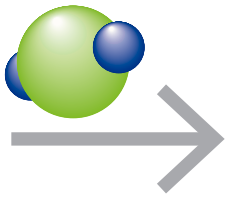
The Kappa Language

A Reference Guide

v4



Walter Fontana, Pierre Boutillier, Jérôme Feret, Jean Krivine

KappaLanguage.org

August 19, 2026

Table of Contents

1 Introduction 1
1.1 Background 1
1.2 Support 2
1.3 Hello ABC 2
2 The Kappa language 5
2.1 Names and labels 5
2.2 Pattern expressions 6
2.2.1 Examples 7
2.2.2 Other notations and renderings 8
2.3 Rule expressions 9
2.3.1 Arrow notation 10
2.3.1.1 Examples 10
2.3.2 Edit notation 15
2.3.2.1 Examples 15
2.3.3 Counters 16
2.4 Kappa Declarations 16
2.4.1 Variables, algebraic expressions, and observables 17
2.4.2 Agent signatures 17
2.4.3 Initial conditions 20
2.4.4 Parameters 20
2.4.5 Tokens and hybrid rules 21
2.5 Intervention directives 22
2.5.1 Timing and conditioning of interventions 23
2.5.2 Model perturbation 23
2.5.3 Model observation 24
2.5.4 Hello ABC, modified 26
3 Simulation 27
3.1 Matching 27
3.2 Symmetry 29
3.3 Rule activity 29
3.3.1 Symmetry and rule activity 30
3.4 The core loop 31
3.5 The rate constant 32
3.6 Rescaling a Stochastic System 33
3.7 Ambiguous molecularity 34
3.8 Rate functions 35
Appendices 38
A Syntax of Kappa 38
A.1 Names and labels 38
A.2 Pattern expressions 38
A.3 Rule expressions 38
A.3.1 Chemical notation 38
A.3.2 Edit notation 39
A.3.3 Counters 39
B Syntax of declarations 40

B.1 Variables, algebraic expressions, and observables 40
B.2 Boolean expressions 40
B.3 Observable declarations 41
B.4 Agent signature 41
B.5 Initial condition 41
B.6 Parameter settings 41
B.7 Token expressions 41
B.8 Intervention directives 42
C Counters 43
D Continuous-time Monte-Carlo 44
E The symmetries of a rule 45

List of Grammars

1 Names and labels 5
2 Pattern expressions 6
3 Rule expressions in arrow notation 10
4 Rule expressions in edit notation 15
5 Variable declaration 17
6 Observable declarations 17
7 Algebraic expression 18
8 Boolean expression 18
9 Agent signature 19
10 Initial condition 20
11 Parameters 21
12 Tokens 21
13 Intervention directives 22
14 Counters 43

List of Figures

1 Graph rewriting in chemistry and Kappa 1
2 Elements of the Kappa UI 2
3 Simulation of the ABC model 4
4 The concept of plain graphs and site graphs 7
5 Patterns 8
6 Transfer of chains 14
7 Intervening in the ABC model 27
8 Embedding a graph into a host graph 28
9 Embedding of binding types 28
10 Rule application 32
11 Ambiguous molecularity 34
12 Rate functions 37
13 Rules and symmetries 46
14 Automorphisms 47
15 Embedding location 48
16 Symmetry correction 48

1 Introduction

This manual aims at providing an up-to-date description of the Kappa language and accompanying software tools. It is a work in progress. We welcome feedback, but keep in mind that a manual is not a modeling tutorial.

1.1 Background

Kappa is a rule-based modeling language. More specifically, it is a graph-rewrite language supported by software for representing, reasoning about, and simulating systems of interacting structured entities (graphs). The insistence on a grammatical structure distinguishes rule-based models from generic agent-based models[1]. Kappa follows the same idea underlying the symbolic representation of organic molecules as graphs and the specification of their transformations as graph-rewrite directives, Figure 1. In chemistry, the concept of a rule emphasizes the distinction between the transformation of a structure fragment and the reaction instance that results when that fragment is transformed within the context of specific entities that contain it. In this sense, a rule represents the *mechanism* of an interaction. That is precisely the intended meaning of a rule in Kappa.

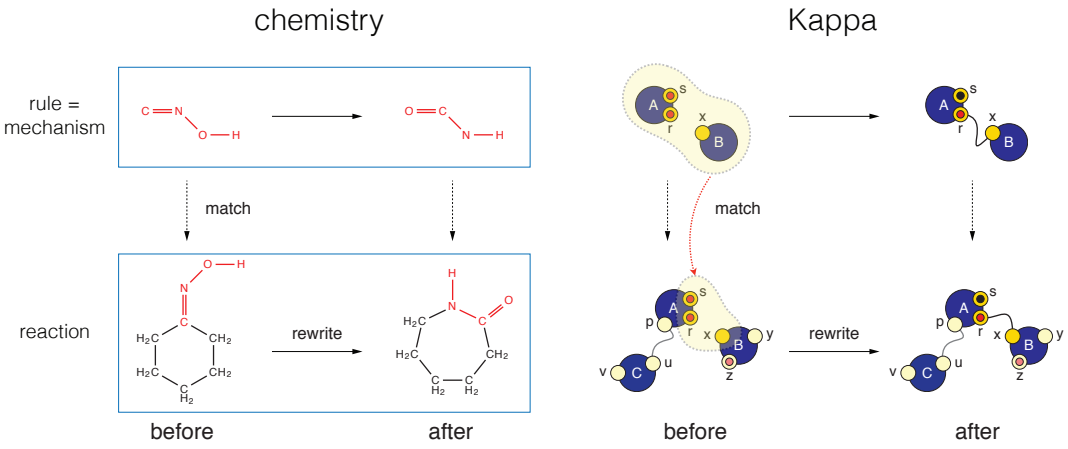


Figure 1: Graph rewriting in chemistry and Kappa. In chemistry, atoms have specific valences through which they bind other atoms. In Kappa, proteins (here blue nodes with a type identified by a name) have sites (here small nodes attached to the blue nodes and identified by names) through which they bind other proteins. In addition to their binding state, sites can hold internal state (here color marks) typically denoting post-translational modifications. A rule specifies the transformation of a graphical fragment. In Kappa as in chemistry, when the fragment on the left hand side of a rule can be matched to a target graph, the matched part is rewritten in place giving rise to a reaction.

Kappa originated with in mind applications to systems of protein-protein interaction where "structured entities" are complexes of non-covalently bound proteins as they arise in signaling and assembly processes. In Kappa, individual proteins appear as agents with a minimal abstract structure given by an interface of *sites* that hold state required for interaction, such as binding and post-translational modification. This said, Kappa is perhaps best thought of as a versatile framework for thinking about the statistical dynamics induced by the mass-action of interacting heterogeneous agents, regardless of how one chooses to interpret them.

Because rules avoid the need to pre-specify all possible molecular species, they enable reasoning about the

[1] The term "agent-based" is often used informally to refer to a modeling style in which discrete units of interaction (the agents) are defined ad hoc, without a systematic internal structure. In such a setting, the complex of a kinase and a substrate might be considered an agent. In Kappa, in contrast, an agent is an atomic entity and a complex of agents explicitly reveals—by virtue of a graphical representation—its composition and connectivity in terms of atoms.

behavior of systems that are marked by combinatorially explosive complexity. Rule-based models can be concise, transparent, and readily extensible, making them candidates for supporting model-based reasoning in bioinformatics.

1.2 Support

The Kappa portal, `http://kappalanguage.org`, is the easiest way to access the latest software (and previous versions). The ecology of Kappa tools consists of several software agents that communicate through an ad hoc JSON-based protocol and expose high-level functionalities through an HTTP REST service. A Python client (API) enables scripting to tailor work flows and is available as the *kappy* package in *pip*.

Modeling in a rule-based language is much like writing large and complex programs, which is greatly facilitated by an integrated development environment. An evolving browser-based User Interface (UI) is aimed at integrating various Kappa web services. The UI is accessible online and also available as a self-contained downloadable application referred to as the `Kappapp`.



Figure 2: Elements of the Kappa UI. Left: Main window with editor and contact graph. Center: XY plots of observables. Right: Patchwork (treemap) rendering of the system contents at a particular time point.

At one glance:

- Items of general interest and downloads can be found at `http://kappalanguage.org`
- Bug reports should be posted to `https://github.com/Kappa-Dev/KaSim/issues`
- The Kappa-user mailing list at `http://groups.google.com/group/kappa-users` is a quick way for asking questions, finding answers, or sharing frustration.
- If you wish to contribute to the Kappa project, please contact Pierre Boutillier.

1.3 Hello ABC

To get a quick intuition about what a Kappa model looks like, consider the following simple system, which is also pre-loaded and ready to run in the online version of the UI.

In this toy model, agents of type `A` can doubly phosphorylate agents of type `C`. However, unphosphorylated `C` can bind `A` only in a complex with `B`. Once phosphorylated, `C` can bind an individual `A`, which then phosphorylates `C` on a second site. The verbal statement of such a model is highly underspecified. To make precise what we mean, we pin down its mechanisms in terms of clear rules.

```
1  // Signatures
2
3  %agent: A(x,c)               // Declaration of agent A
4  %agent: B(x)                 // Declaration of agent B
5  %agent: C(x1{u p},x2{u p})   // Declaration of agent C
```

```
6
7  // Variables
8
9  %var: 'on_rate'  1.0E-4      // per molecule per second
10 %var: 'off_rate' 0.1         // per second
11 %var: 'mod_rate' 1           // per second
12
13 // Rules
14
15 // A and B bind and dissociate:
16 'rule 1' A(x[.]), B(x[.]) <->
17          A(x[1]), B(x[1]) @ 'on_rate', 'off_rate'
18
19 // AB binds unphosphorylated C:
20 'rule 2' A(x[_],c[.]), C(x1{u}[.]) ->
21          A(x[_],c[2]), C(x1{u}[2]) @ 'on_rate'
22
23 // site x1 is modified:
24 'rule 3' C(x1{u}[1]), A(c[1]) ->
25          C(x1{p}[.]), A(c[.]) @ 'mod_rate'
26
27 // A conditionally binds C:
28 'rule 4' A(x[.],c[.]), C(x1{p}[.],x2{u}[.]) ->
29          A(x[.],c[1]), C(x1{p}[.],x2{u}[1]) @ 'on_rate'
30
31 // site x2 is modified:
32 'rule 5' A(x[.],c[1]), C(x1{p}[.],x2{u}[1]) ->
33          A(x[.],c[.]), C(x1{p}[.],x2{p}[.]) @ 'mod_rate'
34
35 // Observables
36
37 %obs: 'AB'  |A(x[x.B])|
38 %obs: 'Cuu' |C(x1{u},x2{u})|
39 %obs: 'Cpu' |C(x1{p},x2{u})|
40 %obs: 'Cpp' |C(x1{p},x2{p})|
41
42 // Initial condition
43
44 %init: 1000  A(), B()
45 %init: 10000 C(x1{u},x2{u})
```

A file with these sections is also referred to as a "Kappa file". The signatures section (lines 3–5) declares for each agent a set of sites (its interface) and the possible values each site can take. For example, line 5 informs us that agents of type `C` have two sites `x1` and `x2` whose internal state may have the label `u` (for unphosphorylated, say) or `p` (for phosphorylated). In the rules section, the rule labeled 'rule 2' starts on line 20 and asserts that if an agent of type `A` is bound (to someone not further specified—that's the underscore) at its site `x` and if it is free (unbound) at its site `c`, then it can bind an agent of type `C` provided `C`'s site `x1` is free and unphosphorylated. In this rule, the state of agent `C` at its site `x2` is not mentioned and therefore irrelevant to the applicability of the rule. This is why a rule is not a reaction. The left- and right-hand sides of a rule are usually patterns—partially specified molecular species. Hence, a rule subsumes many possible reactions.

As mentioned, an `A` can modify both sites of `C` once it is bound to them. However, only an `A` bound to a `B` can connect to a `C` on `x1` and only a free `A` can connect to `C` on `x2`. Note also that site `x2` is available for binding only when `x1` is already modified.

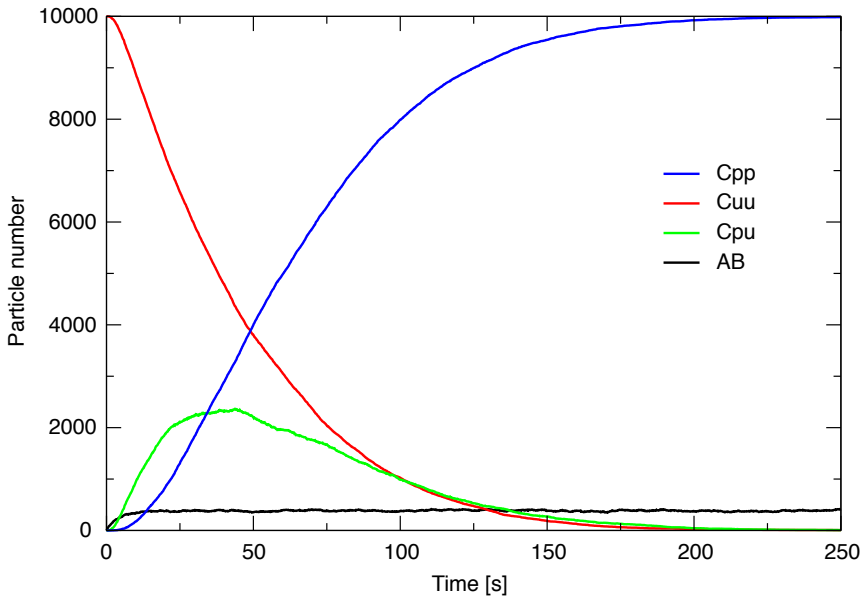


Figure 3: Simulation of the ABC model. The population of unmodified `Cs` (observable `Cuu` in red) drops rapidly and is replaced, at first, by simply modified `Cs` (observable `Cpu` in green), which are in turn replaced by doubly modified `Cs` (observable `Cpp` in blue). The population of `AB` complexes (observable `AB` in black) stabilizes slightly below 400 individuals after about 20s.

We will call this model "ABC" and shall use it to illustrate concepts in others sections of this manual. For now, let us simply run "ABC". To this end, download the KaSim executable for your operating system. At its core, KaSim implements a continuous-time Monte Carlo (Gillespie) simulation.

One possibility is to run for 100,000 interaction events (10 times the number of agents in the initial system) using the command line:

```
> KaSim ABC.ka -u event -l 100000 -p 1000 -o abc.csv
```

Another possibility is to push the start button on the online UI and switch to the plot tab to see the developing trajectory of the observables. The command line produces a csv output file whose contents can be plotted by any number of tools, such as `gnuplot`. From Figure 3 we see that the observables have become stationary after 250 (simulated) seconds. We could therefore specify a meaningful time limit instead of an event limit in subsequent simulations:

```
> KaSim ABC.ka -l 250 -p 0.25 -o abc.out
```

2 The Kappa language

We overload the term Kappa language (or Kappa for short) with both a broad and a narrow meaning. In a narrow sense, Kappa refers to a language for specifying patterns of certain graphs—"site graphs". The narrow sense also includes rules that specify the rewriting of such patterns. Understood in a broad sense, Kappa includes the above plus a collection of declarations with which inputs are provided to the simulator `KaSim`. These inputs enable the execution of a model and the observation of its behavior. It should be clear from context which sense we mean.

A Kappa model consists of a set of files whose concatenation constitutes the Kappa input file or KF for short. The KF serves as input to the Kappa tool in question, usually `KaSim`. This input could be a single file, but splitting it up can be convenient.

A KF consists of *declarations*, which can be

- *rules* (section 2.3)
- *variables* (section 2.4.1)
- *signatures* of agents (section 2.4.2) and tokens (section 2.4.5)
- *initial conditions* (section 2.4.3)
- *intervention directives* (section 2.5)
- *configuration settings* (section 2.4.4)

The structure of the KF is quite flexible. The order of declarations is not important with the exception of variable declarations and intervention directives as detailed in sections 2.4.1 and 2.5, respectively).

Comments work much like in the C language. A comment can start with a `//` marker, which instructs `KaSim` to ignore the remainder of the line. A `/* comment */` can also be placed between a pair of matching delimiters.

In the following sections, formal grammars (BNF forms) are used to define various language elements. All grammars are gathered in Appendices A and B for quick reference. Terminal symbols are in red. Optional expansions are enclosed in square brackets. The symbol ε represents the empty list.

The substantive language element in the context of a model is a rule. But to discuss rules, we first need to work through elements of Kappa from which rules are built, specifically identifiers and patterns. After having defined Kappa rules, we proceed with Kappa declarations and intervention directives that complete Kappa in the broad sense—the input language to the simulator `KaSim`.

