A Simple Query Language

- Given a Pnet state ps(P,O) there are two types of query
  - subnet
  - findPath
- For each type there are three parameters (requirements)
  - G: a goal set---occurrences required to be present at the end of a path
  - A: an avoid set---occurrences that must not appear in any transition fired
  - H: as list of identifiers of transitions that must not be fired
- findPath returns a pathway (transition list) generating a computation satisfying the requiremments.
- subnet returns a subnet containing all (minimal) such pathways.

PNet Query Functions

Computing a subnet

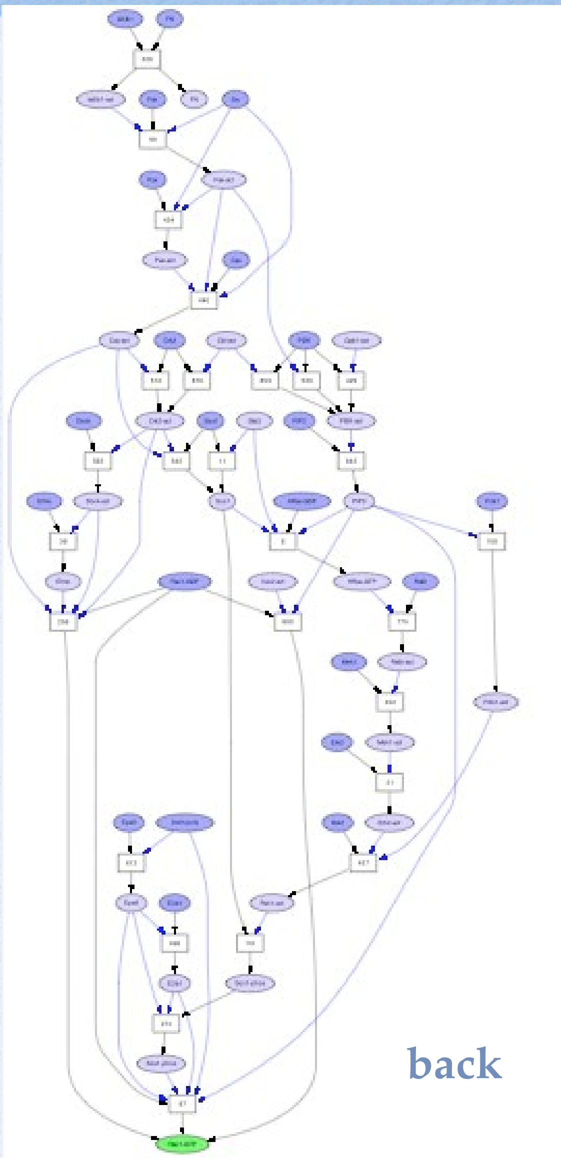
```
****                          ioccs goals avoids hides
op relSubnet : PNTransList Occs Occs Occs QidList -> PNTL3 .

ceq relSubnet(pntl,ioccs,goals,avoids,rids) =
            {fpntl,ioccs',unused,used}
if  pntl' := avoidOccs(omitRules(pntl,rids),avoids)
/\  bpntl := (if goals == none
              then pntl'
              else bwdCollect(pntl',goals) fi)
/\ {fpntl,ioccs',unused,used} :=
             fwdCollect(bpntl,Odiff(ioccs, avoids)) .
```

Finding a path -- invoke a model-checker asserting goals are unreachable
from initial occurrences using avoidOccs(omitRules(pntl,rids),avoids)

