Computing approximations for graph transformation systems

Vincent Danos
School of Informatics
University of Edinburgh
vdanos@inf.ed.ac.uk

Tobias Heindel
School of Informatics
University of Edinburgh
theindel@inf.ed.ac.uk

Ricardo Honorato-Zimmer
School of Informatics
University of Edinburgh
r.honorato@sms.ed.ac.uk

Sandro Stucki
sandro.stucki@epfl.ch
Programming Methods Laboratory, EPFL

We describe a tool that can compute a differential equation for the mean occurrence counts of a fixed graph observable in a given stochastic graph transformation system. It is an open problem whether the function that gives the mean occurrence count of the graph motif at a fixed time on the positive real line is approximable to arbitrary precision. However, the tool allows to express common practices to approximate the function using mean-field and refined approximation techniques. In the long term, we plan an extension to stochastic bisimulation checking for graph transformation systems.

1 Introduction

In the context of dynamic systems where states have complex structure, graph transformation [2, 5] is a natural candidate for a general formalism to describe all possible transitions of systems. Graph transformation is a particularly useful specification formalism if all possible transitions between system states fall into a finite number of transition classes such that each transition class has a common principle of a local structural modification that can be captured by a single *rule* of graph transformation.

Graph transformation systems in which rules are equipped with rates gives rise to a continuous-time Markov chain [8]. Intuitively, transitions between states have now a propensity to take place in an infinitesimal small time step; formally, they define the infinitesimal generator of a Markov chain which has isomorphism classes of finite graphs as states.

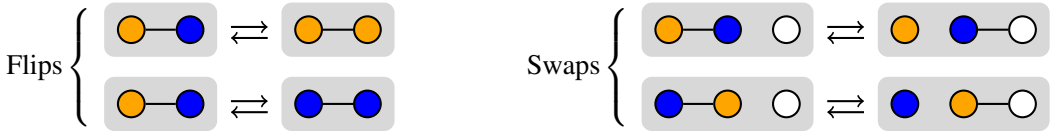
The central topic of the paper is the calculation of the mean occurrence count of certain graphs of interest in function of time in such a stochastically evolving graph. A classic example that comes to mind is the count of triangles in a random graph; for stochastic graph transformation systems one might thus ask how their expected numbers change over time. In general, given a stochastically evolving graph (specified by a stochastic graph transformation system), the question that we seek to answer is how many occurrences of a certain fixed graph, the *observable graph pattern*, can one expect to have at a given time in the future. It is an open problem whether this function can be computed to arbitrary precision in general. We present a tool that allows to compute (approximations to) this continuous-time behaviour by deriving an ordinary differential equation (ODE) for the rate of change of the mean occurrence count of any graph motif. The procedure has been informally described in [3] and we provide an abriged version in §3. It is a well-known problem to give guarantees for the quality of the approximations. "Analytical approximation approaches vary in their complexity, and there is usually an associated trade-off in accuracy (as measured, for example, by comparing the prediction of the theory with a large-scale Monte Carlo simulation of the dynamics)" [11, p. 18]. Roughly, the problem is that the function that counts the number of occurrences of a graph does not have any bound *a priori*; we can nevertheless derive

Submitted to:
MeMo 2015

© V. Danos, T. Heindel, R. Honorato-Zimmer & S. Stucki

fundamental equations for variances of counting functions and sometimes even get precise solutions. The tool has the form of a Scala library named *graph-rewriting* and is available for download from `https://github.com/rhz/graph-rewriting/`.

We use the voter model treated in Ref. [3] as running example and show how to obtain its results automatically by the tool. In this model, nodes can be in either of two states, 'red' or 'blue', and there are two types of rewriting rules, so-called 'flips' and 'swaps':



White nodes are to be thought colourless, i.e. they represent nodes that can be in any of the two states. White nodes represent nodes that can be in any of the two states. Typically node colours are interpreted as people's opinions and thus these rules attempt to minimise disparity among acquaintances. These rules show an interesting behaviour that has been studied in depth in Ref. [4].