2.1 Names and labels

The name construct ⟨*Name*⟩ refers to any string generated by the regular expressions indicated below. It is used to name agents, sites, states, and variables. The ⟨*Label*⟩ construct is similar, but must be in single quotes. It can contain any sequence of characters, excluding the newline or the quote characters. A ⟨*Label*⟩ is used to name rules.

Grammar 1: Names and labels

⟨*Name*⟩	::= [a-z A-Z] [a-z A-Z 0-9 _ ~ − +]*	// cannot start with a digit
	| [_] [a-z A-Z 0-9 _ ~ − +]+	// initial underscore can't stand alone
⟨*Label*⟩	::= ' [^ \n ']+ '	//no newline or single quote in a label

2.2 Pattern expressions

A Kappa expression denotes a *site graph*. In a site graph, nodes possess a set of sites, called the *interface* of the node. The sites, not the nodes themselves, are the endpoints of edges, Figure 4. A site graph formalizes the resources that an interaction requires, such as physical surfaces in the case of a binding interaction.

The name of an agent denotes its type. Thus, RAS denotes a type of protein, not a specific instance. To tell it from other instances of the same type, a node in a graph has an associated identifier (node id). The node id is (typically) not explicitly mentioned in graphical or line-oriented expressions as it is implicit in the node layout or the sequence of agent occurrences, respectively.

Grammar 2: Pattern expressions

⟨*pattern*⟩	::=	⟨*agent*⟩ ⟨*more-pattern*⟩	
⟨*agent-name*⟩	::=	⟨*Name*⟩	
⟨*site-name*⟩	::=	⟨*Name*⟩	
⟨*state-name*⟩	::=	⟨*Name*⟩	
⟨*agent*⟩	::=	⟨*agent-name*⟩ (⟨*interface*⟩)	
⟨*site*⟩	::=	⟨*site-name*⟩ ⟨*internal-state*⟩ ⟨*link-state*⟩	
	|	⟨*site-name*⟩ ⟨*link-state*⟩ ⟨*internal-state*⟩	
	|	⟨*counter*⟩	// see Grammar 14
⟨*interface*⟩	::=	⟨*site*⟩ ⟨*more-interface*⟩	
	|	ε	
⟨*more-pattern*⟩	::=	[,] ⟨*pattern*⟩	
	|	ε	
⟨*more-interface*⟩	::=	[,] ⟨*site*⟩ ⟨*more-interface*⟩	
	|	ε	
⟨*internal-state*⟩	::=	{ ⟨*state-name*⟩ }	
	|	{#}	// wildcard
	|	ε	
⟨*link-state*⟩	::=	[⟨*number*⟩]	
	|	[.]	
	|	[_]	
	|	[#]	// wildcard
	|	[⟨*site-name*⟩ . ⟨*agent-name*⟩]	
	|	ε	
⟨*number*⟩	::=	$n \in \mathbb{N}_0$	

We distinguish two kinds of site graphs: contact graphs and patterns. A contact graph is a site graph where every agent node has a different name and a site can be the endpoint of multiple edges. Contact graphs represent a static summary of all agent types that occur in a model alongside their potential binding interactions. The contact graph is the diagram that is displayed in the Kappa UI on the right side of the model editor. A pattern, on the other hand, is a site graph that can contain multiple nodes with the same name, representing different agents of the same type, but each site can be the endpoint of at most one edge. A pattern is meant to represent a partially specified molecular species, which is to say a realizable object whose resources are single-use at any given moment. A full-fledged molecular species is represented by the special case of a pattern in which every agent exhibits all its sites in a definite state.

It bears emphasis that all sites of an agent must have distinct names. (The interface is a set, not a multiset.) As a consequence, an embedding (or matching) of a connected site graph into another (i.e. a sub-graph isomorphism) is completely defined by the matching of a single node. (Kappa sub-graph isomorphisms are in P.) We call this property the *rigidity* of Kappa. Rigidity is critical for the efficiency of the Kappa platform.

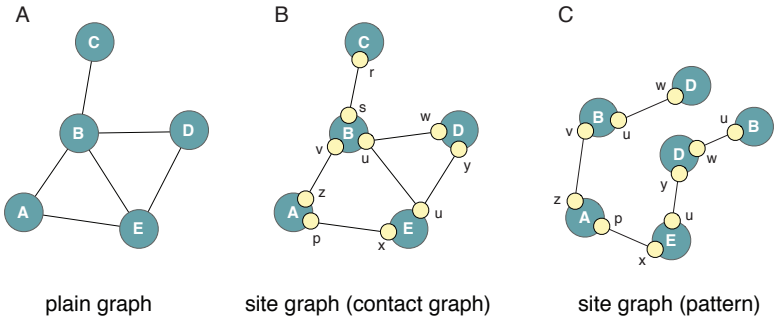


Figure 4: The concept of plain graphs and site graphs. **A:** In a plain graph edges directly connect nodes. **B:** In a site graph, nodes possess sites (lower-level nodes) that act as endpoints of edges. One can think of a site graph as a refinement (a higher level of resolution) of a plain graph. In a contact graph, every node type occurs exactly once and a site can exhibit more than one edge, summarizing all its interaction partners. **C:** A pattern allows for multiple nodes of the same type to occur, but a site can exhibit at most one edge (and some sites might not be mentioned at all).

According to grammar 2, the anatomy of a Kappa pattern is as follows.

- An agent expression consists of its type (the name of the agent) followed, between parentheses, by a list of site names (optionally comma-separated).
- The state of a site is specified after its name. We distinguish two kinds of states: an internal state (usually representing a modification state) and a link state (usually representing a bond). The order in which they are specified is not significant.
- The internal state of a site is a label written between curly braces, for example `{happy}`.
- The link state of a site is usually a non-negative integer written between square brackets, for example `[42]`. Refer to ⟨*link-state*⟩ in Grammar 2 for more options and consult the examples below.
- An edge is identified by an arbitrary non-negative integer n that is unique in the scope of the pattern. The two sites that are the endpoints of an edge exhibit its label as a link state. Thus, in a pattern expression, each link label appears exactly twice.
- The link state of a site that is unbound (free) is written as a dot: `[.]`.

2.2.1 Examples

The idea of a pattern is to leave some state unspecified (”don’t care, don’t write”), as exemplified next.

- ► `A(x[.],z[.])` means an agent of type `A` whose sites `x` and `z` are free but whose internal state is left unspecified (Figure 5, graph I). What exactly an expression leaves unspecified is relative to the agent signature (Grammar 9) and ultimately the rules. If in our example `A`’s sites can have no internal state and there are no sites other than `x` and `z`, then nothing is left unspecified and the expression is also a molecular species.
- ► `A(x[.],z)` means that both the internal state and the binding state at `z` are unspecified. This is the same as writing `A(x[.])`.

- `A(loc{membrane},z{p}[0]),B(loc{cytosol},x[0])` denotes a complex of two agents bound to each other, the bond having the label 0. Agent `A`'s site `loc` in state `membrane`, so maybe this could be taken as crude localization information. `A`'s site `z` is in state `p` (phosphorylated, say) and bound to site `x` of agent `B` located in the cytosol; so maybe this pattern refers to a transmembrane protein.
- `A(x{u}[#])` means agent `A` whose site `x` is in state `u` and *explicitly* in an unspecified binding state (Figure 5, graph II). In the context of pattern expressions, this is the same as writing `A(x{u})`. The rationale for the pound sign (wildcard) will be discussed in the context of rules, section 2.3.
- `A(x[1],y[3]),A(x[2],y[1]),A(x[3],y[2])` is a three-membered ring made of agents of type `A` (Figure 5, graph III).
- `A(x[1],y[_]),A(x[1],y[.])` is a dimer of `A`s, in which one `A` is bound to an unspecified agent at site `y` (Figure 5, graph IV).
- `A(x[.],y[x.B])` means `A` is free on site `x` and bound to the site `y` of *some* `B`. However, the so-called "binding type" `x.B` (sometimes also called a "bond stub") only specifies the binding site and *type* of the bound partner and nothing else (Figure 5, graph V).

  ⚠ Careful: The binding type construct is best understood in the context of matching a pattern, section 3.1. Suffice it to emphasize here that the difference between $\mathscr{P}_1$ = `A(y[x.B])` and $\mathscr{P}_2$ = `A(y[1]),B(x[1])` is that $\mathscr{P}_2$ refers to an agent with name `B` as a stoichiometric resource, whereas $\mathscr{P}_1$ refers to the type (here `B`) of bound agent. A type is not a stoichiometric resource. For example, the binding type `x.B` in the pattern `A(x[.],y[x.B]),B(y[.])` *may* refer to the `B`-agent explicitly mentioned in the pattern (that's `B(y[.])` or it *may* refer to a second `B`-agent, outside the pattern. Likewise, the binding types in `A(x[y.B]),C(z[t.B])` may refer to one and the same or two distinct `B`s.

2.2.2 Other notations and renderings

When discussing Kappa expressions abstractly it is more succinct to use a "mathematical" notation instead of ASCII. In the math notation, bond labels become superscripts of sites and internal states become subscripts. For example, the expression for the three-membered ring reads $A(x^1,y^3),A(x^2,y^1),A(x^3,y^2)$. The ASCII expression `A(b{p}[x.C]),B(a[.]),C(y{u})` becomes $A(b_p^{x.C}),B(a^{\cdot}),C(y_u)$.

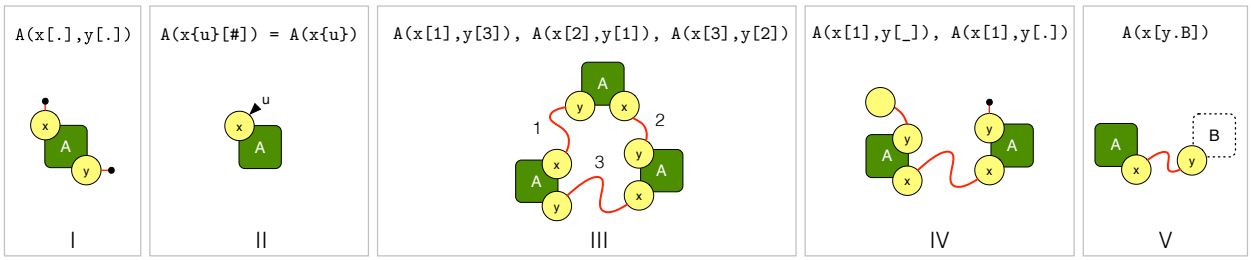


Figure 5: Patterns. Examples of Kappa patterns described in section 2.2.1. In graph III, the numbers labeling the edges are not part of the graph, but only serve to illustrate the labeling scheme in the line notation.

Kappa expressions are line-oriented encodings of site graphs. In graphical displays we indicate the free (unbound) state of a site with a "bond" to a dot, which stands for "nothing" (Figure 5 I). A site whose binding state is unspecified is simply shown without any binding state (Figure 5 II). If the site is bound to some unspecified agent type (an underscore in the line notation), it is shown with a bond to an unnamed orphaned site (Figure 5 IV). A binding stub is a bond to a named site that belongs to an outlined agent type (Figure 5 V). Occasionally, to reduce clutter, we omit the names of sites. When we do so, we will assume an

implicit naming derived from the geometric position of the site on the agent. This works best with few sites that can be arranged on geometric landmarks of an agent. For example, if an agent is drawn as a square, the sides (E, W, S, N) or corners (NE, NW, SE, SW) might implicitly serve as site names. On the other hand, if a site is explicitly named, then the position on the agent has no meaning. In any case, graphical renderings are up to the modeler (or the interface designer), as they are not directly parsed by `KaSim`.

2.3 Rule expressions

Rules have the basic shape $\mathscr{L} \rightarrow \mathscr{R}$. The intended meaning is that the right-hand side graph $\mathscr{R}$ replaces the left-hand side graph $\mathscr{L}$. This replacement usually occurs in the context of a larger graph $\mathscr{G}$ (for example representing a molecular species) that matches $\mathscr{L}$. Matching here means that $\mathscr{L}$ is subgraph-isomorphic to $\mathscr{G}$. We define matching in section 3.1, but the intuitive meaning should be fairly obvious.

There are two ways of specifying rules:

1. The "arrow notation" (or chemical format): This is the familiar format, $\mathscr{L} \rightarrow \mathscr{R}$, in which the "before" ($\mathscr{L}$) and the "after" ($\mathscr{R}$) are two pattern expressions (section 2.2). The arrow requires a mapping between $\mathscr{L}$ and $\mathscr{R}$ that specifies which agents in $\mathscr{L}$ corresponds to which agents in $\mathscr{R}$. If an agent appearing in $\mathscr{L}$ has no correspondence in $\mathscr{R}$, the rule destroys that agent. Likewise, if an agent appearing in $\mathscr{R}$ has no correspondence in $\mathscr{L}$, the rule creates (an instance of) that agent. The grammar of the arrow notation and its interpretation make this mapping explicit.
2. The "edit notation" is more compact and avoids the need of a mapping between two pattern expressions by simply writing edit directives into the "before" pattern. This also avoids errors that might arise when duplicating the invariant part of the $\mathscr{L}$ pattern on the right in the arrow notation.

Both styles are understood by the parser and can be freely mixed. Regardless of style, rules *can* be prefixed by a name and *must* end with rate information.

For discussing the anatomy and expressiveness of rules it suffices to assume for now that the expression ⟨*rate*⟩ in Grammars 3 and 4 is simply a number $k \in \mathbb{R}$. We discuss the meaning of a rate in Kappa in section 3.5 and algebraic expressions, which can be used to express rate *functions*, in section 2.4.1.

There are three flavors of rules.

1. Forward rule: This is the standard one-way format read from left to right (→) in the arrow notation. It requires one rate expression denoting the forward rate of the rule. To express a mechanistically exactly reversible interaction, one would use two forward rules in which the patterns on the left and right are swapped. This can be achieved more compactly using the reversible rule format (below)[2].
2. Reversible rule: This format expresses a mechanistically reversible interaction and is a shorthand for two forward rules in which the left and right are swapped. It requires two rate expressions, one for the forward and one for the backward rate of the rule.
3. Rule with ambiguous molecularity ("ambi-rule"): A rule expresses a mechanism, and a mechanism is essentially a *local* interaction. Local means that all context necessary for an interaction is explicitly specified. This can lead to an ambiguity regarding the number of distinct molecular species involved in an interaction, which, in biological applications, affects how a rate constant scales with volume. An "ambi-rule" requires two rate expressions, unless it is reversible (at which point it requires three or four, depending on whether the reverse direction is also ambiguous). Ambi-rules will be discussed in more depth in section 3.7, but see the last example in section 2.3.1.1.

[2] In situations far from equilibrium, certain interactions are modeled as irreversible. The undoing of such an interaction is often achieved by a different mechanism altogether, which is not a case of reversibility. For example, a phosphorylation that requires a bound kinase is often undone by a dephosphorylation that requires a bound phosphatase.

2.3.1 Arrow notation

The syntax of the arrow notation is specified in Grammar 3.

Grammar 3: Rule expressions in arrow notation

⟨*f-rule*⟩	::=	[⟨*Label*⟩] ⟨*rule-expression*⟩ [| ⟨*token*⟩] @ ⟨*rate*⟩
⟨*fr-rule*⟩	::=	[⟨*Label*⟩] ⟨*rev-rule-expression*⟩ [| ⟨*token*⟩] @ ⟨*rate*⟩ , ⟨*rate*⟩
⟨*ambi-rule*⟩	::=	[⟨*Label*⟩] ⟨*rule-expression*⟩ [| ⟨*token*⟩] @ ⟨*rate*⟩ { ⟨*rate*⟩ }
⟨*ambi-fr-rule*⟩	::=	[⟨*Label*⟩] ⟨*rev-rule-expression*⟩ [| ⟨*token*⟩] @ ⟨*rate*⟩ { ⟨*rate*⟩ } , ⟨*rate*⟩
⟨*rule-expression*⟩	::=	(⟨*agent*⟩ | .) ⟨*more*⟩ (⟨*agent*⟩ | .)
⟨*more*⟩	::=	, (⟨*agent*⟩ | .) ⟨*more*⟩ (⟨*agent*⟩ | .) , | ->
⟨*rev-rule-expression*⟩	::=	(⟨*agent*⟩ | .) ⟨*rev-more*⟩ (⟨*agent*⟩ | .)
⟨*rev-more*⟩	::=	, (⟨*agent*⟩ | .) ⟨*rev-more*⟩ (⟨*agent*⟩ | .) , | <->
⟨*rate*⟩	::=	⟨*algebraic-expression*⟩

As mentioned, the arrow notation requires an explicit mapping between agents on the left and agents on the right. This is achieved by requiring the same number of comma-separated "slots" on both sides of the arrow. A slot is either occupied by an agent or it is vacant, in which case it is represented by a lonely dot-symbol between commas. Thus,

```
'rule' A(x[.]), B(x[.]) -> A(x[1]), B(x[1]) @ 0.001
//      #1L       #2L         #1R       #2R
```

Slots are imagined as numbered from left to right on each side of the arrow with the first slot on the left (#1L) mapping to the first slot on the right (#1R); the second slot on the left (#2L) to the second slot on the right (#2R); etc. From the viewpoint of the pattern grammar (Grammar 2), `A(x[1]),B(x[1])` and `B(x[1]),A(x[1])` mean exactly the same thing: an `A` bound to a `B` at their respective sites `x`. However, if we were to swap `A(x[1]),B(x[1])` with `B(x[1]),A(x[1])` in rule `'rule'`, the parser would throw an error because of an agent mismatch.

```
⛔ 'error'  A(x[.]), B(x[.]) -> C(x[1]), B(x[1]) @ 0.001
//          #1L       #2L         #1R       #2R
```

A rule cannot transmute one agent type into another.

