When rule-based models need to count

Pierre Boutillier[1] Ioana Cristescu[2]

Fontana Lab
Harvard Medical School
Boston, USA

Abstract

Rule-based modelers dislike direct enumeration of cases when more efficient means of enumeration are available. We present an extension of the Kappa language which attaches to agents a notion of *level*. We detail two encodings that are more concise than the former practice.

Rule-based languages are a well-established framework for modeling protein-protein interactions.

Kappa [2,1] is a rule-based language relying on *site-graphs*. The nodes of site-graphs are called *agents*. Agents interact by binding/unbinding through *sites*. Sites are binding resources, each site is part of at most one edge.

A model in Kappa consists of a set of graph rewrite rules with rates. A rule describes a potential interaction given a context. Rates represent the probability of a rule to fire.

In a biological context, it is often the case that a notion of internal state (such as `active`, `methylated`, ...) is required in order to describe possible interactions. In Kappa, sites are equipped with internal states, facilitating the modeling efforts of the user. However, as shown in the following sections a more systematic encoding of internal states is possible.

Another common practice is to attach a *level* to agents and make an interaction sensitive on the level of its participating agents.

We propose here a language extension to store, test and change levels explicitly. Moreover, we present an encoding of levels that induce a linear (in number of levels) blow up of the number of rules. This is in contrast to previous encodings, which induce an exponential blow up in the number of rules.

[1] Email: `Pierre_Boutillier@hms.harvard.edu`
[2] Email: `Ioana_Cristescu@hms.harvard.edu`

1 When enumeration is necessary

The following motivating example [6] demonstrates a typical problem in which levels are necessary:

> KaiC proteins have 6 independent phosphorylation sites. (De)phosphorylation of every site is independent. The more sites are phosphorylated, the bigger the probability that KaiC binds KaiA is.

A typical way to deal with this example consist in explicitly encoding rules for the internal states of the sites of interest. However, doing so induces an exponential blow-up in the number of rules (in the number of levels).

We now show that adding a syntactic layer to Kappa that offers support for counting will avoid the explosion in the number of rules.

2 Encoding counters

In Kappa, agents are typed and their *signature* is given. In the extension we are presenting, counters are part of an agent's signature and the upper bound of the counter has to be specified.

Extended rules can test counter values ("equals" and "bigger than"). They can also modify their value (increase, decrease or assign a new value). Lastly, levels can be used in the rates of the rules.

We now present two possible ways of encoding counters.

2.1 Unary numbers

We define a new agent type `succ` with 2 sites `p` and `n`. A chains of $k + 1$ `succ` agents denotes a counter with value k. Agents equipped with counters are encoded as agents with an extra site for each counter, which is used to bind to a chain of `succ` agents. For example, in figure 1a an agent `A` has a counter `c` set to 1. Note that representing level 0 by a chain of length 1 is a trick: we do not have to make a special case for it in the rules!.

Testing whether a counter is equal to k consists in checking whether there is a chain of $k + 1$ `succ` agents. A greater than k test checks whether there is a chain $k + 1$ `succ` agents, where the site `n` of the last `succ` agent does not matter.

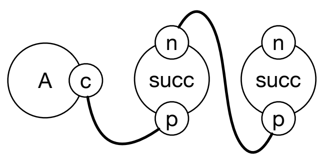
A counter is incremented by adding `succ` agents between the agent and its `succ` chain. The rule depicted on the right increments `c` of `A` by one. Removing the beginning of the `succ` chain decreases the level.

It is important to stress that counter modifications are independent of its value as the encoding only manipulates the beginning of a `succ` chain.

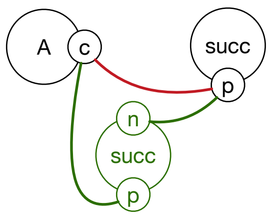
In our encoding, a rule whose rate depends on levels is expanded into several rules, one for each level. This operation imposes the user-defined upper bound on the levels.