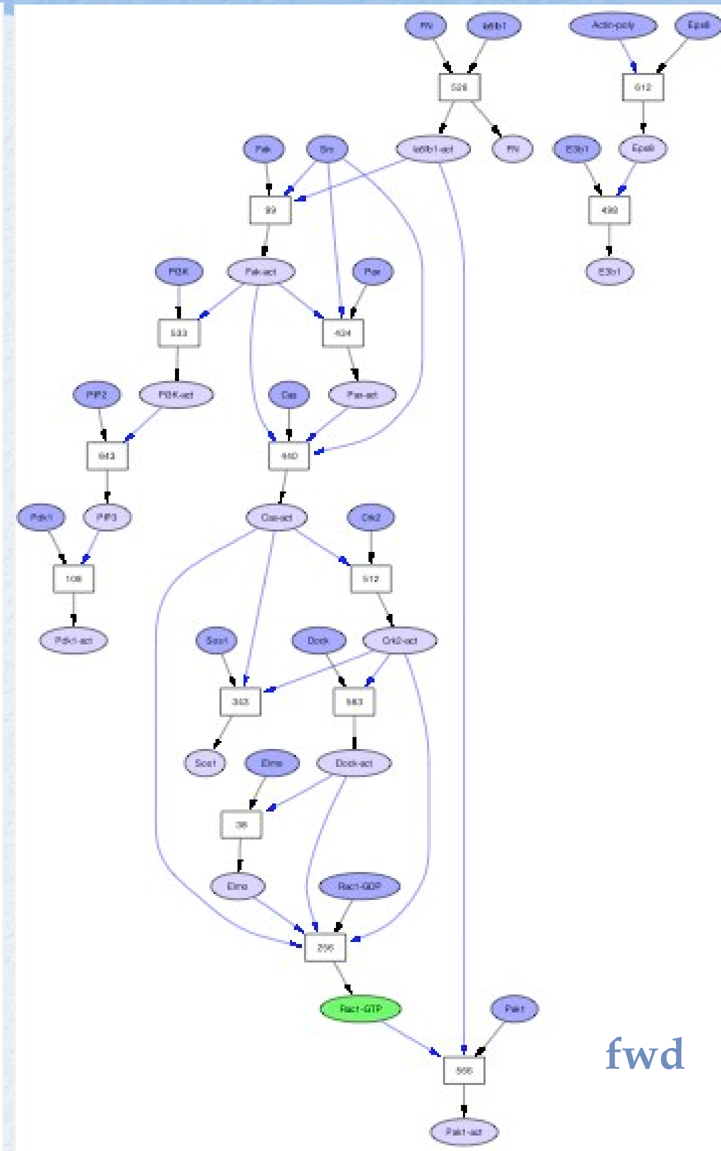
Subnet Adequacy

- Given a Pnet state ps(P,O) , goals G, avoids A and hides H,
  - findPath(ps(P,O),G,A,H) succeeds iff
  - findPath(relSubnet(ps(P,O),G,A,H),G,none,nil) does
- subnetting
  - reduces the search space for finding a path
  - simplifies the network to be understood by a biologist