For the sake of completeness: The naming of a rule (here `'rule'` or `'error'`) is arbitrary and optional. The number after the @-symbol is the forward rate constant associated with the rule. The number `1` on the right appears exactly twice, as it identifies a bond with two endpoints. This edge identifier can be an arbitrary non-negative natural number.

2.3.1.1 Examples

A few examples will convey a sense for rules in the arrow notation.

▶
```
'asymmetric dimerization'
A(x[.]), A(y[.]) -> A(x[1]), A(y[1]) @ 0.001
'symmetric dimerization'
A(x[.]), A(x[.]) -> A(x[1]), A(x[1]) @ 0.001
```

These are simple dimerizations between two `A`s. In rule `'asymmetric dimerization'` the two `A`s bind each other asymmetrically—one uses site `x`, the other site `y`. The repeated application of this rule to an initial pool of free `A`s will result in polymerization. On the other hand, the repeated application of the rule `'symmetric dimerization'` will result in a population of strict dimers. The example also shows that rules can extend over multiple lines without the need for a continuation character.

▶
```
'conditional asymmetric dimerization'
A(x[.]), A(y{p}[.]) -> A(x[1]), A(y{p}[1]) @ 0.001
```

Here the binding depends on site `y` not only being free but also being in state `p` (phosphorylated, say). This could allow moderation of the polymerization degree through the activity of another rule that phosphorylates `A` and site `y`.

▶
```
'degrade'  A(x[1]), A(y{p}[1]) -> . , A(y{p}[.]) @ 0.001
```

A binding partner is destroyed. This removes the bond labeled `1`; in fact, it removes *all* bonds the destroyed agent might have entertained with other agents not mentioned in the rule. Be aware of potential side effects!

▶
```
'create'  A(x[.]), . -> A(x[1]), A(y{p}[1]) @ 0.001
```

An additional `A` is created and bound to the `A` mentioned on the left. If the internal state at `y` is not specified, the freshly created agent would have a default state. The default state can be defined in the agent signature (section 2.4.2).

▶
```
'force1'    A(y{#}) -> A(y{p}) @ 0.001
⛔ 'nono1'  A(y)    -> A(y{p}) @ 0.001
⛔ 'nono2'  A()     -> A(y{p}) @ 0.001
'force2'    A(y[#]), B(x[.]) -> A(y[1]), B(x[1]) @ 0.001
```

In these examples we force a site into a definite state. The symbol (`#`) is a "wildcard" and means an unspecified state. Rule `'force1'` puts site `y` from an arbitrary internal state into a definite state `p`. Assuming that the permissible values at `y` are `{u,p}`, `'force1'` really acts as if it were two rules (a refinement or split by cases):

```
'refinement1' A(y{u}) -> A(y{p}) @ 0.001
'refinement2' A(y{p}) -> A(y{p}) @ 0.001
```

Upon matching an unphosphorylated `A`, `'force1'` phosphorylates it; but matching an already phosphorylated `A` results in no change. Yet, from the point of view of simulation (section 3), the "no change" event (i.e. the firing of `'refinement2'`) is an event that advances the simulated time. Rule `'force1'` phosphorylates `A` at the same rate as `'refinement1'` alone, but is computationally more expensive because of the null events.

Although the pattern `A(y{#})` is equivalent to `A(y)`, the parser will reject rule `'nono1'` based on the rationale that `A(y)` is too error prone a construct in the context of a rule. It is easy to forget specifying a particular state for a site, which results in semantic errors that can be difficult to find in a complex model. This justifies the explicit symbol (`#`) for *deliberately* asserting an unspecified state.

The same reasoning applies to rule `'nono2'`: *Sites that appear on the left must appear on the right and vice versa or the parser will raise an error.*

Finally, `'force2'` illustrates the forcing of a binding state.

```
'side effect'          A(x[_])    -> A(x[.]) @ 0.001
'also side effect'     A(x[y.B]) -> A(x[.]) @ 0.001
```

This is the forcing of an underspecified (binding) state. Perfectly legal. However, by unbinding `A` at site `x` any agent at the other end of the bond is disconnected from `A` as well. In the context of rules, the underscore (`_`) has a *side effect* in the sense that the rule modifies the state of an agent not explicitly mentioned. The underscore can be quite useful provided the user is aware of the scope of side effects. The so-called "binding type", as in rule `'also side effect'`, is often used in lieu of the underscore. It specifies the site and *type* of the agent at the other end of a bond without including the agent itself in the expression. This improves readability and provides a minimum of type safety.

```
'many modifications at once'
A(y{#}[1]),     .    , B(x[1],y[_]), C()   ->
A(y{p}[7]), D(q[0]), B(x[0],y[7]),   .       @ 0.001
```

In a useful model, rules are mostly "elementary", meaning that they represent one single action, albeit guarded by a possibly complex condition. The above rule is the opposite; a multi-action rule in which many changes occur simultaneously (including a bond swap to a newly created agent). While perfectly legit, it seems a rather absurd mechanism. This said, in some situations multi-actions are unavoidable. For example, when destroying an agent or when expressing the fusion of two chains (Figure 6).

```
'nonsense'                A(x[.]) -> A(x[_]) @ 0.001
'bidirectional nonsense'  A(x[#]) <-> A(x[_]) @ 0.001, 0.001
'more nonsense'           A(x[#]) -> A(x[y.B]) @ 0.001
'not allowed'             A(x{p}) -> A() @ 0.001
```

The rule `'nonsense'` puts a defined state (unbound) into an underdefined state. The problem here is that while the action could be executed in principle by choosing a random agent and bind it to `A`, it no longer represents a *deterministic* mechanism. In the reverse direction of `'bidirectional nonsense'`, the underscore is replaced by `#`. A mechanism in which an agent forgets the state it is in makes little physical sense (and what happens to whoever is bound to `A`?). The forward direction is not meaningful either (in analogy to `'force1'`, the forward direction of `'bidirectional nonsense'` can be refined, one case being `'nonsense'`). The rule `'more nonsense'` does not execute in a predictable way either. For example, the simulator might pick a random `B` to attach to `A`, but nothing guarantees that the picked `B` is free at site `y`, at which point it would have to try again. Finally, `'not allowed'` is semantically a "no operation"—it does nothing—but violates the constraint that sites appearing on the left of a rule must also show up on the right.

```
'reversible association'
K(s[.]),S(e[.],x{u}) <-> K(s[1]),S(e[1],x{u}) @ 0.001, 0.1
'phosphorylation'
K(s[1]),S(e[1],x{u})  -> K(s[1]),S(e[1],x{p}) @ 0.1
'dephospho'
S(x{p}) -> S(x{u}) @ 0.001
```

The meaning of bidirectional rules is obvious. Their purpose is to save writing, but remember to add rate information for the reverse interaction. Yet, in biological applications far from thermodynamic equilibrium, one uses uni-directional rules when the state conditions in one direction differ from

those in the (functionally) "opposite" direction. For example, rule `'phosphorylation'` changes the state of substrate `S` when in a complex with kinase `K`. Perhaps the only route for `S` to revert back to state `u` is to spontaneously dephosphorylate, as in `'dephospho'`. In this case, phosphorylation and dephosphorylation are not mechanistically opposite to each other.

- `'ambiguous molecularity'`

```
A(x[.]), A(y[.]) -> A(x[1]), A(y[1]) @ 0.001 {0.1}
```

We return to the example of asymmetric dimerization. As pointed out, when repeatedly applied to a pool of initial `A`-agents, this rule will generate polymers because it does not care whether any of the two `A`-agents is already bound to others at the respective site not mentioned. In particular, the rule is oblivious of whether the two `A` are the two ends of a single chain. In that case, it will convert the chain into a ring (with the same number of protomers). In rule `'asymmetric dimerization'` the two distinct reaction instances induced by the rule occur with the same rate constant (as only one is suplied). Yet, chemically, the elongation of a chain is a bi-molecular interaction, involving the collision between two entities, whereas the ring formation is a uni-molecular interaction involving no collision at the "entity"-level of abstraction. Chemically, this warrants two distinct rate constants, as they must scale differently with the system volume the user might choose (section 3.5). If the uni- and bi-molecular applications of a rule should be kinetically distinguished, the user must supply rate information for the uni-molecular case in curly braces. The rate syntax is γ_2 $\{\gamma_1\}$ with γ_2 the bi-molecular rate constant (or function) and γ_1 the uni-molecular rate constant (or function). If only one rate is supplied, no distinction is made between the two cases. Rules with ambiguous molecularity require KaSim to track appropriate information to correctly implement the stochastic kinetics. This can slow down simulation considerably; see section 3.7.

We end with a more complex example to exercise the Kappa muscle. Imagine two agents, each carrying a polymeric tail. The agents bind, and one agent concatenates its polymeric tail to that of the other as illustrated in Figure 6A. This scenario occurs between E2 ligases that carry ubiquitin chains. Making a rule that specifies such a transfer between polymer tails of defined length is straightforward. For example, the following rule concatenates a tail of length 2 and a tail of length 3:

```
'2+3'
E(e[1],start[2]),U(h[2],t[6]),U(h[6],t[.]),
E(e[1],start[4]),U(h[4],t[7]),U(h[7],t[8]),U(h[8],t[.])
->
E(e[1],start[.]),U(h[2],t[6]),U(h[6],t[.]),
E(e[1],start[4]),U(h[4],t[7]),U(h[7],t[8]),U(h[8],t[2]) @ 0.001
```

The protomers of type `U` form head-to-tail chains. The transfer of one chain to the end of the other occurs by first releasing the bond between the head (`h`) of the first protomer in a chain and its carrier and then binding it to the free tail (`t`) of the other chain (Figure 6A). This, however, implies a distinct mechanism (i.e. a separate rule) for any pair of polymer lengths, which seems unlikely. Figure 6B slightly generalizes the rule `'2+3'` to `'m+3'`, which adds a chain of length 3 to a chain of arbitrary length m. The chain of length m only shows up with the unit that attaches to the carrier `E`. This suffices because the chain of length three is represented explicitly, thus specifying the (free) tail point for attaching the m-chain. Next is an encoding, `'m+n bloated'` (also rendered graphically in Figure 6C) for concatenating two chains of any length with a single rule, hence a single mechanism.

```
'm+n bloated'
E(e[1],start[2],end[3]),U(h[2],t[h.U],l[.]),U(h[t.U],t[.],l[3]),
E(e[1],start[6],end[5]),U(h[6],t[h.U],l[.]),U(h[t.U],t[.],l[5])
->
E(e[1],start[.],end[.]),U(h[2],t[h.U],l[.]),U(h[t.U],t[.],l[5]),
E(e[1],start[6],end[5]),U(h[6],t[h.U],l[.]),U(h[t.U],t[2],l[.]) @ 0.001
```

It is perhaps instructive that to accomplish a generic concatenation required introducing a new site (a new logical resource), `end`, that points to the end of a chain. (Likewise, a new complementary site `l` is required for agents of type `U`.) This is because *both* chains have unspecified length. The fictitious sites `end`, `l`, and the bond between them can be interpreted as information about the "location" of the tail of a chain of arbitrary length. A system of interactions that constructs chains in this manner is guaranteed to have U_1 (the agent of type `U` with identifier 1 in Figure 6C) connected to U_2, despite this not being explicit in the pattern. (The same holds for the other chain.) We can exploit this behavior to completely omit the binding types. In addition, upon reflection, several sites appearing on the left of the rule in Figure 6C need not be mentioned, as they are not necessary conditions for concatenation. These observations yield a more succinct version shown in Figure 6D.

```
'm+n minimalist'
E(e[1],start[2],end[3]),U(h[2]),U(l[3]),
E(e[1],start[6],end[5]),U(h[6]),U(l[5],t[.])
->
E(e[1],start[.],end[.]),U(h[2]),U(l[5]),
E(e[1],start[6],end[5]),U(h[6]),U(l[.],t[2]) @ 0.001
```

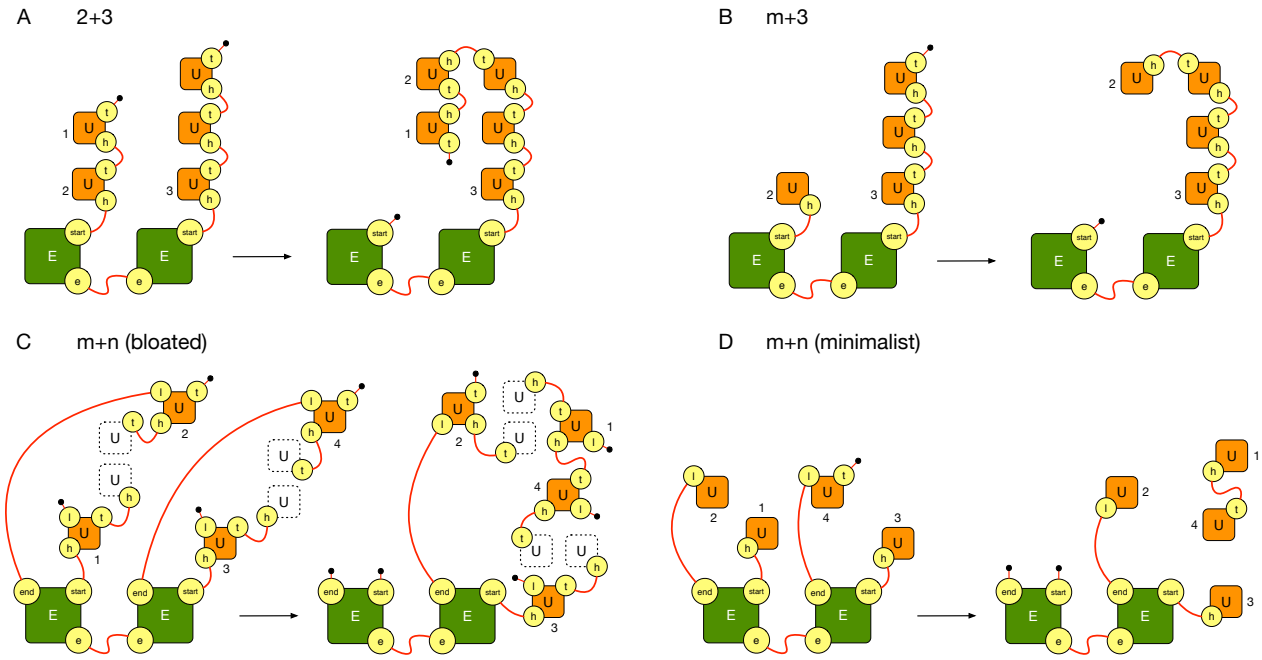


Figure 6: Transfer of chains. **A:** The transfer of a chain from one agent `E` onto the end of the chain of another proceeds by head-to-tail concatenation (rule `'2+3'`). **B:** The transfer of a chain of length *m* onto a chain of length 3. **C:** The transfer of one chain onto another when both have unspecified lengths requires that the agents `E` keep information about the tail ends of their chains. **D:** A minimalist rule for concatenating two chains of arbitrary length. Note that the right hand side consists of two disconnected patterns although the molecular species produced by the underlying reaction is connected.

It is worth emphasizing again that `'m+n bloated'` and `'m+n minimalist'` are not the *same* rule, despite both concatenating two strings of arbitrary length. They express the same principle, but they don't express the same mechanism. For example, `'m+n bloated'` *requires* that sites `l` and `t` of agents U_1 and U_2 be free, whereas `'m+n minimalist'` does not. This innocuous seeming difference may have consequences when the rules act in the context of others. Some other rule might bind an agent `C` to site `l` of U_1, in which case subsequent concatenations would be blocked under `'m+n bloated'` but not under `'m+n minimalist'`. While this might be unlikely when `U` is ubiquitin, it should convey appreciation for the idea

that the conditions for interaction expressed by a rule—i.e. the states tested but not modified—are as important as the action itself (the modification or difference between left and right side).

