Symbolic Systems Biology and Pathway Logic

Patrick Lincoln and Carolyn Talcott

SRI International
333 Ravenswood Avenue, Menlo Park, CA 94025, USA
[lincoln,clt]@csl.sri.com

Abstract. Pathway Logic (PL) is an approach to modeling and analysis of biological processes based on rewriting logic. This chapter describes the use of PL to model signal transduction processes. It begins with a general discussion of Symbolic Systems Biology, followed by some background on rewriting logic and executable specifications. Finally, representation and analysis of a small model of Ras and Raf activation is presented in some detail.

Key words: Symbolic systems biology, rewriting logic, signal transduction, Pathway Logic

1 Symbolic Modeling of Cellular Processes

Biological processes are complex. They exhibit dynamics with a huge range of time scales: microseconds to years. The spatial scales cover 12 orders of magnitude: metabolite to single protein to cell to organ to whole organism. Just considering the cellular level, cells interact with their environment, both sensing and affecting. They have many behaviors: they can grow, proliferate, migrate, differentiate, or die. Underlying these behaviors are a variety of processes such as gene regulation, signal transduction, and metabolism that interact with one another in complex ways. Genes are regulated by proteins (and other molecular entities) binding to promoter regions. This determines which genes are expressed (turned on) and thus which new proteins are produced. These proteins may in turn regulate the same or other genes. A cell senses its environment by receptors in the membrane that recognize specific types of molecules or conditions. This results in *signal transduction* that transmits the information to appropriate components inside the cell. Mechanisms underlying the flow of information include modification of protein state, formation of complexes, and change of location. The flow is controlled by mechanisms that activate or inactivate proteins in the signaling path. Metabolism involves both synthesis and degradation of chemicals to generate energy, synthesize protein building blocks (amino acids), and cell structure components amongst other things. Metabolic processes are controlled by enzymes which may in turn be activated or inhibited by signaling processes. Furthermore, metabolites such as glucose play a role in controlling signal flow.

Oceans of experimental biological data are being generated, from both traditional and emerging high throughput techniques. How can we use this data to develop better models? Important intuitions are captured in mental models that biologists build of

biological processes and the cartoons they draw. The trouble is that these models are not amenable to computational analysis.

High level statistical models can be developed, for example, to discover possible correlations and causal relations. Such models may suggest useful insights, but have many limitations such as features that can not be modeled. Low level, detailed kinetic or stochastic models can be developed for small subsystems, but often require parameter fitting, so that the reaction rates used reflect unknown biological context.

Symbolic systems biology is the *qualitative and quantitative* study of biological processes as integrated systems rather than as isolated parts. Our focus is on modeling causal networks of biomolecular interactions in a logical framework at multiple scales. The aim is to develop formal models that are as close as possible to domain experts (biologists) mental models. Furthermore, it is important to be able to compute with and analyze these complex networks. The latter includes techniques for abstracting and refining the logical models; using simulation and deduction to compute or check postulated properties; and make testable predictions about possible outcomes, using experimental results to update the models.

There are many challenges in developing symbolic systems models. One challenge is choosing the right abstractions. Biological networks (metabolic, protein, or regulatory, for example) are large and diverse. It is important to balance computational complexity against model fidelity and to be able to move between models of different levels of detail, using different formalisms in meaningful ways. Biological networks combine to produce high levels of physiological organization, for example, circadian clock subnetworks are integrated with metabolic, survival, and growth subnetworks. A second challenge is to be able to compose different views or models of different components into integrated system models.

Symbolic/logical models allow one to represent partial information and to model and analyze systems at multiple levels of detail, depending on information available and questions to be studied. Such models are based on formalisms that provide language for representing system states and mechanisms of change such as reactions. These languages come with a well-defined semantics, and tools for analysis are based on this underlying semantics. Formal theories can include both specific facts and general principles relating and categorizing data elements and processes. New data structures for representing biological entities and their relations and properties can easily be defined. Theories concerning different types of information can also be combined using well-understood operations for combining logical theories. A wide range of analytical tools developed for the analysis of computer system specifications is being adapted to carry out new kinds of analysis of experimental data curated into formal theories.

Of particular interest are symbolic models that are *executable*. An executable model describes system states and provides rules specifying the ways in which the state may change. Such models can be used for simulation of system behavior. In addition, properties of processes can be stated in associated logical languages and checked using tools for formal analysis. Given an executable model, the path graph of a given initial state is a graph whose nodes are the reachable states and whose edges are the rules connecting them. Paths through the graph then correspond to possible ways a system can evolve. An execution strategy picks out a particular path among those possible. For such a model,

there are many kinds of analysis that can be carried out, including: static analysis, forward simulation, forward search, backward search, model checking, constraint solving, and meta analysis.

Static analysis allows one to examine the structure of the model and to understand how the elements are related and organized (the sort structure). It can be used to infer flow of control and dependencies. Static analysis also provides a means to check for inconsistencies or ill-formed declarations and to look for missing information.

Forward simulation runs the model from a given initial state using a specified strategy either for a fixed number of steps, or until no more rewrites apply. This is extremely fast, and very useful for initial exploration.

Forward search is a breadth-first search of all paths through the transition graph for a given initial state. If the graph (number of states) is finite search will find *all* possible outcomes from a given initial state. Assuming finite branching, search will find all outcomes at a finite depth. Search can also be constrained to find only states satisfying a given property.

Backward search runs the model backwards. For models satisfying certain constraints, backwards search can answer the question: "From what initial states can we get to this state?". For example it can be used to find all possible precursors to a particular checkpoint, or to prove that a bad state cannot be reached from an initial state.

Model checking expands the collection of properties that can be investigated. Search concerns only properties of individual states. Model-checking tools are based on algorithms to determine if all computations of a system (pathways / sequences of steps) satisfy a given property. For example we can ask if molecule X is never produced before molecule Y has been produced. If not, a pathway that fails to satisfy the property (molecule Y is produced and molecule X is produced before it) is returned. Turning this around, to find a pathway satisfying a property of particular interest, one asserts that no such pathway exists and a counterexample will be one of the desired pathways. An example of another kind of property that can be model checked is: "If we reach a state that satisfies P then do we always later reach a state satisfying Q?"

Constraint solving attempts to find values for a set of variables that satisfy a given set of constraints. Maximal satisfiability (MaxSat) problems are a generalization of constraint satisfaction problems where there may be conflicting constraints, and hence no assignment of values to variables that will satisfy them all. Weights (importance measures) are assigned to constraints and a MaxSat solver finds a solution maximizing the total weight of the satisfied constraints. Many static analysis problems can be formulate as constraint systems. Steady state analyses such as determining possible flows of information or chemicals through a system can be formulated as constraint problems.

Meta analysis allows us to reason about the models themselves. Essential features of models can be abstracted to form families of related models, allowing us to work with uncertainty about reactions. Starting with a base set of known reactions, different instantiations of sets of reactions can be explored. For example, we can search for models where a given path property is true in a given initial state. In addition, rules themselves can be abstracted into families of rules, each family corresponding, for example, to a particular type of reaction, such as activation, inhibition, or translocation. It also allows the knowledge base to be queried as data base, for example finding all

rules that involve a given protein (in any or a specified state or location). Finally, using mappings of logics a model can be mapped to another formalism to take advantage of additional tools.

A variety of formalisms initially developed to model and analyze concurrent computer systems have been used to develop symbolic models of biological systems, including: Petri nets [37, 46, 21, 41, 17, 25, 31, 14, 28, 53]; the pi-calculus [33, 34, 43] and its stochastic variants [39]; membrane calculi [42, 35, 27]; statecharts [18, 8], life sequence charts [22]; rule-based systems including Rewriting Logic [32, 6, 9, 10, 47, 49]; BioCham [12, 4, 2]; and P-systems [36, 40]; and hybrid systems [19, 29, 51, 15]. For a recent review of 'executable specification approaches' see [13]. A comprehensive review of rule-based modeling can be found in [20]. Overviews of different Petri net formalisms and their application to modeling biological processes can be found in [16, 5]. A series of abstract machines each suited to modeling biological process associated to a different class of macromolecules is presented in [3] giving an nice introduction to the concepts to be modeled.