One question Ref. [4] sets out to answer is what happens in the long run. Clearly, if the graph reaches a state where everyone has the same opinion, no further rule applications are possible. Also, if the graph disconnects (due to the application of a swap), as soon as every connected component becomes monochrome the system halts. What will be the number of red and blue nodes in these halted states in average? Of course, this will depend, among other factors, on the likelihood of each flip with respect to one another and the fraction of red and blue nodes at the start. Will the network split before one colour wins over the other? Since the only way to split the network is by applying a swap, the likelihood of these two rules with respect to the flips will be crucial. To answer these questions quantitatively, we equip our rules with rates and compute the derivative of the number of red (or blue) nodes over time. In the running example, the rates are k_{01}, k_{10} for the flips from red (0) to blue (1) and from blue (1) to red (0), respectively; and k_0, k_1 are the rates for the swaps.

2 Graph transformation

Rules are like productions of Chomsky grammars, according to the analogy from the first paper of algebraic graph transformation [7]; in particular they have a left- and right-hand side. The first obvious difference are the kinds of structures being rewritten, i.e. graphs vs. strings. However, one subtle but important difference that the analogy does sweep under the carpet is the fact that applications of graph transformation rules always involve a single occurrence of the left- and right-hand side. For each rule application we always know where and how the rule is applied. This is essential for the definition of the continuous-time stochastic semantics of graph transformation.

A rule of transformation for any general type of structure consists of a left-hand side L, a right-hand side R, and a pair of partial maps from nodes and edges of L to those of R such that they jointly preserve the structure of the left hand side. as in single pushout graph transformation [10]. The nodes and edges that have an image along these partial maps are preserved by the rule, although their label might change. Instead, the nodes and edges in the left-hand side that do not have an image are destroyed, while those in the right-hand side that do not have a pre-image are created by an application of the rule. Note that the application of a rule always is relative to an occurrence of the left-hand side L in a graph G, which is formalised by an injective (total) graph morphism from L to G.

2.1 Graphs

The concrete graphs that are handled by the tool are node- and edge-labelled directed multi-graphs (although the general method applies to several other graph-like structures). Directed multi-graphs definition is as usual. The labelling of nodes and edges is formalised by a partial function from nodes (resp. edges) to colours for each state of the system, thus equipping graphs with a partial labelling of nodes (resp. edges). Colourless nodes are then those that are not in the domain of definition of the labelling function. The ocurrence count of a graph G is written as $[G]$ and $G+H$ denotes the disjoint union of G and H.

In the `graph-rewriting` library, graphs are parametric in the type of nodes, edges, and labels they contain. In particular, the type of nodes and labels can be anything. Edges however can only be an instance of a class implemeting the `DiEdgeLike` interface. Any class that defines a `source` and `target` method can implement this interface. For multi-edges an `id` is needed to differentiate between edges that have the same `source` and `target`. The library comes with a default class for multi-edges, `IdDiEdge`, and graphs must be comprised of edges of this type to be fed to the ODE generation algorithm.

The example introduced in the previous section uses undirected graphs instead of directed graphs. To encode the model, we could split each rule into two versions, one for each direction (if the rules had more than one edge it would be a combinatorial expansion though!). Here we take a simpler approach instead: edges always go from red to blue.

A multi-edge can be created by using `IdDiEdge`'s constructor, e.g. the expression `IdDiEdge(0,"u","v")` will create an edge from node "u" to "v" with an `id` of type `Int` and value `0`. Also, an edge can be created by using `~~>`, as in `"u"~~>"v"`. This will implicitly define an `id` for that edge by means of a global counter that will not generate the same `id` twice but could produce an `id` that has been already used by the user.

Graphs are mutable and so can be constructed progressively by adding nodes, edges and labels to them. When a new graph is instantiated, an initial set of nodes, edges and labels can be defined. For example, `Graph("u"->"l1","v"->"l2")("u"~~>"v")` will create a graph with two nodes "u" and "v" with labels "l1" and "l2" respectively and an unlabelled edge from "u" to "v". More specifically, the `Graph` constructor takes as a first set of parameters any number of nodes with an optional label each, indicated by an arrow (`->`). Then it takes a second set of parameters for edges and their optional labels in the same fashion. Note that some nodes and edges can be left unlabelled while others are labelled. For instance, the left-hand side of the first swap rule may be constructed by the following expression: `Graph("u"->"red","v"->"blue","w")("u"~~>"v")`.