2.3.2 Edit notation

In the edit notation (Grammar 4), state modifications are directly indicated for each site with a "before" / "after" syntax; agent destruction (creation) is annotated with a '−' ('+') sign following the respective agent. The edit notation has no bi-directional rules; these must be written as two uni-directional ones.

Grammar 4: Rule expressions in edit notation

⟨*f-rule*⟩	::=	[⟨*Label*⟩] ⟨*f-rule-expression*⟩ [| ⟨*token*⟩] @ ⟨*rate*⟩
⟨*ambi-rule*⟩	::=	[⟨*Label*⟩] ⟨*f-rule-expression*⟩ [| ⟨*token*⟩] @ ⟨*rate*⟩ { ⟨*rate*⟩ }
⟨*f-rule-expression*⟩	::=	⟨*agent-mod*⟩ ⟨*more-agent-mod*⟩
	|	ε
⟨*more-agent-mod*⟩	::=	, ⟨*agent-mod*⟩ ⟨*more-agent-mod*⟩
	|	ε
⟨*agent-mod*⟩	::=	⟨*agent-name*⟩ (⟨*interface-mod*⟩)
	|	⟨*agent-name*⟩ (⟨*interface*⟩) (+ | −)
⟨*site-mod*⟩	::=	⟨*site-name*⟩ ⟨*internal-state-mod*⟩ ⟨*link-state-mod*⟩
	|	⟨*site-name*⟩ ⟨*link-state-mod*⟩ ⟨*internal-state-mod*⟩
	|	⟨*counter-name*⟩ ⟨*counter-state-mod*⟩
⟨*interface-mod*⟩	::=	⟨*site-mod*⟩ ⟨*more-mod*⟩
	|	ε
⟨*more-mod*⟩	::=	, ⟨*site-mod*⟩ ⟨*more-mod*⟩
	|	ε
⟨*internal-state-mod*⟩	::=	{ (⟨*state-name*⟩ | #) / ⟨*state-name*⟩ }
	|	{ (⟨*state-name*⟩) }
	|	ε
⟨*link-state-mod*⟩	::=	[(⟨*number*⟩ | . | _ | ⟨*site-name*⟩.⟨*agent-name*⟩ | #) / (⟨*number*⟩ | .)]
	|	⟨*link-state*⟩
	|	ε
⟨*counter-state-mod*⟩	::=	{ ⟨*counter-expression*⟩ / ⟨*counter-mod*⟩ }
	|	{ ⟨*counter-expression*⟩ }
	|	{ ⟨*counter-mod*⟩ }
⟨*rate*⟩	::=	⟨*algebraic-expression*⟩

2.3.2.1 Examples

Rehashing a few of the examples given in the arrow notation should convey a sense for the edit notation.

- `'asymmetric dimerization' A(x[./1]),A(y[./1]) @ 0.001`
- `'degrade' A(x[1])−, A(y{p}[1/.]) @ 0.001`
- `'create' A(x[./1]), A(y{p}[1])+ @ 0.001`

- `'many modifications at once'`
  `A(y{#/p}[1/7]), D(q[0])+, B(x[1/0],y[_/7]), C()-  @ 0.001`
- `'dephospho' S(x{p/u}) @ 0.001`
- `'also side effect' A(x[y.B/.]) @ 0.001`
- `'ambiguous molecularity' A(x[./1]), A(y[./1]) @ 0.001 {0.1}`
- `'m+n minimalist'`
  `E(e[1],start[2/.],end[3/.]),U(h[2]),U(l[3/5]),`
  `E(e[1],start[6],end[5]),U(h[6]),U(l[5/.],t[./2]) @ 0.001`

2.3.3 Counters

The core of Kappa is deliberately small, if not minimal: Agents, sites, and two types of actions and their reverse (binding and internal state modification)[3]. One consequence of this terseness is the possibility of formal analysis; one drawback is that some biophysical aspects of importance to biology are not captured "naturally". Expressing certain dependencies can still be combinatorial. For example, in core Kappa, one cannot directly express a pattern in which, say, `A` is phosphorylated at 5 of its 10 sites. Consequently, there is no direct way of asserting that "`A` binds `B` if 5 of 10 sites of `A` are phosphorylated". It appears that one might have to list all "10 choose 5" (=252) combinations as separate rules. In actuality it is not that bad if one tolerates "encoding", which is the use of agents, sites, and rule actions that have nothing to do with biological mechanisms but only with accounting mechanisms, i.e. with managing "counters". For example, to implement a counter that tracks the number of phosphorylated sites of an agent, one can attach a chain of dummies to an agent and add or remove a dummy alongside a phosphorylation or dephosphorylation event, respectively. In this way, "`A` binds `B` if 5 of 10 sites of `A` are phosphorylated" is a rule whose left hand side exhibits a pattern with a chain of exactly 5 dummies. This, of course, is tedious to do by hand. To avoid this headache, Kappa provides an *experimental* construct ⟨*counter*⟩, whose grammatical terms show up in Grammars 2 and 4. At present, the simulator internally translates counter constructs into chains just as described (but the static analyzer KaSa deals with them abstractly) This construction does not cause much of an overhead during simulation—especially with the latest algorithms implementing the Kappa simulator—and is hidden from the user's view in state dumps (section **??**) and causal analysis (section **??**).

Because of the experimental character of counters at the time of writing, their description is consigned to appendix C.

2.4 Kappa Declarations

Sections 2.1–2.3 covered the Kappa language in the narrow sense, which is mainly concerned with graph-rewriting. The core language is suited for reasoning formally about static properties (section **??**) of rule-based models. One purpose of the Kappa platform is to enable the simulation of models and the exploration of their dynamic properties. Unlike in static analysis, rate constants play a role in simulation and we need more flexible ways of defining rates. Moreover, we need to specify observables, initial conditions, and, importantly, interact with the model by specifying interventions both scheduled in advance and in real-time. In short, we need to be able to *experiment* with the model. This is the purpose of Kappa declarations, which, together with the core language, constitute Kappa in the wider sense.

[3] One could dispense with the distinction between binding and internal state modification and consider the latter as a form of un/binding that (typically) requires a third agent—an enzyme—to occur. However, this difference is sufficient to warrant state modification to be a separate action.

2.4.1 Variables, algebraic expressions, and observables

Many components of KF rely on the declaration of variables. For example, variables might be used as model parameters: If a user redefines the system volume, stochastic rate constants of bimolecular interactions need to scale inversely with the volume. It would be useful, then, to be able to declare the volume as a variable that can show up in an algebraic expressions defining a rate constant.

A variable is declared with the `%var:` directive (Grammar 5).

Grammar 5: Variable declaration

⟨*variable-declaration*⟩ ::= %var: ⟨*declared-variable-name*⟩ ⟨*algebraic-expression*⟩
⟨*declared-variable-name*⟩ ::= ⟨*Label*⟩ // not ⟨*Name*⟩

The construction of an ⟨*algebraic-expression*⟩ is defined by Grammar 7.

►
```
// declare a variable
%var: 'pattern matchings' |A(x[1]),A(x[1])|
%var: 'homodimers' 'pattern matchings' / 2
```

The first declaration defines a variable `'pattern matchings'` as the number of occurrences of the pattern `A(x[1]),A(x[1])`. This pattern has a symmetry (section 3.2). Whatever site graph, such as a molecular species, the first `A` matches, the second can match too. As a result there will be twice as many matchings than objects of interest. We therefore correct this by dividing the variable `'pattern matchings'` by 2 and call the new variable `'homodimers'`. Of course we could have divided by 2 in the first declaration.

Any variables used in the declaration of another variable must be declared beforehand. Variables can be used in rate expressions (section 3.8).

Grammar 6: Observable declarations

⟨*plot-declaration*⟩ ::= %plot: ⟨*declared-variable-name*⟩
⟨*observable-declaration*⟩ ::= %obs: ⟨*Label*⟩ ⟨*algebraic-expression*⟩

The value of a variable can be written to an output file (section **??**) using the `%plot:` or `%obs:` declarations:

►
```
// print a variable
%plot: 'homodimers'
```

An `%obs:` directive declares an observable, which is a shortcut for both declaring a variable and plotting it.

►
```
// declare an observable (var+plot shortcut)
%obs: 'homodimers'  |A(x[1]),A(x[1])| / 2
```

2.4.2 Agent signatures

A signature, Grammar 9, defines the interface of an agent type, i.e. its full complement of sites, including all internal state values that are possible for each site and all potential binding partners. *It also defines*

Grammar 7: Algebraic expression

⟨*algebraic-expression*⟩ ::= ⟨*float*⟩
| ⟨*defined-constant*⟩
| ⟨*declared-variable-name*⟩ // variable must be declared using `%var`
| ⟨*reserved-variable-name*⟩
| ⟨*algebraic-expression*⟩ ⟨*binary-op*⟩ ⟨*algebraic-expression*⟩
| ⟨*unary-op*⟩ (⟨*algebraic-expression*⟩)
| [max] (⟨*algebraic-expression*⟩) (⟨*algebraic-expression*⟩)
| [min] (⟨*algebraic-expression*⟩) (⟨*algebraic-expression*⟩)
| ⟨*boolean-expression*⟩ [?] ⟨*algebraic-expression*⟩ [:] ⟨*algebraic-expression*⟩

⟨*reserved-variable-name*⟩ ::= [E] // productive events since simulation start
| [E-] // number of null events
| [T] // simulated physical time
| [Tsim] // cpu time since simulation start
| | ⟨*declared-token-name*⟩ | // concentration of token
| | ⟨*pattern-expression*⟩ | // occurrences of pattern
| inf // denotes ∞

⟨*binary-op*⟩ ::= +
| −
| *
| /
| ^
| [mod]

⟨*unary-op*⟩ ::= [log]
| [exp]
| [sin]
| [cos]
| [tan]
| [sqrt]

⟨*defined-constant*⟩ ::= [pi]

⟨*float*⟩ ::= $x \in \mathbb{R}$

Grammar 8: Boolean expression

⟨*boolean-expression*⟩ ::= ⟨*algebraic-expression*⟩ (= | < | >) ⟨*algebraic-expression*⟩
| ⟨*boolean-expression*⟩ || ⟨*boolean-expression*⟩
| ⟨*boolean-expression*⟩ && ⟨*boolean-expression*⟩
| [not] ⟨*boolean-expression*⟩
| ⟨*boolean*⟩

⟨*boolean*⟩ ::= [true]
| [false]

counters. The signature section of a KF lists the signatures of all agents that appear in the rule section.

Unless one uses counters, the signature is not essential information, since the simulator `KaSim` determines the agent signatures directly from the rules in the KF.

⚠ Careful: The signature section is "all or nothing": if it is defined for one agent type, it must be defined for all agent types.

The point of a signature is to provide readability and a simple semantic check. If a user defines an agent signature, `KaSim` can compare it to the signature it derives from the rules and issue an error upon mismatch. Leaving out the signature section is perfectly admissible, but a simple mistake in which the user's writing differs from the user's intention might go unnoticed by the parser and can be difficult to discover.

Grammar 9: Agent signature

⟨*signature-declaration*⟩ ::= %agent: ⟨*signature-expression*⟩
⟨*agent-name*⟩ ::= ⟨*Name*⟩
⟨*site-name*⟩ ::= ⟨*Name*⟩
⟨*state-name*⟩ ::= ⟨*Name*⟩
⟨*signature-expression*⟩ ::= ⟨*agent-name*⟩ (⟨*signature-interface*⟩)
⟨*signature-interface*⟩ ::= ⟨*site-name*⟩ ⟨*set-of-internal-states*⟩ ⟨*set-of-link-states*⟩ ⟨*more-signature*⟩
| ⟨*site-name*⟩ ⟨*set-of-link-states*⟩ ⟨*set-of-internal-states*⟩ ⟨*more-signature*⟩
⟨*site-name*⟩ { = ⟨*integer*⟩ / += ⟨*integer*⟩ }
⟨*more-signature*⟩ ::= [,] ⟨*signature-interface*⟩
| ε
⟨*set-of-internal-states*⟩ ::= { ⟨*set-of-state-names*⟩ }
| ε
⟨*set-of-state-names*⟩ ::= ⟨*state-name*⟩ ␣ ⟨*set-of-state-names*⟩
| ε
⟨*set-of-link-states*⟩ ::= [⟨*set-of-stubs*⟩]
| ε
⟨*set-of-stubs*⟩ ::= ⟨*site-name*⟩ . ⟨*agent-name*⟩ ␣ ⟨*set-of-stubs*⟩
| ε

For example, a signature line in the KF might read

```
%agent: A(x[a.B y.A]{u p}, y[x.A], z{z1 z2 z3})   // sig of A
%agent: B(a[x.A],c{= 0 / += 10})                  // sig of B
```

This signature will declare an agent-type `A` whose site `x` can bind agents of type `B` on site `a` or agents of type `A` (the same agent instance or a different instance of the same type) on site `y`. The declaration of potential binding partners in the signature is a situation where the binding type notation is deployed. Moreover, site `x` can hold an internal state of two possible values, `u` and `p`. Likewise, site `z` can hold an internal state with three possible values `z1`, `z2` and `z3`. (Internal state names cannot start with a number, Grammar 1.) A counter site `c` is declared for agent `B`.

⚠ Careful: The use of binding types in the signature is also "all or nothing": If none of the agent signatures specifies a binding type, `KaSim` will not cross-check signature and rules with regard to binding

consistency. In other words, any site of any agent is allowed to bind any site of any agent as far as semantic checks go. If some agent signature specifies a binding type, then all signatures must specify all binding types.

If you are sure that your binding rules express your intentions, you could write the above signature in abbreviated form:

►
```
%agent: A(x{u p}, y, z{z1 z2 z3})   // sig of A
%agent: B(a,c{= 0 / += 10})         // sig of B
```

If `B` had no counter, you could omit the signature entirely and forgo any semantic sanity check.

The agent signature plays one additional significant role: it defines the *default* state of an agent when it is created (produced) afresh either by a rule or in an initialization declaration (section 2.4.3) when no further specifications override it. The default internal state is the first value appearing in the internal state list of a site. The default binding state is always "free".

2.4.3 Initial conditions

For a model to behave dynamically, a set of agents initially present must be specified. We imagine a pool of agents or complexes that constitute at any given time the state of the system. Rules apply stochastically to that state (section 3.4), transitioning it to a new state. Transition by transition this process produces a trajectory of events. The declaration of an initial state is simple and follows Grammar 10.

Grammar 10: Initial condition

⟨*init-declaration*⟩ ::= %init: ⟨*algebraic-expression*⟩ ⟨*pattern*⟩
| %init: ⟨*algebraic-expression*⟩ ⟨*declared-token-name*⟩

The ⟨*algebraic-expression*⟩ is evaluated prior to the start of the simulation. Hence, all Kappa expressions (and tokens, section 2.4.5) are evaluated to 0. The `%init:` declaration will add as many copies of ⟨*pattern*⟩ to the system as ⟨*algebraic-expression*⟩ evaluates to. These copies are added in their default state as specified in the respective agent signatures (see end of section 2.4.2), unless aspects of the default state are overridden in the initialization.

►
```
%agent: A(x{u p}, y[x.B, y.C], z{z1 z2 z3})  // sig of A
%var: 'n' 150
%init: 'n' A()
%init: 'n' A(z{z3})
```

In this example, the creation of 150 instances of `A()` amounts to 150 instances of the molecular species in its signature-specified default state `A(x{u}[.],y[.],z{z1}[.])`. (If a site has no binding interactions, it is obviously free at all times.) The second initialization of `A` overrides the default at site `z` from `z1` to `z3`. Agent numbers add up if `%init:` is used multiple times. In this example, the initial state consists of 300 agents of type `A`, 150 of which have their site `z` in state `z1` and the other 150 in state `z3`.

2.4.4 Parameters

In section **??** we discuss the `KaSim` command in more detail. `KaSim` has a number of options that can be set on the command line but also from within the KF. For the sake of syntactical completeness we provide the syntax of simulation parameter definitions here (Grammar 11). Usage examples and a table of reserved

names and value ranges can be found in section **??**.

Grammar 11: Parameters

⟨*parameter-setting*⟩	::=	%def: ⟨*parameter-name*⟩ ⟨*parameter-value*⟩
⟨*parameter-name*⟩	::=	reserved names listed in table **??**
⟨*parameter-value*⟩	::=	defined range associated with each ⟨*parameter-name*⟩, table **??**.

2.4.5 Tokens and hybrid rules

At the level of abstraction captured by Kappa, biomolecular processes can span several time and concentration scales. For example, molecular species, such as ATP, redox couples like $NADP^+$/NADPH, or various ions are often abundant relative to those agents that are of principal mechanistic interest in a model. Kappa provides a way of treating such species as pool variables with continuous values. They are called tokens and give rise to "hybrid rules" in which the mechanistic part of the rule is linked to a *change* in token values: Each time a rule fires it not only modifies a graph pattern but also causes an update in token values. Tokens are structureless species (basically a proper name) and their value can be referred to in rate expressions.