Creation of agents with levels also creates the necessary chain of `succ` to represent the levels.

Deletion is more problematic: the chain of `succ` is disconnected from the deleted agent, but not deleted. A possible solution is to collect free chains of `succ` by using a



(a) Agent `A` has site `c` connected to a chain of two `succ` agents, representing a counter `c` of value one.



(b) A rule that increments by one the counter `c` of agent `A`: the red part is tested and erased by the rule; the green part is added and the black part is tested and preserved by the rule.

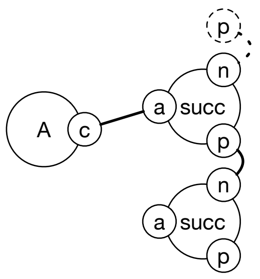
Fig. 1. An agent `A` with a counter `c`

rule that says that `succ` agents with their sites `p` free are deleted "at infinite speed". It is however only a problem for the fine grained performance of the simulator.

2.2 Ruler

A second encoding allows tests "smaller than" in addition of "equal" and "greater than", but is also more verbose and increases the size of states.

As before, counters are encoded as a chain of `succ` agents. In this encoding however, a `succ` agent has 3 sites: `p` and `n` to form chains but also `a`, which is where the other agents bind. Every agent with a counter bounded by n has attached a chain of (always) n `succ`. The value of the counter is given by which `succ` agent it is bound to on site `a`. For example, in the figure on the right, an agent `A` has a counter `c` set to 1.



Since site `p` and `n` are distinct, a direction can be attached to the chain of `succ`. The level is 0 if the agent is bound to the "bottom" of the chain, and it is n if it is bound to the "top".

Incrementing/decrementing the levels are implemented by sliding on the `succ` chain. Testing whether a level is equal or greater than k consists of inspecting the chain of `succ` "below" the connection to the agent. Whereas inspecting the length of the chain "above" the connection to the agent enables to test if the level is smaller than a value k.

3 A simple example

Our first example 1 describes the phosphorylation of two sites of an agent `A`[3]. The rates of the phosphorylation/dephosphorylation depend on the number of phosphorylated sites.

```
%agent: A(s1~u~p,s2~u~p)
%init: 100 A(s1~u,s2~u)

```

[3] taken from http://www.di.ens.fr/ feret/CMSB2017-tool-paper/index.shtml#multi_phos.

```
A(s1~p,s2~u) -> A(s1~p,s2~p) @ 3*5^1
A(s1~p,s2~u) -> A(s1~u,s2~u) @ 2*7^1

A(s1~p,s2~p) -> A(s1~p,s2~u) @ 2*7^2
A(s1~p,s2~p) -> A(s1~u,s2~p) @ 2*7^2

A(s1~u,s2~u) -> A(s1~u,s2~p) @ 3*5^0
A(s1~u,s2~u) -> A(s1~p,s2~u) @ 3*5^0

A(s1~u,s2~p) -> A(s1~u,s2~u) @ 2*7^1
A(s1~u,s2~p) -> A(s1~p,s2~p) @ 3*5^1
```

Code 1: A Kappa program which phosphorylates and dephosphorylates the sites `s1` and `s2` of an agent `A`.

The program can be rewritten using the explicit counters syntax, as in the code snippet 2 below. In figure 2 we introduce the grammar of counters in Kappa. Please see the reference manual of KaSim [1] for the full grammar of Kappa.

```
%agent: A(s1~u~p,s2~u~p,c:0 +=2)
%init: 100 A(s1~u,s2~u,c:0)

A(s1~u,c:x) -> A(s1~p,c +=1) @ 3*5^x
A(s2~u,c:x) -> A(s2~p,c +=1) @ 3*5^x

A(s1~p,c:x) -> A(s1~u,c += -1) @ 2*7^x
A(s2~p,c:x) -> A(s2~u,c += -1) @ 2*7^x
```

Code 2: A Kappa program in which the rules increment or decrement the counter `c` of agent `A`. The rates depend on the counter's value `x`.