Whenever we want to create a Graph that does not have edges or nodes or labels, we must specify a type for them. Otherwise Scala's type inference will assign them a `Nothing` type. There are 4 type parameters in total for `Graph`'s constructor: the node type, node label type, edge type, and edge label type. The first two are specified before the first set of parameters and the other two right before the second set of parameters, as in `Graph[String,String]("u")[IdDiEdge[Int, String],String]()`. Repeating the type parameters again every time we would like to instantiate a `Graph` can rapidly become unwieldy and in models usually all `Graph`s will have the same type signature. For this reason the library let us create custom constructors with specific type parameters defined beforehand by using `Graph`'s `withType` method, e.g.

```
val G = Graph.withType[String,String,IdDiEdge[Int,String],String]
val oneNodeGraph = G("u")()
```

2.2 Rules

As mentioned earlier, rules are described by two graphs (L and R) and a pair of partial maps between them for nodes and edges. The library uses Scala's `Map` to define the latter. Additionally, rules are equipped with a `Rate`, which are comprised of a name and a value and constructed as `Rate(‘‘rate name’’, rateValue)` with `rateValue` a `Double`, or implicitly from a `String`, as in the following example (with an implicit value of `1.0`). Consider the first swap rule in our example. One way to construct this rule would be:

```
val lhs = Graph("u"->"red","v"->"blue","w")("u"~~>"v")
val rhs = Graph("u"->"red","v"->"blue","w")("v"~~>"w")
val swap0 = Rule(lhs, rhs, Map("u"->"u","v"->"v","w"->"w"),
                 Map(), "k0")
```

Where `"k0"` is the name for the rate. In the generated ODEs it will appear by this name instead of its numerical value. This allows an easier interpretation of where the terms of each ODE come from. In the definition of this `Rule` all nodes are preserved while no edge is, since the partial map for edges is empty. Note that, when mapping edges, attention has to be paid to the `ids` of edges, not just their `source` and `target`. In our example, the flip rule preserves the edge in its left-hand side and can be constructed using the following code:

```
val e = "u"~~>"v"
val lhs = Graph("u"->"red","v"->"blue")(e)
val rhs = Graph("u"->"red","v"->"red")(e)
val flip0 = Rule(lhs, rhs, Map("u"->"u","v"->"v"),
                 Map(e->e), "k01")
```

The complete code needed to define all rules can be found in Appendix A.

3 ODE generation

The ODE generation algorith depends heavily on a graph construction called *minimal glueings* in Ref. [3] and related to local co-products. This construction allows us to characterise all pair of matches of any two graphs onto a common target graph into a unique family among finitely many. Each family represents a way in which the two graphs can overlap. Intuitively, we use it to enumerate all possible ways in which the application of a rule could create or destroy instances of the desired observables. It is worth noting that, although finite, there may be exponentially many minimal glueings (i.e. families) for a pair of graphs!

Call $m(A,B)$ the set of minimal glueings of A and B. The algorithm proceeds as follows, given an observable G: 1) for each rule left-hand side L, compute $m(G,L)$; 2) for each rule right-hand side R, compute $m(G,R)$ and apply the inverse rule to each element of $m(G,R)$ – call this set $m'(G,R)$; 3) generate an ODE for $[G]$ of the form:

$$\frac{d}{dt}[G] = \sum_{(L \rightharpoonup R,k)\in\mathscr{R}} k \left(\sum_{H\in m'(G,R)} [H] - \sum_{F\in m(G,L)} [F] \right)$$

4) We repeat the procedure for the new observables $[H]$ and $[F]$.

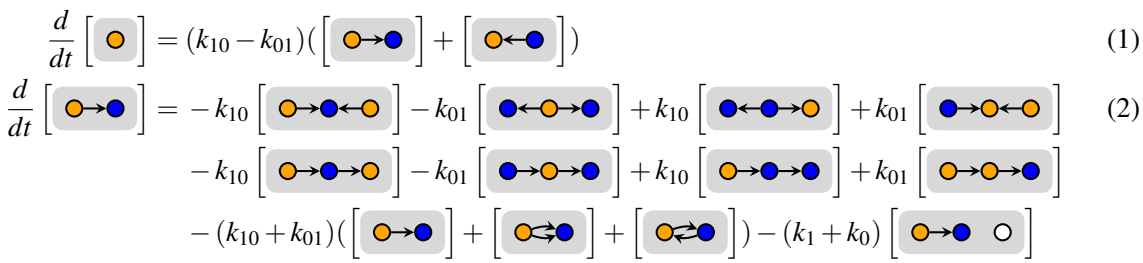
3.1 Equations

We have just seen how to build flip and swap rules and so we are ready to use our algorithm to obtain a system of differential equations that can describe the average count of an observable in time. To do this we use the `generateMeanODEs` method in `moments`. This method accepts 3 mandatory parameters, namely, the maximum number of ODEs to be discovered, a list of rules and a list of observables. Optionally we can provide so-called 'transformers' which are explained in the next section. As output, this method returns a list of equations. We can use `ODEPrinter` to print them to the standard output or to save them in an Octave script that will integrate the system of differential equations.