Grammar 12: Tokens

⟨*token*⟩	::=	⟨*algebraic-expression*⟩ ⟨*declared-token-name*⟩ ⟨*another-token*⟩
⟨*another-token*⟩	::=	, ⟨*token*⟩
	|	ε
⟨*declared-token*⟩	::=	%token: ⟨*declared-token-name*⟩
⟨*declared-token-name*⟩	::=	⟨*Name*⟩

An example might clarify the intent.

```
%token: ATP
%token: ADP
%init:  9.6  ATP  // mM per E. coli cell, about 1E7 molecules
%init:  0.55 ADP  // mM per E. coli cell, about 1E6 molecules
// arrow notation
'hybrid rule a' S(x{u}[1]),K(y[1]) ->
                S(x{p}[1]),K(y[1])   | -1E-6 ATP +1E-6 ADP @ 'k'
// edit notation
'hybrid rule e' S(x{u/p}[1]),K(y[1]) | -1E-6 ATP +1E-6 ADP @ 'k'
```

The names `ATP` and `ADP` are declared as tokens and initialized much like structured agents. In this example we imagine a kinase `K` phosphorylating a substrate `S` to which it is bound in an *E. coli* cell. The concentration of ATP in such a cell is about 9.6 mM, which corresponds roughly to 10^7 molecules; the concentration of ADP is roughly 0.55 mM (about 10^6 molecules). Upon phosphorylation, 1 molecule (roughly 1 nM = 10^{-6} mM) of ATP is consumed and 1 molecule of ADP produced. Thus the change in terms of mM values is -10^{-6} for `ATP` and $+10^{-6}$ for `ADP`, as

indicated in the hybrid rule after the vertical bar.

A few points should be kept in mind at the current state of implementation:

- The propensity of the hybrid rule to fire (section 3.3) does not automatically depend on the concentration of tokens even though they are mentioned. If a user wishes to create a dependency, then the rate constant must be an explicit function of the token concentration.
- The simulator does not perform any checks on whether the token concentration stays non-negative.
- The change of a token can be an ⟨*algebraic-expression*⟩, which means that tokens can change as a function of anything else in the system, including other tokens. Use with caution, as a model should be transparent.
- Tokens are an experimental feature. Please give feed-back if you run into problems.

2.5 Intervention directives

The simulator `KaSim` executes an event loop (section 3.4) whose basic cycle consists of advancing the simulated wall-clock time, selecting a rule for application, and computing updates to reflect the result. It is useful being able to intervene in a simulation by scheduling perturbations, such as injecting a certain number of agents of a given type or modifying a variable, or invoking data reporting tasks. To this end, Kappa provides a mini-language for specifying interventions and their timing. Some of these constructs can also be used to interact with `KaSim` while the simulator is running. The syntax is given in Grammar 13.

Grammar 13: Intervention directives

⟨*effect-list*⟩ ::= ⟨*effect*⟩ ; ⟨*effect-list*⟩ | *ε*

⟨*effect*⟩ ::= $ADD ⟨*algebraic-expression*⟩ ⟨*pattern*⟩
| $DEL ⟨*algebraic-expression*⟩ ⟨*pattern*⟩
| $APPLY ⟨*algebraic-expression*⟩ ⟨*rule-expression*⟩ [| ⟨*token*⟩]
| $SNAPSHOT ⟨*string-expression*⟩
| $STOP ⟨*string-expression*⟩
| $DIN ⟨*string-expression*⟩ ⟨*boolean*⟩
| $TRACK ⟨*label*⟩ ⟨*boolean*⟩
| $UPDATE ⟨*var-name*⟩ ⟨*algebraic-expression*⟩
| $PLOTENTRY
| $PRINT ⟨*string-expression*⟩ > ⟨*string-expression*⟩
| $SPECIES_OFF ⟨*string-expression*⟩ ⟨*pattern*⟩ ⟨*boolean*⟩

⟨*string-expression*⟩ ::= *ε*
| `string` . ⟨*string-expression*⟩
| ⟨*algebraic-expression*⟩ . ⟨*string-expression*⟩

⟨*intervention*⟩ ::= %mod: (*ε* | alarm ⟨*float*⟩) ⟨*boolean-expression*⟩ do ⟨*effect-list*⟩ repeat ⟨*boolean-expression*⟩

An ⟨*intervention*⟩ is declared using `%mod:` and specifies the temporal granularity with which `KaSim` checks the subsequent ⟨*boolean-expression*⟩ whose satisfaction triggers execution of ⟨*effect-list*⟩. This basic scheme repeats as long as the ⟨*boolean-expression*⟩ following the `repeat` keyword is satisfied. The temporal granularity of the checks comes in two flavors: either at specified intervals of (simulated) time or at each event. A few examples will clarify the broad scope of the intervention grammar.

2.5.1 Timing and conditioning of interventions

- ```
  // test at a specified time
  %mod: alarm 2.3 [true] do $PLOTENTRY; repeat [true]
  ```

  Here the `alarm` keyword is followed by a real-valued number specifying the time interval between checks of the subsequent boolean condition, here `[true]`. Thus, the directive means that KaSim should check after 2.3 simulated time units whether `[true]` is true—which of course it is—and execute the effect `$PLOTENTRY`, i.e. print a line to the output file with the values of all variables declared for plotting (`%plot`) and all observables (`%obs`). This will occur every 2.3 time units as long as `[true]` is true, which is forever.

- ```
  // test at each event
  %mod: |A()| > 1000 do $PLOTENTRY; repeat |B(x[_])| < |B(x[.])|
  ```

  When the `alarm` keyword is omitted the subsequent condition is checked at each event cycle. In our example, if the number of agents of type `A` is greater than 1,000 a `$PLOTENTRY` is executed, and this directive repeats until the number of `B`s bound at site `x` exceeds those that are unbound. Note that the `$PLOTENTRY` action could be triggered intermittently, as long as `|B(x[_])| < |B(x[.])|`, because the number of `A`s might fluctuate around 1,000. However, once the repeat condition is violated, the directive ceases to be active forever after.

- Omitting the `repeat` keyword (and the associated condition) is equivalent to `repeat [false]`. This defines a "one-shot" intervention, which makes most sense if the timing of checks is specified by an `alarm`. Otherwise the condition is checked at the first event, which is of limited utility.

  ```
  // conditional one-shot directive
  %mod: alarm 5 |A()| > 1000 do $PLOTENTRY;
  // unconditional one-shot directive
  %mod: alarm 5 do $PLOTENTRY;
  ```

  The first code snippet means that a print is generated after the first time interval of length 5, if `A` is present in more than 1,000 copies. In essence this directs `KaSim` to print all observable values at `[T]=5`, where `[T]` is the ⟨*reserved-variable-id*⟩ holding the current simulated wall-clock time (Grammar 7). The second code snippet omits the condition following the timing specification. This is tantamount to setting it to `[true]`. Thus, at `[T]=5` a `$PLOTENTRY` is executed.

2.5.2 Model perturbation

- ```
  // model perturbation: adding and deleting agents
  %var: 'n' |B(x[.])|
  %mod: alarm 10.0 do $ADD 'n' C(x1{p}[.]); /* add some C */
  %mod: alarm 15.5 do $DEL |B(x[_])| B(x[_]); // delete all bound B
  ```

  Here we perturb the system at time `[T]=10.0` by injecting as many agents of type `C` as there are unbound copies of `B` at that moment. The additional `C`-agents are specified to be unbound and in state `p` at site `x1`. This perhaps contrived intervention illustrates the use of variables in intervention directives. The subsequent directive deletes all instances of bound `B`-agents present at `[T]=15.5`.

- ```
  // timing is everything
  %mod: [T] = 10.0 do $ADD 1000 C(); /* no no */
  %mod: alarm 10.0 do $ADD 1000 C(); /* yes yes */
  ```

  Once more: Without the `alarm` keyword, the directive's condition (`[T]=10`) is checked at every

event. It is extremely unlikely that an event occurs at a simulated time `[T]` that exactly equals a pre-specified value. Thus, the condition will never be satisfied. If a perturbation is needed at an exactly specified point in simulated time, use the `alarm` construction.

```
// model perturbation: updating variables
%obs: 'Cpp' |C(x1{p},x2{p})|
%var: 'k' 0.001
%mod: 'Cpp' > 500 do $UPDATE 'k' 'k'/100;  // change variable
%mod: 'Cpp' > 500 do $UPDATE 'rule 1' inf; // change rate constant
```

The intervention grammar allows us to update values of variables. In the first one-shot directive, the first time the number of doubly phosphorylated `C` agents exceeds 500, the variable `'k'` (which could be a rate constant) is decreased hundredfold. The second directive refers to the rate constant of `'rule 1'` and sets it to infinity.

```
// apply a rule at a specified time
%mod: alarm 5 do $APPLY |C(x{p})| C(x{p})->C(x{u}); repeat [true]
// change the value of a token at a specified time
%mod: alarm 15 do $APPLY 1 | -|money| money;
```

The effect of `$APPLY` is to apply a rule a specified number of times at a specified instant. In the first case, the rule is applied every 5 time units as many times as there are agents of type `C` whose site `x` is in state `p`. Note that the rule is applied deterministically, i.e. it has no propensity. If a rule cannot be applied at runtime, a warning is issued but no fail condition ensues. The second example deploys a rule that specifies no pattern transformation but exploits the token section to reset a token value after the first 15 time units have elapsed. This is useful since only variables, not token values, can be set with the `$UPDATE` effect.

2.5.3 Model observation

```
// interrupt
%mod: alarm 25 do $STOP 'dump.ka'; // interrupt run and dump state
```

The directive `$STOP` interrupts the simulation. If the simulator is running in batch mode, this will terminate the simulation. If the simulator is running in interactive mode, control returns to the user, who could issue intervention directives by hand and resume the simulation (section **??**). The ⟨*Label*⟩ after the `$STOP` is the optional name of an output file. If a name is given, the state of the system (i.e. the set and count of molecular species present at that time) is written to a file so named.

```
// one-time snapshots of the system state
%mod: [E-]/([E-]+[E]) > 0.9 do $SNAPSHOT "my_prefix";
```

The directive `$SNAPSHOT` produces a snapshot of the system state (which is what happens after the `$STOP` directive in the previous example). The snapshot specifies the full state of all agents present in the system at that point in time. Each line in the snapshot output consists of a molecular species (i.e. a connected component of the system graph) along with the count of its occurrences and, as a comment, the size of the species in terms of agents. Output lines are constructed in such a way that the snapshot file can be used as the initialization component of a KF.

```
// Snapshot [Event: 2069]
%def: "T0" "10"
%init: 743 /*2 agents*/ B(x[1]), A(x[1] c[.])
%init: 257 /*1 agents*/ B(x[.])
```

```
%init: 257 /*1 agents*/ A(x[.] c[.])
```

For example, one could run a model to steady state, then dump a snapshot and use that file as a representative initial condition for subsequent simulations. This might save time when investigating the response of a model system to a variety of perturbations that are applied at steady state. In the example above the snapshot is produced once as soon as the fraction of non-productive events has exceeded 90%. The name of the snapshot file is "my_prefix_*n*.ka", where *n* is the event number at the time of the snapshot. The "my_prefix" string can be constructed by string concatenation (`.`) using local variables. When the prefix is omitted, the snapshot filename defaults to "snap.ka".

- ```
  // periodic snapshots of the system state
  %mod: ( [E] [mod] 1000 ) = 0 do
           $SNAPSHOT "abc.ka"; $SNAPSHOT "abc.dot";
        repeat [true]
  ```

  This directive will produce a snapshot every 1,000 productive events. If the filename already exists, a counter is automatically appended to prevent overwriting. In this case, the result will be a series of files named "abc_*n*.ka" with $n = 1, 2, \ldots$. The second invocation of `$SNAPSHOT` uses the filename extension "dot" to instruct KaSim to output a file in a format that can be processed by `graphviz` or similar program, visualizing (however crudely at present) the molecular species populating the system. The extension "html" is also an option.

- As mentioned, `$PLOTENTRY` prints the current value of observables. In the example below, `$PRINT` ⟨*string-expression*⟩ `>` ⟨*string-expression*⟩ outputs the first string to a file named by the second string.

  ```
  // print to file
  %token: A
  %mod: |A| < 0 do
           $PRINT ("A is: ".|A|." at time=".[T]) > ("A_".[E].".dat");
        repeat [true]
  ```

  In this case, the first string expression is assembled using variables and string concatenation. A new filename is assembled using the event number `[E]` each time the value of token `A` drops below zero (and a print event is generated). Omitting the filename and the '>' character defaults to standard out.

- The following toy example illustrates the joint operation of multiple intervention directives. Imagine the need to gather simulation data over a window of time set by the variable `'interval'` and to record these data in a file. Moreover, we wish to slide this window by `'shift'` time units, each time starting a new file and closing the file that has been recording for `'interval'` time units. This should continue until the last file opens at time `'Tend'`. Such a situation arises in section **??** when we record data for the Dynamic Influence Network (DIN) to be visualized as a "movie". Here we simply "open" and "close" files verbally by writing a reporting string to standard out.

  ```
  // sliding data window and output
  %var: 'Tend'     100
  %var: 'shift'      0.5 // in time units
  %var: 'interval'   3   // in clock beats
  %var: 'tick'       0   // auxiliary
  %var: 'clock'      0   // auxiliary

  %mod: [T] > 'clock' && 'tick' > 'interval' - 1 do
          $PRINT "close file ".'tick'-'interval'." at time ".[T];
        repeat 'tick' < 'Tend'/'shift' + 'interval'
  ```
  ```

```
%mod: [T] > 'clock' do
        $PRINT "open file ".'tick'." at time ".[T];
      repeat 'tick' * 'shift' < 'Tend'
%mod: [T] > 'clock' do
        $UPDATE 'clock' 'clock'+'shift'; $UPDATE 'tick' 'tick'+1;
      repeat 'tick' < 'Tend'/'shift' + 'interval'
```

The directive conditions are checked at each event (no `alarm` keyword). When multiple `%mod:` directives are present, their conditions are tested and effects executed in the order in which they are declared in the KF. The `repeat` condition is evaluated at each event after execution and determines whether the directive is eligible for consideration at the next event cycle. A sample output follows.

```
open file 0 at time 0.00143794913206
open file 1 at time 0.501596099326
open file 2 at time 1.00261155525
close file 0 at time 1.50649098743
open file 3 at time 1.50649098743
close file 1 at time 2.00398432147
...
open file 200 at time 100.002636571
close file 198 at time 100.51907549
close file 199 at time 101.001209551
close file 200 at time 101.509098845
```

Other directives listed in Grammar 13 require more background and their description is deferred to section **??**. A directive is a semi-colon separated list of effects, so don't forget the semi-colon after *each* effect.

2.5.4 Hello ABC, modified

In many scenarios it makes sense to let some components of a system reach steady state before studying the effect of introducing other components. For example, a signal molecule might affect a system that has attained steady-state in its absence. We illustrate this in the ABC model (section 1) by letting the binding between `A` and `B` equilibrate before adding `C`. This is easily achieved by splicing in the following code snippet.

