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 one mark on any place -- conservation of matter)
- Graph-based algorithms for analyzing structure, finding 'modules'
- Also represented in Maude
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 >
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 $\rightarrow$ Transition Schemes
  - Components + Transition Schemes $\rightarrow$ Transition list
    (knowledge base)

- Computing with Petri net models
PETRI NETS: OCCURRENCES AND TRANSITIONS
(See pl-aux.maude modules DISH-OPS, PETRI)
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 >
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
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] } ] .

Rule 1 Transition:

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

Rule 5 Transition:

pnTrans('1.EgfR.act,
< Egf, Out > < EgfR,CLm >,
<[Egf - bound],CLo >
  <[EgfR - ct],CLm >,
none)
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)
Operations on transition lists

- \( \text{len} (\text{pntl}) \) is the length of \( \text{pntl} \)

- \( \text{getPre} (n, \text{pntl}) \) is the prefix of \( \text{pntl} \) of length \( n \)

- \( \text{getPost} (n, \text{pntl}) \) is the suffix of \( \text{pntl} \) after the first \( n \)
  
  \[ \text{pntl} = \text{getPre} (n, \text{pntl}) \cup \text{getPost} (n, \text{pntl}) \]

- \( \text{unionTrans} (\text{pntl0}, \text{pntl1}) \)
  
  concatenates \( \text{pntl1} \) to \( \text{pntl0} \) removing duplicates

- \( \text{intersectTrans} (\text{pntl0}, \text{pntl1}) \) -- the transitions in both lists
PETRI NETS: FUNCTIONS FOR TRANSFORMING

(See pl-aux.maude modules RELEVANT)
Auxiliary sort for tupling results

sort PNTL3 .  **** Transitionlist plus 3 Occ sets

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)
Backward Collection:
\[ \text{bwdCollect}(\text{pntl}, \text{goals}) = \text{pntl}' \]
where
\[ \text{pntl}' \] is the sublist of transitions in \( \text{pntl} \) that might contribute to \( \text{goals} \)

\[ \text{pnTrans}(\text{id}, \text{ioccs}, \text{ooccs}, \text{boccs}) \text{ might contribute if } \text{Osame}(\text{ooccs}, \text{goccs}) \neq \text{none} \]

\( \text{goals} \) are contained in \( \text{goccs} \)
\( \text{ioccs} \) and \( \text{boccs} \) are contained in \( \text{goccs} \)
if \( \text{pnTrans}(\text{id}, \text{ioccs}, \text{ooccs}, \text{boccs}) \text{ might contribute} \)

Pruning a net:
\[ \text{omitRules}(\text{pntl}, \text{rids}) \text{ removes transitions from } \text{pntl} \text{ with identifier in } \text{rids} \]
\[ \text{avoidOccs}(\text{pntl}, \text{avoids}) \text{ removes } \text{pnTrans}(\text{id}, \text{ioccs}, \text{ooccs}, \text{boccs}) \]
if \( \text{Osame}(\text{ioccs ooccs boccs}, \text{avoids}) \neq \text{none} \)
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

  \[rl[1.EgfR.on]: \text{?ErbB1L:ErbB1L} \]
  \[[\text{CellType:CellType} | \text{ct } \{\text{CLo | clo} \} \{\text{CLm | clm EgfR}\}] \]
  \[
  \Rightarrow \quad [\text{CellType:CellType} | \text{ct }
  \quad \{\text{CLo | clo } [\text{?ErbB1L:ErbB1L - bound}]\}
  \quad \{\text{CLm | clm [EgfR - act]}\}].
  \]

  becomes

  \[rl[PN1.EgfR.on]: \]
  \[
  < \text{?ErbB1L:ErbB1L, out } > < \text{Egf, CLm } > \]
  \[
  \Rightarrow \quad < [\text{?ErbB1L:ErbB1L - bound}], \text{CLo } > < [\text{EgfR-act}], \text{CLm } > .
  \]

- there are two substitutions binding \text{?ErbB1L:ErbB1L} to \text{Egf} or \text{Tgfa}
giving two PNTransitions

  \[\text{pnTrans('1.EgfR.act,< Egf, Out } > < \text{EgfR,CLm } > ,\]
  \[< [\text{Egf - bound}],\text{CLo } > < [\text{EgfR - act}],\text{CLm } > ,\text{none})\]
  \[\text{pnTrans('1.EgfR.act#1,< EgfR,CLm } > < \text{Tgfa,Out } > ,\]
  \[< [\text{EgfR - act}],\text{CLm } > < [\text{Tgfa - bound}],\text{CLo } > ,\text{none})\]
fwdCollect(smallKB, pl2occ(s(rasDish)))

- R - the SmallKB rule set
- smallKB is TKB(R)
A pnet state carries along its transition list

\[ \text{op } \text{ps} : \text{PNTransList Occs } \rightarrow \text{State } \text{[ctor]} . \]
\[ \text{op } \text{initPs} : \text{PNet } \rightarrow \text{State} . \]
\[ \text{eq } \text{initPs}(\text{pnet}(\text{pntl:PNTransList},i:\text{Occs})) = \text{ps}(\text{pntl:PNTransList},i:\text{Occs}) . \]
\[ \text{crl[psStep]}:\]
\[ \text{ps}(\text{pntl:PNTransList}, \ i:\text{Occs} \ b:\text{Occs} \ occs:\text{Occs}) \Rightarrow \]
\[ \text{ps}(\text{pntl:PNTransList}, \ o:\text{Occs} \ b:\text{Occs} \ occs:\text{Occs}) \]
\[ \text{if } \text{pntl:PNTransList} := \]
\[ \text{pntl0:PNTransList} \ \text{pnTrans(rid:Qid,i:Occs,o:Occs,b:Occs)} \]
\[ \text{pntl1:PNTransList} . \]

It may also carries along a history of rules fired

\[ \text{op } \text{pssp} : \text{PNTransList Occs QidList } \rightarrow \text{State } \text{[ctor]} . \]
\[ \text{op } \text{initPsp} : \text{PNet } \rightarrow \text{State} . \]
\[ \text{eq } \text{initPsp}(\text{pnet}(\text{pntl:PNTransList},i:\text{Occs})) = \]
\[ \text{pssp}(\text{pntl:PNTransList},i:\text{Occs},\text{nil}) . \]
\[ \text{crl[psspStep]}:\]
\[ \text{pssp}(\text{pntl:PNTransList}, \ i:\text{Occs} \ b:\text{Occs} \ occs:\text{Occs}, \ rids:QidList) \Rightarrow \]
\[ \text{pssp}(\text{pntl:PNTransList}, \ o:\text{Occs} \ b:\text{Occs} \ occs:\text{Occs}, \ rids:QidList \ rid:Qid) \]
\[ \text{if } \text{pntl:PNTransList} := \]
\[ \text{pntl0:PNTransList} \ \text{pnTrans(rid:Qid,i:Occs,o:Occs,b:Occs)} \]
\[ \text{pntl1:PNTransList} . \]
For a set of rules $R$, a sequence

$$R |- D_0 -rl_1-> ... -rl_k-> D_k$$

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

$$P |- O_0 -pnt_0-> ...-pnt_k-> O_k$$

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 $pnt_i$
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 $O_0$

$$R \vdash D_0 - rl_1 \rightarrow ... - rl_k \rightarrow D_k \leftrightarrow P \vdash O_0 - pnt_0 \rightarrow ... - pnt_k \rightarrow O_k$$

where $O_0 = pl2occ(D_0)$, 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 requirements.
- subnet returns a subnet containing all (minimal) such pathways.
**PNet Query Functions**

Computing a subnet

****

\[ \text{op relSubnet} : \text{PNTransList Occs Occs Occs QidList} \rightarrow \text{PNTL3} \]

\[
\text{ceq relSubnet}(\text{pntl}, \text{ioccs}, \text{goals}, \text{avoids}, \text{rids}) = \{\text{fpntl}, \text{ioccs'}, \text{unused}, \text{used}\}
\]

\[
\text{if } \text{pntl'} := \text{avoidOocs}(\text{omitRules}(\text{pntl}, \text{rids}), \text{avoids})
\]

\[
\text{bpntl} := (\text{if goals }== \text{none}
\]

\[
\quad \text{then } \text{pntl'}
\]

\[
\quad \text{else } \text{bwdCollect}(\text{pntl'}, \text{goals}) \text{ fi}
\]

\[
\text{\text{fpntl}, \text{ioccs'}, \text{unused}, \text{used}} := \text{fwdCollect}(\text{bpntl}, \text{Odiff(ioccs, avoids)})
\]

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

Given a Pnet state $ps(P,O)$, goals $G$, avoids $A$ and hides $H$,

- $\text{findPath}(ps(P,O),G,A,H)$ succeeds iff
- $\text{findPath}(\text{relSubnet}(ps(P,O),G,A,H),G,\text{none},\text{nil})$ does subnetting

- reduces the search space for finding a path
- simplifies the network to be understood by a biologist
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
- $\text{PLA} = \text{IMaudePLA} \oplus_{\text{IOP}} \text{JLambdPLA}$
INTEROPERABILITY PLATFORM

IOP
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.
Architecturally IOP consists of
- A dynamic pool of actors
- A `main` 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 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 $[\text{inQ}, S, \text{outQ}]$
- 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.
A PLAIMaude state has the form

\[ st(\text{control}, \text{wait4s}, \text{requestQ}, \text{eset}, \text{log}) \]

- 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
  \[ e(\text{etype}, \text{args}, \text{notes}, \text{evalue}) \]
- The log component is a place to record success or failure information
  -- for debugging
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)
)

**** can the request be execute now?
\[\text{eq isReq('dish2pnet) = true .} \]
\[\text{eq enabled(wait4s,} \]
\[\text{req('dish2pnet,ql(kbname dname pname toks),reqQ))} \]
\[= \text{true .} \]

**** update the entry set with the pnet for the dish `dname'
**** `pname' is the name of the new pnet
\[\text{rl[dish2pnet]:} \]
\[\text{[nil,} \]
\[\text{st(processing(req('dish2pnet, ql(kbname dname pname toks),} \]
\[\text{reqQ')), } \]
\[\text{**** what to do with the pnet} \]
\[\text{wait4s,reqQ,es,log),} \]
\[\text{outQ]} \]
\[= \text{]} \]
\[\text{[nil,} \]
\[\text{st(ready, wait4s, (reqQ reqQ'),} \]
\[\text{dish2pnet(es,kbname,dname,pname), log),} \]
\[\text{outQ] .} \]
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 knowledgebase transition list from the entry set
  \( \text{tm(modname',pntlT) :=}
    \text{getVal(es,'tval, 'tkb kbname,tm('BOOL,'true.Bool))}
**** do the forward collection
  \( \text{'}\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\text{'}[pntlT',ioccsT,uoccsT,roccsT] :=}
    \text{getTerm(metaReduce([modname],'fwdCollect[pntlT,occsT])})
.....
  \( \text{pnotes := ("source" := ql('dishnet dname)),}
       \text{("rchOcss" := tm(modname, roccsT)),}
       \text{("unusedOcss" := tm(modname, uoccsT)),}
       \text{("dishname" := ql(dname user-dname)),}
       \text{("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 manipulate 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 ...
(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))
          )
(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
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