We see that allowing explicit counters simplifies the model. We can also compare the two approaches, explicit counters and enumeration of sites, from the point of view of the performance. We show the results in figure 3. We see that for the same version of the simulator and using the same number of levels, the two approaches have close execution times. We conclude that, for some models there is a small performance costs attached to the explicit counters approach, but we argue the performance cost is negligible.

4 Related works

In Kappa each site is unique. Some modeling languages [4,5,7,8] allow *indistinguishable* sites. i.e. one can define an agent with n sites that all have the same name. Consequently, a single rule to specify that k sites (out of n) are phosphorylated is enough. Moreover, the number of species is also reduced.

In the BNGL [4] language (and similarly in React-C [8]) our example 1 can be written as in the code snippet 3 below. We see that some enumeration is still necessary if the rate of a rule depends on *the number of phosphorylated sites*, as in

`%agent` : *signature_expression*

$$\begin{array}{ll}
\textit{signature\_expression} & ::= \texttt{Id(}\textit{sig}\texttt{)} \\
\textit{sig} & ::= \texttt{Id}\ \textit{internal\_state\_list}\texttt{,}\ \textit{sig} \mid \texttt{Id}\ \textit{counter\_test}\ \textit{counter\_modif}\texttt{,}\ \textit{sig} \mid \varepsilon \\
\textit{internal\_state\_list} & ::= \sim \texttt{Id}\ \textit{internal\_state\_list} \mid \varepsilon
\end{array}$$

(a) Agents signature. Set the min and max value of a counter using a *counter_test* expression, and a *counter_modif* expression, respectively.

$$\begin{array}{ll}
\textit{counter\_expression} & ::= \texttt{Id}\ \textit{counter\_test}\ \textit{counter\_modif} \mid \varepsilon \\
\textit{counter\_test} & ::= \texttt{:}\ \ \texttt{n} \mid \texttt{:>}\ \texttt{n} \mid \texttt{:>}\ \texttt{variable} \mid \varepsilon \\
\textit{counter\_modif} & ::= \texttt{:}\ \ \texttt{i} \mid \varepsilon
\end{array}$$

(b) Counters expressions. Both variables n and i are integers, with $n > 0$. A counter test can do three things: (i) check that the counter value is equal to a positive integer; (ii) check that the counters value is greater than a positive integer or (iii) declare a variable, to which the counters value is assigned. The variable can then be used in the rate constant. A counter modification increments or decrements the original counter value.

Fig. 2. The grammar of counters in Kappa.

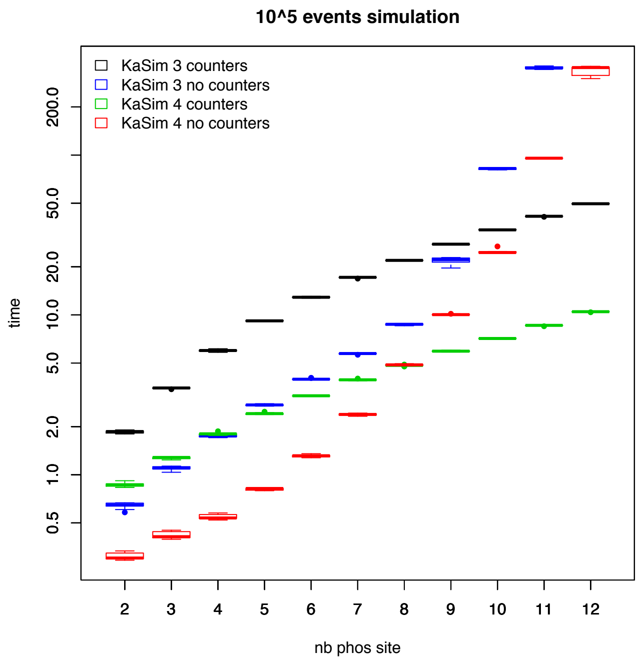


Fig. 3. The execution time of a program with n phosphorylation levels depending on the approach (explicit counters or enumeration of sites) and on the simulator used (KaSim 4 versus KaSim 3).