```
42 // Initial condition
43
44 %init: 1000  A(), B()
45 %init:    0  C(x1{u},x2{u})
46
47 // Intervention
48
49 %mod: alarm 25 do $ADD 10000 C();
```

The amount of `C` is set to zero initially and an intervention directive is added that injects 10,000 copies of `C` in its default state, as specified in its signature, at time 25—which allows plenty of time for the complex formation between `A` and `B` to equilibrate (Figure 3). The result is seen in Figure 7.

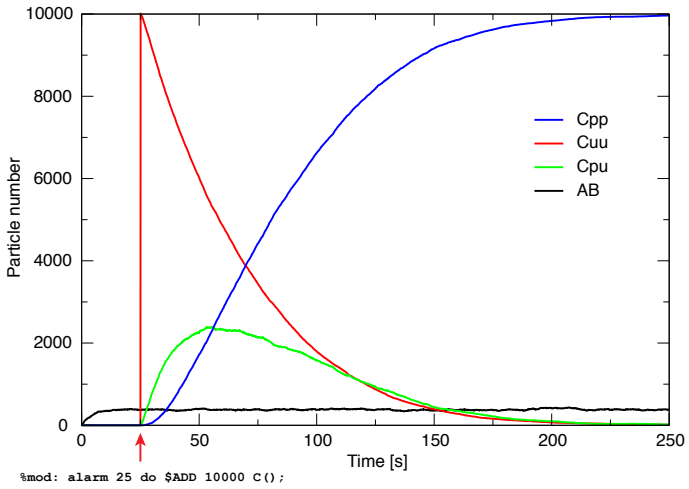


Figure 7: Intervening in the ABC model. 10,000 `C`s are injected after 25 s simulated time into an equilibrated mixture of `A`- and `B`-agents.

3 Simulation

In this section we cover some elementary aspects of stochastic simulation as they pertain to rules. This background should help to better understand how Kappa rules work and what happens when a model is simulated by `KaSim`. For details, please consult any of the numerous textbooks and reviews that cover stochastic chemical kinetics and Monte-Carlo in the context of reaction networks.

The simulator `KaSim` is given an input file (a KF), which contains among other things (section 2) the specification of a model and an initial condition. The initial condition is a collection of *molecular species* represented in Kappa. Recall that a molecular species is a Kappa expression that specifies the full state of all agents explicitly; it is the degenerate case of a pattern that leaves nothing unspecified. A collection of molecular species is called a *mixture*. In the context of a model, the mixture is the state of the system. The state changes because of events that occur. An *event* is the reaction induced by the application of a rule to the mixture. The *application* of a rule is based on *matching* the pattern on its left-hand-side to the mixture and executing the transformation specified by the rule. The choice of which rule to apply where in the mixture and when is stochastic and follows a scheme known as *continuous time Monte-Carlo*—CTMC for short, but often simply referred to as "Gillespie algorithm" (section 3.4). The basic tenet of CTMC is that the conditional probability of a specific interaction occurring between time T and $T + t$, given that it did not occur up to T, is independent of T. An event therefore marks time and `KaSim` simulates the physical time T *of* the system by advancing it from event to event. `KaSim` implements the stochastic chemical kinetics induced by the rules of a model without requiring the (often unfeasible, if not outright impossible) explicit enumeration of all implied reactions.

3.1 Matching

We refer to a match also as an *embedding* of a graph, usually an observable or the left pattern of a rule, into a host graph, usually a mixture of molecular species—but it could be any site graph. The term embedding connotes that the host graph must be of equal size or larger than the pattern so that the pattern can "fit" into the host graph. If we disregard binding types for a moment (see below), an embedding or match is formally a subgraph isomorphism: Every node mentioned in the pattern must have a corresponding node of the same type in the host graph; for each node so matched, every site mentioned in its scope must have a corresponding site in the host node and for each site so matched, its state—whether internal or binding—must be the same or be less specific than the one of the host site. As far as internal state is

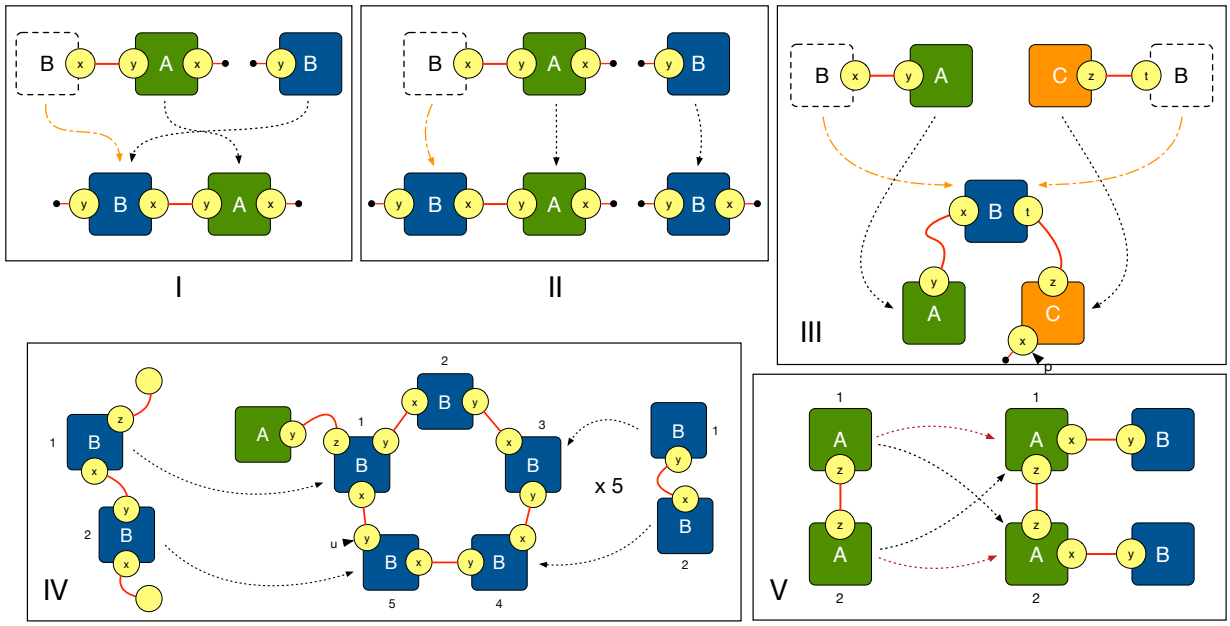


Figure 8: Embedding a graph into a host graph. Cases **I**—**III** illustrate the matching of binding types. In case **IV** the central graph is the host graph; its ring polymer is "oriented" by virtue of the `x`-`y` bonds. The pattern to the left has one embedding in the host graph because it requires `B` 1 to have a bound `z`-site. Note that the specification of site `y` does not care about its internal state and so it is compatible with the `u`-state in the host graph. In case **IV** the pattern on the right has 5 embeddings. In case **V** the pattern has a twofold symmetry and therefore two embeddings in the host shown. We discuss symmetry in section 3.2.

concerned, the wildcard (#) is less specific than a state identifier. As far as binding is concerned, the wildcard (#) is less specific than the underscore (_), which is less specific than a bond (or bond label in the line-oriented expression). A bond is matched by following the bond of a matched site in the host graph and extending the matching process from there. Implicit in all this is that any information not provided in the pattern poses no constraints to the match—don't care, don't write: match anything.

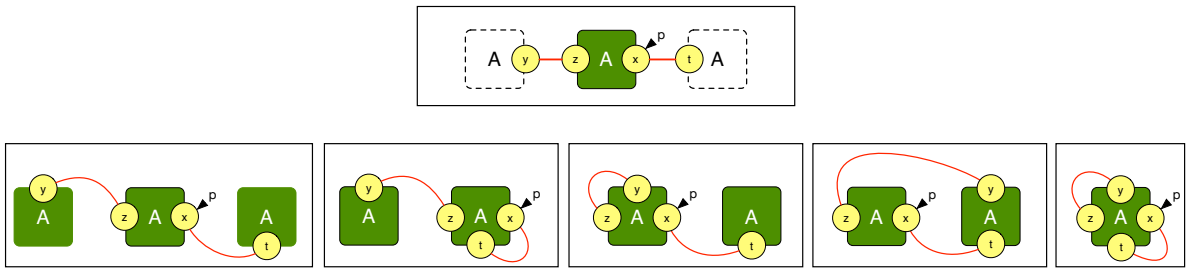


Figure 9: Embedding of binding types. The pattern at the top embeds in all patterns shown at the bottom.

Binding types (section 2.2) are treated differently in the matching process because they only specify type constraints and not resource (stoichiometric) constraints. A binding type matches a type of bound agent and the specified site, *even if that agent has already been matched*. Thus, the matching of binding types can be many-to-one (a homomorphism). A binding type is less specific than a labelled bond and more specific than the underscore (_).

Figure 8 provides some graphical examples, while Figure 9 centers on the matching of binding types.

3.2 Symmetry

A *symmetry* of a site graph is an embedding into itself, also called an automorphism. An example is the pattern in case V of Figure 8. If we permute the agent identifiers $\{1 \to 2, 2 \to 1\}$, we obtain a graph that is indistinguishable from the original. We say that `|A(z[1]),A(z[1])|` has two automorphisms: the identity (which maps the pattern into itself without shuffling identifiers, $\{1 \to 1, 2 \to 2\}$) and the permutation $\{1 \to 2, 2 \to 1\}$. Every pattern has the identity as a trivial symmetry. For example, the pattern to the right of case IV has only the trivial symmetry, because one `B` is bound at site `y` while the other is bound at site `x`. The left hand side of the rule in panel C of Figure 6 has one non-trivial symmetry (thus two in total), whereas the left pattern of the minimal rule in panel D has lost that symmetry because the `U` with identifier `4` must be free at site `t`, whereas no such condition is imposed on the `U` with identifier `2`.

The mixture, which is the (well-stirred) "soup" of molecules that constitute the state of the system at time t, should be thought of as one big graph in which molecules are connected subgraphs. In fact, `KaSim` does not "know" how many instances of a given molecular species are present at any time, unless the `$SNAPSHOT` directive is used. If the mixture contains 451 copies of a complex, these will be represented as 451 isomorphic graphs, i.e. connected subgraphs of the mixture-graph. Rather than counting species `KaSim` maintains, and cleverly administers, all embeddings from rules into the mixture graph.

Because the mixture is a graph whose nodes have not only names but also identifiers, it represents a "microstate": It contains everything one can possibly know about the system at the level of abstraction set by Kappa, including the identifiers. A more coarse-grained notion of state is one in which we disregard the identifiers, paying attention only to which graph components (i.e. molecules) are the same. This yields the notion of a "macrostate".

Two site graphs are indistinguishable if they are related by an isomorphism. A symmetry, or automorphism, is an isomorphism of a graph to itself, which is a *permutation* of the identifiers that yields a graph identical to the original *even when taking into account the identifiers*. An isomorphism establishes indistinguishability at the macro level (sameness), whereas an automorphism establishes indistinguishability at the micro level (identity). For example, Figure 14 in section 3.2 shows a microstate in which we can tell apart all molecules by virtue of their identifiers, whereas disregarding the identifiers yields a macrostate with three molecules of class I, two of class II and one of class III.

Symmetry awareness is important in Kappa models. For example, we encountered symmetry in section 2.4.1, where we defined the number of embeddings `|A(x[1]),A(x[1])|` in the mixture for the purpose of counting the instances of homodimers `A(x[1]),A(x[1])`. Clearly, we don't care about the agent identifiers when counting homodimer objects. However, by virtue of symmetry, the pattern `A(x[1]),A(x[1])` has two embeddings in any molecular species `A(x[1]),A(x[1])` contained in the mixture. We therefore need to compensate for the symmetry by dividing the number of embeddings by the total number of symmetries, here 2.

Symmetry shows up again in the context of rule activity, section 3.3.

⚠ Careful: `KaSim` only determines embeddings, not symmetries. *It is the user's responsibility to be symmetry aware.* While the static analyzer can detect symmetries, this feature is not yet integrated with `KaSim` and the UI.

3.3 Rule activity

The simulation core loop (section 3.4 below) selects a rule for application to a mixture $\mathcal{M}$ with a probability that is proportional to a quantity called the *rule activity* (or rule propensity). The activity of rule $i : \mathcal{L}_i \to \mathcal{R}_i \ @ \ \gamma_i$ is given by

$$\alpha_i = |\{\mathcal{L}_i \curvearrowright \mathcal{M}\}| \, \Omega_i \gamma_i, \tag{1}$$

where $\{\mathscr{L}_i \curvearrowright \mathscr{M}\}$ is the set of embeddings of $\mathscr{L}_i$ into $\mathscr{M}$ and $|\cdot|$ returns the size of the set; γ_i is the rate constant. The count $|\{\mathscr{L}_i \curvearrowright \mathscr{M}\}|$ takes care of mass-action kinetics by reporting the number of opportunities for the rule to apply in $\mathscr{M}$.

The factor Ω_i depends on one's interpretation of the "physics" underlying a model, as detailed next.

3.3.1 Symmetry and rule activity

Symmetry can affect rule activity (1) in two fundamental ways.

Stance 1 (ND): One stance is to consider only the symmetries of the left pattern $\mathscr{L}$. In this view, the focus is exclusively on the matching of $\mathscr{L}$. The assertion is that two embeddings of $\mathscr{L}$ that are related by symmetry constitute the same match; they are equivalent much like a square can be juxtaposed onto another in a number of indistinguishable ways due its rotational and reflectional symmetry. In this view, the factor Ω_i in (1) becomes

$$\Omega_i = \frac{1}{\omega_{\mathscr{L}_i}}, \tag{2}$$

where $\omega_{\mathscr{L}_i}$ is the number of symmetries of $\mathscr{L}_i$, the left pattern of rule i. (Recall that the identity is a trivial symmetry, so $\omega_{\mathscr{L}_i} \geq 1$.) The factor Ω_i compensates for counting *all* embeddings in $|\{\mathscr{L}_i \curvearrowright \mathscr{M}\}|$ of (1).

Stance 2 (D): The other stance considers the symmetries of the *rule*, by which we mean the symmetries of the left pattern $\mathscr{L}$ *that are preserved across the rule arrow* and still occur in the right pattern $\mathscr{R}$. Here the focus is not on whether two embeddings of $\mathscr{L}$ are indistinguishable (as in stance 1), but whether the resulting actions of the rule are indistinguishable. Two embeddings of $\mathscr{L}$ related by a symmetry that is preserved in $\mathscr{R}$ yield the same outcomes *in microscopically identical ways*. As such they should not be considered distinct actions. However, if the symmetry is broken in $\mathscr{R}$, the outcomes are achieved in microscopically distinguishable ways, see Appendix E for details. In other words: if the embeddings of $\mathscr{L}$ are related by a preserved symmetry, the outcomes are two identical mixtures (i.e. the same down to the identifiers); otherwise the outcomes are two isomorphic mixtures (i.e. the same if one disregards identifiers). In this view, the factor Ω_i in (1) becomes

$$\Omega_i = \frac{1}{\omega_{\mathscr{L}_i \to \mathscr{R}_i}}, \tag{3}$$

where $\omega_{\mathscr{L}_i \to \mathscr{R}_i}$ is the number of symmetries of the rule, i.e. the number of symmetries of $\mathscr{L}_i$ that are preserved in $\mathscr{R}_i$.

The two viewpoints have a natural interpretation that is best illustrated by example. Consider the following rules:

```
'1p'   A(x[1]{u}), A(x[1]{u}) -> A(x[1]{p}), A(x[1]{u}) @ k
'2p'   A(x[1]{u}), A(x[1]{u}) -> A(x[1]{p}), A(x[1]{p}) @ k
```

and the mixture $\mathscr{M}$ = `A(x[1]u),A(x[1]u)`. Both rules `'1p'` and `'2p'` have symmetric left patterns, but `'1p'` breaks that symmetry on its right side, whereas `'2p'` preserves it. The symmetry of the molecule(s) in the mixture is irrelevant, since a rule "perceives" molecules only through the mask placed on them by its left pattern. Upon adopting stance 1, rules `'1p'` and `'2p'` have the same activity ($k/2$) in $\mathscr{M}$. Upon adopting stance 2, rule `'1p'` has twice the activity of `'2p'` and twice the activity than under stance 1. The interpretation in chemical language is that the pattern `A(x[1]u),A(x[1]u)` has two "reaction centers" with respect to the action specified in rule `'1p'`: each `A`-agent can be a target of phosphorylation. In contrast, the same pattern has only one reaction center with respect to the action

specified in `'2p'`. If this seems a natural viewpoint, on what grounds is stance 1 defensible? Why should the activity of `'1p'` ever be the same as that of `'2p'`? The answer is to view rule `'1p'` as *non-deterministic* (hence the label ND): when its left pattern matches $\mathscr{M}$, the phosphorylation action of `'1p'` executes at random on one of the two `A`-agents related by symmetry. In contrast, stance 2 adopts the view that, given an embedding, the execution of a rule is deterministic (which agent gets phosphorylated is determined by the chosen embedding, not afterwards). Stance 2 corresponds to the notion of a classical mechanism—a locally deterministic process (hence the label D). In stance 2, all randomness occurs in the dynamics: which agents interact when through what rule.

Stance 1 is also known as the stochastic simulation algorithm (SSA) convention. Its persistent use perhaps traces back to a notion of agent-based models that are a stochastic version of highly abstract reaction expressions, such as $X + X \to Z$ (see footnote 1). In this reaction, X and Z are just proper names that do not formally expose the internal structure of the molecules to which they refer. Such molecules cannot have a symmetry: there is no way of deducing from this reaction whether Z is a symmetric combination of Xs or not. As a result, the only symmetry to deal with is at the level of "reaction configurations" on the left; in this case, one should only distinguish unordered pairs and hence divide by 2. Given that Kappa deals with structured entities that can have symmetry, the most natural way of giving meaning to stance 1 within Kappa is in terms of a non-deterministic rule action, as indicated above.

Finally, there is a third possibility:

Stance 3 (Null): Symmetry is ignored and every embedding triggers an action. In this case,

$$\Omega_i = 1. \tag{4}$$

It is unclear what *physical* interpretation should be given to this viewpoint.

At present, `KaSim` defaults to "Null". For a model to have a physical interpretation, the user must decide on how to view symmetry and correct rate constants accordingly by hand. (Future releases of `KaSim` will have a more integrated symmetry support.) In any case, nothing in Kappa hinges formally on how symmetries are viewed. Different interpretations can be chosen simply by correcting rate constants with combinatorial factors.

3.4 The core loop

The principles of continuous-time Monte-Carlo are well-known and can be found in any number of textbooks. They were laid out for chemical reaction networks by Gillespie in the mid 70s. A brief refresher is given in appendix D.

A Kappa simulation is largely initialized by the information provided in the KF, which includes a set of rules with rate constants, numbers of agents present initially as well as the specification of their state, observables, and possibly a schedule of intervention directives.

Let T denote the simulated wall-clock time, which is initialized to some value, typically $T = 0$; let the $\alpha_i(T), i = 1, \ldots, r$ be the rule activities as discussed in section 3.3 and let $\lambda(T) = \sum_i^r \alpha_i(T)$ denote the system activity. Note that the α_i are dependent on T because they reflect the available embeddings into the mixture $\mathscr{M}(T)$, which is evolving with T. The simulation core loop then consists of three conceptual steps:

Time to the next event

Determine the time interval t until the next event occurs according to

$$P\,[\text{next event occurs at } t] = \lambda(T) \exp(-\lambda(T)\, t) \tag{5}$$

Type of next event

Choose which rule induces the next event according to

$$P\,[\text{rule } i \text{ fires} \mid \text{next event occurs at } t] = \frac{\alpha_i(T)}{\lambda(T)}. \tag{6}$$

Note that expression (6) is independent of t. A particular embedding (section 3.1) of the left pattern of rule i—i.e. a "location" in the mixture graph at which i acts—is chosen uniformly from all $|\{\mathscr{L}_i \curvearrowright \mathscr{M}(T)\}|\,\Omega_i$ embeddings of i deemed to be distinct potential events. Rule i is then applied to the mixture graph using the selected embedding (Figure 10).

Update

Update the wall-clock time by setting $T \leftarrow T + t$ and update the embeddings for every rule j affected by the altered mixture $\mathscr{M}(T)$. Repeat.

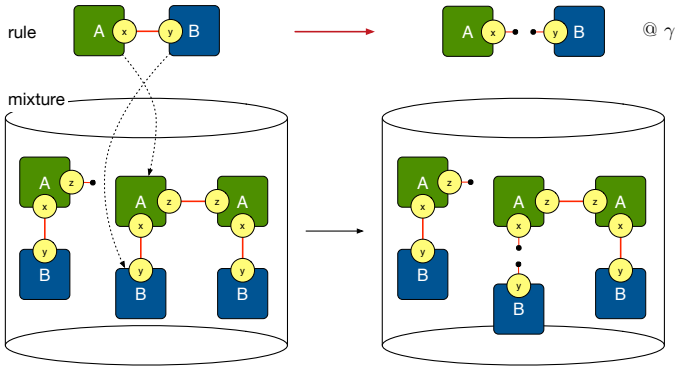


Figure 10: Rule application.

By avoiding unnecessary computations in the core loop, KaSim implements this basic specification with a computational cost per event bounded by a constant that is independent of the size of the mixture. In general, the update phase of the core loop is time consuming since many embeddings can appear or disappear as a consequence of an event. The cost of an update is proportional to the number of rules (and Kappa observables) in the model. The number of rules can be large, as in some applications rules might be generated programmatically. `KaSim`'s update scheme relies on the computation of the so called "extension basis" of the model: a data structure that is built prior to the simulation and guides the search for matchings of a graph pattern by stepwise extension of an initial anchor. The data structure permits the factorization of extension steps whenever two patterns share a sub-pattern. In this way, the computational cost of matching is maximally shared among all patterns that must be checked in the update phase.

3.5 The rate constant

When using Kappa in a fashion that is informed by basic chemical kinetics, it is useful to be aware of a few basics concerning rate constants.

A rate *constant* k is the expression of a single mechanism underlying a given type of reaction. It is the probability rate that an event due to that mechanism occurs between time t and $t + dt$ between specific reactants, conditioned by the knowledge that no such event occurred up to t. Such a conditional probability rate is also known as an event "risk". The rate constant is the risk that a specific event occurs in the next dt. In a continuum setting, the risk that some reaction event of a given type, such as A + B → *products*, occurs is simply $k[A][B]$ with $[\cdot]$ denoting a concentration[4]. This is known as the reaction velocity or flux.

[4] To be conceptually consistent in a continuum picture, a concentration $[A]$ should not be thought of as "particles per volume", as there are no particles, only a "smear of A-stuff per volume". Note also in passing that since particles do not exist in a continuum picture, there is no symmetry that A-stuff could possess. In reference to section 3.3.1, a homodimerization in Kappa, where entities

In a discrete setting, exemplified by the rule `'dim' A(x[.]),B(x[.])->A(x[1]),B(x[1]) @` γ', the risk that a particular `A` interacts with a particular `B` depends on the likelihood that they bump into each other, i.e. that the `B` happens to be in the fraction σV ($\sigma \leq 1$) of *system volume* V swept out by that `A` during a time interval dt. In a simplistic picture, this fraction depends on the size of `A`, the size of `B`, and their relative velocity. The question then is: what is the activity α of `'dim'` that *corresponds to the continuum version of the reaction* in which `A` and `B` have concentrations $[A]$ and $[B]$, respectively? In the discrete picture, $[A] = n_A/V$ (ditto for $[B]$), where n_A and n_B are the particle numbers (or embeddings) of `A` and `B` and we obtain $\alpha = \gamma'\sigma V(n_A/V)(n_B/V) = \gamma/V\, n_A n_B$. Thus, γ/V corresponds to the k in the continuum picture. Quite generally, the stochastic rate constant γ associated with a rule must be a volume-scaled version of the rate constant k in the corresponding continuum setting:

$$\gamma = \frac{k}{V^{n-1}}\ [\mathrm{s}^{-1}\ \mathrm{mol}^{n-1}] \quad \text{or} \quad \gamma = \frac{k}{(\mathscr{A}V)^{n-1}}\ [\mathrm{s}^{-1}\ \mathrm{molecules}^{n-1}]\,, \tag{7}$$

where n is the molecularity of the interaction, k the deterministic rate constant, V the systems volume, and $\mathscr{A} \approx 6.022 \cdot 10^{23}$ is Avogadro's number.

Note that the dependency on volume does not imply that the system is spatially inhomogeneous. Kappa models are homogeneous ("well-stirred" containers), but in a stochastic setting there is a notion of system size.

In summary, a bimolecular stochastic rate constant scales inversely with the system volume (size); a unimolecular rate constant is independent of it (indeed, the interaction does not require a collision between two objects); and a zeroth order flow rate is proportional to the volume (as every volume element is a source).

3.6 Rescaling a Stochastic System

To speed up a simulation it is sometimes useful to *rescale* a model. For the rescaling to be meaningful, it should keep the average (i.e. deterministic or continuum) behavior of the model the same. This means that any deterministic rate constant k should stay invariant across the rescaling.

Let agent types be indexed by i, rules by r, and rescaled variables be marked with a prime. A rescaling with scale parameter σ then means

1. The system volume and all molecule numbers are multiplied by σ (and the latter presumably rounded to some integer):

$$V' = \sigma V \tag{8}$$
$$n_i' = \sigma n_i \qquad \text{for all agent types } i \tag{9}$$

2. All stochastic rate constants of rules whose molecularity is n are divided by σ^{n-1} (refer to equation 7):

$$\gamma_r' = \frac{1}{\sigma^{n-1}}\gamma_r \qquad \text{for all rules } r \text{ of molecularity } n \tag{10}$$

Obviously, if $\sigma < 1$ (> 1), the rescaled system is smaller (larger) than the original.

---

and reactions can have symmetry as in `A(x[.]),A(x[.])<->A(x[1]),A(x[1]) @` γ_1, γ_{-1}, would proceed with activity $\gamma_1\, n_A(n_A-1)/2 \sim \gamma_1\, n_A^2/2$, whereas in the continuum case the forward flux or reaction velocity (which is the analog of the activity) is $k_1[A]^2$, where k_1 is the volume-scaled rate constant corresponding to γ_1, as detailed later in this section. This is a true discrepancy, intrinsic to the continuum description. There is no point in trying to sweep the factor $1/2$ into the k_1, as that would require that we also rescale by $1/2$ the rate constant k_{-1} of the reverse reaction (or else we would not be considering the "corresponding" continuum reaction with the same equilibrium). However, there is no reason for altering k_{-1}.

⚠ Careful: While the average behavior of the system is not changed by rescaling, the fluctuations are. For $\sigma < 1$ one is considering a smaller volume with fewer particles (at constant concentration), but fewer particles lead to larger fluctuations.

3.7 Ambiguous molecularity

The graph-rewrite part of a Kappa rule represents a mechanism. As such it does not specify whether two disconnected components of the left pattern $\mathscr{L}$ should or should not be embedded in the same molecular species (i.e. connected component) of the mixture, Figure 11.

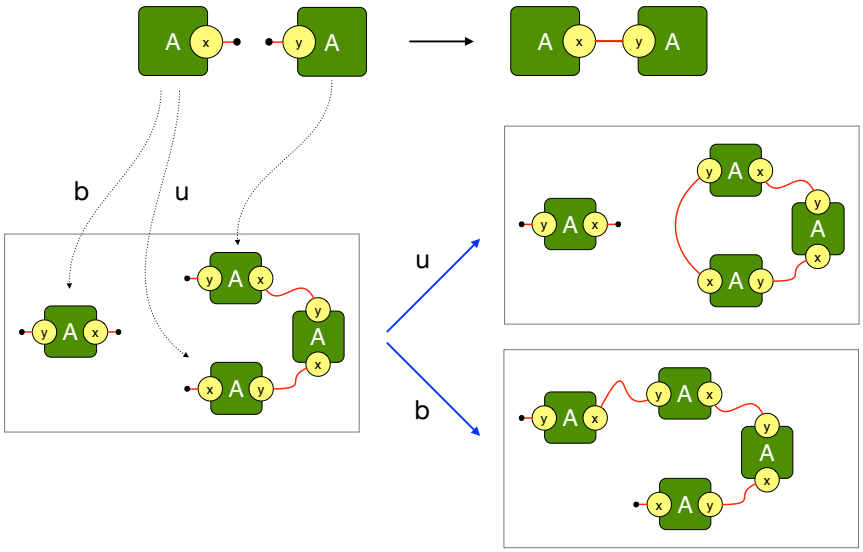


Figure 11: Ambiguous molecularity. The two A agents in the left pattern of the dimerization rule can embed in the shown mixture either within the same molecule (the trimer) or in different molecules (the trimer and the monomer). The former leads to a unimolecular (u) kinetics, the latter to a bimolecular (b) kinetics. As an aside in reference to symmetry corrections (section **??**): Because of the non-trivial symmetry of the left pattern, there are two embeddings for the unimolecular case. However, because that symmetry is preserved by the rule, a correction factor of 1/2 should be applied, as the two embeddings refer to the same physical event. In the bimolecular case there are four embeddings: two locations (section **??**) and two embeddings per location due to symmetry. Again, because the rule preserves the symmetry, a correction factor of 1/2 should be applied at each location, yielding one potential physical event per location. Since there are two locations, intermolecular bindings occur with twice the frequency of unimolecular bindings in this example.

The view adopted by Kappa is that, all else being equal, uni- and bimolecular binding interactions proceed by the same mechanism. This view is also suggested by chemistry, where, for example, the electronic displacements that result in an esterification do not depend on whether the carboxyl and alcohol groups are within the same molecule or belong to distinct molecules, assuming no other constraints are in place. Obviously, if constraints, such as geometry, prevent an intramolecular reaction (for example, the reactants groups may face in opposite directions), these constraints must become part of the mechanism and would have to be specified as context in the graph-rewrite rule.

Thus, if the mechanism is the same, the only distinction between intra- and intermolecular interactions is the effective reaction volume. In the intermolecular case, the effective reaction volume is the system volume, as reactants have to meet within the boundaries of the system. In the intramolecular case, however, the effective reaction volume is constrained by the dimensions of the molecule and therefore independent of the system volume. As a consequence, the distinction between uni- and bimolecular interactions is kinetic and is captured by the distinction between uni- and bimolecular rate constants—the former being

independent of system volume, while the latter depend on it inversely, see section 3.5.

A Kappa rule whose molecularity is ambiguous must be equipped with two rate constants, `A(x[.]),A(y[.])` → `A(x[.]),A(y[.])` `@` γ_2 `(`γ_1`)`, as indicated in Grammar 3 for the production of ⟨*ambi-rule*⟩. The simulator `KaSim` ensures that the bimolecular version of the mechanism is realized with the proper stochastic rate constant γ_2 and the unimolecular version with γ_1.

This raises the question: How does one determine whether the molecularity of a rule is *potentially* ambiguous? While the static analyzer `KaSa` can determine molecular ambiguity, this capability is presently not integrated in the User Interface. It is up to the modeler to ensure that the kinetic aspect of molecularity is properly specified. In case of doubt (because it might be impossible for a human to consider all reachable molecular species to which the rules of a model might apply), candidate rules should be refined by adding more context to eliminate ambiguity.

Molecular ambiguity *can* be very costly. `KaSim` assigns identifiers to molecular species in the mixture and tracks for every agent the identifier of the species it is a part of; it also tracks all embeddings of the left rule pattern within molecular species, as well as across species. The present implementation results in an overestimation of the bimolecular rule activity, which is corrected by rejecting attempted bimolecular events that turn out to be unimolecular (but advancing the wall-clock time). Costs like these are unavoidable, but some clever computer science could improve efficiency.

⚠ Careful: If only one rate constant is supplied, `KaSim` will not complain and execute both intra- and intermolecular rule applications with the same rate constant.

Other rule-based languages offer syntactical constructs to exclude or enforce a particular molecularity of a rule. To maintain the "purity" of the mechanism, Kappa does not provide such constructs. Rather, a user can enforce exclusively uni- or bimolecular applications of a rule by setting the corresponding rate constant to zero.

⚠ Careful: The treatment of molecular ambiguity is limited to patterns with two disconnected graphical components (rules with arity 2); it does *not* extend to rules with higher arity and thus combinatorial ambiguity. Do not use the two-rate-constants construct for such rules.

3.8 Rate functions

It is rarely possible to model consistently at a single level of mechanistic description. Sometimes, processes need to be collapsed into a single "overall" process, if not for conceptual then for practical reasons. This aggregation is often done by accounting for the kinetic consequences of not explicitly exposed mechanisms through rate functions that are not mass-action monomials and that may even depend on entities not mentioned in the rule. Kappa permits such rate functions, but the user must understand that the price to pay is a decreased transparency of the model and a curtailment of the causal analysis that is perhaps the most compelling opportunity enabled by rule-based modeling.

As an example consider the standard Michaelis-Menten scheme and its quasi steady-state approximation.