Symbolic executable models can be mapped to alternative logical formalisms for analysis. As will be discussed later, certain rewriting logic models can be mapped to Petri Nets for analysis by special purpose, efficient model checkers. In [1] a continuous stochastic logic and the probabilistic symbolic model checker, PRISM, is used to express and check a variety of temporal queries for both transient behaviors and steady state behaviors. Proteins modeled as synchronous concurrent processes, and concentrations are modeled by discrete, abstract quantities. Metabolic or signaling networks can be analyzed using a constraint-based technique that generalizes the well-known flux balance analysis [7] by representing the network as constraints on the reactions, rather than on the reacting components. In [52] this technique is used to compute preferred steady states under different conditions, also represented as constraints. Apart from understanding the steady-state configurations, constraint-based analysis can also be used to identify modules in the network, trace the flow of information in the network, and identify cross talk and conflicts.

In the remainder of this chapter we describe the Pathway Logic approach to symbolic systems biology.

2 Pathway Logic Overview

Pathway Logic (PL) [9, 10, 47, 50, 48, 49] is a symbolic systems biology approach to the modeling and analysis of molecular and cellular processes based on rewriting logic [32]. In PL, biological molecules, their states, locations, and their roles in molecular or cellular processes can be modeled at very different levels of abstraction. For example, a complex signaling protein can be modeled either according to an overall state, its post-translational modifications, or as a collection of protein functional domains and their internal or external interactions. Similarly biological processes can be represented at different levels of granularity using rewrite rules. Each rule represents a step (at the chosen level of granularity) in a biological process such as metabolism or intra-/intercellular signaling. A rule may represent a family of reactions using variables to stand for

families of molecular components. Rules express dependencies on biological context; for example, a scaffold needed to hold proteins in position to interact productively.

A collection of rules together with the underlying data type specifications forms a PL knowledge base. Each biological molecule that is declared in a PL rewrite theory has associated metadata linking it to standard database entries, for example HUGO or UniProt/Swiss-Prot for proteins, along with other information such as category and synonyms. This information is part of the knowledge base. It is important to place the knowledge in a broader context and to be able to integrate it with other knowledge sources. Each rule has associated evidence used to justify the rule, which is also part of the knowledge base.

A PL model consists of a specification of an initial state (cell components and locations) interpreted in the context of a knowledge base. Such models are executable and can be understood as specifying possible ways a system can evolve. Logical inference and analysis techniques are used for simulation to study possible ways a system could evolve, to assemble pathways as answers to queries, and to reason about dynamic assembly of complexes, cascading transmission of signals, feedback-loops, cross talk between subsystems, and larger pathways. Logical and computational reflection are used to transform and further analyze models.

Pathways are not predefined. Instead they are assembled by applying the rules starting from an initial state, searching for a state meeting given conditions. For example, a pathway leading to specific conditions, such as activation of a Ras protein can be generated as the result of a logical query. A subnet (subset of reactions) composed of all possible relevant pathways can also be generated. A subnet consisting of connections to a given set of molecular components can be generated by graph exploration techniques.

PL knowledge is represented and analyzed using Maude [6], a rewriting-logic-based formalism. The Pathway Logic Assistant (PLA) [50] provides an interactive visual representation of PL models. In PLA, models are represented as graphs with nodes for rules and components, and edges connecting reactant components to rules and rules to product components (formally these graphs are Petri Nets). These models can be queried and in silico experiments can be performed to study the effects of perturbations on these networks. Using PLA a biologist can:

- ask for a list of dishes available for study, and modify or create dishes;
- display the network of signaling reactions for a specified model;
- formulate and submit queries to find pathways, for example, activating one protein without activating a second protein, or exhibiting a phenotype signature such as apoptosis;
- compare two pathways;
- find knockouts—proteins whose omission prevents reaching a specified state;
- incrementally explore network connections to given rules or components;

PLA, sample models, tutorial material, papers and presentations are available from the Pathway Logic web site, `http://pl.csl.sri.com/`.

3 Introduction to Formal Executable Specification and Maude

As mentioned in Section 2, Pathway Logic models of biological processes are developed using the Maude system, a formal language and tool set based on rewriting logic. Rewriting logic [32] is a logical formalism that is based on two simple ideas: states of a system are represented as elements of an algebraic data type, specified in an equational theory, and the behavior of a system is given by local transitions between states described by *rewrite rules*. An equational theory specifies data types by declaring constants and constructor operations that build complex structured data from simpler parts. Functions on the specified data types are defined by *equations* that allow one to compute the result of applying the function. A term is a variable, a constant, or application of a constructor or function symbol to a list of terms. A specific data element is represented by a term containing no variables. Assuming the equations fully define the function symbols, each data element has a canonical representation as a term containing only constants and constructors. The canonical representation is obtained by using the equations to rewrite the data term. An equation is applied to a term by matching the left hand side of the equation to a subterm and replacing that subterm by the corresponding righthand side of the equation. For example the natural numbers are constructed from the constant 0 by application of the successor function $s(0), s(s(0)) \ldots$ (usually written as $1, 2, \ldots$). The plus function, $+$, can be defined by two equations: $n + 0 = n$ and $n + s(m) = s(n) + m$, where n and m are variables standing for arbitrary numbers. Using these equations we can compute the canonical form (value) of $s(s(0)) + s(s(s(0)))$ $(2 + 3)$.

The second equation matches $s(s(0)) + s(s(s(0)))$ with $n := s(s(0))$ and $m := s(s(0))$ and the result of rewriting with this equation is $s(s(s(0))) + s(s(0))$. Using the second equation twice more we get $s(s(s(s(0)))) + s(0)$ and then $s(s(s(s(s(0))))) + 0$. Now we apply the first equation to obtain $s(s(s(s(s(0)))))$ (i.e. 5).

One data type might be a subtype (subsort / subset) of another. For example the non-zero numbers are a subset of all numbers. Elements of one data type might consist of lists or multisets of elements from another type. For example a system might be represented by a set of pairs such as `{(A,2) (B,5) (C,0)}`.

A rewrite rule has the form $t \Rightarrow t'$ *if* c where t and t' are patterns (terms possibly containing variables) and c is a condition (a boolean term). Such a rule applies to a system in state s if t can be matched to a part of s by supplying the right values for the variables, and if the condition c holds when supplied with those values. In this case the rule can be applied by replacing the part of s matching t by t' using the matching values for the place holders in t'. The process of application of rewrite rules generates computations (also thought of as deductions). In the case of biological processes these computations correspond to pathways. Note that rewriting with rules is similar to rewriting with equations, in that we match the lefthand side and replace the matched subterm by the instantiated righthand side. The difference is in the way the rewriting is used. Equations are used to define functions by providing a means of computation the value of a function application. This means that the equations of an equational theory should give the same result independent of the order in which they are applied. Furthermore, equational rewriting should terminate. In contrast, rules are used to describe

change over time, rather than computing the value of a function. They often describe non-deterministic possibly infinite behavior.

To summarize, a rewriting logic specification naturally has two parts: an equational part that specifies data types and functions on these types, and a rules part, specifying how systems may evolve. To query the specification, one may further specify one or more terms representing systems (initial states) of interest, to be analyzed using formal tools such as execution, search and model checking.

Maude is a language and tool based on rewriting logic `http://maude.cs.uiuc.edu`. Maude provides a high performance rewriting engine featuring matching modulo associativity, commutativity, and identity axioms; and search and model-checking capabilities. Thus, given a specification S of a concurrent system, one can execute S to find one possible behavior; use search to see if a state meeting a given condition can be reached; or model-check S to see if a temporal property is satisfied, and if not to see a computation that is a counter example.

In the following we use a simple example to introduce Maude notation and give some intuition about how to represent and analyze the structure and behavior of concurrent systems using Maude. We call the example *Magic Marbles*. In the world of magic marbles, a marble can be plain or have some magical potential: positive, negative, or (positive or negative) activator. A positive (resp. negative) activator marble can give positive (resp. negative) potential to a plain marble. When it does so, it changes parity and becomes a negative (resp. positive) activator. If a marble with negative potential contacts a marble with positive potential the potential is cancelled and they both become plain. A marbles world consists of a collection (formally a multiset) of marbles interacting according to the laws described above.

We formalize the marbles world in Maude by defining three modules. The modules `MAGIC-MARBLES-DATA` and `MAGIC-MARBLES-STATE` specify data types representing marbles and marbles world states. The module `MAGIC-MARBLES-RULES` specifies the rules governing magic marble behavior. The modules `MAGIC-MARBLES-DATA` and `MAGIC-MARBLES-STATE` are functional modules specifying an equational theory. They form the equational part of the marbles world specification. The module `MAGIC-MARBLES-RULES` is a system module forming the rules part.

A Maude module begins with the keyword `fmod` (a functional module, specifying one or more data types) or `mod` (a system module, with rules specifying system behavior), followed by the module name, and ends with a corresponding keyword `endfm`, or `endm`, respectively.

