Strategy-Driven Exploration for Rule-Based Models of Biochemical Systems with Porgy

Oana Andrei, Maribel Fernández, Hélène Kirchner, Bruno Pinaud

Strategy-Driven Exploration for Rule-Based Models of Biochemical Systems with Porgy

Oana Andrei[1], Maribel Fernández[2], Hélène Kirchner[3], and Bruno Pinaud[4]

[1] School of Computing Science, University of Glasgow, UK
oana.andrei@glasgow.ac.uk
[2] King's College London, UK
maribel.fernandez@kcl.ac.uk
[3] Inria, France
helene.kirchner@inria.fr
[4] University of Bordeaux, CNRS UMR5800 LaBRI, France
bruno.pinaud@u-bordeaux.fr

Abstract. This paper presents Porgy – an interactive visual environment for rule-based modelling of biochemical systems. We model molecules and molecule interactions as port graphs and port graph rewrite rules, respectively. We use rewriting strategies to control which rules to apply, and where and when to apply them. Our main contributions to rule-based modelling of biochemical systems lie in the strategy language and the associated visual and interactive features offered by Porgy. These features facilitate an *exploratory approach* to test different ways of applying the rules while recording the model evolution, and tracking and plotting parameters. We illustrate Porgy's features with a study of the role of a scaffold protein in Raf-1/MEK/ERK signalling.

Key words: Computational systems biology, Biochemical networks, Rule-based modelling, Graph transformations, Strategic rewriting, Visual Analytics Software

1 Introduction

The study of biochemical networks is a difficult task due to the usually high number of underlying processes and large body of data, some of which are only partially available at best. Rule-based modelling techniques [11] have been successfully applied in this area, giving rise to a methodology where the state of the biochemical network at a given point in time is represented as a data structure, and its dynamic behaviour as a set of rules (or transformations) describing changes of state. This methodology is supported by several software tools (see Section 1.2).

To facilitate the tasks associated with the specification, simulation, and analysis of biochemical networks, we propose a modelling framework based on the use of *port graphs with attributes* to represent system states. A port graph is a graph where edges connect to nodes at specific points, called ports. Nodes, ports and edges in the graph describe the network components and their relationships, while the attributes encapsulate the data values associated with each element. We use port graph rewrite rules to describe the evolution of the system by means of particular graph transformations.

Port graph rewrite rules are graphical representations of transitions in the system, thus they provide a direct, visual mechanism to observe the system's behaviour, as opposed to textual specifications, which usually require more effort to be interpreted by humans[5].

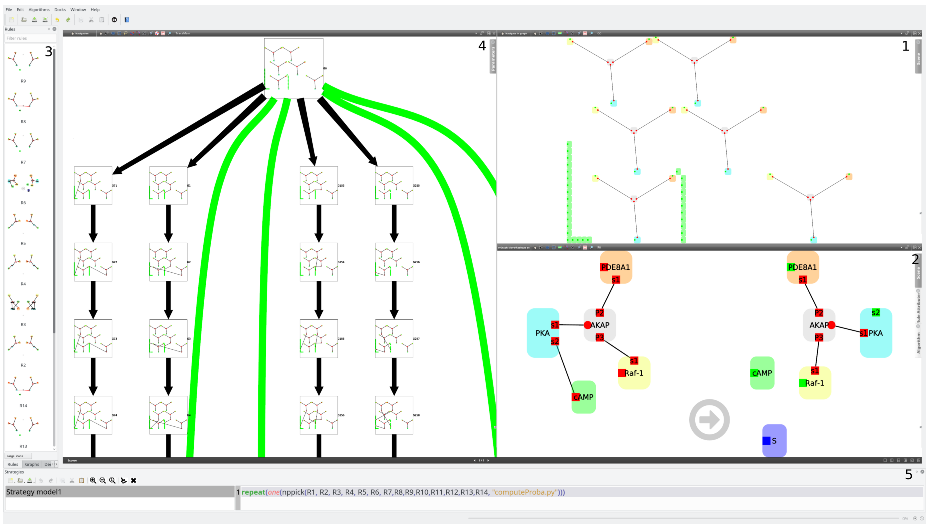
In addition to port graphs and rewrite rules, we consider it essential to add explicitly a third ingredient, which is often left implicit in this kind of modelling framework: together with an initial port graph and a set of port graph rewrite rules, our models include a *strategy expression*, which controls the application of the rules. Strategies help modellers when priorities over rule applications are known rather than exact reaction rates, when they want to bound the number of certain rule applications, or when they want to restrict the rule applications to certain parts of the graph. Thus, starting from a port graph representing the initial state of the system, and given a set of port graph rewrite rules, the strategy expression defines which transformation steps (among potentially many possible rule applications) are feasible. It may be the case that more than one transformation is possible at a given state, in which case instead of a single transformation step, we may have several alternatives, and in turn, generate several different sequences of steps starting from a given state. The various transformation sequences are organised as a tree, which we call the *derivation tree.* In other words, given an initial state and a set of rules, instead of implementing a specific strategy of application of the rules, we allow the users to specify the way rules should apply. This explicit control is done using a set of control operators, that indicate *when and where* to apply the rules.

This approach has been implemented in Porgy – an interactive visual environment for rule-based modelling of complex systems. Porgy provides support for the definition of graph-based models, the representation of rules describing model transformations, and the specification of strategies to control the application of rules (see Fig. 1). The derivation tree is itself a data structure that gives users access to the history of the system (i.e., the various states that preceded the current state). Porgy provides an interface to interact with the model, which includes visualisation of the derivation tree. In this way, users can analyse the system, track specific molecules, measure quantities of relevant elements at different points, compare concentrations, scatterplot, visualise alternative traces, etc.

1.1 Contributions

Our approach contributes to the general rule-based modelling trend of complex systems. Although Porgy is a general purpose modelling environment, not ex-

[5] "A picture is worth a thousand words" – Traditional idiom.



```
Strategy model1
1 repeat(one(nppick(R1, R2, R3, R4, R5, R6, R7,R8,R9,R10,R11,R12,R13,R14, "computeProba.py")))
```

Fig. 1. Overview of PORGY. The panels (see the black numbers in the top right corner) permit to (1) edit a graph (representing a state of the model); (2) edit a rule; (3) show the available rules; (4) display the derivation tree, a complete trace of the computing history; (5) write a strategy, using the strategy editor.

clusively dedicated to biology, its design has been strongly inspired from the study of biochemical systems and we focus here on this specific domain.

Our purpose is to introduce the main concepts of PORGY and present what distinguishes it from other rule-based modelling frameworks:

– PORGY is an interactive visual environment that offers graphical representations of different ingredients: the model species as port graph elements, the reactions as port graph rewrite rules, the biological pathways as strategic derivations. Thus, we do not need to go through writing an encoding of the model species and reactions, but simply use these visual representations. PORGY also provides useful visual tools for representing the system's evolution history and for plotting parameters evolution. Visualisation is quite helpful in a modelling environment to get intuitions when searching for an expected event or to debug a model.
– When studying biochemical systems that evolve along transformation steps, it is important to be able to have access to the evolution history. For that purpose, the derivation tree is displayed and provides visual execution traces of the process. The history of transformations leading to a state is available, as well as branching points where different choices are possible. This enables the causal analysis of specific events or components by placing a derivation under scrutiny.
– Through its strategy language, PORGY provides an explicit and flexible control mechanism for applying rules. By making the strategy explicit, we give

the user the possibility to specify where, in which order and how rules are applied. This finer description is often useful to specify the behaviour of complex systems and to be able to ensure some properties like termination of the transformation process or absence of conflicts. Moreover the user can also easily modify the strategy while keeping the same rules and so perform different in-silico experiments. For instance, we will see (in Section 3) how when we only have partial or approximate knowledge of the reaction rate constants, we can easily modify the strategy to perform different in-silico experiments.

1.2 Related Works

Pfaltz [29] was an early advocate of graph grammars, using graphs to represent pictures and geometrical problems. Bunke [10] proposes the use of *attributed graphs* and *graph transformations* to interpret diagrams and flowcharts. This work gave rise to numerous applications in a variety of domains (e.g., music notation, programming language implementation, software engineering, synthetic biology). In all these works, graph transformations are usually specified by means of rules [13, 17].

More recently, graph rewriting has been implemented in a variety of programming languages and modelling tools, however, in most cases a fixed strategy is used for rule application. Tools such as BioNetGen [19], RuleBender [36], Mosbie [39] and Kappa [15] integrate visualisation with modelling and simulation of rule-based intracellular biochemistry with emphasis on visual model exploration and integrated execution of simulations. States are represented by graphs describing the system components, often called agents; their interactions are defined by rules governed by associated rate constants, which determine how frequently the rules apply. BioNetGen explicitly uses the structure of port graphs, while the other tools use graph-based structures with labels.

Our contribution and main difference with respect to the rule-based tools enumerated above is the general strategy language, which allows the modeller to define explicit control over the application of a set of rules, and the inclusion of the derivation tree as part of the visualisation framework, which gives the modeller access to all the traces (i.e., sequences of transformation steps) to facilitate the analysis of the system's evolution.