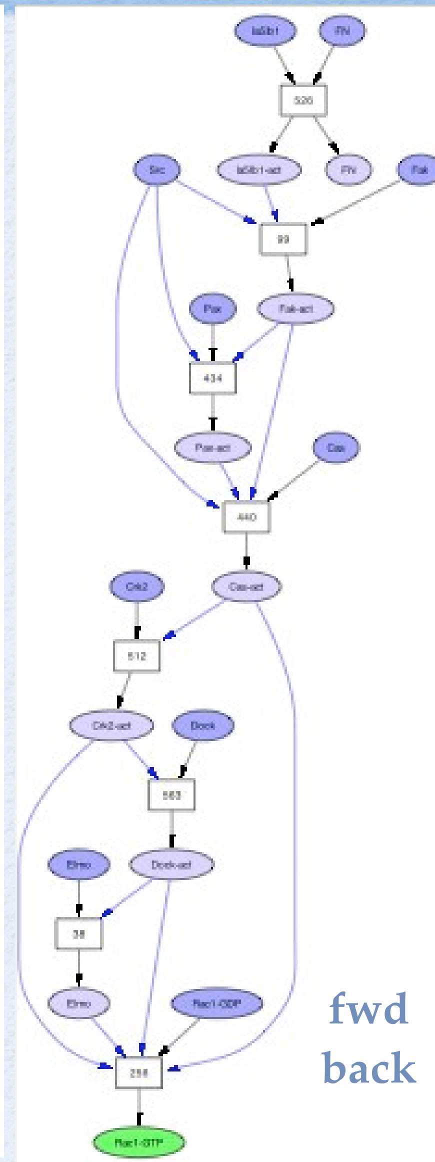
Example collection results



back



fwd



fwd
back

PLA

- Provides a means to interact with a PL model
- Manages multiple representations
  - Maude module (logical representation)
  - PetriNet (process representation for efficient query)
  - Graph (for interactive visualization)
- Exports Representations to other tools
  - Lola (and SAL model checkers)
  - Dot -- graph layout
  - JLambda -- interactive visualization
  - SBML

PLA : Overview

- IOP
- IMaude -- actors in Maude
- JLambda
- PLA = IMaudePLA $+_{IOP}$ JLambdPLA

Interoperability Platform IOP

IOP Aims/Motivations

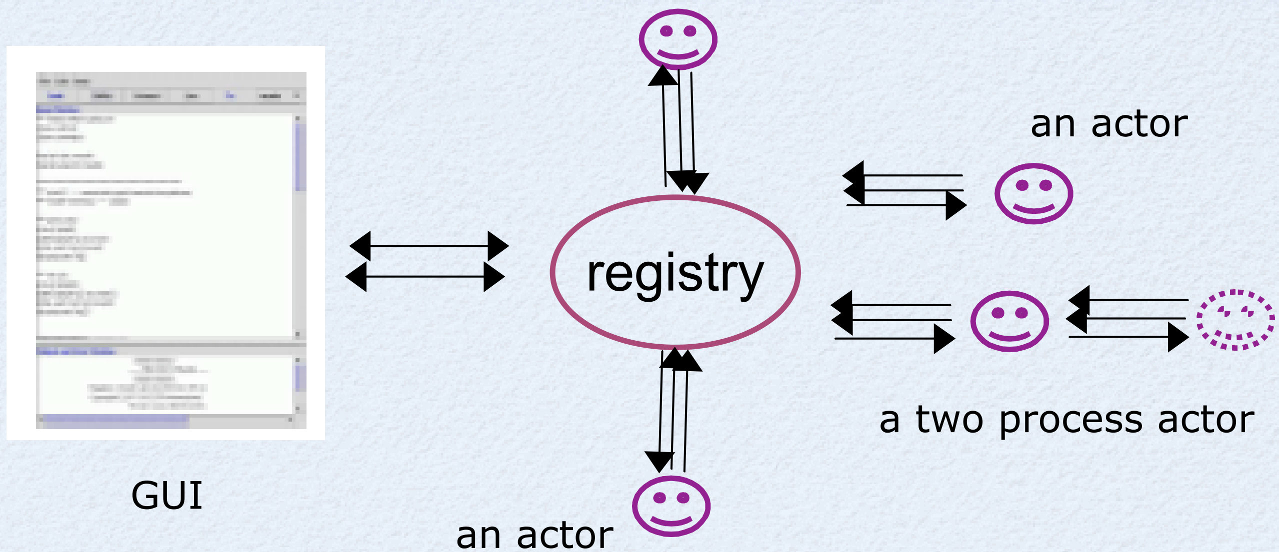
- Long term
  - infrastructure for simple message passing tool interoperation
- Short term---giving Maude interactive capabilities
  - communication with other tools, including itself
  - accessing web resources
  - manipulating files
  - using visualization tools
  - accessing the underlying OS
- Two sides to Maude interoperation:
  - The world must be prepared to talk to Maude (IOP)
  - Maude must be prepared to talk to the world (IMaude)

IOP DESIGN

Based on the **actor model** of distributed computation.