To answer one of our initial questions about the model – namely how many red (or blue) nodes there are in the halted state – let us consider the following simple observable: a single red node.

```
val redNode = G("u" -> "red")()
val equations = meanfield.mfa(2,
  List(flip0,flip1,swap0r,swap0b,swap1r,swap1b), List(redNode))
```

Here we use the custom `Graph` constructor defined above, `G`. To print these `equations`, we use `ODEPrinter(equations).print`. The output, rendered graphically, is:



If we were to ask for the ODEs of these observables, we would get bigger observables and from the ODEs of these even bigger observables in a process that would never terminate. However, we are interested in obtaining a system of differential equations that is closed. To this end, we have to make some approximations which can be integrated algorithmically by using 'transformers'.

The reader might wonder why the second ODE listed above has a term for the graph with a red and a blue node with two edges between them. This comes from the fact that whenever we apply a flip to a pair of nodes that have more than one edge between them, more than one instance of the red-to-blue pattern is deleted *at once*. It is indeed sufficient to look at each pair of multi-edges to account for all those deletions.

3.2 Transformers: approximations and simplifications

A transformer is any function defined on `Graphs` that returns a graph monomial or nothing (i.e. an `Option[Mn[N,NL,E,EL]]` with `N`, `NL`, `E`, and `EL` the `Graph`'s type parameters). A transformer is applied whenever a new observable is discovered by the ODE generation mechanism. If it returns a graph monomial, an algebraic (instead of differential) equation is generated for that observable, equating the observable to the returned monomial. Here is where the user is given the freedom to equate expressions that are not actually equal and thus introduce approximations. It is worth noting that in the abscence of transformers, all ODEs discovered by the algorithm are exact. If the transformer returns nothing, then the algorithm will compute an ODE for the observable (as long as the maximum number of ODEs to be generated hasn't been reached).

The first transformer that we introduce in our example will help us to break down disconnected graph observables into their constituent connected components, keeping the size of some graph observables bounded. Formally, this amounts to say that the expected value of the ocurrence counts of a disconnected graph observable (given the time-dependent probability distribution p on the the state space), $\mathrm{E}_p([G])$ with $G = A + B + \ldots + C$, is approximately $\mathrm{E}_p([A]) \cdot \mathrm{E}_p([B]) \cdot \ldots \cdot \mathrm{E}_p([C])$. This approximation is akin to that used in the theory of Petri nets for computing rate equations, e.g. that $\frac{d}{dt}\mathrm{E}_p([A]) = \mathrm{E}_p([2A]) \simeq \mathrm{E}_p([A])^2$ in the system comprised of the single reaction $2A \to 3A$. It is based on the assumption of independence between observables. In the following code snippets, we assume type `N` is `String`, `E` is `IdDiEdge[Int,N]`, and `NL`, `EL` are any types.

```
def splitConnectedComponents(g: Graph[N,NL,E,EL]): Option[Mn[N,NL,E,EL]] =
  if (g.isConnected) None else Some(Mn(g.components))
```

This is indeed a general transformation and thus it is supplied as part of the library. This transformer will return `None` when the graph is connected, meaning that connected graph observables will not be approximated by it.

The second approximation introduced is the so-called "pair approximation", which in the most general case entails approximating a mean ocurrence count of a graph G as the product of two overlapping subgraphs G_1, G_2 such that $G_1 \cup G_2 = G$, divided by their intersection I, i.e. $\mathrm{E}_p([G]) \simeq \mathrm{E}_p([G_1])\mathrm{E}_p([G_2])/\mathrm{E}_p([I])$. This approximation assumes conditional independence of the overlapping subgraphs.

```
def pairApprox(g: Graph[N,NL,E,EL]): Option[Mn[N,NL,E,EL]] =
  if (g.nodes.size == 3 && g.edges.size == 2 && g.isConnected) {
    val List(e1, e2) = g.edges.toList
    val intersection = e1.nodes &~ (e1.nodes &~ e2.nodes)
    Mn(g.inducedSubgraph(e1.nodes)) *
       g.inducedSubgraph(e2.nodes) /
       g.inducedSubgraph(intersection)
  } else None
```

Where `&~` is Scala's set difference operator. The code presented here performs a "pair approximation" on graphs with 3 nodes and 2 edges like the ones obtained in Eq. 2, splitting them into two graphs with 2 nodes and 1 edge each, and the intersection being the node in the middle.