The module `MAGIC-MARBLES-DATA` begins by declaring a sort `Marble`, the data type consisting of all marbles, and a subsort (think subset or subtype) `PlainMarble`, of plain marbles. This is followed by an `ops` declaration naming several specific plain marbles, for example a red marble `redM`. Next a sort `MagicMarble` of marbles with magical potential is declared. It is also a subsort of `Marble`. Magic potential is represented abstractly by a sort `Potential`. Four different potentials are defined (the second `ops` declaration):

- `+`, `-` represent positive and negative potentials
- `*`, `@` represent the potential of an activator marble to generate a positive or negative potential respectively.

A marble with potential p is constructed by annotating a plain marble m with p, written `[m | p]`.

```
fmod MAGIC-MARBLES-DATA is
  sort Marble .
  sort PlainMarble . subsort PlainMarble < Marble .
  ops redM blueM greenM purpleM whiteM blackM
        : -> PlainMarble [ctor] .

  sort MagicMarble .  subsort MagicMarble < Marble .

  sort Potential .
  ops + - * @ : -> Potential [ctor] .
  op `[_|_`] : PlainMarble Potential -> MagicMarble [ctor] .
endfm
```

The declaration beginning `op `[_|_`]` is an example of *mixfix* syntax, where an operator is collection of symbols including blanks (_) that are place holders for the arguments. In the above case there are two arguments, the first of sort `PlainMarble`, the second of sort `Potential`. (The backquotes are to ensure the brackets are parsed as part of the operator symbol.) As examples, we have

- `[whiteM | *]` a white marble with positive activator potential
- `[whiteM | @]` a white marble with negative activator potential
- `[greenM | +]` a green marble with positive potential

The module `MAGIC-MARBLES-STATE` extends `MAGIC-MARBLES-DATA` (using the inclusion statement beginning `inc`) specifying a sort `Mix` of multisets of marbles. (A multiset or bag is a collection of elements where the number of occurrences of an given element matters, but the order does not.) The constant `none` represents the empty multiset, and multiset union is represented by the binary operator, _ _, that is associative and commutative with identity `none` (ACI).

```
fmod MAGIC-MARBLES-STATE is
  inc MAGIC-MARBLES-DATA .
sort Mix .
subsort Marble < Mix .
op none : -> Mix [ctor] .
op _ _ : Mix Mix -> Mix [assoc comm id: none] .
endfm
```

Thus `redM blueM [whiteM | *]` is a mix of three marbles, two plain and one with positive activating potential.

The syntax, _ _, is an example of empty syntax, a special case of mixfix syntax in which application of the operator is juxtaposition. The attribute `[assoc comm id: none]` at the end of the declaration specifies the ACI property, which axiomatizes the notion of multiset. Associative means that the two ways of joining three elements with two applications of the operator give the same multiset. For example,

```
(redM blueM) whiteM = redM (blueM whiteM) .
```

That is, grouping of arguments doesn't matter, and thus parentheses can be omitted. Commutative means that permuting the order of arguments gives the same result. For example,

```
redM blueM = blueM redM .
```

The "identity `none`" property means that adding `none` to the mix does not change the mix. For example,

```
redM blueM none = blueM redM .
```

The module `MAGIC-MARBLES-RULES` specifies how marbles interact using three rules. A rule begins with the key word `rl` followed by the rule label enclosed in `[]`s. The lefthand side (premiss) and righthand side (conclusion) of a rule are separated by the `=>` sign. The rules labeled `plus` and `minus` formalize the informal statements "A positive (resp. negative) activator marble can give positive (resp. negative) potential to a plain marble." "a marble with the `*` potential is a positive activator", and "a marble with the `@` potential is a negative activator." The rule labeled `cancel` formalizes the statement "If a marble with negative potential contacts a marble with positive potential the potential is cancelled and they both become plain".

```
mod MAGIC-MARBLES-RULES is
  inc MAGIC-MARBLES-STATE .
  vars pm0 pm1 : PlainMarble .
  rl[plus]:  pm0 [ pm1 | *] => [ pm0 | + ] [ pm1 | @ ] .
  rl[minus]:  pm0 [ pm1 | @] => [ pm0 | - ] [ pm1 | * ] .
  rl[cancel]:  [pm0 | +] [ pm1 | -] =>  pm0   pm1 .
endm
```

The variables `pm0`, `pm1` stand for arbitrary plain marbles. The change of activator potential in rules `plus,minus` formalize "When it does so, it changes parity and becomes a negative (resp. positive) activator."

Now we have an executable formal specification of magical marbles. What can we do with it? The simplest thing to do is to pick a starting state and use the rewrite and continue commands to watch it run. The command `rew [1] t .` rewrites the term `t` one step. The command `cont 1 .` continues rewriting one more step. Suppose we have an initial state `[whiteM | *] redM blueM` with two plain marbles and a positive activator. The rule `plus` applies to the subterm `[whiteM | *] redM` matching `pm0` to `redM` and `pm1` to `whiteM` (using multiset matching where the order of multiset elements doesn't matter), and replacing the matched subterm by the corresponding instance of the rule's righthand side, `[redM | +] [whiteM | @]`.

```
Maude> rew [1] [whiteM | *] redM blueM .
result Mix: blueM [redM | +] [whiteM | @]
```

Rewriting can be continued by rewriting with the `minus` rule and then by the `cancel` rule.

```
Maude> cont 1 .
result Mix: [redM | +] [blueM | -] [whiteM | *]    *** by [minus]
Maude> cont 1 .
result Mix: redM blueM [whiteM | *]                *** by [cancel]
```

This computation could be continued as many steps as you like. There are many other possible computations starting from our initial state, each making different choices of which plain marble to use in the `plus` step.

The command `search [n] istate =>+ pattern` searches the states reachable from `istate` for states matching `pattern`, stopping when it has found `n` solutions, or it runs out of states. It starts by finding all states that result from application of one rewrite rule, the finding all states that result from application of one rewrite rule to each of these states, and so on. Using the search command we can ask whether it is possible to make the red and blue marbles simultaneously positive, starting from the our initial state.

```
Maude> search [1] [whiteM | *] redM blueM
         =>+ M:Mix [redM | + ] [blueM | + ] .
Maude> no Solution .
```

The answer is no. If we add another positive activator then getting two positive marbles is easy, as we will see below.

An alternative to search is to use model checking. A model checker checks properties of the possible computations starting from a given initial state. The properties are expressed in Linear Temporal Logic (LTL). The module `MAGIC-MARBLES-MC` defines model checking states for Magic Marbles and defines two state propositions. The `{_}` operator encapsulates a mix, thus defining a boundary. Propositions are defined using the relation, `{mx}|= prop`, read the mix `mx` satisfies the proposition `prop`. A predicate on mixes (and other arguments) can easily be turned into a proposition, by defining a corresponding operator that maps the remaining arguments to the sort `Prop`.

As an example we define the function `contains` using two equations.

```
var mx mx0 mx1 : Mix .
op contains : Mix Mix -> Bool .
eq contains(mx0 mx1, mx0) = true .
eq contains(mx,mx0) = false [owise] .
```

The first equation uses multiset matching to identify the true case. The pattern `mx0 mx1` matches a mix term `mx` just if `mx` contains `mx0` (`mx1` is the rest of the mix). In this case `contains(mx,mx0)` will rewrite to `true`. An equation, such as the second equation above, tagged with the `[owise]` attribute, can only be used if no other equation applies. In our case the second equation applies to a term `contains(mx,mx0)` just if `mx` does not contain `mx0`. In this case the term rewrites to `false`. Then `hasMix(mx0)`,a proposition corresponding to the predicate `contains(mx,mx0)` is defined such that {`mx`} satisfies `hasMix(mx0)` if `contains(mx,mx0)`. We also define a proposition `nPot(n,p)` such that {`mx`} satisfies `nPot(n,p)` if `count(mx,p) == n`, where `count(mx,p)` is the number `n` of marbles in `mx` with potential `p`.

```
mod MAGIC-MARBLES-MC is
  inc MAGIC-MARBLES-RULES .
  inc MODEL-CHECKER .

  op `{_`} : Mix -> State .

  vars mx mx0 : Mix .  var p : Potential .   var n : Nat .
  op nPot : Nat Potential -> Prop .
  eq {mx} |= nPot(n,p) = count(mx,p) == n .
  op hasMix : Mix -> Prop .
  eq {mx} |= hasMix(mx0) = contains(mx,mx0) .
endm
```

If `P` is a state proposition, then the property `[]P` says that every state in a computation satisfies `P` and `[]~P` says that no state in a computation satisfies `P`. Thus to see if a state satisfying `P` can be reached, we can use the Maude model checker to evaluate `modelCheck({mx},[]~P)`. If a state can be reached satisfying `P`, the model checker will return a counter-example showing the transitions (state and rule label) of a computation containing such a state. For example, starting from a state with two plain red marbles, two plain blue marbles, and two positive activator marbles we can ask (1) can we get one red and one blue marble with positive potential; (2) can we get three marbles with positive potential; (3) what about four? The following is the model-checking query to answer the first question.

```
Maude> red modelCheck(
  {[whiteM | *] [blackM | *] redM redM blueM blueM},
                        [] ~ hasMix([redM | +] [blueM | +]) ) .
reasult ModelCheckResult: counterexample(
{{redM redM blueM blueM [whiteM | *] [blackM | *]},'plus}
{{redM blueM blueM [redM | +] [whiteM | @] [blackM | *]},'plus}
{{blueM blueM [redM | +] [redM | +] [whiteM | @] [blackM | @]},
                                                      'minus}
{{blueM [redM | +] [redM | +] [blueM | -]
                              [whiteM | *] [blackM |  @]},'plus}
{{[redM | +] [redM | +] [blueM | +] [blueM | -]
                     [whiteM | @] [blackM | @]},'cancel}, ....)
```

A counterexample is a list of transitions that fails to satisfy the property being model checked. A transition is represented by the starting state and the label of the rule applied. Thus the starting state of the first transition in the above counterexample is the initial state, and the `plus` rule is applied to one of the red marbles. The second transition applies the `plus` rule to the other red marble. This is followed by `minus` transition to recover a positive activator marble, and finally the remaining blue marble is made postive. The `...`s indicate that the counter example continues. This is an artifact of the model checker requirement that counter examples are infinite. The remainder of the computation is building a loop and can be ignored for our purposes.

The second and third questions are answered by the following queries. In fact the counter example for the first question also answers the second. The true result for the

third query says that in fact it is not possible to obtain four marbles with positive potential given our initial state.

```
Maude> red modelCheck(
    {[whiteM | *] [blackM | *] redM redM blueM blueM},
     [] ~ nPot(3,+)) .
result:  same as for (1)

Maude> red modelCheck(
    {[whiteM | *] [blackM | *] redM redM blueM blueM},
     [] ~ nPot(4,+)) .
result Bool: true
```

4 Building a Pathway Logic Knowledge Base

Now we describe a small PL knowledge base, SmallKB, that represents initial signaling events in response to Epidermal Growth Factor (Egf) stimulation. Egf initiates epidermal growth factor receptor (EgfR) signaling, which regulates growth, survival, proliferation, and differentiation in mammalian cells. In particular we will look at an initial part of the MAPK (Mitogen-Activated Protein Kinase) pathway [45, 11, 26, 24]. [1] This pathway is often represented by the a linear sequence

$$\text{Egf} \rightarrow \text{EgfR} \rightarrow \text{Grb2} \rightarrow \text{Sos1} \rightarrow \text{Ras} \rightarrow \text{Raf1} \rightarrow \text{Mek} \rightarrow \text{Erk}$$

Recall that a rewriting logic specification has two parts: an equational part specify structure and static properties of system states, and a rules part specifying system behaviors. A PL system is structured in four layers: (1) sorts and operations, (2) molecular components, (3) rules, and (4) queries. Layers 1-3 make up a Pathway knowledge base (PL KB) with layers 1 and 2 being the equational part. Layer 4 specifies models (initial states of interest).

4.1 The Equational Part

The *sorts and operations* layer declares the main sorts, subsort relations, and operators to construct representations of cellular states. The sorts of entities include Chemical, Protein, Complex, and Location (position within cellular compartments), and Cell. These are all subsorts of the sort, Soup, that represents 'liquid' mixtures as multisets of entities. The sort Modification is used to represent post-translational protein modifications. They can be abstract, to specify that a protein is activated, bound, or phosphorylated, or more specific, for example, phosphorylation at a particular site. Modifications are applied using the operator `[_-_]`. (Note the similarity to the annotation of marbles with potential in section 3.) For example, the term `[Raf1 - act]` represents `Raf1` in an activated state, and `[Hras - GTP]` represents the protein `Hras` in its "on" state

[1] The Wikipedia article on signal transduction `http://en.wikipedia.org/wiki/Signal_transduction` contains an excellent overview and is a good place for the reader not familiar with cellular signaling notions to start reading to learn more [23].

(loaded with GTP). (Hras is a specific member of the Ras family.) The term `[Gab1 - Yphos]` represents `Gab1` phosphorylated on a tyrosine site while `[Gab1 - phos(Y 627)]` represents `Gab1` phosphorylated on tyrosine 627. Complex formation is represented by the operation `(_:_)`. For example, the term `(Egf : [EgfR - act])` represents the complex resulting from binding of Egf to EgfR and subsequent activation of EgfR. A cell state is represented by a term of the form

```
[cellType | locs] .
```

The symbol `cellType` specifies the type of cell, for example Macrophage or Fibroblast. The symbol `Cell` is used to indicate an unspecified cell type. The symbol `locs` represents the contents of a cell organized by cellular location. Each location is represented by a term of the form `{ locName | components }` where `locName` identifies the location, for example `CLm` for cell membrane, and `components` stands for the mixture of proteins and other compounds in that location. For example,

```
[Cell | {CLm | EgfR  PIP2}
        {CLi | [Hras - GDP]  Src}
        {CLc | Gab1 Grb2 Pi3k Plcg Sos1}]) .
```

represents a generic cell with three locations: the membrane (location tag `CLm`) contains `EgfR` and a chemical `PIP2` (see below); the inside of the membrane (location tag `CLi`) contains `Hras` loaded with `GDP` and `Src`; and the cytoplasm (location tag `CLc`) contains `Gab1`, `Grb2`, `Pi3k`, `Plcg`, and `Sos1`.

The *components* layer specifies particular entities (proteins, genes, chemicals) and introduces additional sorts for grouping proteins by family or shared behavior. For example `ErbB1L` is a subsort of `Protein` whose elements are ErbB1 (`EgfR`) ligands. Components are declared as constants, giving their sort, and metadata giving synonyms and standard names that can be linked to databases providing other information. For example the epidermal growth factor `Egf` with sort `ErbB1L`, and metadata giving its HUGO and Swiss-Prot names, its *category*, and two synonyms is declared as follows.

```
op Egf : -> ErbB1L [metadata "(\
  (spname EGF_HUMAN)\
  (spnumber P01133)\
  (hugosym EGF)\
  (category Ligand)\
  (synonyms \"Pro-epidermal growth factor precursor, EGF\" \
       \"Contains: Epidermal growth factor, Urogastrone \"))"] .
```

Similarly, `EgfR` is declared simply to be a protein.

```
op EgfR : -> Protein [metadata "(\
  (spname EGFR_HUMAN)\
  (spnumber P00533)\
  (hugosym EGFR)\
  (category Receptor)\
  (synonyms \"Epidermal growth factor receptor precursor\" \
       \"Receptor tyrosine-protein kinase ErbB-1, ERBB1 \"))"] .
```

PIP2 is a chemical (a lipid) residing in the membrane. Its phosphorylated form, PIP3, plays an important role in a number of signaling pathways, either directly or through its cleavage products. Chemicals have metadata linking them to a KEGG database entry, where much information can be found.

```
op PIP2 : -> Chemical [metadata "(\
   (category Chemical)\
   (keggcpd C04569)\
   (synonyms \"Phosphatidylinositol-4,5P \" ))"] .
```

The *rules* layer is the heart of a PL KB. It contains rewrite rules specifying individual reaction steps. In the case of signal transduction rules represent processes such as activation, phosphorylation, complex formation, or translocation. The rules layer is discussed in Section 4.2 below.

The *queries* layer specifies initial states (called dishes) to be studied. Initial states can be thought of as describing "in silico experiments". They represent in silico Petri dishes containing a cell and ligands of interest in the surrounding mixture. An initial state is represented by a term of the form

```
PD(out cell)
```

where `cell` represents a cell state and `out` represents a soup of ligands and other molecular components in the cells surroundings. In fact a dish can contain many cells, however the current PL analysis tools only treat single cells. For example an initial state to study Ras activation in `SmallKB` is given by the dish term

```
op rasDish  : -> Dish .
eq rasDish =
     PD(Egf [Cell | {CLm | EgfR  PIP2}
                    {CLi | [Hras - GDP]  Src}
                    {CLc | Gab1 Grb2 Pi3k Plcg Sos1}]) .
```

representing a dish containing `Egf` and the cell discussed above.