our example.

```
begin molecule types
A(s~u~p,s~u~p)
end molecule types

```

```
begin seed species
%init: A(s~u,s~u) 100
end seed species

begin reaction rules

A(s~u,s~u) -> A(s~u,s~p) kp0
A(s~p,s~u) -> A(s~p,s~p) kp1
A(s~p,s~u) -> A(s~u,s~u) ku1
A(s~p,s~p) -> A(s~p,s~u) ku2

end reaction rules
```

Code 3: An BNGL program with two undistinguishable sites `s` for an agent `A`.

.

Variants of Kappa such as Kappa with symmetries [5] or meta-Kappa [7,3] allow a restricted usage of indistinguishable sites. Both languages impose restrictions on the usage of *the number of phosphorylated sites* in the rates.

5 Conclusions

While fairly trivial, encoding counters in the Kappa simulator has greatly simplified some of the models written by the Kappa team. However, it comes with two drawbacks. First, there is a computational cost (mainly in term of memory management) to synthetize and degrade `succ` agents. Secondly, the simulator works with a plain Kappa model. There is an implementation cost to go back from the plain Kappa model to the user-written model, which is necessary when giving feedback to the user (for example, dumping the current state of the system). As a result, we plan to implement a native treatment of the counters in the Kappa simulator.

Another direction for future work is the static detection of overflow for counters. Currently, the maximal level of the counter has to be declared by the user. Nevertheless, this means the rules have to be written such that they can never increase a level beyond the defined boundary. The level of the counter can be checked by static analysis using interval computation, but for the meantime we have simply added watchdogs that dynamically raise the alarm if the site of a level becomes free, or if a chain of `succ` that is too long appears.

Acknowledgements.
We gratefully acknowledge the discussions with Walter Fontana, and the helpful comments of the reviewers.

References

[1] Pierre Boutillier, Jérôme Feret, and Jean Krivine. Kappa and KaSim development page. http://kappalanguage.org.

[2] Vincent Danos, Jérôme Feret, Walter Fontana, Russell Harmer, and Jean Krivine. Rule-based modelling, symmetries, refinements. In *Formal Methods in Systems Biology, First International Workshop, FMSB 2008, Cambridge, UK, June 4-5, 2008. Proceedings*, pages 103–122, 2008.

[3] Vincent Danos, Jérôme Feret, Walter Fontana, Russell Harmer, and Jean Krivine. Rule-based modelling and model perturbation. *Trans. Computational Systems Biology*, 11, 2009.

[4] James R. Faeder, Michael L. Blinov, and William S. Hlavacek. *Rule-Based Modeling of Biochemical Systems with BioNetGen*, pages 113–167. Humana Press, Totowa, NJ, 2009.

[5] Jérôme Feret. An algebraic approach for inferring and using symmetries in rule-based models. *Electron. Notes Theor. Comput. Sci.*, 316(C):45–65, September 2015.

[6] Susan S Golden. Integrating the circadian oscillator into the life of the cyanobacterial cell. In *Cold Spring Harbor symposia on quantitative biology*, volume 72, pages 331–338. Cold Spring Harbor Laboratory Press, 2007.

[7] Russ Harmer. Rule-based modelling and tunable resolution. In *Proceedings Fifth Workshop on Developments in Computational Models–Computational Models From Nature, DCM 2009, Rhodes, Greece, 11th July 2009.*, pages 65–72, 2009.

[8] Mathias John, Cédric Lhoussaine, Joachim Niehren, and Cristian Versari. Biochemical reaction rules with constraints. In Gilles Barthe, editor, *Programming Languages and Systems: 20th European Symposium on Programming, ESOP. Proceedings*, 2011.