Some tools offer users basic mechanisms to define strategies. In AGG [18], application of rules can be controlled by defining *layers* and then iterating through and across layers. PROGRES [34] allows users to define the way rules are applied and includes non-deterministic constructs, sequence, conditional and loops. The Fujaba Tool Suite [28] offers a basic strategy language, including conditionals, sequence and method calls, but no concurrency. GROOVE [33] permits to control the application of rules, via a control language with sequence, loop, random choice, try()else() and simple function calls. In GReAT [6] the pattern-matching algorithm always starts from specific nodes called "pivot nodes"; rule execution is sequential and there are conditional and looping structures. GrGen.NET [23] uses the concept of search plans to represent different matching strategies. GP [32] is the closest to Porgy in that it is a rule-based, non-deterministic language, where users can program their own strategy to define which rules are applied (but not where; there are no positioning constructs in its strategy language). GP's strategy language has three main control constructs: sequence, repetition and conditional. Only one derivation is built, although early versions of GP used a Prolog-like backtracking technique to explore the derivation tree. The tool described in [1] for chemical space exploration uses graphs to represent models of chemical compounds and rules to simulate their reactions, however, since the purpose is the exploration of very large chemical spaces, the notion of transformation step has been adapted to include partial rule applications, and the strategy language is geared towards systematic exploration of spaces of graphs, including predicates to prune unwanted derivations.

None of the available tools permits users to visualise the derivation tree, as in Porgy, where users can interactively navigate on the tree, visualise alternative derivations, follow the development of specific redexes (reactants), etc.

Porgy's strategy language is strongly inspired by GP and PROGRES, and by strategy languages developed for term rewriting, such as Elan [7] and Stratego [38]. The sublanguage to manipulate rewrite positions in Porgy is a lower level version of the built-in (predefined) traversal mechanisms available in term-based languages.

The probabilistic primitives in Porgy's strategy language (in particular, the ppick commands) allow users to model basic dynamic behaviour in non-deterministic and probabilistic systems. These features are used to deal with uncertainties and large volumes of data. The current implementation permits to use constants to specify probabilities in ppick commands, and offers a more general version of the command that incorporates a probability distribution as a parameter. In this way, it is possible to incorporate sophisticated behaviours (for example, where the application rate of rules depends on specific internal or external parameters).

To perform stochastic simulation in biological signalling pathways specifically [14], the Kappa language [15, 16] and the BioNetGen system [19] provide for each state of the system and each rule, a rate law used to determine the probability that a reaction occurs within a given fixed time step. How to compute this probability is detailed for instance in [12]; tools such as KaSim [14,15], LBS-Kappa and LMS-Kappa have been implemented to facilitate the creation and manipulation of Kappa models.

Both Kappa and BioNetGen offer mechanisms to combine modules in order to build systems in an incremental way. In Porgy, strategies can be named and then used as macros in other strategies. This offers basic support for modular design at strategy level. Higher-order port-graphs [22] formalise a notion of hierarchical port-graph that could be used to support incremental definition of models; this feature is under development for Porgy.

1.3 Outline

The paper is structured as follows. In Section 2, we describe the Porgy tool, which implements the rule-based modelling approach presented in this paper. Section 3 illustrates the approach via examples. We study a model of an A-kinase anchor protein (AKAP) and its mediating role in the crosstalk between the cyclic AMP (cAMP) and the Raf-1/ERK/MEK signalling pathway with respect to the activity of the cAMP-specific phosphodiesterase-8A1 (PDE8A1). We introduce the concepts of port graphs and port graph rewrite rules, and show how we use these structures to represent the molecular species and reaction rules. We then introduce the strategy language, give examples of strategy expressions for our AKAP model, and discuss some possible experiments and analysis options offered by Porgy. Section 4 provides additional information about port graph rewriting systems and strategies, and gives technical details about Porgy's implementation and installation.

2 Software

Porgy [4,20,30] is a visual environment that allows users to define port graphs and port graph rewrite rules, and to apply the rewrite rules in an interactive way, or via the use of strategies. A distinctive feature of Porgy's strategy language is that it allows users to use not only operators to combine graph rewriting rules but also operators to define the location in the target graph where rules should, or should not, apply. Users can create graph rewriting derivations and specify graph traversals using the language primitives to select rewriting rules and the subgraphs where the rules apply.

In order to support the various tasks involved in the study of a port graph rewriting system, the system provides facilities:

- to view each component of the rewriting system: the current graph being rewritten (or any other previous state), the derivation tree, the rules and the strategy, with drag-and-drop mechanisms to apply rules and strategies on a given state,
- to explore a derivation tree with all possible derivations,
- to perform on-demand reduction using a strategy expression, which permits to restrict or guide reductions,
- to track the reduction process throughout the whole derivation tree,
- to navigate in the derivation tree, for instance, backtracking and exploring different branches,
- to plot the evolution of a chosen parameter (a specific element in the port graph structure) along a derivation. The system supports synchronisation between the different views: selecting points on the plot view triggers the selection of the corresponding nodes in the derivation tree. Such a mechanism helps to track properties of the output graph along the rewriting process (see Sect. 3.5).

These features have been successfully applied to propose a visual analytics approach to compare propagation models in social networks in [21, 37].

PORGY is implemented on top of the open-source visualisation framework TULIP [5] as a set of TULIP plugins. The latest version of PORGY (including TULIP) can be downloaded from `http://porgy.labri.fr` either as source code or binaries for MacOS and Windows machines. The TULIP library natively supports many features for graph generation, manipulation and visualisation. We refer the reader to [20, 31] for more details about the interactive features of PORGY and how they are implemented with TULIP. See Note 4.5 for information about how to install PORGY.

3 Methods

We illustrate the use of PORGY with a model of an A-kinase anchor protein (AKAP) and its mediating role in the crosstalk between the cyclic AMP (cAMP) and the Raf-1/ERK/MEK signalling pathway with respect to the activity of the cAMP-specific phosphodiesterase-8A1 (PDE8A1). The Raf-1/ERK/MEK pathway plays an important role in cell growth, prevention of apoptosis, cell cycle arrest and induction of drug resistance in cells [27]. The interactions between these molecules and pathway are complex and still under study in the laboratory [8,9]. An initial rule-based model was formalised in [2,3] using a population-based continuous-time Markov model. Our illustrative biological example follows the hypotheses put forward by biologists about the cAMP-degrading effect of PDE8A1 either bound or not an AKAP scaffold. Using PORGY as a modelling, simulation and analysis tool, we show that indeed the fully filled scaffold increases the output of the signalling more than the partially filled scaffold.

In the following we walk through the AKAP model to introduce the concepts of port graphs and port graph rewrite rules, and show how we use these structures to represent the molecular species and reaction rules. We then introduce the strategy language, give examples of strategy expressions for our AKAP model, and discuss some possible experiments and analysis tools available in PORGY.

3.1 Molecules as Port Graphs

Graphical notation is a simple communication tool to use by both (computational) biologists and computer scientists. A port graph is a graph where nodes have explicit connection points called *ports*; edges are attached to ports. Nodes, ports and edges are labelled by a set of attributes. For instance, a port may be associated with a state (e.g., active/inactive or principal/auxiliary) and a node may have properties such as colour, shape, etc. Attributes (see Note 4.1) may be used to define both the behaviour of the modelled system and for visualisation purposes. We represent molecular species as port graphs in a simple way: each molecule is represented by a node whose ports correspond to its binding or phosphorylation sites.

In PORGY, nodes, ports and edges between ports have sets of attributes attached to them, whose values may vary along the simulation. Standard attributes for all nodes are *Name* for identifying the type of species, as well as *Colour* and *Shape*, which have the same values for all nodes with the same *Name*. Additional attributes specific to the modelling can be defined.

In intracellular signal transduction pathways scaffolds are proteins exhibiting two main functions [26]: *anchors* for particular proteins in specific intracellular locations for receiving signals or transmitting them and *catalysts* for increasing the output of a signalling cascade or decreasing the response time for a faster output under certain circumstances. A-Kinase Anchoring Proteins (AKAP s) are a family of scaffolds proteins with the ability of binding the regulatory subunit of protein kinase A (PKA). Recently many computational models were studied in order to provide insight about how AKAP regulates signalling dynamics and cardiovascular pathophysiology [25]. Our AKAP scaffold models have three binding sites: one for the protein PKA, one for the enzyme Raf-1, and one for the enzyme PDE8A1, with the latter not always bound to AKAP. In this paper we are investigating two slightly different AKAP scaffold models: M_1, where all binding sites are filled, and M_2, where PDE8A1 is not bound.

The molecular species of our AKAP models are the following:

- scaffold protein AKAP with three binding sites, s_1, s_2, and s_3;
- nucleotide cAMP with one binding site s_1;
- protein PKA with one site s_1 bound to AKAP's site s_1 and one site s_2 for binding to cAMP's site s_1; we say that PKAis active when bound to cAMP;
- Raf-1 enzyme with two sites: the site s_1 bound to AKAP's site s_2 and the phosphorylation site s_2;
- enzyme PDE8A1 with one site s_1 for binding to the scaffold's site s_3 and one phosphorylation site s_2; we say that PDE8A1 is more active when the site s_2 is phosphorylated;
- activation signal SA –an artificially introduced entity whose role is explained later in the description of the reaction rules.

Fig. 2 illustrates the port graph representations of the molecular species in our AKAP model: a filled scaffold protein AKAP binding together PKA, Raf-1, and PDE8A1; a partially filled protein AKAP binding together only PKA and Raf-1; an unbound cAMP; a signal molecule; an unbound PDE8A1. In the graphical representation, we fill in the sites with black when bound to another site, with a grey shadow when activated, otherwise the sites are white-filled.

3.2 Model Behaviour and Reaction Rules

The AKAP scaffold binds together PKA, Raf-1, and sometimes PDE8A1. One of the biologists' hypothesis concerns the effect of PDE8A1 on degrading cAMP and, as a consequence, on the Raf-1 activation. In the following we detail the overall behaviour of our AKAP model.

If the concentration of cAMP rises above a given threshold, cAMP activates PKA by binding to it. Activated PKA catalyses the transfer of phosphates to the

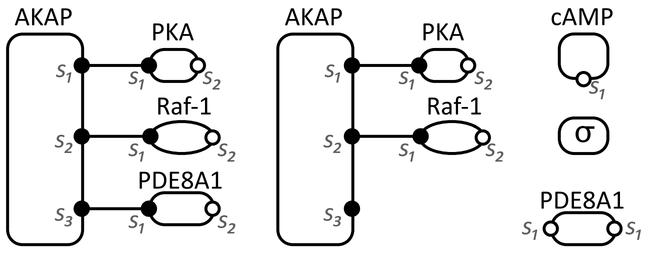


Fig. 2. Molecular species in the AKAP model.

phosphorylation site s_2 of Raf-1. Only when s_2 is dephosphorylated, the pathway Raf-1/MEK/ERK is activated and the signalling cascade begins; we represent this by the creation of a signal molecule σ. The catalytic function of PKA sometimes couples with the AKAP, by binding PKA together with phosphodiesterase PDE8A1 on the scaffold to form a complex that functions as a signal module. Under these conditions, as the cell is stimulated, cAMP activates PKA, and then PKA is responsible for the activation of PDE8A1 (by phosphorylation). PDE8A1 degrades cAMP, but if phosphorylated, PDE8A1 degrades more cAMP, hence rapidly reducing the amount of cAMP that can activate PKA. This leads to a feedback mechanism for downregulating PKA.

In order to analyse the effect of PDE8A1 on the cAMP degradation, we analyse two models:

- M_1 only with filled scaffolds (the three sites are bound to PKA, Raf-1, and PDE8A1) and no unbound PDE8A1 molecules,
- M_2 only with unfilled scaffolds (the binding site for PDE8A1 is free, the other two are bound) and unbound PDE8A1 molecules.

In the following, we detail the two sets of rules for each model, before describing their representation as port-graph rewrite rules.

Overview of the reaction rules in M_1. There are fourteen rules in this model: $R_1, \ldots, R_{14}$; we explain them in turn. Free cAMP activates PKA by binding to its free port (R_1). Active PKA catalyses the transfer of phosphates to the phosphorylation site s_2 of Raf-1 both when PDE8A1 is unphosphorylated (R_2) or when PDE8A1 is phosphorylated (R_3). Active PKA also acts to phosphorylate PDE8A1 and, as a consequence, to enhance PKA's activity in both cases when the site s_2 of Raf-1 is unphosphorylated (R_4) or phosphorylated (R_5) respectively. When cAMP is released as it unbinds from PKA, PKA becomes inactive and, consequently, Raf-1 and PDE8A1 are unphosphorylated, and a signal molecule SA is created to mark that the Raf-1/MEK/ERK is activated (see Fig. 3). Unphosphorylated PDE8A1 degrades cAMP when PKA is active and Raf-1 inactive (R_7), when PKA is active and Raf-1 unphosphorylated (R_8), when PKA is inactive and Raf-1 phosphorylated (R_9), and when PKA is active and Raf-1 phosphorylated (R_{10}). Phosphorylated PDE8A1 also degrades cAMP when PKA is inactive and

Raf-1 unphosphorylated (R_{11}), when PKA is active and Raf-1 unphosphorylated (R_{12}), when PKA is inactive and Raf-1 phosphorylated (R_{13}), and when PKA is active and Raf-1 phosphorylated (R_{14}).

Overview of the reaction rules in M_2. There are seven rules in this model. Free cAMP activates PKA by binding to its free port (R_1^u). Active PKA phosphorylates Raf-1 on site s_2 (R_2^u). Active PKA phosphorylates free (unbound) PDE8A1 in both cases when Raf-1 is not phosphorylated (R_3^u) or is phosphorylated (R_4^u). When cAMP is released as it unbinds from PKA, PKA becomes inactive and, consequently, Raf-1 is unphosphorylated, and a signal molecule SA is created to mark that the Raf-1/MEK/ERK is activated (R_5^u). Free cAMP is degraded by both unphosphorylated PDE8A1 (R_6^u) and phosphorylated PDE8A1 (R_7^u).

Reaction rates. We associate reaction rate constants (from r_1 to r_{14} for M_1 and from r_1^u to r_7^u for M_2) to each reaction. These reactions have mass-action kinetics. The existing experimental data suggest only approximate ratios of the reaction rates. We only have partial information on the ratio between some reaction rates, such as in M_1, PKA phosphorylates Raf-1 and PDE8A1 at the same rate and pPDE8A1 degrades approximately three times more cAMP than unphosphorylated PDE8A1. Additionally, for M_2, we know that PKA phosphorylates three times less PDE8A1 than Raf-1. If we consider r_1 to be 1.0 as a baseline, then we obtain the following values for the reaction rate constants: $r_1 = r_2 = r_3 = r_4 = r_5 = r_6 = r_{11} = r_{12} = r_{13} = r_{14} = 1.0$, $r_7 = r_8 = r_9 = r_{10} = \frac{1}{3}$, $r_1^u = r_2^u = r_5^u = r_7^u = 1.0$, $r_3^u = r_4^u = \frac{1}{3}$, $r_6^u = \frac{1}{9}$.

Port graph rewrite rules. We represent these reactions as port graph rewrite rules as explained below. First, recall that a *port graph rewrite rule* $L \Rightarrow R$ is a port graph consisting of two subgraphs L and R together with a node (called *arrow* node) that encodes the correspondence between the ports of L and the ports of R. L and R are called the *left-* and *right-*hand side respectively. For more details, see Note 4.3. A *port graph rewrite system* $\mathcal{R}$ is a finite set of port graph rewrite rules.

To illustrate the idea, we show the representation of the reaction R_6 as a port graph rewrite rule in Fig. 3. A phosphorylation action activates a site (or port) and we represent a phosphorylated site by a grey-filled port; a black-filled port corresponds to a bound site, while a white-filled port corresponds to a free (i.e., not bound) and inactive (i.e., not phosphorylated) site.

Figure 4 shows rules R_6 (left panel) and R_{10} (right panel) of model M_1 as they are visually represented in Porgy. The layout and shape of the graph are different from the ones used in Fig. 3. See Note 4.7 for more details on Porgy's layout algorithms and techniques.

Rule application is briefly described below and more completely in Note 4.3.

Let $L \Rightarrow R$ be a port graph rewrite rule and G a port graph such that there is an injective port graph morphism g from L to G. By replacing the subgraph

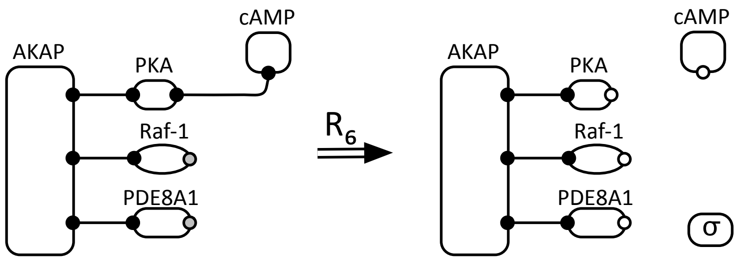


Fig. 3. Rule R_6 in the filled AKAP model M_1 (note we no longer label the sites' names in this graphical representation of the rewrite rules, but it is mandatory in PORGY).

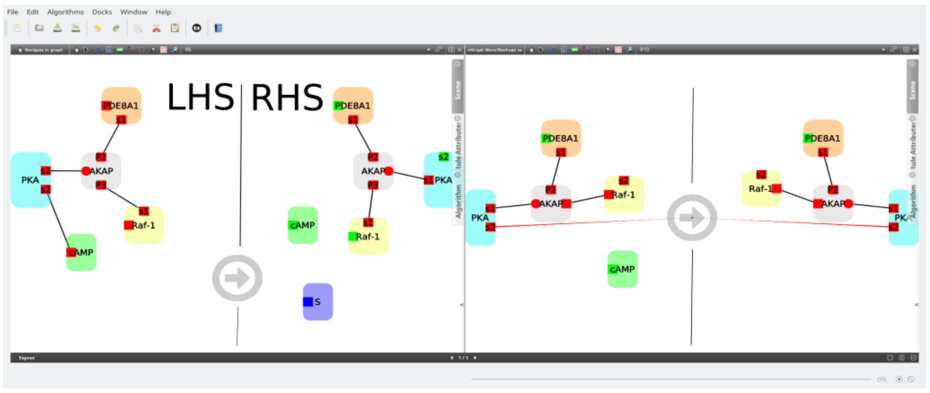


Fig. 4. Rules R_6 (left panel, also presented in Fig. 1) and R_{10} (right panel) of model M_1. A rule is a graph composed of a left-hand side (LHS) and a right-hand side (RHS), linked by an arrow node. A red edge connected to the arrow node from LHS to RHS indicates a direct correspondence between two ports. The port colour is used to show its state: a phosphorylated port is shown in red, while an unphosphorylated port is shown in green.

$g(L)$ of G by $g(R)$ and connecting it with the rest of the graph, we obtain a port graph G' representing a result of *one-step rewriting* of G using the rule $L \Rightarrow R$, written $G \rightarrow_{L\Rightarrow R} G'$. Several injective morphisms g from L to G may exist leading to possibly different rewriting results. These are built as solutions of a *matching* problem from L to a subgraph of G. If there is no such injective morphism, we say that G is *irreducible* with respect to $L \Rightarrow R$. Given a set $\mathcal{R}$ of rules, a port graph G *rewrites* to G', denoted by $G \rightarrow_{\mathcal{R}} G'$, if there is a port graph rewrite rule r in $\mathcal{R}$ such that $G \rightarrow_r G'$. This induces a transitive relation on port graphs. Each *rule application* is a rewriting step and a *derivation* is a sequence of rewriting steps, also called a computation. A port graph is in *normal form* if no rule can be applied on it. Rewriting is intrinsically non-deterministic since several subgraphs of a port graph may be rewritten under a set of rules.

PORGY provides a special matching construct Match that checks whether a rule matches the current graph and computes all solutions of a given matching problem. This construct is used in rule application. It is important to note that attributes play an important role in the matching process, since their values often determine whether a rule is applicable or not. By default, all attributes are taken into account in the matching process, but PORGY also offers the possibility to specify that an attribute is not used in matching. This feature gives a great flexibility in the design of rules due to the many combinatorial possibilities. It can also be used to optimise the matching process when some attributes are specified as non relevant.

3.3 Strategies for Rule Application

In PORGY, graph transformations can be defined by a single rule, or two or more rules composed in parallel or in a probabilistic manner. There are several ways of performing graph transformations (denoted by T below) in PORGY: the strategy language provides various constructs to specify how transformations should be applied; here we just illustrate how strategies are used to simulate the two models of the AKAP Scaffold Protein. The reader should refer to Note 4.4 for more information on the strategy language. To choose between several rules, the following operators are available:

one: `one`(T) computes only one of the possible applications of the transformation T and ignores the others; more precisely, it makes a choice between all the possible applications, with equal probabilities. PORGY also supports `all`(t) which computes all possible applications of the transformation T.

ppick: When probabilities $\pi_1, \ldots, \pi_n \in [0, 1]$ are associated to rules $T_1, \ldots, T_n$ such that $\pi_1 + \ldots + \pi_n = 1$, the strategy `ppick`$(T_1, \pi_1, \ldots, T_n, \pi_n)$ picks one of the rules for application, according to the given probabilities. More generally, this strategy can also take as inputs a list of rules $R_1, \ldots, R_n$ and a user-defined function *prob.py* that computes the respective probability for each rule to apply on the current graph: `ppick`$(R_1, R_2, \ldots, R_n,$ "*prob.py*"). The probabilities may be computed from the current system state instead of a fixed distribution. The function has to be written as a Python script (see Note 4.8). It is for instance possible to perform probabilistic rule application according to mass-action kinetics, as in Gillespie's stochastic simulation algorithm [24]. Previous versions of PORGY's strategy language offered only the first format (that is, the particular case where the function *prob* is a fixed distribution of probabilities that does not depend on the current system state).

Beyond the different choices for rule application, many other choices have to be made to control rewriting: choose where to apply a rule in a graph, define a sequence of rules which are correlated, iterate a rule or a sequence of rules, etc. For this, we can use the following constructs.

– id and fail are two atomic strategies that respectively denote success and failure.

– The expression S;S' represents sequential application of S followed by S' if S succeeds.
– `repeat`$(S)(n)$ simply iterates the application of S until it fails, but, if n is specified, then the number of repetitions cannot exceed n.
– `try`(S) behaves like the strategy S if S succeeds, but if S fails, it still returns id, thus never fails.

In the following we illustrate how to use the above strategy constructs in simulating the behaviour of the AKAP scaffold protein. Strategy 1 is the strategy used for model M_1. The function "ComputeProba.py" which computes the respective application probability π_i for each rule R_j to apply on a port graph G is computed as follows: $P_G(R_j) = \frac{m_j * r_j}{\sum_{i=1,..,14} m_i * r_i}$ where m_i (resp. m_j) is the number of rule R_i (resp. R_j) matches in the port graph G and r_i (resp. r_j) is the application rate of rule R_i (resp. R_j). The strategy for model M_2 is similar; one has just to replace the name of each rule and set the application rates in the Python script accordingly. See Note 4.8 for the associated Python code.

Strategy 1: Strategy of model M_1.

```
repeat(
  one(
    ppick($R_1$, $R_2$, $R_3$, $R_4$, $R_5$, $R_6$, $R_7$, $R_8$,
       $R_9$, $R_{10}$, $R_{11}$, $R_{12}$, $R_{13}$, $R_{14}$,"ComputeProba.py")
  )
)
```

Having a strategy language allows easily changing the control on rule application. Let us give two examples of this flexibility:

– Let us assume that we want to change the reaction rate constants. These rates are declared as constants in the Python script. One has just to change the corresponding value.
– A strategy is specially useful to express anteriority or priority between rules. For instance, if we want to express that the degradation process of the cAMP molecule occurs after other rules application, the previous strategy can be changed as described in Strategy 2.

3.4 Derivations and Derivation Tree

Once the representation of each species has been defined as above, it is easy to replicate as many port nodes as wanted and to draw the edges between them if needed.

For example, let us consider an initial port graph consisting of 30 unbound cAMP molecules; 6 structures built upon an AKAP protein binding an inactive PKA an unphosphorylated PDE8A1 and an unphosphorylated Raf-1 for model

Strategy 2: Updated strategy of model M_1 where the degradation process (rule R_6) occurs after the other rule applications.

```
repeat(
 one(
  ppick($R_1$, $R_2$, $R_3$, $R_4$, $R_5$, $R_7$, $R_8$,
    $R_9$, $R_{10}$, $R_{11}$, $R_{12}$, $R_{13}$, $R_{14}$,“ComputeProba.py”)
 );
 try(one($R_6$))
)
```

M_1 and 30 unbound cAMP molecules; 6 structures built upon an AKAP protein binding an inactive PKA and an unphosphorylated Raf-1 plus 3 unphosphorylated PDE8A1 proteins not bound to an AKAP scaffold protein for model M_2.

A *derivation*, or computation, is a sequence $G \rightarrow^*_{\mathcal{R}} G'$ of rewriting steps. Each rewriting step involves the application of a rule at a specific position in the graph. Considering a whole derivation and its successive steps provides access to the evolution history and is particularly useful for understanding and explaining how a specific state of the system has been reached. In general, several derivations are possible from any single state, giving rise to the notion of derivation tree, a data structure that represents all the different alternatives.

The derivation tree can be visualised and analysed in PORGY (we give more details in Section 3.5). Although it is in general a large data structure, it provides an organised, indexed representation of the evolution of the system, where each node in the derivation tree represents one system state. PORGY offers zooming mechanisms to analyse the tree and work with this structure at different levels; derivations can be visualised in different ways. One such zooming mechanism is the *small multiples* and we illustrate it in Fig. 5 for some steps of one execution of Strategy 1. This view allows to see the graph like a comics. Each thumbnail shows an overview of the graph or the morphism of the LHS/RHS of a rule. The layout of the graph is not changing, so it is easy to compare two states of the graph being rewritten. For instance, Fig. 6 shows a blue bordered node of interest (on the left of each thumbnail). The border is present until the node is modified by the application of a rewriting rule (border disappeared since G21).

3.5 Experimentation and analysis

PORGY allows users to interact and experiment with port graph rules in a visual and interactive way. It offers different views on each component of the rule system: the current graph being rewritten, the derivation tree, the rules and the strategy. The different components are shown for the AKAP example in Fig. 1.

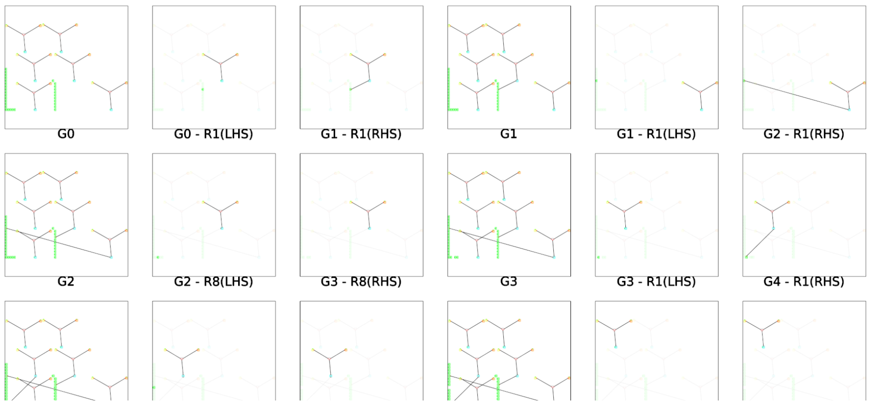


Fig. 5. Small multiples view of a derivation of Strategy 1. The first thumbnail (G_0) is an overview of the initial graph, the next one allows to view the LHS instance of rule r_1 in G_0, then the RHS instance of r_1 in the next graph G_1 and so on.

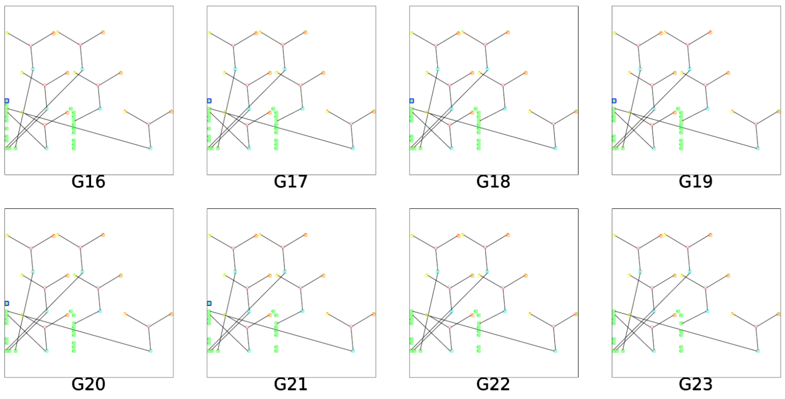


Fig. 6. Same small multiples views like in Fig. 5 (same graph, continuing) without LHS and RHS. The selected node in blue disappeared from G_{22} because it has been changed by the application of a rule.

In order to illustrate the capabilities of PORGY, we describe in this section different experiments performed on our running example. These experiments are calibrated in size in order to be able to draw understandable pictures. Indeed for real analysis, more and bigger experiments are necessary and we do not pretend giving any biological conclusion from these experiments. We just show some interesting capabilities of PORGY. PORGY has been designed with the Visual information-seeking mantra of Shneiderman [35] in mind: *Overview first, zoom and filter, then details on demand.*

Overview First. Because we use probabilistic rewriting, it may be relevant to execute several times the same rewrite program on the same input to look for potential variations. To launch a strategy, one has just to drag and drop the strategy to execute onto a node of the derivation tree. We show the results obtained for ten runs in Fig. 7 and 8. A black edge represents one application of a rule. A green edge represents one application of a strategy. Its extremities are respectively the starting and resulting nodes of the strategy. This structure is of course hardly readable, it has to be used as an overview and as such, it is a good starting point for an analysis. Fig. 5 and 6 were obtained from a branch of these trees. Instead of drawing a Small Multiples, PORGY is also able to animate the changes over a branch of the whole tree. We may compare the overall results of each branch but also the sequences of rules applied in a particular branch.

Zoom and Filter. Typically, the user may be interested in plotting the evolution of a parameter computed out of each intermediate state to filter and zoom on some interesting states. For example, in the AKAP model, the behaviour of the SA protein, as predicted by the biologists, can be examined by plotting the curve of the evolution of the number of SA protein throughout the rewriting process. The evolution of the rewriting process is here modelled by the depth of a branch of the derivation tree.

After selecting a branch inside the derivation tree, PORGY allows us to isolate this branch and compute the number of nodes of some given types. Thanks to TULIP, the scatter plot is dynamically built from the nodes of the derivation tree. A scatterplot can be built to visually compare for models M_1 and M_2 (resp. Fig. 9 and Fig. 10). We have selected one branch inside the derivation tree of each model with equivalent length (78 and 100). We can immediately see that both plots have a step shape. However, the evolution rate is very different. Moreover, all graphical views are synchronised. For instance, if some interesting points are selected inside the scatter plot, they are also immediately selected inside the corresponding branch of the derivation tree (Fig. 9). PORGY is also able to easily show where a given rule was used inside the derivation tree (See Fig. 11).

Details on Demand. Now, that some differences have been highlighted between both models, we can investigate further and see where the differences are more precisely. By zooming further on the derivation tree, one can, for instance, analyse the sequence of rules used to produce the selected graphs with model M_1 by just hovering the mouse pointer on a node (Fig. 12). See Note 4.6 to understand how this sequence of rules can be retrieved as a strategy ready to run. It is also possible to see which elements were changed by the application of a rule by hovering the mouse pointer over an edge (Fig. 13). The nodes changed in the rewriting step are emphasised in the picture, to clearly show the elements that have evolved inside the graph.

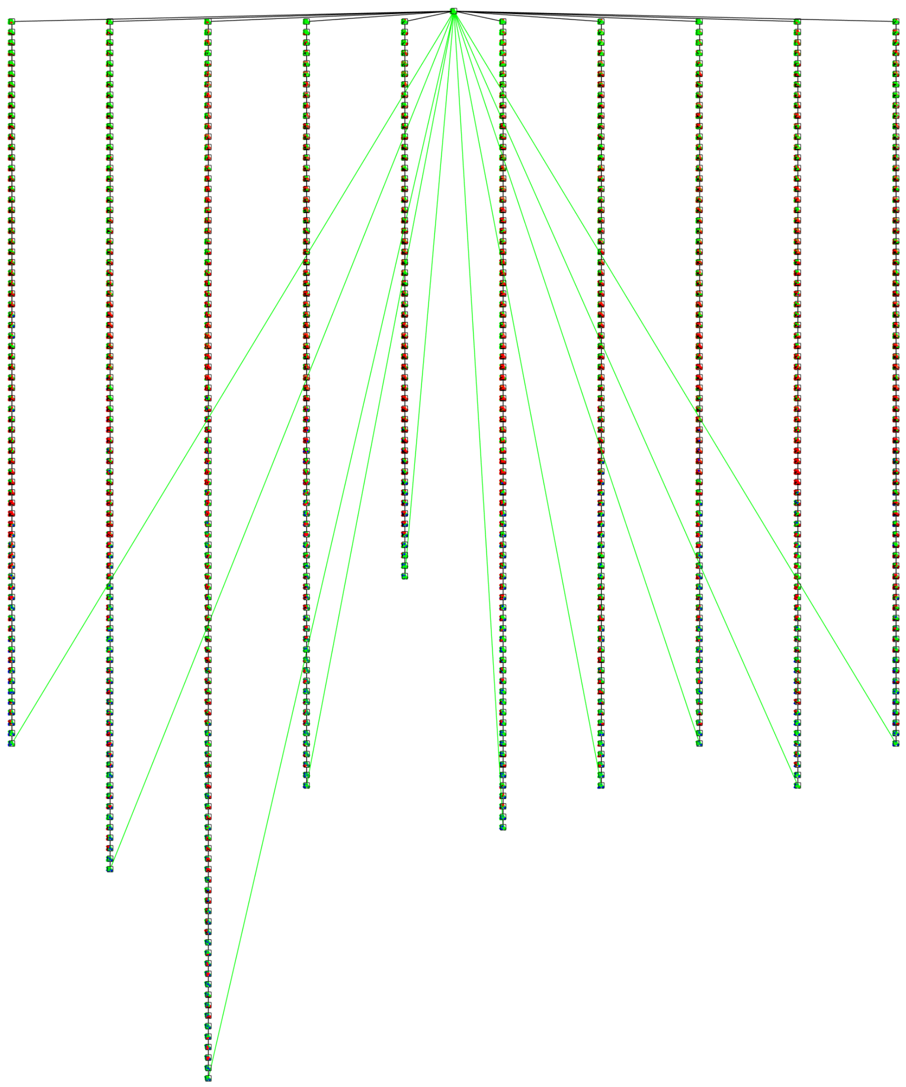


Fig. 7. The full derivation tree after running strategies for model M_1 10 times from the same graph. Note that the depth of each branch is not the same because of the probabilistic rewriting. The longest branch has 100 intermediate states between the starting graph and the final state where no rule can be applied any longer.

3.6 Concluding Remarks

We have illustrated via examples the use of Porgy as a tool for the development and analysis of rule-based models of biological systems. Since Porgy is a general purpose formal specification environment (based on strategic rewrite

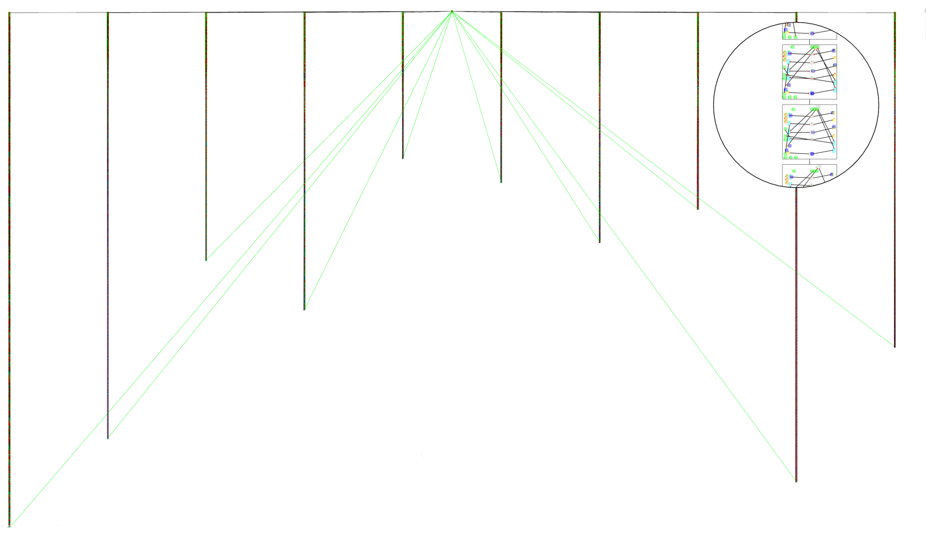


Fig. 8. The full derivation tree after running strategies for model M_2 10 times from the same graph including a close-up on a branch (built-in in PORGY). Note that the depth of each branch is not the same because of the probabilistic rewriting. The full tree is hardly readable because the longest branch has 343 intermediate states between the starting graph and the final state where no rule can be applied any longer.

programs) and has not been designed exclusively for biological modelling, some features specific to this application domain may be missing. However PORGY is an open source software and its architecture makes it easy to develop domain-specific instances of the general framework and to extend and refine the features presented here.

A main contribution of PORGY is its strategy language, partly demonstrated here. The full expressivity has not been illustrated here; in particular we have omitted some constructs to select positions in a graph. More information on the full strategy language is provided in [21], where the use of other strategy constructs is illustrated in the domain of social networks. The strategy language has evolved: it has been refined to take into account the specific needs of new application domains. The deliberate choice of separation between rules and control gives this flexibility of evolution. Many questions are still opened concerning strategies. How to compare and optimise them? How to synthesise them, i.e., how to find a sequence of rules leading to a certain port graph?

We have argued that visualisation techniques are important to guide intuition and design biochemical systems. However there remain also big challenges in this domain: for instance provide capability to easily change the display of molecules or to define new views.

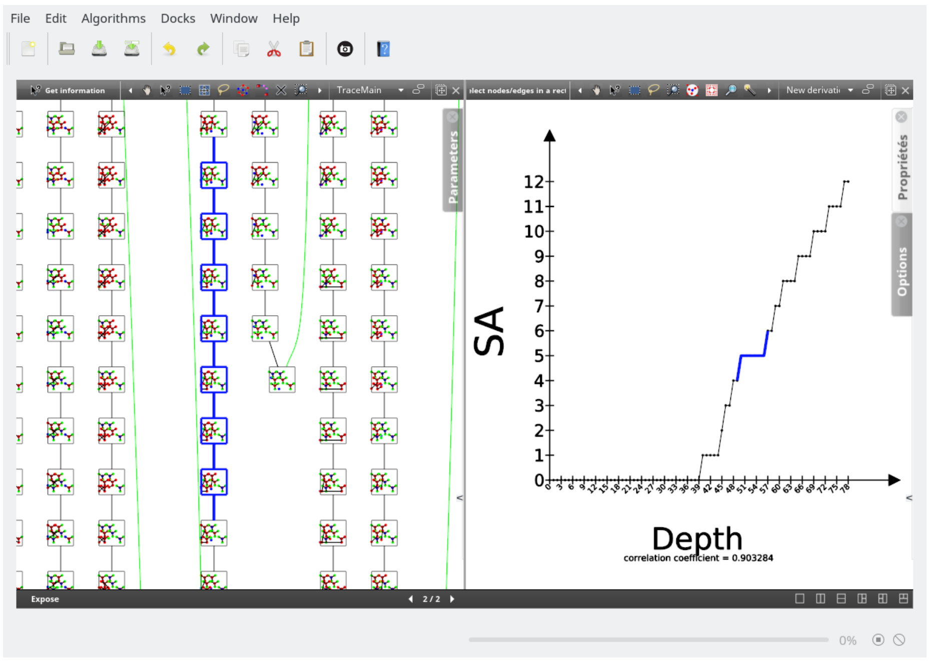


Fig. 9. A close-up on the derivation tree of Fig. 7 (left panel) and a scatterplot (right panel) which shows the evolution of the number of SA s for model M_1. A portion of interest of the scatterplot is selected (in blue) and this selection is automatically reported on the derivation tree. The Depth axis represents the number of rewriting rules applied to create the branch. It is the depth of the selected branch of the derivation tree.

4 Notes

4.1 Attributes

All attributes of nodes, ports and edges are represented in records over a given signature ∇ [20]. A record r is a set $\{(a_1, v_1), \dots, (a_n, v_n)\}$ of pairs, where each a_i occurs only once in r, and there is one pair where $a_i = Name$. The function $Atts$ applies to records and returns the labels of all the attributes: $Atts(r) = \{a_1, \dots, a_n\}$ if $r = \{(a_1, v_1), \dots, (a_n, v_n)\}$. As usual, $r.a_i$ denotes the value v_i of the attribute a_i in r.

The attribute $Name$ identifies the record in the following sense: For all r_1, r_2, $Atts(r_1) = Atts(r_2)$ if $r_1.Name = r_2.Name$.

4.2 Port graph

We recall the general definition of port graph; further examples can be found in [20].

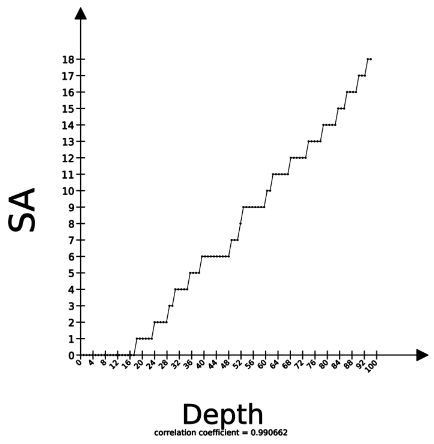


Fig. 10. A scatterplot built like the one of Fig 9 showing the evolution of the number of SA s for model M_2.

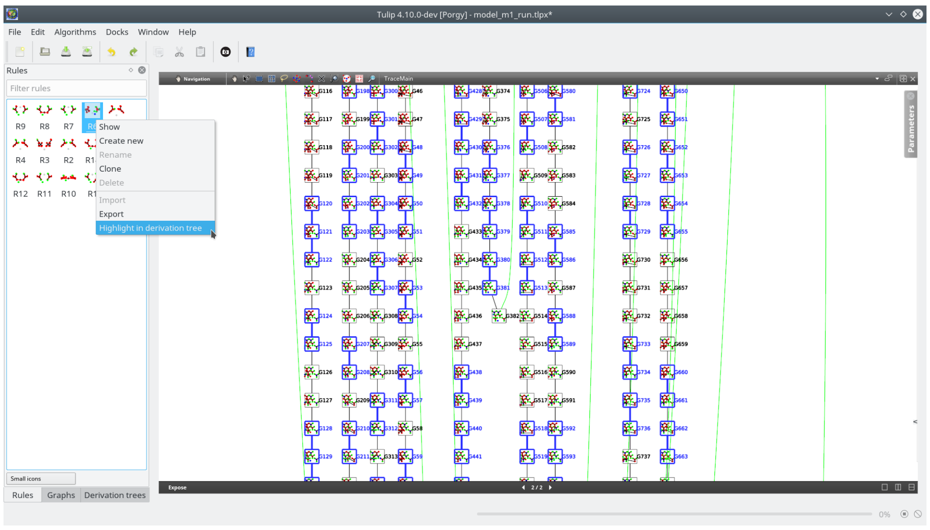
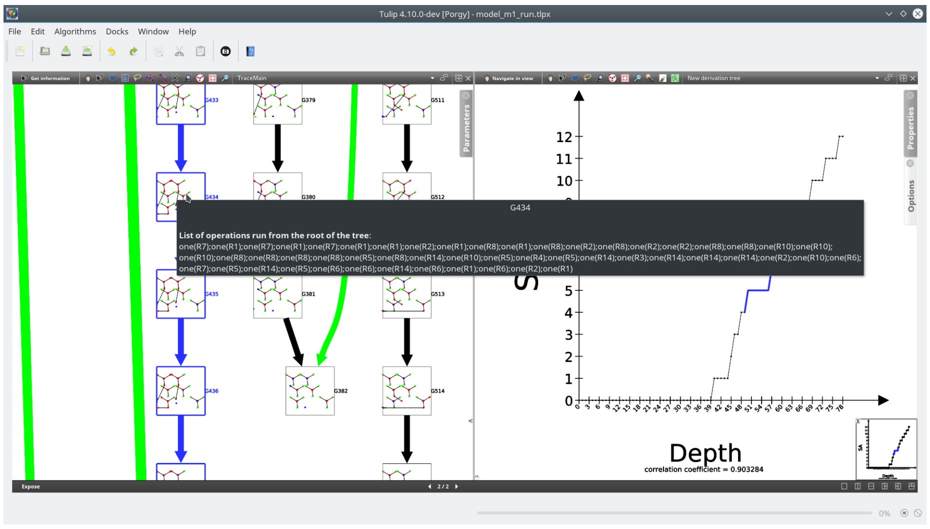

Tulip 4.10.0-dev [Porgy] - model_m1_run.tlpx*
File Edit Algorithms Docks Window Help
Rules
Filter rules
R9 R8 R7
R4 R3 R2
R12 R11 R10
Show
Create new
Rename
Clone
Delete
Import
Export
Highlight in derivation tree
Small icons
Rules Graphs Derivation trees
Navigation
TraceMain
Parameters
Expose

Fig. 11. The context menu associated with each rule displayed on the left panel allows to highlight this rule in the derivation tree (right panel).

Definition 1 (Port graph). *A* port graph *over a* signature ∇ *is a tuple* $G = (V, P, E, \mathcal{L})$ *where*

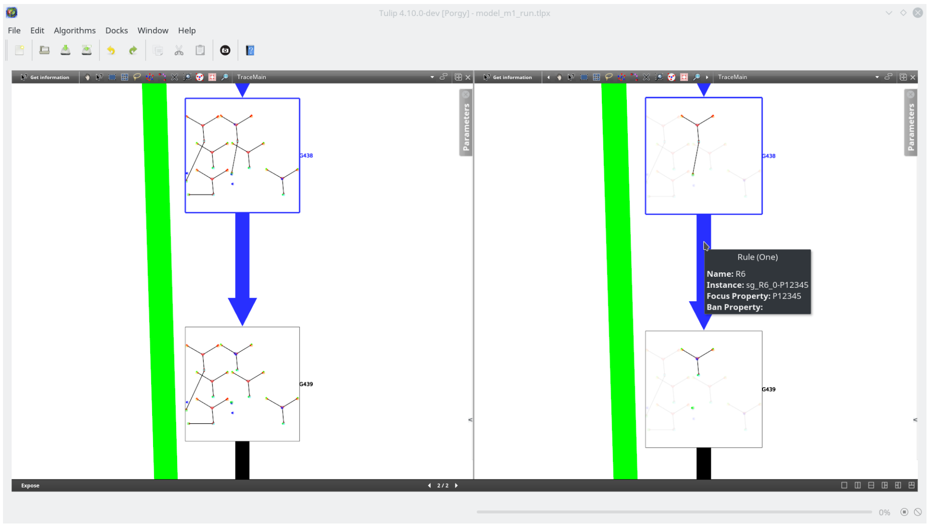
– $V \subseteq \mathcal{N}$ *is a finite set of nodes;* $n, n_1, \ldots$ *range over nodes.*
– $P \subseteq \mathcal{P}$ *is a finite set of ports;* $p, p_1, \ldots$ *range over ports.*
– $E \subseteq \mathcal{E}$ *is a finite set of edges between ports;* $e, e_1, \ldots$ *range over edges. Edges are undirected and two ports may be connected by more than one edge.*



G434

List of operations run from the root of the tree:
one(R7);one(R1);one(R7);one(R1);one(R7);one(R1);one(R1);one(R2);one(R1);one(R8);one(R1);one(R8);one(R2);one(R8);one(R2);one(R2);one(R8);one(R8);one(R10);one(R10);one(R10);one(R8);one(R8);one(R8);one(R8);one(R5);one(R8);one(R14);one(R10);one(R5);one(R4);one(R5);one(R14);one(R3);one(R14);one(R14);one(R14);one(R2);one(R10);one(R6);one(R7);one(R5);one(R14);one(R5);one(R6);one(R6);one(R14);one(R6);one(R1);one(R6);one(R2);one(R1)

Fig. 12. After selecting and zooming on some interesting nodes of the derivation tree (Fig. 9), hovering the mouse pointer over a node displays the list of rewriting operations used to obtain this state of the rewritten graph. See Note 4.6 for more information.


Tulip 4.10.0-dev [Porgy] - model_m1_run.tlpx
File Edit Algorithms Docks Window Help
TraceMain
G438
G439
Parameters
Expose
2 / 2

Rule (One)

Name: R6
Instance: sg_R6_0-P12345
Focus Property: P12345
Ban Property:

Fig. 13. Hovering the mouse pointer over an edge of the derivation tree allows seeing what parts of the graph were changed by the application of a rule. The rule (and some runtime information) are displayed as a tooltip.

– *$\mathcal{L}$ is a labelling function that returns, for each element in $V \cup P \cup E$, a record such that:*

- *for each edge $e \in E$, $\mathcal{L}(e)$ contains an attribute $Connect$ whose value is the pair $\{p_1, p_2\}$ of ports connected by e.*
- *for each port $p \in P$, $\mathcal{L}(p)$ contains an attribute $Attach$ whose value is the node n to which the port belongs, and an attribute $Arity$ whose value is the number of edges connected to this port.*
- *For each node $n \in V$, $\mathcal{L}(n)$ contains an attribute $Interface$ whose value is the set of names of ports in the node: $\{\mathcal{L}(p_i).Name \mid \mathcal{L}(p_i).Attach = n\}$. We assume that $\mathcal{L}$ satisfies the following constraint:*

$$\mathcal{L}(n_1).Name = \mathcal{L}(n_2).Name \Rightarrow \mathcal{L}(n_1).Interface = \mathcal{L}(n_2).Interface.$$

4.3 Rules and rewriting

In order to provide a better control for rule application, PORGY uses the concept of *located graph* G_P^Q that consists of a port graph G and two distinguished subgraphs P and Q of G, called respectively the *position subgraph*, or simply *position*, and the *banned subgraph*. In a located graph G_P^Q, P represents the subgraph of G where rewriting steps may take place (i.e., P is the focus of the rewriting) and Q represents the subgraph of G where rewriting steps are forbidden. The intuition is that subgraphs of G that overlap with P may be rewritten, if they are outside Q.

When applying a port graph rewrite rule, not only the underlying graph G but also the position and banned subgraphs may change. A *located rewrite rule* specifies two disjoint subgraphs M and M' of the right-hand side R that are respectively used to update the position and banned subgraphs. If M (resp. M') is not specified, R (resp. the empty graph $\emptyset$) is used as default. Precise definitions are given in [20].

4.4 Strategies

To control the application of the rules, a strategy language is presented in [20,21] where the grammar rules for strategy expressions is given.

A *strategic graph program*, consists of a *located graph* as defined above in Note 4.3, a set of rewriting rules, and a strategy expression. PORGY provides a strategy language to define those strategy expressions. In addition to the well-known constructs to select rewrite rules, the strategy language provides position primitives to select or ban specific positions in the graph for rewriting. The latter is useful to program graph traversals in a concise and natural way, and is a distinctive feature of the language.

A complete formal definition of strategic graph programs and their semantics can be found in [20]. Correctness and completeness of strategic port graph rewriting are stated and imply in particular that the derivation tree in which each rewrite step is performed according to the strategy –let us call it the *strategic derivation tree*– is actually a subtree of the derivation tree of the rewrite system without strategy. The strategic derivation tree is a valuable concept because it records the history of the transformations and provides access to generated models. It is, by itself, a source of challenging questions, such as detecting isomorphic models and folding the tree, finding equivalent paths and defining the "best ones", abstracting a sequence of steps by a composition strategy, or managing the complexity of the tree and its visualisation.

4.5 Downloading and installing PORGY

PORGY is built on top of the TULIP visualisation framework (`http://tulip.labri.fr`), as a set of TULIP plugins. PORGY is coded in C++11 and uses the Qt and Boost libraries. From the Boost library, we particularly use Spirit (see `http://boost-spirit.com/home/`) for the strategy language interpreter. A binary distribution of TULIP containing PORGY for MacOS (universal binary) and Windows (64-bit Windows only) machines and ready to compile source files can be downloaded from the PORGY page of the Tulip website (`http://porgy.labri.fr`). We also have automatic nightly builds for binaries and source files which are built from the latest development source trees of TULIP and PORGY.

4.6 Obtaining a strategy from the derivation tree

Fig. 12 illustrates how the list of operations done from the root node of the derivation tree to a given node can be retrieved. This is achieved by selecting the "Get Information" interactor from the interactor toolbar which is on top of each graphical view (look for the mouse pointer close to a question mark).

This list of operations can also be retrieved as a new strategy ready to run. Select a branch or a portion of a branch of the derivation tree, then from the "new strategy" menu, choose "New strategy from the derivation tree".

4.7 About the layout of the graphs in PORGY

Graph drawing is a research field in itself (see for instance the proceedings of the annual Graph Drawing & Network Visualization conference). The graphs used in different application domains usually look quite different. To produce a drawing that looks like the diagrams used in a particular domain, it is necessary to develop drawing algorithms that are specifically tuned for that domain. However, PORGY is a generic tool not linked to a particular application domain. This is why the current version of PORGY cannot deal with all drawing conventions and constraints. PORGY uses traditional and well known graph drawing algorithms available with TULIP, adapted to display port graphs. These algorithms are known to produce good drawings in reasonable time. However, thanks to the plugin mechanism of TULIP, new layout algorithms can be added easily and used inside PORGY.

4.8 TULIP and the embedded Python support

First introduced in TULIP 3.5, the TULIP framework now provides Python binding of all TULIP main features. It empowers users with easy scripting capabilities, facilitated by the property-based nature of TULIP. We used a common approach to bind C/C++ definitions with the SIP tool.[6]

The bindings are also publicly available from PyPI and can be independently installed from the TULIP framework as a standard Python package[7]. Users can then manipulate their graphs, create visualisation and export images completely independently from the TULIP perspectives and GUI previously mentioned.

In PORGY we use a feature which allows to call Python code directly from C++ code. The Python script path needs to be given as a parameter of the `ppick()` construct. The basename of the given file (i.e. filename without path information and extension) is used as the name of the function to call inside the Python script. The function must have 5 parameters which are in this order: the graph used to apply the rules on, a list of rules to test, the position subgraph and the banned subgraph. It must return a Python array (the C++ TULIP library does not support conversion from Python dictionary, this is planned for a future TULIP release) which has as elements the name of a rule followed by its application probability and so on for each rule. Reaction rates are given inside the Python script. Note that, all modifications made by the Python script are not kept.

The Python code used to compute the application probability for model M_1 is given in Listing 1. For model M_2, one has just to update the reaction rates and rule names accordingly. The script works by calling the "Check Rule" TULIP plugin for PORGY which computes the number of possible rule applications given a position subgraph and a banned subgraph. The plugin returns the number of application found. "Check Rule" is a core plugin of PORGY which is called every time a rule is tentatively applied. It computes all the morphisms of the rule LHS in a given graph.

For more flexibility, the reaction rates can be seen as rule parameters instead of being hardcoded in the Python script. In a future release of PORGY, we plan to support rule parameters directly stored in the TULIP graph describing each rule.

Acknowledgements

We thank Guy Melançon and Olivier Namet for their work in the initial PORGY project (2009–2012); their ideas and enthusiasm were invaluable during the early stages of development of this tool. We also thank Jason Vallet for implementing several features of PORGY and writing the documentation.

[6] Riverbank Computing Limited. SIP –A tool for automatically generating Python bindings for C and C++ libraries. `http://www.riverbankcomputing.co.uk/software/sip/`

[7] This package is available at `https://pypi.python.org/pypi/tulip-python` and can be installed using the command *pip install tulip-python*

References

1. Jakob L. Andersen, Christoph Flamm, Daniel Merkle, and Peter F. Stadler. Generic strategies for chemical space exploration. *I. J. Computational Biology and Drug Design*, 7(2/3):225–258, 2014.
2. Oana Andrei and Muffy Calder. A Model and Analysis of the AKAP Scaffold. *Electr. Notes Theor. Comput. Sci.*, 268:3–15, 2010.
3. Oana Andrei and Muffy Calder. Trend-Based Analysis of a Population Model of the AKAP Scaffold Protein. *Trans. Computational Systems Biology*, 14:1–25, 2012.
4. Oana Andrei, Maribel Fernández, Hélène Kirchner, Guy Melançon, Olivier Namet, and Bruno Pinaud. PORGY: Strategy-Driven Interactive Transformation of Graphs. In Rachid Echahed Echahed, editor, *Proc. of the 6th Int. Workshop on Computing with Terms and Graphs (TERMGRAPH 2011)*, volume 48, pages 54–68, 2011.
5. David Auber, Daniel Archambault, Romain Bourqui, Maylis Delest, Jonathan Dubois, Bruno Pinaud, Antoine Lambert, Patrick Mary, Morgan Mathiaut, and Guy Melancon. Tulip III. In *Encyclopedia of Social Network Analysis and Mining.* Springer-Verlag New York, 2014.
6. Daniel Balasubramanian, Anantha Narayanan, Christopher P. van Buskirk, and Gabor Karsai. The Graph Rewriting and Transformation Language: GReAT. *ECE-ASST*, 1, 2006.
7. Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. *ENTCS*, 15, 1998.
8. Kim M. Brown, Jon P. Day, Elaine Huston, Bastian Zimmermann, Kornelia Hampel, Frank Christian, David Romano, Selim Terhzaz, Louisa C. Y. Lee, Miranda J. Willis, David B. Morton, Joseph A. Beavo, Masami Shimizu-Albergine, Shireen A. Davies, Walter Kolch, Miles D. Houslay, and George S. Baillie. Phosphodiesterase-8A binds to and regulates Raf-1 kinase. *Proceedings of the National Academy of Sciences*, 110(16):E1533–E1542, 2013.
9. Kim M. Brown, Louisa C.Y. Lee, Jane E. Findlay, Jonathan P. Day, and George S. Baillie. Cyclic AMP-specific phosphodiesterase, PDE8A1, is activated by protein kinase A-mediated phosphorylation. *FEBS Letters*, 586(11):1631–1637, 2012.
10. Horst Bunke. Attributed programmed graph grammars and their application to schematic diagram interpretation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 4(6):574–582, 1982.
11. Lily A Chylek, Leonard A Harris, James R Faeder, and William S Hlavacek. Modeling for (physical) biologists: an introduction to the rule-based approach. *Physical Biology*, 12(4), 2015.
12. Joshua Colvin, Michael I Monine, James R Faeder, William S Hlavacek, Daniel D Von Hoff, and Richard G Posner. Simulation of large-scale rule-based models. *Bioinformatics*, 25(7):910–917, 04 2009.
13. Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation - part i: Basic concepts and double pushout approach. In *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, pages 163–246. World Scientific, 1997.
14. Vincent Danos, Jérôme Feret, Walter Fontana, Russell Harmer, and Jean Krivine. Rule-Based Modelling of Cellular Signalling. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *Proc. of CONCUR'07*, volume 4703 of *Lecture Notes in Computer Science*, pages 17–41. Springer, 2007.

15. Vincent Danos, Jérôme Feret, Walter Fontana, and Jean Krivine. Scalable Simulation of Cellular Signaling Networks. In Zhong Shao, editor, *Proc. of APLAS'07*, volume 4807 of *Lecture Notes in Computer Science*, pages 139–157. Springer, 2007.
16. Vincent Danos and Cosimo Laneve. Formal Molecular Biology. *Theoretical Computer Science*, 325(1):69–110, 2004.
17. Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1-3.* World Scientific, 1997.
18. Claudia Ermel, Michael Rudolf, and Gabriele Taentzer. The AGG approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages, and Tools*, pages 551–603. World Scientific, 1997.
19. James Faeder, Michael Blinov, and William Hlavacek. Rule-Based Modeling of Biochemical Systems with BioNetGen. In I. V. Maly, editor, *Systems Biology*, volume 500 of *Methods in Molecular Biology*, pages 113–167. Humana Press, 2009.
20. Maribel Fernández, , Hélène Kirchner, and Bruno Pinaud. Strategic Port Graph Rewriting: An Interactive Modelling and Analysis Framework. Research Report, Inria, January 2016.
21. Maribel Fernández, Hélène Kirchner, Bruno Pinaud, and Jason Vallet. Labelled Graph Rewriting Meets Social Networks. In Dorel Lucanu, editor, *Rewriting Logic and Its Applications, WRLA 2016*, volume 9942 of *LNCS*, page 25, Eindhoven, Netherlands, April 2016. Springer International Publishing Switzerland.
22. Maribel Fernández and Sébastien Maulat. Higher-order port-graph rewriting. In Sandra Alves and Ian Mackie, editors, *Proceedings 2nd International Workshop on Linearity, LINEARITY 2012, Tallinn, Estonia, 1 April 2012.*, volume 101 of *EPTCS*, pages 25–37, 2012.
23. Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam Szalkowski. GrGen: A Fast SPO-Based Graph Rewriting Tool. In *Proc. of ICGT*, volume 4178 of *LNCS*, pages 383–397. Springer, 2006.
24. Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.*, 81(25):2340–2361, 1977.
25. Eric C. Greenwald and Jeffrey J. Saucerman. Bigger, better, faster: principles and models of AKAP signaling. *J Cardiovasc Pharmacol.*, 58(5):462–469, 2012.
26. Jr. James E. Ferrell. What Do Scaffold Proteins Really Do? *Sci. STKE*, 2000(52):1–3, 2000.
27. James A. McCubrey, Linda S. Steelman, William H. Chappell, Stephen L. Abrams, Ellis W.T. Wong, Fumin Chang, Brian Lehmann, David M. Terrian, Michele Milella, Agostino Tafuri, Franca Stivala, Massimo Libra, Jorg Basecke, Camilla Evangelisti, Alberto M. Martelli, and Richard A. Franklin. Roles of the Raf/MEK/ERK pathway in cell growth, malignant transformation and drug resistance. *Biochimica et Biophysica Acta (BBA) - Molecular Cell Research*, 1773(8):1263 – 1284, 2007.
28. Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *ICSE*, pages 742–745, 2000.
29. John L. Pfaltz and Azriel Rosenfeld. Web Grammars. In *Proc. of the 1st Int. Joint Conf. on Artificial Intelligence, Washington, DC, May 1969*, pages 609–620, 1969.
30. Bruno Pinaud, Guy Melançon, and Jonathan Dubois. PORGY: A Visual Graph Rewriting Environment for Complex Systems. *Computer Graphics Forum*, 31(3):1265–1274, 2012.

31. Bruno Pinaud, Guy Melançon, and Jonathan Dubois. PORGY: A Visual Graph Rewriting Environment for Complex Systems. *Computer Graphics Forum*, 31(3):1265–1274, 2012.
32. Detlef Plump. The Graph Programming Language GP. In S. Bozapalidis and G. Rahonis, editors, *CAI*, volume 5725 of *LNCS*, pages 99–122. Springer, 2009.
33. Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. In *AGTIVE*, volume 3062 of *LNCS*, pages 479–485. Springer, 2003.
34. Andy Schürr, Andreas J. Winter, and Albert Zündorf. The PROGRES Approach: Language and Environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages, and Tools*, pages 479–546. World Scientific, 1997.
35. Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proc. of the IEEE Symp. on Visual Languages*, pages 336–343. IEEE Computer Society Press, 1996.
36. Adam M. Smith, Wen Xu, Yao Sun, James R. Faeder, and G.Elisabeta Marai. Rulebender: integrated modeling, simulation and visualization for rule-based intracellular biochemistry. *BMC Bioinformatics*, 13(8), 2012.
37. Jason Vallet, Hélène Kirchner, Bruno Pinaud, and Guy Melançon. A visual analytics approach to compare propagation models in social networks. In *Proceedings Graphs as Models, GaM 2015, London, UK, 11-12 April 2015.*, pages 65–79, 2015.
38. Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In *Proc. of RTA'01*, volume 2051 of *LNCS*, pages 357–361. Springer-Verlag, 2001.
39. John E.Jr. Wenskovitch, Leonard A. Harris, Jose-Juan Tapia, James R. Faeder, and G. Elisabeta Marai. Mosbie: a tool for comparison and analysis of rule-based biochemical models. *BMC Bioinformatics*, 15(1), 2014.

```
from tulip import *
def computeProba(graph, rules, position, ban):
  react={} #Reaction rates for $M_1$
  react["R1"] = float(1)
  react["R2"] = float(1)
  react["R3"] = float(1)
  react["R4"] = float(1)
  react["R5"] = float(1)
  react["R6"] = float(1)
  react["R7"] = float(1)/float(3)
  react["R8"] = float(1)/float(3)
  react["R9"] = float(1)/float(3)
  react["R10"] = float(1)/float(3)
  react["R11"] = float(1)
  react["R12"] = float(1)
  react["R13"] = float(1)
  react["R14"] = float(1)

  num={} #number of applications of each rule
  total=0.0 #Overall number of rule applications
  for g in rules: #Check if the rule can be applied
    params = tlp.getDefaultPluginParameters("Check Rule", graph)
    params["Rule Name"] = g
    params["Property for Position"]=graph.getBooleanProperty(position)
    params["Property for Ban"]=graph.getBooleanProperty(ban)
    graph.applyAlgorithm("Check Rule", params)
    #retrieve the number of possible rule applications
    number = params["number of instances"]
    num[g]=float(number)
    total += number * react[g]

#compute application probability for each rule
  proba={}
  for g in rules:
    if(total==0):
      proba[g]=0
    else:
      proba[g]=num[g]*react[g]/total;

  list_proba=[] #result array: a rule followed by its probability
  for i in proba:
    list_proba.append(i)
    list_proba.append(str(proba[i]))

  return list_proba
```

Listing 1: Python code used to compute the probabilities for model M_1. This code is called from the strategy (see Strategy 1) for every `ppick()` call.