4.2 The Rules Part

PL rules are curated from the literature, and each rule has associated evidence items describing experimental data that serve as evidence for the rule. Discussion of evidence is beyond the scope of the present document, as it requires some understanding of experimental methods to be meaningful. Rule 1 (label `1.EgfR.act`) describes the binding of an ErbB1 ligand to EgfR, formalizing the first step of the signaling sequence above.

```
rl[1.EgfR.act]:
  ?ErbB1L:ErbB1L
  [?CellType:CellType | ct {CLm | clm EgfR}]
  =>
  [?CellType:CellType | ct
    {CLm | clm ([EgfR - act] : ?ErbB1L:ErbB1L)} ] .
```

The term `?ErbB1L:ErbB1L` is a variable that matches any ErbB1 ligand, for example `Egf`, and `?CellType:CellType` is a variable that matches any cell type. `{CLm | clm EgfR}` matches any cell membrane location that contains `EgfR`, since `clm` is a variable that will match the rest of the membrane contents. Thus the left hand side subterm

```
[?CellType:CellType | ct {CLm | clm EgfR}]
```

matches any cell that contains `EgfR` in the cell membrane, since the variable `ct` will match any additional locations. For example, it matches the initial state `rasDish` with

```
?CellType:CellType := Cell
?ErbB1L:ErbB1L := Egf
clm := PIP2
ct := {CLi | [Hras - GDP]  Src} {CLc | Gab1 Grb2 Pi3k Plcg Sos1}
```

and the result of rewriting `rasDish` with rule 1 is

```
PD([Cell |
   {CLm | ([EgfR - act] : Egf)  PIP2}
   {CLi | [Hras - GDP]  Src}
   {CLc | Gab1 Grb2 Pi3k Plcg Sos1}]) .
```

which is obtained by instantiating the rules right hand side using the variable bindings from the left hand side match.

Once the receptor is activated it can recruit `Grb2` to the membrane interior. This is describe by the rule labeled `5.Grb2.reloc`.

```
rl[5.Grb2.reloc]:
  {CLm | clm ([EgfR - act] : ?ErbB1L:ErbB1L)  }
  {CLi | cli                   }
  {CLc | clc Grb2              }
  =>
  {CLm | clm ([EgfR - act] : ?ErbB1L:ErbB1L) }
  {CLi | cli [Grb2 - reloc]    }
  {CLc | clc                   } .
```

Notice that on the left, `Grb2` is in the `CLc` location (cytoplasm) while on the right it is in `CLi` location. The modification `reloc` makes the change in location explicit. It is not strictly necessary, but makes the changes easier to follow. Continuing the rewriting of `rasDish` with rule `5.Grb2.reloc` we get

```
PD([Cell |
   {CLm | ([EgfR - act] : Egf)  PIP2}
   {CLi | [Hras - GDP]  Src [Grb2 - reloc]}
   {CLc | Gab1 Pi3k Plcg Sos1}]) .
```

Now `Sos1` can be recruited to the membrane complex by binding to `Grb2`. This is described by rule `13.Sos1.reloc`. Note that in this particular representation the complex formation is abstracted to colocation. We could also make the complex explicit if it were needed for some analysis.

```
rl[13.Sos1.reloc]:
  {CLi | cli [Grb2 - reloc]                }
  {CLc | clc Sos1                          }
  =>
  {CLi | cli [Grb2 - reloc] [Sos1 - reloc] }
  {CLc | clc                               } .
```

The resulting state is

```
  PD([Cell |
     {CLm | ([EgfR - act] : Egf)  PIP2}
     {CLi | [Hras - GDP]  Src [Grb2 - reloc] [Sos1 - reloc]}
     {CLc | Gab1  Pi3k Plcg }]) .
```

In the next section we will see how to transform the Maude terms in to a graph representation that makes it easier to visualize and understand reaction networks and their evolution. In particular, a graphical representation of the above three step computation is shown below in Figure 2.

5 Computing with a PL KB

The Pathway Logic Assistant (PLA) provides interactive graphical access to a PL knowledge base. For this purpose, rule sets and computations are represented using Petri nets [38, 37, 46], which have a natural graphical representation, additionally, there are very efficient tools for analyzing the Petri net models generated by PLA.

5.1 PL Petri nets

Petri Nets were invented to model execution of concurrent processes and thus are nicely suited to modeling signals propagating through a cell. A Pathway Logic Petri net (simply called Petri net, in what follows) can be thought of as graph with two kinds of nodes: rule nodes (shown as squares) and occurrence nodes (shown as ovals). Rule nodes, called transitions in the Petri net community, represent reactions, and occurrence nodes, called places in the Petri net community, represent reactants, products, or modifiers. Occurrences can be thought of as atomic propositions asserting that a protein (in a given state) or other component (small molecule, complex, ...) occurs in a given compartment. In this view, rules are logical implications.

An occurrence oval is labeled by a string representation of the corresponding Maude term. For example the string representation of `Efg` outside a cell is `Egf-Out` and `Egf:EgfR-act-CLm` is the string representation of `Egf : [EgfR - act]` in the cell membrane. The reactants of a rule are the occurrences connected to the rule by arrows from the occurrence to the rule. The products of a rule are the occurrences connected to the rule by arrows from the rule to the occurrence. The modifiers of a rule (enzymes and other components that must be present but are unchanged) are the occurrences connected to the rule by a dashed arrow. For example, the Petri net representation of the rule for recruitment of `Sos1` is shown in Figure 1.

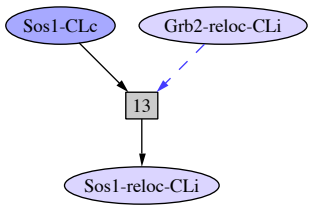
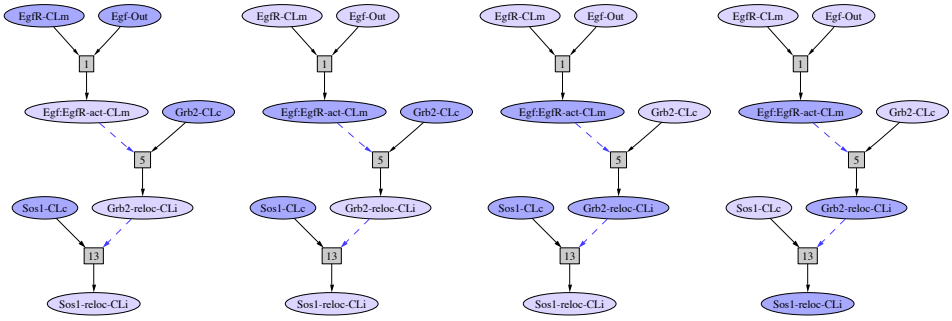


Fig. 1. Petri net transitions for rule 13.Sos1.reloc

The rule is represented by the rectangle labeled `13` (short form of `13.Sos1.reloc`). The reactant `Sos1` in the cytoplasm is represented by the oval labeled `Sos1-CLc` with an arrow from the oval to the rule rectangle. The product `[Sos1 - reloc]` at the membrane interior is represented by the oval labeled `Sos1-reloc-CLi` with an arrow from the rule rectangle to the oval. `[Grb2-reloc]` drives the reaction but is not changed (at our level of representation), thus it is represented by the oval labeled `Grb2-reloc-CLi` with a dashed arrow from the oval to the rule rectangle.

A set of Petri net rules corresponding to the rules of a PL knowledge base is called a transistion knowledge base (TKB). The analog of a PL dish is a PL Petri net state, which specifies which occurrences are present, that is, it specifies the state and location of each molecular component. Given a state, a Petri net rule is enabled if all of its occurrences connected by incoming arrows (reactants and modifiers) are present in the state. When an enabled rule fires, the reactant occurrences are removed from the state and the product occurrences are added. The modifier occurrences are left unchanged.

Corresponding to a PL model, a Petri net model consists of a set of rules (a TKB) and an initial state. To execute a Petri net model one puts tokens on the ovals corresponding to occurrences present in the initial state, and moves tokens as rules become enabled and fired. Figure 2 shows the execution of a Petri net model of the process that recruits `Sos1` to the membrane interior.