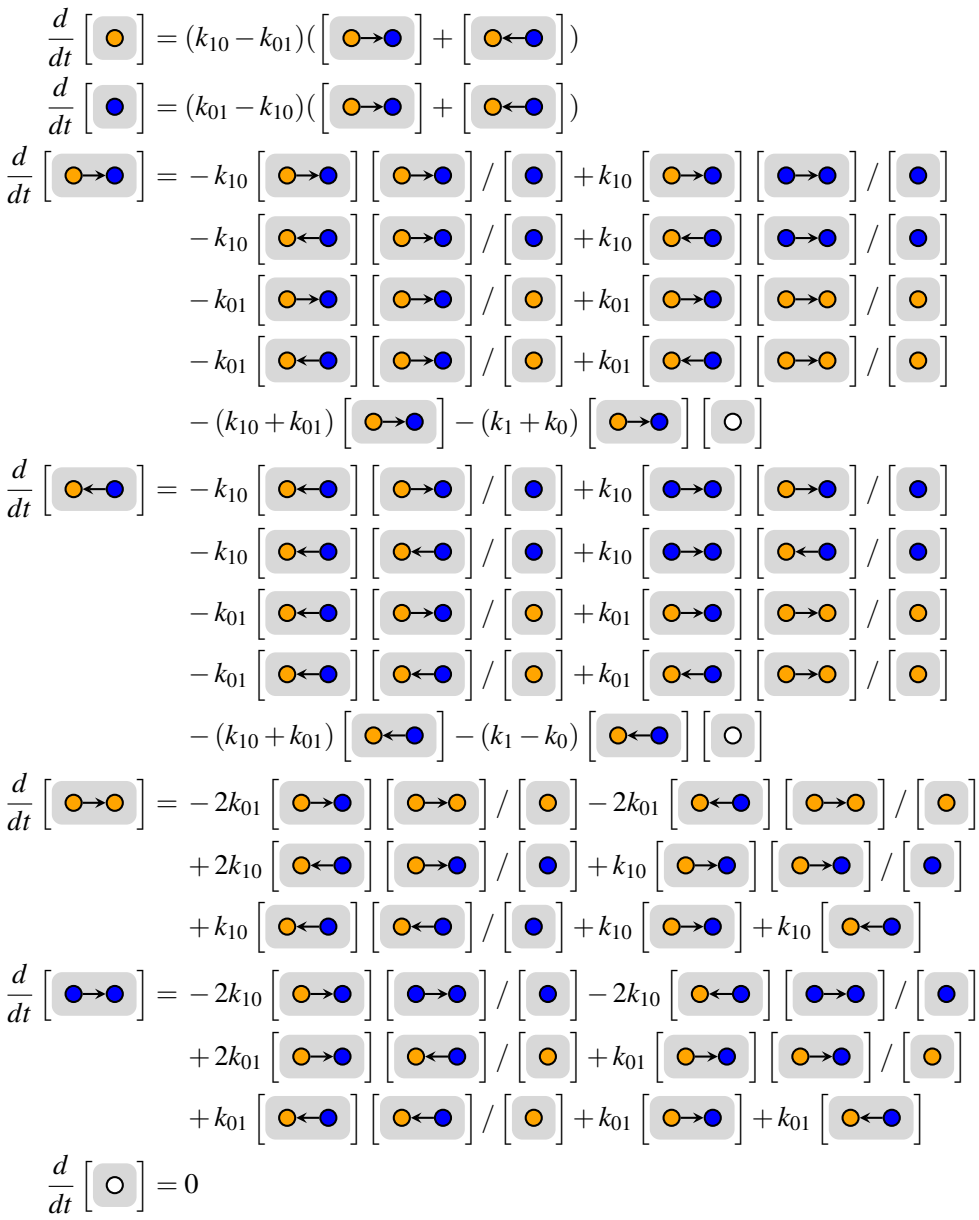
Finally, the only observables left that blow up the expansion are the terms with multi-edges between two nodes. However, if the initial state is a sparse graph, a multi-edge is highly unlikely to ever occur. Hence we approximate this observable's average ocurrence count as zero.

```
def noParallelEdges(g: Graph[N,NL,E,EL]): Option[Mn[N,NL,E,EL]] =
  if (g.nodes.size == 2 && g.edges.size == 2) Some(Mn.zero)
  else None
```

4 Example results

In this section we show the results obtained from running the model in our example as presented in Appendix A. Please note that we have manually translated the textual output notation given by the tool

into the graphical notation used throughout this paper.



Note that the algorithm discovered that the number of nodes with any colour is invariant. By generating an Octave script using `saveAsOctave` method in `ODEPrinter`, the system of differential equations can be integrated numerically. In particular, the solution to the set of ODEs above has been computed and is shown in Fig. 1.

4.1 Variance and other moments

In addition to the mean number of red nodes ($\mathrm{E}_p([0])$), we can compute what the variance of this number is by using the formula $\mathrm{V}([0]) = \mathrm{E}_p([0]^2) - \mathrm{E}_p([0])^2$. Minimal glueings let us express $[0]^2$ as a sum of

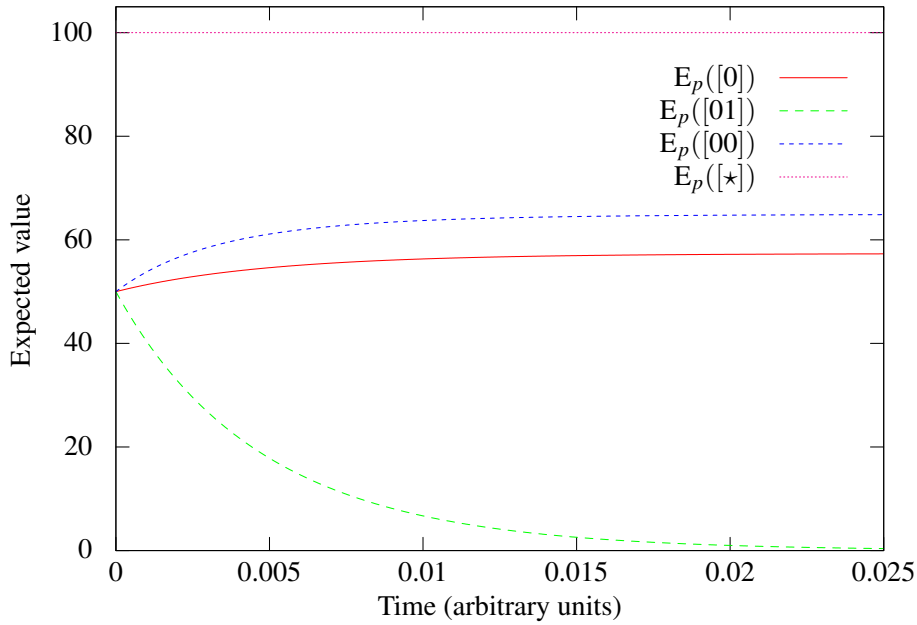


Figure 1: Solution to the system of ordinary differential equations given the following initial conditions: $[0] = 50, [1] = 50, [01] = 50, [10] = 50, [00] = 50, [11] = 50$, and $[\star] = 100$ with $\star$ the graph consisting of a single colourless node and rates $k_{10} = 15$, $k_{01} = 0.1$, $k_0 = 1$, and $k_1 = 1$.

observables: $[0]^2 = [0+0] + [0]$. For this reason, we compute an ODE for $E_p([0+0])$ without using any transformers (`splitConnectedComponents` would transform $[0+0]$ back into $[0]^2$ and the other two are irrelevant in this case).

$$\frac{d}{dt}\Big[\circ\ \circ\Big] = 2(k_{10} - k_{01})\Big(\Big[\circ\ \circ\!\!\rightarrow\!\!\bullet\Big] + \Big[\circ\ \circ\!\!\leftarrow\!\!\bullet\Big]\Big) + 2k_{10}\Big(\Big[\circ\!\!\rightarrow\!\!\bullet\Big] + \Big[\circ\!\!\leftarrow\!\!\bullet\Big]\Big) \tag{3}$$

Formulas exist that relate the value of any moment to lower moments. Hence these higher moments can be computed in a similar fashion to the one shown here.

5 Related and future work

A complete list of related work is beyond the scope of the paper: We expect a large number of concrete models that have been studied extensively in application-specific, *ad hoc* terminology to be captured equally well by graph transformation. We refer the reader to Ref. [11] for a recent overview on dynamical systems on networks that are of particular interest to the authors and gives references for a relevant part of the immense body of literature; in particular, the type of approximations that we expect to be used as transformers in our tool are related to those listed under the heading of Mean-Field Theories, Pair Approximations, and Higher-Order Approximations in the latter work.