```
1  %agent: E(s[e.S])
2  %agent: S(e[s.E], x{u p})
3
4  %agent: _E(s)
5  %agent: _S(e, x{u p})
6
7  %var: 'k1'  0.001
8  %var: 'k_1' 0.1
9  %var: 'k2'  1
10
```

```
11 %var: 'Km' ('k_1'+'k2')/'k1'
12
13 E(s[.]), S(e[.],x{u}) <-> E(s[1]), S(e[1],x{u})  @ 'k1', 'k_1'
14 E(s[1]), S(e[1],x{u}) -> E(s[.]), S(e[.],x{p})   @ 'k2'
15
16 _E(s[.]), _S(e[.],x{u}) -> _E(s[.]), _S(e[.],x{p})
17 @ 'k2'/('Km'+|_S(e[.],x{u})|)
18
19 %init: 100     E(s[.])
20 %init: 100000  S(e[.], x{u})
21
22 %init: 100     _E(s[.])
23 %init: 100000  _S(e[.], x{u})
24
25 %obs: 'P'  |S(e[.],x{p})|
26 %obs: 'ES' |E(s[1]), S(e[1],x{u})|
27 %obs: 'S'  |S(e[.], x{u})|
28
29 %obs: '_P' |_S(e[.],x{p})|
```

In the code snippet, lines 13 and 14 are the standard Michaelis-Menten scheme in which an enzyme `E` reversibly binds unphosphorylated substrate `S(x{u})` and converts it to released product `S(x{p})`. The quasi steady-state assumption is a way of eliminating the explicit binding interaction by positing that the the enzyme-substrate complex `E(s[1]),S(e[1],x{u})` is maintained at steady-state even though our system is closed and everything ends up in product because of the irreversible enzymatic reaction. The model for the quasi steady-state case then simplifies to the single rule of line 16 (where the enzyme and substrate names are prefixed by an underscore, so we can run both models simultaneously). Note that the `_E(s[.])` is invariant context that cannot be eliminated from the left and the right pattern of the rule, since it plays a role in determining the number of embeddings of the left pattern (and thus the activity of the rule).

From elementary course work we know that that the rate of product formation in the quasi steady-state approximation is given (in the deterministic setting) by

$$\frac{k_2 E_t[S]}{K_m + [S]}, \tag{11}$$

where $[S]$ is the concentration of *free* substrate, E_t the total concentration of enzyme, and k_2 the catalytic rate constant. We therefore replace the rate constant of the rule on line 16 with a rate function (on the continuation line 17). Note that the factor $E_t[S]$ in (11) has been omitted because that is precisely the number of embeddings that `KaSim` determines for the rule. In situations in which the rate function does not contain the number of embeddings of the left pattern $\mathscr{L}$ as a factor, simply divide by the number of embeddings $|\mathscr{L}|$ to compensate.

The quasi steady-state approximation works well when substrate is in large excess of enzyme. In lines 19 and 20 (as well as 22 and 23) we define an initial amount of substrate that saturates the enzyme, in which case product formation becomes linear. The enzyme is saturated when substrate is in excess of the (stochastic) Michaelis constant, which in this example is $\kappa_M = (k_{-1} + k_2)/k_1 = (0.1 + 1)/0.001 = 1100$ molecules. In our code snippet, we initialize substrate at an amount 10fold above the saturation regime, so as to extend the time that the enzyme-substrate complex is at steady-state. Figure 12A shows the result. The orange curve is the amount of enzyme-substrate complex (right ordinate). It changes little on average until free substrate (green) is depleted and the enzyme falls out of saturation. As expected, the agreement between the aggregate model with rate function and the mechanistic model is near perfect (of course, both must coincide when all substrate is depleted). In Figure 12B we run the comparison for the opposite case in

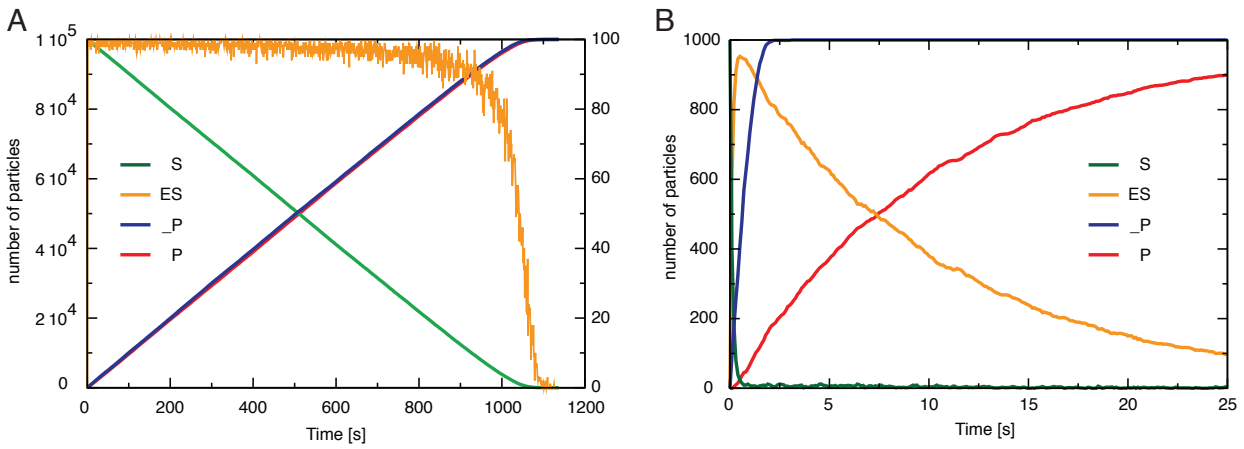


Figure 12: Rate functions. Two Michaelis-Menten rule sets are compared under different conditions. The rule set that produces `P` is mechanistically more detailed than the one producing `_P`. The latter mechanism compensates for the lack of detail with the usual hyperbolic Michealis-Menten rate function, which rests on the assumption of a quasi steady-state in the abundance of the enzyme-substrate complex. **A:** Substrate `S` is in vast excess over enzyme `E`, resulting in a quasi steady-state of the enzyme-substrate complex (orange curve). Because the modeled mechanisms is irreversible, this quasi-equilibrium eventually breaks down. Under these conditions, the production of `P` (blue curve) by the more detailed mechanism is practically identical to the production of `_P` (red curve) by the less detailed mechanism. (They are linear during quasi steady-state, because `S` is here also in excess of the Michaelis constant K_m.) **B:** In this scenario, the catalytic rate constant `'k2'` is slower than in panel A and enzyme is in excess of substrate. As a result, most substrates are immediately bound by an enzyme and then slowly converted to product without the possibility of establishing a quasi steady-state. The more detailed model (red curve) is much more sensible, whereas the aggregate model (blue curve) is completely wrong.

which enzyme is in excess of substrate. As expected, the dynamics differs dramatically between the two models.

Appendices

A Syntax of Kappa

A.1 Names and labels

⟨*Name*⟩ ::= [a-z A-Z] [a-z A-Z 0-9 _ ~ − +]∗ // cannot start with a digit
| [_] [a-z A-Z 0-9 _ ~ − +]+ // initial underscore can't stand alone

⟨*Label*⟩ ::= ' [^ \n ']+ ' //no newline or single quote in a label

A.2 Pattern expressions

⟨*pattern*⟩ ::= ⟨*agent*⟩ ⟨*more-pattern*⟩

⟨*agent-name*⟩ ::= ⟨*Name*⟩

⟨*site-name*⟩ ::= ⟨*Name*⟩

⟨*state-name*⟩ ::= ⟨*Name*⟩

⟨*agent*⟩ ::= ⟨*agent-name*⟩ (⟨*interface*⟩)

⟨*site*⟩ ::= ⟨*site-name*⟩ ⟨*internal-state*⟩ ⟨*link-state*⟩
| ⟨*site-name*⟩ ⟨*link-state*⟩ ⟨*internal-state*⟩
| ⟨*counter*⟩ // see Grammar 14

⟨*interface*⟩ ::= ⟨*site*⟩ ⟨*more-interface*⟩
| ε

⟨*more-pattern*⟩ ::= [,] ⟨*pattern*⟩
| ε

⟨*more-interface*⟩ ::= [,] ⟨*site*⟩ ⟨*more-interface*⟩
| ε

⟨*internal-state*⟩ ::= { ⟨*state-name*⟩ }
| {#} // wildcard
| ε

⟨*link-state*⟩ ::= [⟨*number*⟩]
| [.]
| [_]
| [#] // wildcard
| [⟨*site-name*⟩ . ⟨*agent-name*⟩]
| ε

⟨*number*⟩ ::= $n \in \mathbb{N}_0$

A.3 Rule expressions

A.3.1 Chemical notation

⟨*f-rule*⟩ ::= [⟨*Label*⟩] ⟨*rule-expression*⟩ [| ⟨*token*⟩] @ ⟨*rate*⟩

⟨*fr-rule*⟩ ::= [⟨*Label*⟩] ⟨*rev-rule-expression*⟩ [| ⟨*token*⟩] @ ⟨*rate*⟩ , ⟨*rate*⟩

⟨*ambi-rule*⟩ ::= [⟨*Label*⟩] ⟨*rule-expression*⟩ [| ⟨*token*⟩] @ ⟨*rate*⟩ { ⟨*rate*⟩ }

⟨*ambi-fr-rule*⟩ ::= [⟨*Label*⟩] ⟨*rev-rule-expression*⟩ [| ⟨*token*⟩] @ ⟨*rate*⟩ { ⟨*rate*⟩ } , ⟨*rate*⟩

⟨*rule-expression*⟩ ::= (⟨*agent*⟩ | .) ⟨*more*⟩ (⟨*agent*⟩ | .)

⟨*more*⟩ ::= , (⟨*agent*⟩ | .) ⟨*more*⟩ (⟨*agent*⟩ | .) ,
| ->

⟨*rev-rule-expression*⟩ ::= (⟨*agent*⟩ | .) ⟨*rev-more*⟩ (⟨*agent*⟩ | .)

⟨*rev-more*⟩ ::= , (⟨*agent*⟩ | .) ⟨*rev-more*⟩ (⟨*agent*⟩ | .) ,
| <->

⟨*rate*⟩ ::= ⟨*algebraic-expression*⟩

A.3.2 Edit notation

⟨*f-rule*⟩ ::= [⟨*Label*⟩] ⟨*f-rule-expression*⟩ [| ⟨*token*⟩] @ ⟨*rate*⟩

⟨*ambi-rule*⟩ ::= [⟨*Label*⟩] ⟨*f-rule-expression*⟩ [| ⟨*token*⟩] @ ⟨*rate*⟩ { ⟨*rate*⟩ }

⟨*f-rule-expression*⟩ ::= ⟨*agent-mod*⟩ ⟨*more-agent-mod*⟩
| ε

⟨*more-agent-mod*⟩ ::= , ⟨*agent-mod*⟩ ⟨*more-agent-mod*⟩
| ε

⟨*agent-mod*⟩ ::= ⟨*agent-name*⟩ (⟨*interface-mod*⟩)
| ⟨*agent-name*⟩ (⟨*interface*⟩) (+ | −)

⟨*site-mod*⟩ ::= ⟨*site-name*⟩ ⟨*internal-state-mod*⟩ ⟨*link-state-mod*⟩
| ⟨*site-name*⟩ ⟨*link-state-mod*⟩ ⟨*internal-state-mod*⟩
| ⟨*counter-name*⟩ ⟨*counter-state-mod*⟩

⟨*interface-mod*⟩ ::= ⟨*site-mod*⟩ ⟨*more-mod*⟩
| ε

⟨*more-mod*⟩ ::= , ⟨*site-mod*⟩ ⟨*more-mod*⟩
| ε

⟨*internal-state-mod*⟩ ::= { (⟨*state-name*⟩ | #) / ⟨*state-name*⟩ }
| { (⟨*state-name*⟩) }
| ε

⟨*link-state-mod*⟩ ::= [(⟨*number*⟩ | . | _ | ⟨*site-name*⟩.⟨*agent-name*⟩ | #) / (⟨*number*⟩ | .)]
| ⟨*link-state*⟩
| ε

⟨*counter-state-mod*⟩ ::= { ⟨*counter-expression*⟩ / ⟨*counter-mod*⟩ }
| { ⟨*counter-expression*⟩ }
| { ⟨*counter-mod*⟩ }

⟨*rate*⟩ ::= ⟨*algebraic-expression*⟩

A.3.3 Counters

⟨*counter*⟩ ::= ⟨*counter-name*⟩ { (⟨*counter-expression*⟩ | ⟨*counter-var*⟩ | ⟨*counter-mod*⟩) }

⟨*counter-name*⟩ ::= ⟨*Name*⟩

⟨*counter-expression*⟩ ::= (= | >=) ⟨*integer*⟩

⟨*counter-var*⟩ ::= = ⟨*variable-name*⟩

⟨*counter-mod*⟩ ::= (− | +) = ⟨*integer*⟩ // only on the right of a rule

⟨*integer*⟩ ::= $i \in \mathbb{Z}$

⟨*variable-name*⟩ ::= ⟨*Name*⟩

B Syntax of declarations

B.1 Variables, algebraic expressions, and observables

⟨*variable-declaration*⟩ ::= %var: ⟨*declared-variable-name*⟩ ⟨*algebraic-expression*⟩
⟨*declared-variable-name*⟩ ::= ⟨*Label*⟩ // not ⟨*Name*⟩

⟨*algebraic-expression*⟩ ::= ⟨*float*⟩
| ⟨*defined-constant*⟩
| ⟨*declared-variable-name*⟩ // variable must be declared using `%var`
| ⟨*reserved-variable-name*⟩
| ⟨*algebraic-expression*⟩ ⟨*binary-op*⟩ ⟨*algebraic-expression*⟩
| ⟨*unary-op*⟩ (⟨*algebraic-expression*⟩)
| [max] (⟨*algebraic-expression*⟩) (⟨*algebraic-expression*⟩)
| [min] (⟨*algebraic-expression*⟩) (⟨*algebraic-expression*⟩)
| ⟨*boolean-expression*⟩ [?] ⟨*algebraic-expression*⟩ [:] ⟨*algebraic-expression*⟩

⟨*reserved-variable-name*⟩ ::= [E] // productive events since simulation start
| [E-] // number of null events
| [T] // simulated physical time
| [Tsim] // cpu time since simulation start
| | ⟨*declared-token-name*⟩ | // concentration of token
| | ⟨*pattern-expression*⟩ | // occurrences of pattern
| inf // denotes ∞

⟨*binary-op*⟩ ::= +
| −
| *
| /
| ^
| [mod]

⟨*unary-op*⟩ ::= [log]
| [exp]
| [sin]
| [cos]
| [tan]
| [sqrt]

⟨*defined-constant*⟩ ::= [pi]

⟨*float*⟩ ::= $x \in \mathbb{R}$

B.2 Boolean expressions

⟨*boolean-expression*⟩ ::= ⟨*algebraic-expression*⟩ (= | < | >) ⟨*algebraic-expression*⟩
| ⟨*boolean-expression*⟩ || ⟨*boolean-expression*⟩
| ⟨*boolean-expression*⟩ && ⟨*boolean-expression*⟩
| [not] ⟨*boolean-expression*⟩
| ⟨*boolean*⟩

⟨*boolean*⟩ ::= [true]
| [false]

B.3 Observable declarations

⟨*plot-declaration*⟩ ::= %plot: ⟨*declared-variable-name*⟩
⟨*observable-declaration*⟩ ::= %obs: ⟨*Label*⟩ ⟨*algebraic-expression*⟩

B.4 Agent signature

⟨*signature-declaration*⟩ ::= %agent: ⟨*signature-expression*⟩
⟨*agent-name*⟩ ::= ⟨*Name*⟩
⟨*site-name*⟩ ::= ⟨*Name*⟩
⟨*state-name*⟩ ::= ⟨*Name*⟩
⟨*signature-expression*⟩ ::= ⟨*agent-name*⟩ (⟨*signature-interface*⟩)
⟨*signature-interface*⟩ ::= ⟨*site-name*⟩ ⟨*set-of-internal-states*⟩ ⟨*set-of-link-states*⟩ ⟨*more-signature*⟩
| ⟨*site-name*⟩ ⟨*set-of-link-states*⟩ ⟨*set-of-internal-states*⟩ ⟨*more-signature*⟩
⟨*site-name*⟩ { = ⟨*integer*⟩ / += ⟨*integer*⟩ }
⟨*more-signature*⟩ ::= [,] ⟨*signature-interface*⟩
| ε
⟨*set-of-internal-states*⟩ ::= { ⟨*set-of-state-names*⟩ }
| ε
⟨*set-of-state-names*⟩ ::= ⟨*state-name*⟩ ␣ ⟨*set-of-state-names*⟩
| ε
⟨*set-of-link-states*⟩ ::= [⟨*set-of-stubs*⟩]
| ε
⟨*set-of-stubs*⟩ ::= ⟨*site-name*⟩ . ⟨*agent-name*⟩ ␣ ⟨*set-of-stubs*⟩
| ε

B.5 Initial condition

⟨*init-declaration*⟩ ::= %init: ⟨*algebraic-expression*⟩ ⟨*pattern*⟩
| %init: ⟨*algebraic-expression*⟩ ⟨*declared-token-name*⟩

B.6 Parameter settings

⟨*parameter-setting*⟩ ::= %def: ⟨*parameter-name*⟩ ⟨*parameter-value*⟩
⟨*parameter-name*⟩ ::= reserved names listed in table **??**
⟨*parameter-value*⟩ ::= defined range associated with each ⟨*parameter-name*⟩, table **??**.

B.7 Token expressions

⟨*token*⟩ ::= ⟨*algebraic-expression*⟩ ⟨*declared-token-name*⟩ ⟨*another-token*⟩
⟨*another-token*⟩ ::= , ⟨*token*⟩
| ε

⟨*declared-token*⟩ ::= %token: ⟨*declared-token-name*⟩
⟨*declared-token-name*⟩ ::= ⟨*Name*⟩

B.8 Intervention directives

⟨*effect-list*⟩ ::= ⟨*effect*⟩ ; ⟨*effect-list*⟩ | ε

⟨*effect*⟩ ::= $ADD ⟨*algebraic-expression*⟩ ⟨*pattern*⟩
| $DEL ⟨*algebraic-expression*⟩ ⟨*pattern*⟩
| $APPLY ⟨*algebraic-expression*⟩ ⟨*rule-expression*⟩ [| ⟨*token*⟩]
| $SNAPSHOT ⟨*string-expression*⟩
| $STOP ⟨*string-expression*⟩
| $DIN ⟨*string-expression*⟩ ⟨*boolean*⟩
| $TRACK ⟨*label*⟩ ