Declarative Specification of Gene Regulatory Networks and Query Evaluation in PathSys* *(Extended Abstract)*

Amarnath Gupta† Yang Yang
University of California
San Diego
{gupta,yyang}@sdsc.edu

Aditya Bagchi
Indian Statistical
Institute, Calcutta
aditya@isical.ac.in

Animesh Ray
Keck Graduate
Institute
Animesh_Ray@kgi.edu

1 Introduction

Biological pathways depict how biological entities interact among themselves to perform a certain task. Metabolic pathways present the complex cascade of reactions that occur to break down a metabolite into constituent chemicals for eventual uptake by the organism. Signal transduction pathways describe how activation of proteins and genes are carried out by a complex network of chemical information-passing that take place through events like transcription of genes and translocation of chemical objects from one compartment to another. Since these interactions are very complex, and may include a large number of smaller steps, a wide variety of modeling paradigms have been used to study and analyze pathways. Some scientists study them as compactly represented Boolean circuits where elements are activated and deactivated like binary switches. Discrete and hybrid Petri Net models of pathways have been proposed to study the state-transition, reachability, deadlock and liveliness properties of a biological network. Many computational cell biologists create deep models of biochemical reactions as ordinary differential equations involving the rate of production and consumption of the individual reactants and reaction products. The nature of model used to study a pathway depends heavily on the degree of available information and the purpose of analysis. Very often even for one pathway, information about different parts is known to scientists at different levels of detail – while some reactions in a large, intricate pathway can be modeled quantitatively, some other reactions may be known only superficially, and are easier to model as discrete events.

The PathSys system aims to develop a general-purpose environment to construct, query, simulate and analyze biological networks by using multiple levels and paradigms. Toward this end, we have developed a graph-based data model to capture the inter-connectivity and nature of interactions between elements of a biological network. The graph-model is connected to a customized Petri Net engine that simulates the behavior of the network. A query engine sits atop both these modules to enable a user to query on both the graph-structured network as well as the states and transitions initiated by a simulation. A simulation can be run independently of the query module, and its results can be stored in a database. Alternatively, it can be invoked as part of a query execution.

*work supported by NSF QuBIC Grant and NSF ITR Grant
†Contact author

In this paper, we first describe *BioNetL*, a graph-based specification language by which the properties of network elements and their connectivity are declared to the system. The BioNetL compiler converts the network to a graph-structured database schema and instance. We then describe *TransGen*, a transformation language that uses a set of rules to convert a BioNetL specification into a state-transition model, specifically a Perti Net. Finally, we describe how a network developed using BioNetL can be queried through *BioNetSQL*, a version of SQL extended to perform graph operations on both the network and its state transition graph.

2 *BioNetL*: a Specification Language for Biological Networks

Informally, the vocabulary of a BioNetL model is based upon the notions of **objects**, **object-sites**, object **properties**, object **interactions**, and object **states**. A BioNetL model specifies a graph of when and how different sites of each object interact with corresponding sites of other objects, that lead to a change in the states of all interacting objects. To formalize this, BioNetL admits the following type structure:

i. **atomic types:** An atomic type is standard, and represents basic types like integer, float, string and Boolean. Typically, the value of an object property has an atomic type. For example, for the enzyme *phosphatase*, the property `concentration` may be declared as a `float`.

ii. **structured types:** A structured (or collection) type is also standard, and contains sets, bags, totally-ordered sets and record (defined as the product of other atomic or structured types). While other, more complex, types can be added, we find this set to be sufficient for our purposes.

iii. **site types:** A site type is an atomic or structured type with the special semantics that it represents a *location*, and that an interaction may occur at a site instance. A site is typically attached to an object. One could state that protein P has a phosphorylation site $P.ph_1$. But in general, a site can be declared independently. For example, one may model `extracellular_space` as an instance of a **site** type with no associated object. A more complex site such as the Polymerase-II binding site of a gene G can be specified as a range-restriction over a chromosome, which is a string over the alphabet {A C T G}.

iv. **object:** An object type is a record containing an object identifier, a number of object properties and optionally a set of sites belonging to the object. For example, one may declare an enzyme as:

```
object type enzyme {
            name:  string
            EC_Number:  string
            catalyzed_reaction:  equation
            reaction_type:  string
            natural_substrate:  string
            ...
}
```

- **subtype:** An object type can be declared as a subtype of another (using the :: symbol). For example, one can specify kinase as an enzyme that has a special site for kinase activity,

```
object type kinase::enzyme {
            k1:  kinase_action_site:  site
            ...
}
```

The "double declaration" in line 2 is a shortcut, but typical way of declaring that kinase_action_site is an instance of site. If a new kinase called `mykinase` has an additional kinase site, `mykinase::kinase` can be declared similarly.

v. **relationship typing:** A relationship has the default type `string` that can either be explicitly of implicitly declared. The sites declared with an object are internally recognized as instances of the system-defined `part_of` (system-defined types are explained later) relation. Thus, the previous declaration of the kinase is equivalent to the declaration:

```
relationship part_of::part_of(k1:kinase_action_site:site, kinase)
```

where the *type* of the `part_of` relationship is a primitive system-defined type called `part-of`, which has the properties of being antisymmetric, non-reflexive and transitive. On the other hand, the relationship `phosphorylates` is declared as:

```
relationship phosphorylates::event(kinase.kinase_action_site,
molecule.phosphorylation_site)
```

where subtyping the *event* type (a system-defined type) entitles `phosphorylates` to a state-transition description and a precondition-postcondition description (see later) thus making it a temporal entity.

Relationships may also be associated with *state-transition axioms*. Let us assume we want to define a new relationship between two sites on two different molecules called *bind_occupy* after Cook's BioD [1]. In addition to declaring the basic relationship signature, we want to capture the semantics of the `bind_occupy` event. Specifically, we want to specify that when the `bind_occupy` event occurs, the second argument's state variable increases (decreases) with the increase (decrease) of the first molecule's variable. This is expressed in BioNetL as:

```
relationship bind_occupy::event(molecule.site,molecule.site){
            axiom:  {
                    increases(self.$1.StateVar) =>
                    increases(self.$2.StateVar)
            }
}
```

where the symbol `=>` denotes logical implication[1]. The BioNetL also allows system designers to define implicit rules on predicates within axioms (called axiomPredicates later), to state for example, that *decrease* is the inverse of *increase*. Note that this specification is completely declarative and independent of whether the underlying simulation machinery is implemented by Petri Nets, hybrid Petri Nets or differential equations. Further, notice that the choice of the predicate `increases` in the axiom restricts the potentially interacting molecules to those that have linear state variables.

[1]The language also allows bidirectional implication `<=>`

A relationship type can be *conditional*, implying that for any instance of the relationship the condition must hold. The *scope* of the condition can be within the axiom, specifying when the axiom holds. Alternatively, it can be within the body of the relationship, denoting that the relationship itself is subject to the specified condition. Consider the event of phosphorylation:

```
relationship phosphorylates::event(kinase.kinase_action_site,
             molecule.phosphorylation_site){
             axiom:  {
                     flows_out(self.$1.StateVar, 'phosphoryl_group') =>
                     flows_in(self.$2.StateVar, 'phosphoryl_group')
             }
condition:  occurs(process(ATP2ADP)). }
```

Here the modeler treats the ATP to ADP conversion as just a named process without detailing its characteristics. This shows how BioNetL can be used at "variable" degrees of resolution where not everything has to be modeled in the same paradigm, at the same level of detail.

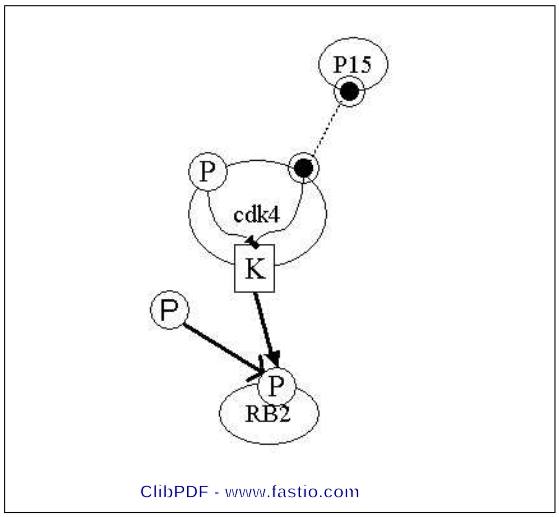
vi. **process:** The process type represents a *named event* with a possibly null attribute called *duration*. In terms of a graph, a process can be thought of as a non-object node that possibly acts as a proxy for a complex subgraph. In addition to being declared as an opaque type, process instances can be produced by "skolemizing" a relationship instance. For example, the relationship instance `phosphorylates(cdk4, 'Rb2')` can be converted to a process instance by the construct `process(phosphorylates(cdk4, 'Rb2'))`, which refers to the phosphorylation process of 'Rb2' by cdk4. This enables us to create an inter-edge relationship. For example, in Figure 1 the fact "the binding of 'p15' to 'cdk4' inhibits phosphorylation of Rb2 by cdk4" will be represented as `inhibits(process(bind_occupy('p15', cdk4)), process(phosphorylates(cdk4, 'Rb2')))`

Instance declaration in BioNetL internally[2] uses a similar syntax as type declaration. Figure 1 shows a small regulatory network and its instance declarations using the types described before. During instance creation, a number of additional constructs can be added.

i. **object states:** The declaration of objects in BioNetL can be supplemented by the declaration of states they can go through. Every object can have a set of attributes designated as the *state variable*. Currently, one state variable can be associated with each site declared for the object. The syntax for state declaration is illustrated as follows:

```
object cdc4:kinase {
       ph1:  phosphorylation_site:  site
       r1:  regulation_site:  site
       stateVar occupancy:  boolean at r1 {
          states=[occupied, unoccupied]
          initial=unoccupied
       }
       stateVar concentration:  float at ph1 {
          states=linear(0,20)
```

[2]An end-user will ultimately use a GUI like that in JDesigner from the SBML [2] group available at http://www.cds.caltech.edu/ hsauro/JDesigner.htm




```
object P15:gene{
  import geneCardwrapper(CDKN2B)
  a1:site
}.
object cdk4:enzyme{
  import geneCardwrapper(CDK4)
  p2:phsophorylation_site:site
  a2:site
}.
relationship
bind_occupy(p15.a1,cdk4.a2).
PP:phosphatase_site.
object Rb2:gene{
...
```

Figure 1: The left panel shows a small regulatory network adapted from [1]. The right panel shows a fragment of BioNetL description that would be compiled by PathSys to populate the database. Some standard properties of the gene are imported by wrapping a website.

```
            initial=0
        }
            ...
}
```

In this declaration, first note that the instance (identified by the : symbol) can add more properties, especially the state variables. For Boolean state variables, the state names can be used as state-inquiry predicates – thus, `exists (s:state, occupied(cdc4.r1, atState(s)))` is a valid query expression using the predicate `occupied`. For linear state variables, predicates `increasing, decreasing, constant`, and functions `valueOf, min, max` are permitted at this time. The initial values are optional.

2.1 The BioNetL Graph Database

The objects, sites and The BioNetL specification is mapped into a database schema using the following principles:

- Every object type (being specializations of `record`) becomes a relation. However the additional attribute and site information is not part of this record.
- Every site type is associated with a set of site instances, each of which refers to an object instance it belongs to, and a set of state variables that is watched at every site. Each state variable may have an initial value. This is modeled as a tree-structured data object represented here in XML syntax:

```
<sites>
    <phosphorylation_sites>
        <p1>
            <object  name="cdk4">
                <statevar name="occupancy" initial="unoccupied">
```

```
                    <value>unoccupied</>
                    <value>occupied</>
                </statevar>
                ...
            </object>
        </p1>
        ...
</sites >
```

In the implementation, this semistructured object is remapped to a relational schema using the node numbering technique described in [3, 4]. An additional relationship called `part_of` is created between the object identifier and the site id (here //sites/phosphorylation_sites/p1) to provide a more direct access from the object to its sites.

- At present system-defined relationships specified in BioNetL represent either binary ontological relationships (like `is_a` or `part_of`) or biological network relationships. The first category is represented in a table `ontoBinRel`:

```
ontoBinRel(
        name            string,
        roleHead        ID,
        headRoleName    string,
        roleTail        ID,
        tailRoleName    string,
        reflexive       [true, false, anti],
        symmetric       [true, false, anti],
        transitive      [true, false, anti],
        rules           Predicate
).
```

The `roleHead` and `roleTail` attributes represent the node ids which are on the "arrow-end" and the "tail-end" of the directional edge. The `headRoleName` and `tailRoleName` attributes represent the name of the corresponding roles. For example, for `phosphorylates`, these role names may be `phosphorylator` and `phosphorylated` respectively. The standard properties of the relationships are directly specified. The `rules` attribute is a predicate logic expression, which can be evaluated for intensional properties other than the standard properties. For example, the rule *if event(A) and part_of(B,A) then duration(B) ≤ duration(A)* should be specified as a predicate.

We used [1], SBML [2] and Cell-ML [5] as the references to design the biological relationships. Consequently, we model four basic types of relationships: representing one or two headed and one or two tailed arrows respectively[3]. For example, the basic two-headed, two-tailed relationship is internally represented as: `bi_bi_Rel`:

```
bi_bi_Rel(
        name              string,
        roleHead1         site,
        roleHead2         site,
        roleTail1         site,
        roleTail2         site,
        condition_global  Predicate,
```

[3]We thank Herbert Sauro from the SBML group for suggesting the sufficiency of these relationships.

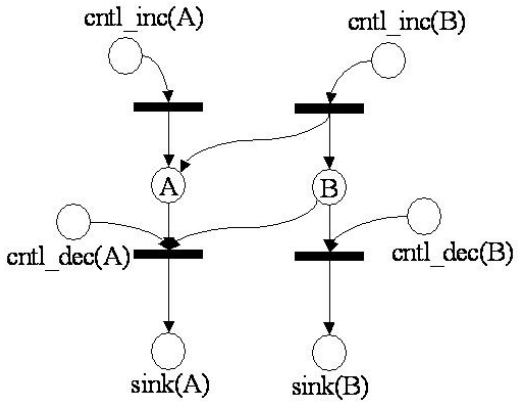


Figure 2: The Petri Net model for the bind_occupy relationship

```
        condition_1_1      Predicate, /* the first arm of the edge */
        condition_2_2      Predicate, /* the second arm of the edge */
        modelType          [LogicEngine, PetriNet, ...]
).
```

The modelType refers to the type of modeling tools that are available for the instance. More specialized relationships like *phosphorylates* are defined as specializations of these base biological relations. We present more details of the biological relationships in the next section.

3 TransGen: Transforming BioNetL Declarations to Petri Nets

Given a BioNetL declaration, the goal of TransGen is to produce a a Petri Net to model the behavior of the network. In itself, TransGen is a metalanguage that specifies *how* to make the transformation. Thus TransGen is not a user's language, but a system-designer's tool. The rationale for having a metalanguage rather than creating a single hardcoded method to construct the Petri Net is that we believe there is no unique, optimal way to make the transformation, and the system designer should have some control over the construction of the target specifications. We illustrate this by a few examples.

Consider the `bind_occupy` edge between p15 and cdk4 in Figure 1. The axiom in the model specifies that increasing the concentration of p15 will increase the occupancy of the binding site of cdk4. To construct a Petri Net from this however, one has to consider the same fact in terms of token flow. We can reinterpret the axiom as follows. As more tokens flow out of p15, more tokens flow into the binding site of cdk4. Thus, we first model cdk4 and p15 to have three places each, one for the "amount" of tokens in the gene, a second for controlling the "increase" event of gene's concentration and the third for controlling its "decrease" event. For book-keeping purposes a *sink* place is added to each to avoid accumulating tokens in places. The resulting sub-network for bind_occupy(A,B) is shown in Figure 2. The basic form of rules in Transgen can be seen from the following general-purpose directives:

```
forall $x in Bionetl.site mapto($x, PN.place).
forall $x in Bionetl.site {
```

```
        forall $y in derive($x.statevar.axiomPredicate) do {
                mapto($y, PN.place)
                create(transition, $t)
                connect(place($x), place($y), $t.input)
        }
}.
```

The *derive* directive creates indirect axiomPredicates – consequently, a place for *decreases* is created while the user has only specified the axiomPredicate *increases*. Rules specific to axiomPredicates can also be added – *if there exists an axiomPredicate called "increases" for a site, add a sink place to the site.*

```
forall $x in Bionetl.site {
        if exists($y in $x.statevar.axiomPredicate and $y="increases") do {
                  create(place, $p:sink)
                  connect($x, $p)
        }
}.
```

For the purpose of PathSys, we have created a set of Transgen rules to model specific forms of biological relationships of Figure 1. Their Petri Net definitions are included in Table 1.

The Petri Net generated by Transgen is stored using a persistent adjacency list and a vector indicating the initial marking. Figure 3 shows the complete Petri Net produced from Figure 1. While we leave the details out of this paper, in constructing a Petri Net, Transgen would often find conflicting choices – at this point the designer needs to resolve them. Techniques to generate correct and consistent Petri Nets automatically is still an unsolved problem.

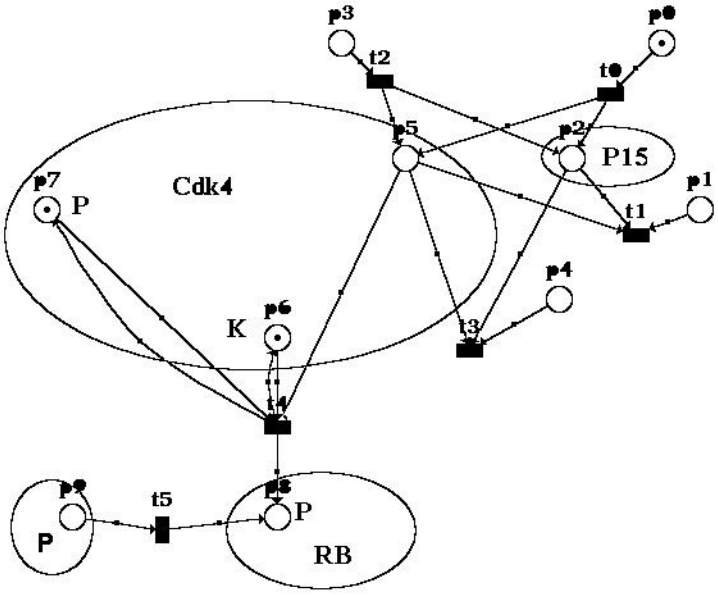


Figure 3: The Petri Net model for the small gene regulatory network for controling the Rb2 gene

4 Querying the Network with BioNetSQL

BioNetSQL extends SQL to query over paths, graphs, and state transition systems. In this extended abstract, we shall introduce the language through examples on our small gene regulatory network, and the secondary networks derived from it. These examples also illustrate the logical schema of these networks:

Example 1 [Path Query]: *Find the chain of genes or proteins downstream of P15 that has a phosphorylation site that is also acted upon by a Phosphatase.*

```
select path P:(gene or protein) from GeneReg
where P.start.name='P15' and type(P.end)='gene' or type(P.end)='protein' and
exists(select S:phosphorylation_site, ph:phosphatase from GeneReg
where flows_from(ph,S)and part_of(S,P.end)).
```

Note that a query can specify the type(s) (the default is all types) of the path elements in the projection clause, as well as in the body of the query. The attributes `start` and `end` are query variables associated with a path.

Example 2 [Subgraph Query]: *Find the regulatory (sub)network rooted at Mitogen that controls the import of cyclin D1 into the nucleus but does not include the sequence Ras-Raf-MEK-ERK.*

```
select graph G from GeneReg where G.root='Mitogen' and
exists(select edge E from GeneReg where E.name like '%cyclin import%'
and located_in(E,'nucleus') and E=G.element)
and not contains(G,'Ras'->'Raf'->'MEK'->'ERK').
```

The $\rightarrow$ symbol denotes a sequence of nodes. The keyword `is` assigns a path constant to the variable P, and the graph attribute `element` means that the desired edge E should be part of the projected graph G. The predicate `located_in` is reserved and stands for an anatomical location where the corresponding object (or process) is situated, while `contains(X,Y)` is `true` if $Y \in nodes(X) \cup edges(X)$.

Example 3 [Subgraph Query with Joins]: *Find the subnetwork of the Petri Net that either belongs to cdk4 or any site connected to it.* This needs the GeneReg graph and PN_GeneReg, its Petri Net graph to be joined.

```
select graph G from GeneReg GG, PN_GeneReg PG
where G.node=PG.node and G.edge=PG.edge and
Pl is PN_mapsto(('cdk4':protein).site) and G.place=Pl and G.transition=connects_to(Pl)
union
select graph G from GeneReg GG, PN_GeneReg PG
where G.node=PG.node and G.edge=PG.edge and
Tr is PN_mapsto(edge_of('cdk4':protein)) and G.Transition=Tr and G.place=connects_to(Tr)
```

The join is effected through the PN_mapsto() function that returns the graph element (node or edge) corresponding to a Petri Net. The union operation operates on both nodes and edges and is used here to guarantee that no graph element belonging to cdk4 is missed.

Example 4 [Querying the Reachability Set of Petri Net]: *What effect does increasing P15 concentration have on the occupancy of Rb2?* This "user query" is essentially a program, i.e., sequence of subqueries that first need to construct the reachability set of the

Petri Net, give it two different query conditions (a nominal concentration and an increased condition) for the state variable called `concentration` at the place for the P15 site, run the Petri Net engine for each of them, gather the results for the `occupancy` variable at the target phosphorylation site of Rb2, and finally compare them.

```
select token_count(Rb2.phosphorylation_site.occupancy) from PN_run
where initially (P15.site.concnetration=1 and P15.site.increasing_control=0 and
Rb2.phosphorylation_site.occupancy=0), and steps=5 into :T1
select token_count(Rb2.phosphorylation_site.occupancy) from PN_run
where initially (P15.site.concnetration=1 and P15.site.increasing_control=1 and
Rb2.phosphorylation_site.occupancy=0) and steps=5 into :T2
select compare_tokens(T1,T2) from T1,T2.
```

The reserved word `initially` specifies the token assignment for the initial marking of the Petri Net, and `steps` is a variable that controls the number of time steps for the network. In our example, the first segment produces a reachability set of 6 states, where the Rb2 site has 5 tokens at the end. In the second case, there are 12 states in the reachability set and the same place has 1 token after the same number of steps - consequently the `compare_tokens` function that accepts the time-trace of the tokens in the target place, reports that the occupancy at the phosphorylation site of Rb2 decreases with the increase compared to the first condition in the query.

5 Conclusion

The PathSys system attempts to bridge the gap between the data management view and modeling-for-simulation view of biological networks by considering them as both a queriable graph-structured database as well as an executable network, produced semi-automatically. The system can be queried and "run" under the control of an extended SQL language. While the implementation of the system is not yet complete, the initial results indicate that both structural and behavior analysis can be addressed in the PathSys framework.

References

[1] D. L. Cook, J. F. Farley, S. J. Tapscott, "A basis for a visual language for describing, archiving and analyzing functional models of complex biological systems", *Genome Biology*, 2(4), 2001.

[2] M. Hucka, A. Finney, H. Sauro, H. Bolouri, "Systems Biology Markup Language (SBML) Level 1: Structures and Facilities for Basic Model Definitions", March, 2001 available at http://www.cds.caltech.edu/erato/sbml/docs/.

[3] D. Shasha, J. T. Wang, R. Giugno, "Algorithmics and Applications of Tree and Graph Searching", *Proc. ACM Princ. of Database Syst. (PODS)*, 39–52, 2002.

[4] Q. Li, B. Moon, "Indexing and Querying XML Data for Regular Path Expressions", In *Proc. of VLDB*, Roma, Italy, September, 361–370, 2001.

[5] W. Hedley, M. Nelson, *CellML Specification*, available at http://www.cellml.org/public/specification/20010810/cellml_specification.html, August, 2001.

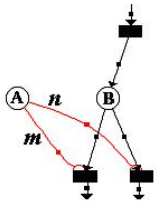
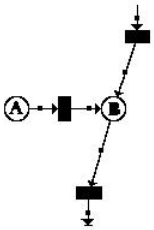
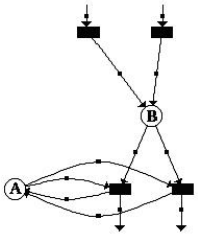
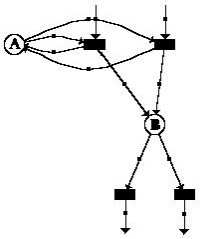
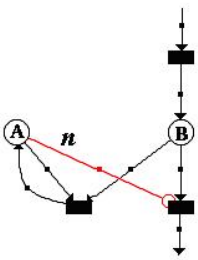
Pattern Name	Petri Net	Explanation
inhibits(A,B,(n, t_1)) $\wedge$ inhibits(A,B,(m, t_2))		A inhibits B on reaction t1 when n or more tokens have from A to B, A inhibits B on reaction t2 when m or more tokens have from A to B
flows_to(A,B,nosat)		tokens pass from A to B, but increasing the flow does not saturate the occupancy of B
activates(A,B)		tokens from A enable all outgoing transitions of B
increases(A,B,nosat)		tokens from A get added to the number of input tokens to B
decreases(A, B, n)		like inhibits, but after the number of tokens in A reaches n, the number of tokens in B will diminish

Table 1: The edge patterns used for biological networks in PathSys