A modelling formalism that is closely related to the algebraic approach to graph transformation are process calculi. Before we sketch possible future research on the connection to this field, especially recent work on stochastic process calculi such as [1], we point out some important differences. The

terminology of *observable* is inspired by (classical) physics, chemistry and biology. In particular, graph observables are *a priori* unrelated to the notion of observable actions from the process calculi literature and there is no direct link to observational equivalences; also, the relation to observability in control theory is unclear at best. Similarly, the question of whether it is suitable to consider a certain graph as observable cannot be answered from the set of rules and the initial state alone; it is necessary to have additional information from the specific application at hand. In examples from biology, such as protein translation as modelled by the TASEP (see Ref. [13] for a tutorial), a graph would be suitable for observation if the evolution of its mean occurrence counts would have a corresponding set of experiments that allow to falsify or validate the model.

For the possible relations of the present paper to the process calculus literature that do exist, note first that rule applications of graph transformation correspond to single step reductions in process calculi. The ground-breaking observation of Leifer and Milner [9] was that one can automatically derive labelled transition systems from reduction semantics (for process calculi). This idea has been successfully transferred to graph transformation [6]. We plan to use techniques from Ref. [6] to canonically associate labelled Markov processes to graph transformation systems. We expect that analogues to the semantics of CCS as labelled Markov processes given in [1] can be derived without complications; moreover, the resulting labelled Markov processes might be meaningful for complex networks modelled by graph transformation (while we would be rather surprised if such interactive semantics have applications in biology or chemistry). In a slightly different direction and as a long term goal, we want to explore whether the data structures and operations on graphs that are implemented in the tool can be re-used for checking (approximate) stochastic bisimulation of graph transformation systems.

References

[1] Luca Cardelli & Radu Mardare (2014): *The Measurable Space of Stochastic Processes*. *Fundam. Inform.* 131(3-4), pp. 351–371, doi:10.3233/FI-2014-1019. Available at `http://dx.doi.org/10.3233/FI-2014-1019`.

[2] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel & Michael Löwe (1997): *Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach*. In Rozenberg [12], pp. 163–246.

[3] Vincent Danos, Tobias Heindel, Ricardo Honorato-Zimmer & Sandro Stucki (2014): *Approximations for Stochastic Graph Rewriting*. In: *ICFEM*, pp. 1–10, doi:10.1007/978-3-319-11737-9_1. Available at `http://dx.doi.org/10.1007/978-3-319-11737-9_1`.

[4] Richard Durrett, James P Gleeson, Alun L Lloyd, Peter J Mucha, Feng Shi, David Sivakoff, Joshua ES Socolar & Chris Varghese (2012): *Graph fission in an evolving voter model*. *Proceedings of the National Academy of Sciences* 109(10), pp. 3682–3687.

[5] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner & Andrea Corradini (1997): *Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with Double Pushout Approach*. In Rozenberg [12], pp. 247–312.

[6] Hartmut Ehrig & Barbara König (2006): *Deriving bisimulation congruences in the DPO approach to graph rewriting with borrowed contexts*. *Mathematical Structures in Computer Science* 16(6), pp. 1133–1163, doi:10.1017/S096012950600569X. Available at `http://dx.doi.org/10.1017/S096012950600569X`.

[7] Hartmut Ehrig, Michael Pfender & Hans Jürgen Schneider (1973): *Graph-Grammars: An Algebraic Approach*. In: *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, IEEE Computer Society, pp. 167–180, doi:10.1109/SWAT.1973.11. Available at `http://dx.doi.org/10.1109/SWAT.1973.11`.

[8] Reiko Heckel, Georgios Lajios & Sebastian Menge (2006): *Stochastic Graph Transformation Systems*. *Fundamenta Informaticae* 74(1), pp. 63–84. Available at http://iospress.metapress.com/content/c7ha18g96nbm7g2e/.

[9] James J. Leifer & Robin Milner (2000): *Deriving Bisimulation Congruences for Reactive Systems*. In Catuscia Palamidessi, editor: *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings*, *Lecture Notes in Computer Science* 1877, Springer, pp. 243–258, doi:10.1007/3-540-44618-4_19. Available at http://dx.doi.org/10.1007/3-540-44618-4_19.

[10] Michael Löwe (1993): *Algebraic Approach to Single-Pushout Graph Transformation*. *Theor. Comput. Sci.* 109(1&2), pp. 181–224, doi:10.1016/0304-3975(93)90068-5. Available at http://dx.doi.org/10.1016/0304-3975(93)90068-5.

[11] Mason A. Porter & James P. Gleeson (2014): *Dynamical Systems on Networks: A Tutorial*. *CoRR* abs/1403.7663. Available at http://arxiv.org/abs/1403.7663.

[12] Grzegorz Rozenberg, editor (1997): *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific.

[13] R.K.P. Zia, J.J. Dong & B. Schmittmann (2011): *Modeling Translation in Protein Synthesis with TASEP: A Tutorial and Recent Developments*. *Journal of Statistical Physics* 144(2), pp. 405–428, doi:10.1007/s10955-011-0183-1. Available at http://dx.doi.org/10.1007/s10955-011-0183-1.

A Voter model

```
import graph_rewriting._
import implicits._
import moments._ // this imports N=String and E=IdDiEdge[Int,N]
type NL = String
type EL = String
val G = Graph.withType[N,NL,E,EL]
// first flip
val e = "u"~~>"v"
val rb = G("u"->"red","v"->"blue")(e)
val br = G("u"->"blue","v"->"red")(e)
val bb = G("u"->"blue","v"->"blue")(e)
val flip0a = Rule(rb, bb, Map("u"->"u","v"->"v"),
  Map(e->e), "k01")
val flip0b = Rule(br, bb, Map("u"->"u","v"->"v"),
  Map(e->e), "k01")
// second flip
val rr = G("u"->"red","v"->"red")(e)
val flip1a = Rule(rb, rr, Map("u"->"u","v"->"v"),
  Map(e->e), "k10")
val flip1b = Rule(br, rr, Map("u"->"u","v"->"v"),
  Map(e->e), "k10")
// first swap (blue rewire)
val rbw1 = G("u"->"red","v"->"blue","w")("u"~~>"v")
val rbw2 = G("u"->"red","v"->"blue","w")("w"~~>"v")
val swap0a = Rule(rbw1, rbw2, Map("u"->"u","v"->"v","w"->"w"),
```

```
  Map(), "k0")
val brw1 = G("u"->"blue","v"->"red","w")("u"~~>"v")
val brw2 = G("u"->"blue","v"->"red","w")("w"~~>"u")
val swap0b = Rule(brw1, brw2, Map("u"->"u","v"->"v","w"->"w"),
  Map(), "k0")
// second swap (red rewire)
val rbw3 = G("u"->"red","v"->"blue","w")("w"~~>"u")
val swap1a = Rule(rbw1, rbw3, Map("u"->"u","v"->"v","w"->"w"),
  Map(), "k1")
val brw3 = G("u"->"blue","v"->"red","w")("w"~~>"v")
val swap1b = Rule(brw1, brw3, Map("u"->"u","v"->"v","w"->"w"),
  Map(), "k1")
def pairApproximation(g: Graph[N,NL,E,EL]): Option[Mn[N,NL,E,EL]] =
  if (g.nodes.size == 3 && g.edges.size == 2 && g.isConnected) {
    val List(e1, e2) = g.edges.toList
    val intersection = e1.nodes &~ (e1.nodes &~ e2.nodes)
    Mn(g.inducedSubgraph(e1.nodes)) *
       g.inducedSubgraph(e2.nodes) /
       g.inducedSubgraph(intersection)
  } else None
def noParallelEdges(g: Graph[N,NL,E,EL]): Option[Mn[N,NL,E,EL]] =
  if (g.nodes.size == 2 && g.edges.size == 2) Some(Mn.zero) else None
val redNode = G("u" -> "red")()
val twoRedNodes = G("u" -> "red", "v" -> "red")()
val odes = generateMeanODEs(10,
  List(flip0a,flip0b,flip1a,flip1b,swap0a,swap0b,swap1a,swap1b),
  List(redNode),
  splitConnectedComponents[N,NL,E,EL] _,
  pairApproximation _,
  noParallelEdges _)
ODEPrinter(odes).print
println()
val varianceODE = generateMeanODEs(1,
  List(flip0a,flip0b,flip1a,flip1b,swap0a,swap0b,swap1a,swap1b),
  List(twoRedNodes))
ODEPrinter(varianceODE).print
```