- IOP consists of a pool of actors, that interact via asynchronous message passing.
- Actors can create other actors
- An actor consists of one or more (UNIX style) processes
- An actors behavior may described in any programming language, possibly using a wrapper to patch it into the mail system.

IOP Architecture



Architecturally IOP consists of

- A dynamic pool of actors
- A <u>main</u> that configures the system
- A registry that keeps track of known actors and maintains the lines of communication
- A GUI front end (the user as an actor)

IMAUDE

IMaude I

- IMaude extends Maude to allow:
  - interactions with the environment to be interleaved with rewriting
  - internal state to persist across interactions
- IMaude begins with the LOOP-MODE module of core Maude.
  - LOOP-MODE provides a basic read-eval-print loop.
- A LOOP-MODE system has the form [inQ,S,outQ]
  - inQ is a list of quoted identifiers read from standard input, and parsed by the Maude tokenizer.
  - outQ is a list of quoted identifiers channeled to standard output.
  - S is the system state, rewritten using application specific rules.

IMaude II

- A PLAIMaude state has the form

  st(control,wait4s,requestQ,eset,log)

- The control component indicates what the current IMaude actor task
- The wait4s component contains handlers for incoming messages (listeners, continuations, ...)
- The requestQ component is a queue of pending tasks
- The eset component is a local environment containing a set of entries of the form

  e(etype,args,notes,evalue)

- The log component is a place to record success or failure information -- for debugging

The Display Petri Request

To build the pnet for a predefined dish and display it

```
(seq
  (predefDish SmallKB graphics2d rasDish dish0 rasDish)
  (dish2pnet SmallKB dish0 pnet1)
  (pnet2graph SmallKB pnet1 graph2)
  (defineGraph  graphics2d graph2)
  (startListener graph2 graphreq graphics2d)
  (showGraph  graphics2d graph2)
)
```

DISH2PNET

```
**** can the request be execute now?
eq isReq('dish2pnet) = true .
  eq enabled(wait4s,
            req('dish2pnet,ql(kbname dname pname toks),reqQ))
       = true .

**** update the entry set with the pnet for the dish `dname'
**** `pname' is the name of the new pnet
  rl[dish2pnet]:
   [nil,
    st(processing(req('dish2pnet, ql(kbname dname pname toks),
                      reqQ')),  **** what to do with the pnet
       wait4s,reqQ,es,log),
    outQ]
     =>
   [nil,
    st(ready, wait4s, (reqQ reqQ'),
       dish2pnet(es,kbname,dname,pname), log),
    outQ] .
```

Dish2pnet entry update function

```
  op dish2pnet : ESet Qid Qid Qid -> ESet .

  ceq dish2pnet(es,kbname,dname,pname) =
**** store the new entry
       addEntry(es,'tval,'pnet pname, pnotes,
                    tm(modname,'pnet[pntlT',ioccsT]))
**** get the dish from the entry set
   if tm(modname,occsT) :=
        getVal(es,'tval,'dish dname,tm('BOOL,'true.Bool))
**** get the knowlegebase transition list from the entry set
   /\ tm(modname',pntlT) :=
        getVal(es,'tval, 'tkb kbname,tm('BOOL,'true.Bool))
**** do the forward collection
   /\ '`{_`,_`,_`,_`}[pntlT',ioccsT,uoccsT,roccsT] :=
           getTerm(metaReduce([modname],'fwdCollect[pntlT,occsT]))
   ....
   /\ pnotes :=  (("source" := ql('dishnet dname)),
                     ("rchOccs" := tm(modname, roccsT)),
                    ("unusedOccs" := tm(modname, uoccsT)),
                    ("dishname" := ql(dname user-dname)),
                    ("kbname" := ql(kbname))) .
```

JLAMBDA

- JLambda is a scheme like interpreted language designed to make programming interactive graphics less painful
  - let, if, closures/apply ... define
- Construct and maniplate objects in any known Java class
- Special purpose classes:
  - Identifiable -- associating objects to strings for external access (actor names)
  - Attributable -- add new fields/methods dynamically
  - Glyph -- interactive graphics -- render and react
  - Graph -- interactive nodes, layout
  - Closure<X> for abstract class X -- listeners, actions ...

Interactive Graphs

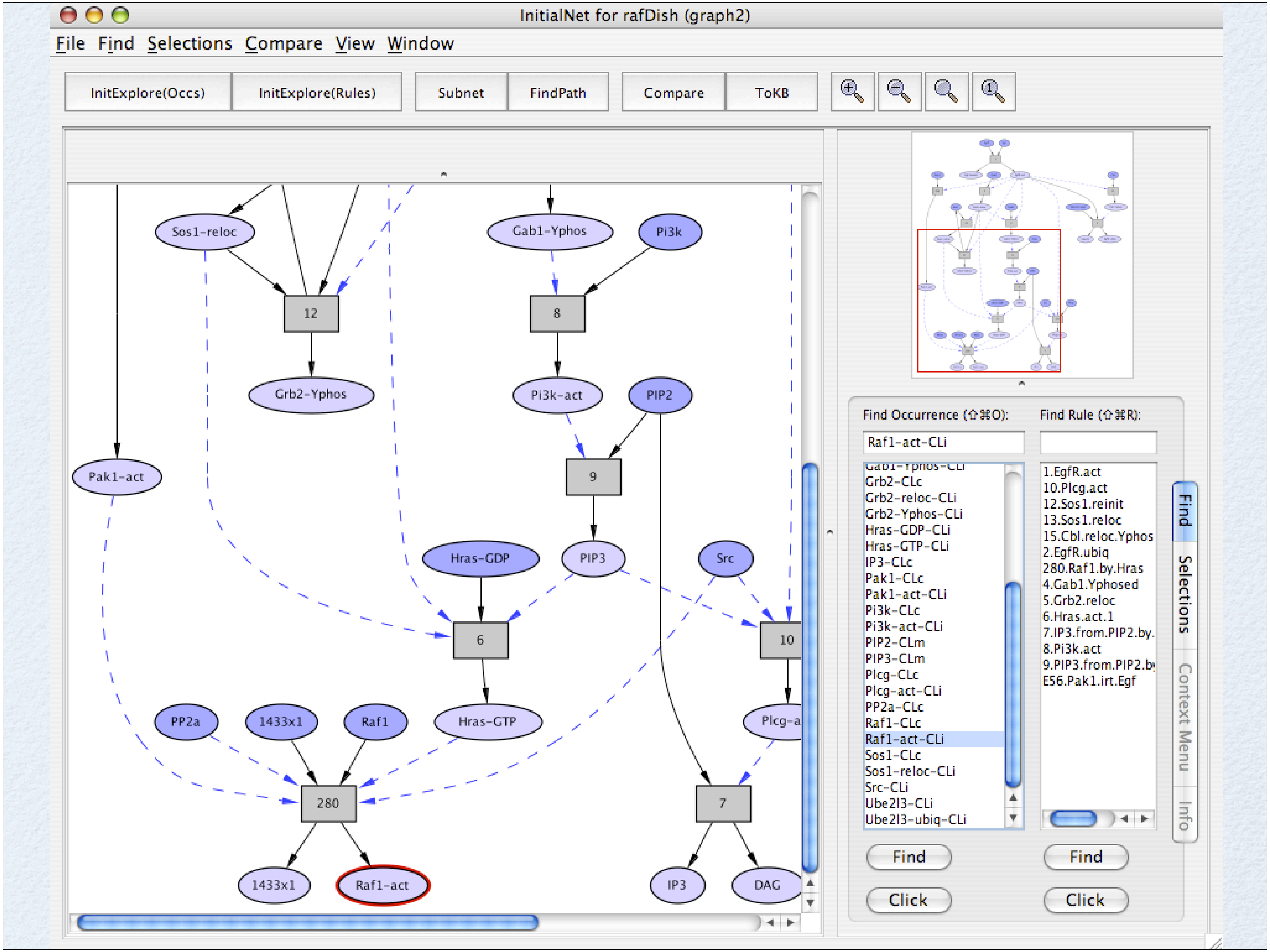
```
(define makeGraph  (graph)
  (let ((node1 (object ("g2d.graph.IOPNode" "node1")))
        (node2 (object ("g2d.graph.IOPNode" "node2")))
        (edge1 (object ("g2d.graph.IOPEdge" node1 node2)))
        )
    (seq
     (invoke node1
         "setMouseAction"
         java.awt.event.MouseEvent.MOUSE_CLICKED
         (lambda (self e)
           (invoke java.lang.System.err "println" e))   )
     (invoke graph "addNode" node1)
     (invoke graph "addNode" node2)
     (invoke graph "addEdge" edge1))))
```

Closures as Actions

```
(define mkAction (label tip closure)
    (object ("g2d.closure.ClosureAbstractAction"
               label
               (object null) ; icon
               tip
               (object null) ; accelerator
               (object null) ; mnemonic
               closure )))   ; action closure

;; adding a button to the toolbar
(invoke toolbar "prepend"
   (object ("pla.toolbar.ToolButton"
       (apply mkAction "FindPath" "find a path to goals"
           (lambda (self event)(apply pathRequest graph))))))

;; sending a request to maude from a graph
;; (received by the graph listener)
(define pathRequest (graph)
   (sinvoke "g2d.util.ActorMsg" "send"
      "maude"
       (invoke graph "getUID")
       (concat "displayPath1" " " (apply mkStatusString graph))))

;; mkStatusString gathers goals, avoids, hides information
```

The PLA Viewer

- Navigation -- find nodes, rules, ends of arrows
- Dish editing and petri net generation/visualization
- Queries -- path/subnet
- Comparing any two graphs/nets
- Exploring -- incremental generation of a subnet
- In context view


InitialNet for rafDish (graph2)
File
Find
Selections
Compare
View
Window
InitExplore(Occs)
InitExplore(Rules)
Subnet
FindPath
Compare
ToKB
Sos1-reloc
Gab1-Yphos
Pi3k
12
8
Grb2-Yphos
Pi3k-act
PIP2
Pak1-act
9
Hras-GDP
PIP3
Src
6
10
PP2a
1433x1
Raf1
Hras-GTP
Plcg-a
280
7
1433x1
Raf1-act
IP3
DAG
Find Occurrence (⇧⌘O):
Find Rule (⇧⌘R):
Raf1-act-CLi
Grb2-CLc
Grb2-reloc-CLi
Grb2-Yphos-CLi
Hras-GDP-CLi
Hras-GTP-CLi
IP3-CLc
Pak1-CLc
Pak1-act-CLi
Pi3k-CLc
Pi3k-act-CLi
PIP2-CLm
PIP3-CLm
Plcg-CLc
Plcg-act-CLi
PP2a-CLc
Raf1-CLc
Raf1-act-CLi
Sos1-CLc
Sos1-reloc-CLi
Src-CLi
Ube2I3-CLi
Ube2I3-ubiq-CLi
1.EgfR.act
10.Plcg.act
12.Sos1.reinit
13.Sos1.reloc
15.Cbl.reloc.Yphos
2.EgfR.ubiq
280.Raf1.by.Hras
4.Gab1.Yphosed
5.Grb2.reloc
6.Hras.act.1
7.IP3.from.PIP2.by.
8.Pi3k.act
9.PIP3.from.PIP2.by
E56.Pak1.irt.Egf
Find
Find
Click
Click
Find
Selections
Context Menu
Info

NEXT SESSION PLA LIVE