(a) initial state (b) step 1 (c) step 2 (d) step 3

Fig. 2. Execution of the Sos1 recruitment pathway

There are three rules (corresponding to the rewrite rules discussed in Section 4). Darker ovals represent occurrences that are present (marked with a token). Figure 2(a) shows the initial state with `Egf-Out`, `EgfR-CLm`, `Grb2-CLc`, and `Sos1-CLc` marked as initially present. The only rule enabled is rule 1. Figure 2(b) shows the result of firing rule 1, removing tokens from `Egf-Out` and `EgfR-CLm` and adding a token to `Egf:EgfR-act-CLm`. Now rule 5 is enabled and Figure 2(c) shows the result of firing rule 5. This enables rule 13 and Figure 2(d) shows the final state.

Starting with a PL KB, we convert it to a Petri net TKB, and convert dishes to occurrence sets, in a way that preserves the possible executions. We can then analyze models by finding subnets relevant to a desired state (goal), finding pathways reaching a goal, comparing subnets and/or pathways, finding knockouts (omissions from the initial state that prevent reaching a goal), or exploring a network of rules by incrementally adding connected components and rules to a given starting subnet. This is explained briefly in the following subsections. Details, including proof that the Petri net representation is equivalent to the rewriting logic representation, can be found in [50].

5.2 Converting a PL KB to a Petri net TKB

Transformation to Petri net representation accomplishes several things. One is support for graphical representation. Another is making things concrete. The full PL representation allows for rules that express families of reactions and for multiple cells and cell types. In PLA we restrict attention to systems with a single cell and for each variable that stands for a single component, we fix a specific (finite) set of components that can be values of that variable. For example, in the `SmallKB` knowledge base, there are two proteins that can be values of the variable `?ErbB1L:ErbB1L` of sort `ErbB1L`, namely `Egf` and `Tgfa`. Producing a Petri net representation of the Pathway Logic knowledge base proceeds in two steps. The first step transforms rules to occurrence form, by transforming the dishes or cells appearing in PL rules into occurrence sets. The second step instantiates remaining variables with known values.

Formally, an occurrence is a pair consisting of a component (a protein, possibly modified, a small molecule, or a complex) and a location name. For example, `<Egf, Out>` is an occurrence representing Efg outside a cell and `< Egf : [EgfR - act], CLm>` is a occurrence representing Egfr complexed with Egf and activated in the cell membrane. The left or right side of a rule is transformed by pairing each component with its location (the name of the enclosing location), dropping the location container, and dropping variables such as `ct` or `clm` that serve only to name location contents that are not important for the rule. Thus rule 1 (`1.EgfR.act`) becomes

```
rl[1.EgfR.act.pn]
    < ?ErbB1L:ErbB1L, Out >  < EgfR , CLm >
   =>
   < ?ErbB1L:ErbB1L : [EgfR - act], CLm >
```

When we instantiate remaining variables, we also convert rules (logical statements) into elements of a data type called `PNetTransition`. This allows us to compute with and reason about the Petri net rules directly. An interpreter to execute rules represented as data can be defined by a single rewrite rule. A pnet transition term has the form

```
pnTrans(label,iOccs,oOccs,bOccs)
```

where `label` is a quoted identifier, and `iOccs`, `oOccs` `bOccs` are multisets of occurrences: `iOccs` are the occurrences required and removed by the transition (connected to the rule by incoming arrows), `oOccs` are the occurrences produced by the transition (connected to the rule by outgoing arrows), and `bOccs` are the occurrences required but not removed by the transition (connected to the rule by dashed arrows). As an example, two pnet transitions are obtained by instantiating the occurrence form of rule 1, the first by instantiating the variable `?ErbB1L:ErbB1L` with `Egf`

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

and the second by instantiating with `Tgfa`.

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

The transition label is the rule label, suffixed with `#1`, `#2`, ... if there are multiple instantiations. Since in rule 1 there are no unchanged occurrences, `iOccs` is simply the instantiated occurrences from the rule lefthand side, `oOccs` is the instantiated occurrences from the rule righthand side, and `bOccs` is `none`, the empty occurrence set.

As another example, the Sos1 recruitment rule (`13.Sos1.reloc`) is transformed into the following pnet transition.

```
pnTrans('13.Sos1.reloc,
        < Sos1,CLc >,
        <[Sos1 - reloc],CLi >,
        <[Grb2 - reloc],CLi >)
```

In this case `bOccs` is `<[Grb2 - reloc],CLi >` which is necessary for the rule to fire, but not used up.

The process of converting a rule set into a list of pnet transitions uses Maude's meta-level, where rules are represented as data and one can manipulate terms with variables (which are also just data in the meta-level).

5.3 PL PNet models

Once we have a TKB we can derive models and compute with them, asking for subnets, pathways, knockouts, and making comparisons. A model consists of a pnet transition list (specifying the possible transitions) and a set of occurrences representing the initial (or current) state. It is derived from a dish and a TKB by transforming the dish into a set of occurrences and doing a *forward collection* in the TKB from the occurrence set to derive the set of transitions that could possibly be enabled in a computation starting

from the initial state. The idea of the forward collection is to iteratively augment the occurrence set with all occurrences that could be produced by firing enabled transitions (from TKB), without removing the `iOccs` part of the transition, and then add enabled transitions to the accumulating list of transitions. The result of forward collection produces a transition list that is possibly an over approximation. That is, any transition of TKB that becomes enabled in some computation from the initial state will be in the accumulated transition list, but there may be some transitions that do not become enabled in any computation from the initial state. The crucial point is that we don't loose any possible computations by restricting the set of transitions to be considered, and the simple over approximation makes the model derivation feasible.

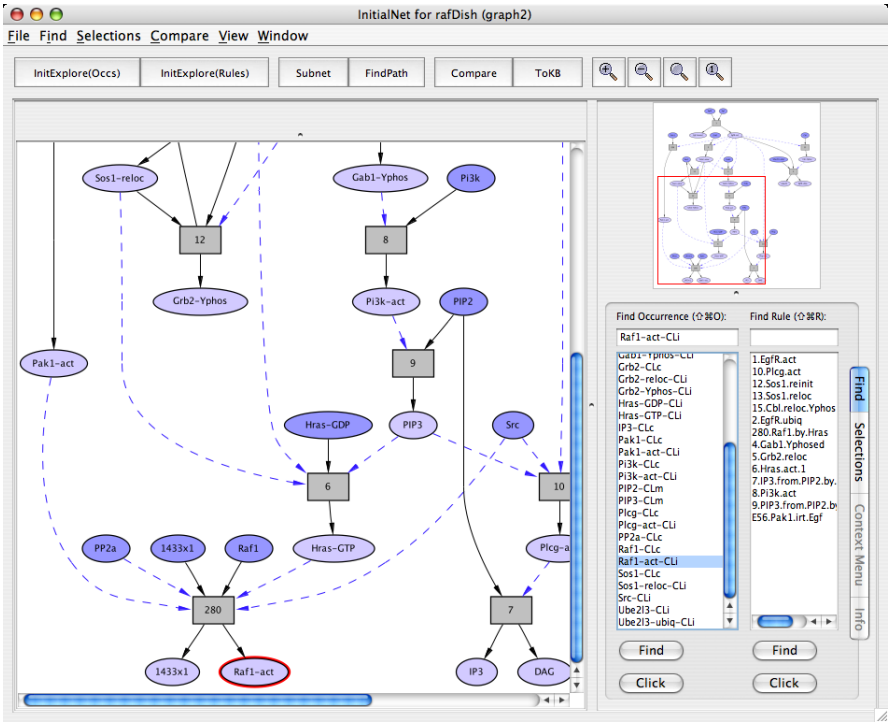


Fig. 3. Raf Activation Model viewed in PLA.

Figure 3 shows a screen shot of the Petri net model of Raf activation, generated by PLA from the dish `rafDish` whose occurrence set is `rafDishOccs`

```
eq rafDishOccs =
  < Egf, Out > < EgfR, CLm >  < PIP2, CLm >
  < [Hras - GDP], CLi> < Src, CLi> < [Ube213 - ubiq], CLi >
  < Cbl, CLc >  < Gab1, CLc >  < Grb2, CLc >  < Pi3k, CLc >
  < Plcg, CLc >  < Sos1, CLc >  < 1433x1, CLc >  < Pak1, CLc >
```

```
< Raf1, CLc >  < PP2a, CLc > .
```

As discussed above, ovals are occurrences, with initial occurrences darker. Rectangles are transitions. Dashed arrows indicate an occurrence that is both input and output. The thumbnail sketch in the upper right shows the full network. The main frame shows a magnified version of the portion of the network in the red rectangle. The view in the main frame can be changed by dragging the red rectangle around in the thumbnail frame. It can also be changed using the scroll bars. The Finder in the lower right allows one to locate occurrences and rules by name, and center the view on the selected node.

PLA provides a simple query language for specifying signaling pathways of interest. A query specifies three sets: goals, avoids, and hides. Goals are a set of occurrences that should appear at the end of a pathway, as they represent properties of a desired state. Avoids are a set of occurrences that should not appear in any state in the execution of the pathway. Hides are a set of rules that should not fire in the pathway. To make a query, goals, avoids, and hides sets can be selected by clicking the occurrence or rule to select, and pressing the corresponding button in the information window that appears. Once query elements have been selected, the user can ask to see the relevant subnet or to find a path. The relevant subnet contains all of rules needed for any (minimal) pathway satisfying the query, while the path is just the first path found by the analysis tool. The relevant subnet is computed directly from the pnet transitions list, together with the initial state and query elements in a manner similar to the forward collection function above.

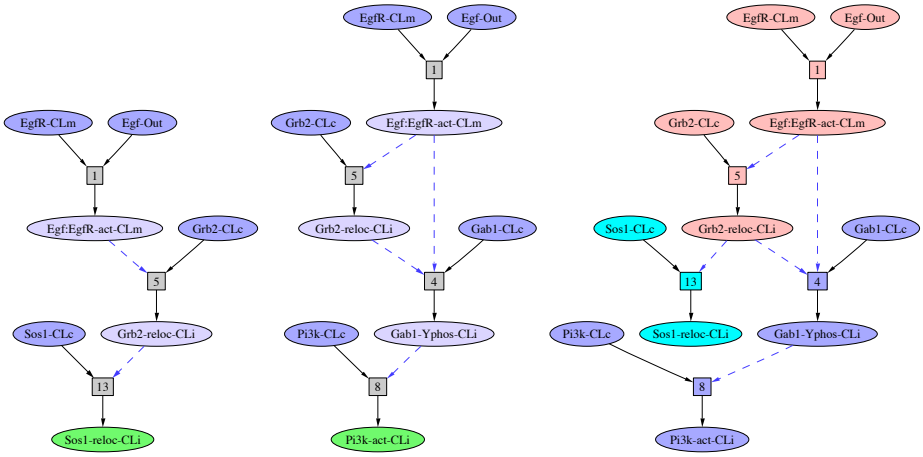
The logic underlying the query language is a temporal logic Goal queries are answered by model-checking the assertion that the goal set is not reachable, from the initial state `ioccs` in a transition list `pntl*` from which transitions that produce an avoid or are in the hides set are removed.

```
(pntl*,ioccs) |= []~ goal
```

A pathway satisfying a query is obtained by translating the reduced pnet transition list and query into the language of the LoLA model checker [44, 30], asserting that no such pathway exists. If a pathway does exist LoLA returns a list of transitions in the pathway, which PLA converts to a Petri net for display and possibly further analysis. The LoLA model checker is highly optimized for Petri nets, and thus allows use to compute with very large models.

Figure 4 shows pathways in the Raf1 model that recruit `Sos1` (a), and activate `Pi3k` (b), obtained by making `Sos1-reloc-CLi` or `Pi3k-act-CLi` a goal (indicated by coloring the oval green) and using FindPath. The key property of a pathway is that executing the pathway Petri net starting from the initial state leads to a state in which the goal(s) are among the occurrences, that is, a state satisfying the goal is reached. Furthermore, none of the avoids or hides appear in the pathway.

In addition to generating pathway subnetworks, two subnets can be compared. For this, the two networks are merged into one. Figure 4(c) shows the result of comparing the `Sos1` and `Pi3k` pathways. Nodes in both pathways are colored pink, nodes only in the `Sos1` pathway are colored cyan, and nodes only in the `Pi3k` pathway are colored dark lavendar.



(a) Sos1 Path (b) Pi3k Path (c) Comparison

Fig. 4. Sos1 and Pi3k activation paths and comparison

The `Sos1` and `Pi3k` pathways are part of the model of Hras activation, and ultimately of Raf activation. Figure 5(a) shows a pathway activating Hras, obtained by specifying `Hras-GTP-CLi` as a goal. Figure 5(b) shows the `Sos1` and `Pi3k` comparison as a subnet of the `Hras` activation pathway (nodes only in the `Hras` path are white).

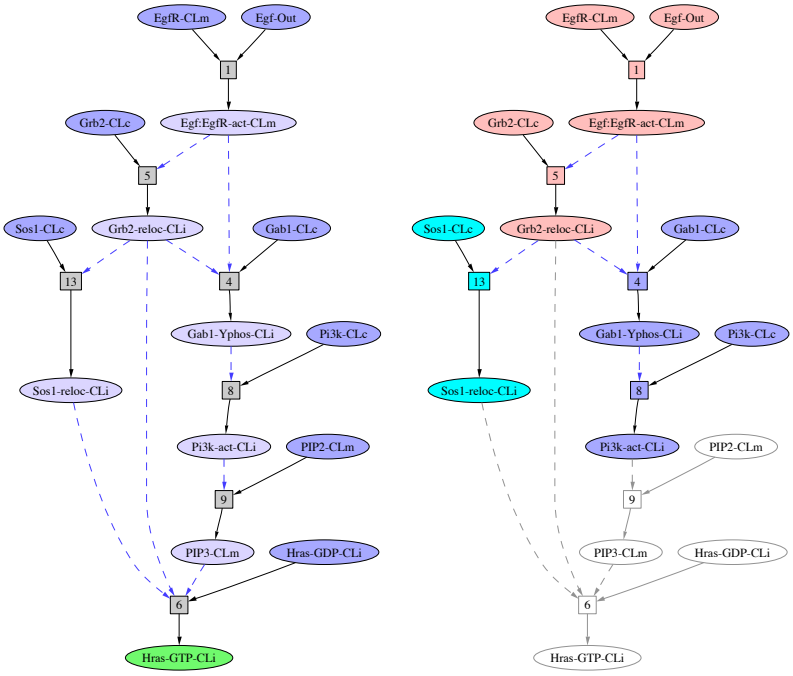
The SmallKB and the Ras and Raf1 activation initial states are available as part of the Pathway Logic Demo available from the Pathway Logic web site `http://pl.csl.sri.com/` along with papers, tutorial material and download of the Pathway Logic Assistant tool.

6 Conclusion

Pathway Logic is a symbolic systems biology approach to modeling biological processes based on rewriting logic. We have described the use of Pathway Logic to model signal transduction processes, and the use of the Pathway Logic Assistant to browse and analyse these models. Pathway logic can also be used to model and analyze metabolic networks and to interpret experimental data. Future challenges include integration of signaling and metabolic network models, and new abstractions to simplify networks and identify meaningful modules.

References

1. Muffy Calder, Vladislav Vyshemirsky, David Gilbert, and Richard Orton. Analysis of signalling pathways using the PRISM model checker. In G. Plotkin, editor, *Proceedings of the Third International Conference on Computational Methods in System Biology*, 2005.



(a) rasPath (b) Sos1 and Pi3k SubPaths

Fig. 5. Ras activation pathway and Sos1,Pi3k subnets

2. Laurence Calzone, Nathalie Chabrier-Rivier, Francois Fages, Lucie Gentils, and Sylvain Soliman. Machine learning bio-molecular interactions from temporal logic properties. In G. Plotkin, editor, *Proceedings of the Third International Conference on Computational Methods in System Biology*, 2005.
3. Luca Cardelli. Abstract machines of systems biology. In *Transactions on Computational Systems Biology III*, volume 3737 of *LNCS*, pages 145–168. 2005.
4. N. Chabrier-Rivier, M. Chiaverini, V. Danos, F. Fages, and V. Schächter. Modeling and querying biomolecular interaction networks. *Theoretical Computer Science*, 351(1):24–44, 2004.
5. C. Chaouiya. Petri net modelling of biological networks. *Briefings in Bioinformatics*, 8:210–219, 2007.
6. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Mart´-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude: A High-Performance Logical Framework*. Springer, 2007.
7. J. S. Edwards, M. Covert, and B. O. Palsson. Metabolic modelling of microbes: The flux-balance approach. *Environmental Microbiology*, 4(3):133–140, 2002.
8. S. Efroni, D. Harel, and I.R. Cohen. Towards rigorous comprehension of biological complexity: Modeling, execution and visualization of thymic t-cell maturation. *Genome Research*, 2003. Special issue on Systems Biology, in press.
9. Steven Eker, Merrill Knapp, Keith Laderoute, Patrick Lincoln, José Meseguer, and Kemal Sonmez. Pathway Logic: Symbolic analysis of biological signaling. In *Proceedings of the*

Pacific Symposium on Biocomputing, pages 400–412, January 2002.
10. Steven Eker, Merrill Knapp, Keith Laderoute, Patrick Lincoln, and Carolyn Talcott. Pathway Logic: Executable models of biological networks. In *Fourth International Workshop on Rewriting Logic and Its Applications*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.
11. G. Pearson et al. Mitogen-activated protein (MAP) kinase pathways: regulation and physiological functions. *Endocr. Rev.*, pages 153–183, 2001.
12. F. Fages, S. Soliman, and N. Chabrier-Rivier. Modelling and querying interaction networks in the biochemical abstract machine BIOCHAM. *Journal of Biological Physics and Chemistry*, 4(2):64–73, 2004.
13. Jasmin Fisher and Thomas A. Henzinger. Executable cell biology. *Nature Biotechnology*, 25(11), 2007.
14. H. Genrich, R. Küffner, and K. Voss. Executable Petri net models for the analysis of metabolic pathways. *Software Tools for Technology Transfer*, 3, 2001.
15. R. Ghosh, A. Tiwari, and C. Tomlin. Automated symbolic reachability analysis with application to delta-notch signaling automata. In O. Maler and A. Pnueli, editors, *Hybrid Systems: Computation and Control HSCC*, volume 2623 of *LNCS*, pages 233–248. Springer, April 2003.
16. David Gilbert, Monika Heiner, and Sebastian Lehrack. A unifying framework for modelling and analysing biochemical pathways using petri nets. In Muffy Calder and Stephen Gilmore, editors, *CMSB*, volume 4695 of *Lecture Notes in Computer Science*, pages 200–216. Springer, 2007.
17. P. J. Goss and J. Peccoud. Quantitative modeling of stochastic systems in molecular biology using stochastic Petri nets. *Proceedings of the National Academy of Science*, 95:6750–6755, 1998.
18. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
19. T.A. Henzinger. The theory of hybrid automata. In *11th IEEE Symposium on Logic in Computer Science*, pages 278–292, 1996.
20. W. S. Hlavacek, J. R. Faeder, M. L. Blinov, R. G. Posner, M. Hucka, and W. Fontana. Rules for modeling signal-transduction systems. *Science STKE*, July 2006.
21. R Hofestädt. A Petri net application to model metabolic processes. *Systems Analysis Modelling Simulation*, 16:113–122, 1994.
22. N. Kam, D. Harel, H. Kugler, R. Marelly, A. Pnueli, J. Hubbard, and M. Stern. Formal modeling of C.elegans development: A scenario-based approach. In *First International Workshop on Computational Methods in Systems Biology*, volume 2602 of *Lecture Notes in Computer Science*, pages 4–20. Springer, 2003.
23. Merrill Knapp, February 2008. personal communication.
24. W. Kolch. Meaningful relationships: The regulation of the Ras/Raf/MEK/ERK pathway by protein interactions. *Biochem J.*, 351:289–305, 2000.
25. R. Küffner, R. Zimmer, and T. Lengauer. Pathway analysis in metabolic databases via differential metabolic display (DMD). *Bioinformatics*, 16:825–836, 2000.
26. J. M. Kyriakis and J. Avruch. Mammalian mitogen-activated protein kinase signal transduction pathways activated by stress and inflammation. *Physiol. Rev.*, 81:807–869, 2001.
27. L.Cardelli. Brane calculi interactions of biological membranes. In *Computational Methods in Systems Biology*, volume 3082 of *LNCS*. Springer, 2004.
28. C. Li, Q. W. Ge, M. Nakata, H. Matsuno, and S. Miyano. Modelling and simulation of signal transductions in an apoptosis pathway by using timed petri nets. *Journal of Bioscience*, 32:113–127, 2007.

29. P. Lincoln and A. Tiwari. Symbolic systems biology: Hybrid modeling and analysis of biological networks. In R. Alur and G. Pappas, editors, *Hybrid Systems: Computation and Control HSCC*, volume 2993 of *LNCS*, pages 660–672. Springer, March 2004.
30. LoLA: Low Level Petri net Analyzer, 2004. `http://www.informatik.hu-berlin.de/~kschmidt/lola.html`.
31. H. Matsuno, A. Doi, M. Nagasaki, and S. Miyano. Hybrid Petri net representation of gene regulatory network. In *Pacific Symposium on Biocomputing*, volume 5, pages 341–352, 2000.
32. J. Meseguer. Conditional Rewriting Logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
33. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
34. R Milner. *Communicating and Mobile Systems: The pi-Calculus*. Cambridge University Press, Cambridge, UK, 1999.
35. F. Nielson, H. R. Nielson, C. Priami, and D. Rosa. Control flow analysis for bioambients. In *BioConcur*, 2003.
36. Gh. Päun. *Membrane Computing. An Introduction*. Springer-Verlag, 2002.
37. J. L. Peterson. *Petri Nets: Properties, analysis, and applications*. Prentice-Hall, 1981.
38. C. A. Petri. Introduction to general net theory. In Brauer, W., editor, *Net Theory and Applications, Proceedings of the Advanced Course on General Net Theory of Processes and Systems, Hamburg, 1979*, volume 84 of *LNCS*, pages 1–19, Berlin, Heidelberg, New York, 1980. Springer-Verlag.
39. C. Priami, A. Regev, E. Shapiro, and W. Silverman. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Information Processing Letters*, 80:25–31, 2001.
40. M.J. Prez-Jimnez and F.J. Romero-Campero. Modelling EGFR signalling cascade using continuous membrane systems. In G. Plotkin, editor, *Proceedings of the Third International Conference on Computational Methods in System Biology*, 2005.
41. V. N. Reddy, M. N. Liebmann, and M. L. Mavrovouniotis. Qualitative analysis of biochemical reaction systems. *Computational Biological Medicine*, 26:9–24, 1996.
42. A. Regev, E. Panina, W. Silverman, L. Cardelli, and E. Shaprio. Bioambients: An abstraction for biological compartments, 2004.
43. A. Regev, W. Silverman, and E. Shapiro. Representation and simulation of biochemical processes using the pi-calculus process algebra. In *Pacific Symposium on Biocomputing*, volume 6, pages 459–470. World Scientific Press, 2001.
44. Karsten Schmidt. LoLA: A Low Level Analyser. In Mogens Nielsen and Dan Simpson, editors, *Application and Theory of Petri Nets, 21st International Conference (ICATPN 2000)*, volume 1825 of *Lecture Notes in Computer Science*, pages 465–474. Springer, 2000.
45. R. Seger and E. G. Krebs. The mapk signaling cascade. *FASEB J.*, 9(9):726–735, 1995.
46. M.-O. Stehr. A rewriting semantics for algebraic nets. In C. Girault and R. Valk, editors, *Petri Nets for System Engineering – A Guide to Modelling, Verification, and Applications*. Springer-Verlag, 2000.
47. C. Talcott, S. Eker, M. Knapp, P. Lincoln, and K. Laderoute. Pathway logic modeling of protein functional domains in signal transduction. In *Proceedings of the Pacific Symposium on Biocomputing*, January 2004.
48. Carolyn Talcott. Formal executable models of cell signaling primitives. In Tiziana Margaria, Anna Philippou, and Bernhard Steffen, editors, *2nd International Symposium On Leveraging Applications of Formal Methods, Verification and Validation ISOLA06*, pages 303–307, 2006.
49. Carolyn Talcott. Symbolic modeling of signal transduction in pathway logic. In L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, editors, *2006 Winter Simulation Conference*, pages 1656–1665, 2006.

50. Carolyn Talcott and David L. Dill. Multiple representations of biological processes. *Transactions on Computational Systems Biology*, 2006.
51. Ashish Tiwari. Abstractions for hybrid systems. *Formal Methods in Systems Design*, 32(1):57–83, 2008.
52. Ashish Tiwari, Carolyn Talcott, Merrill Knapp, Patrick Lincoln, and Keith Laderoute. Analyzing pathways using sat-based approaches. In Hirokazu Anai, Katsuhisa Horimoto, and Temur Kutsia, editors, *Algebraic Biology 2007*, volume 4545 of *LNCS*, pages 155–169, 2007.
53. Ionela Zevedei-Oancea and Stefan Schuster. Topological analysis of metabolic networks based on Petri net theory. *In Silico Biology*, 3(0029), 2003.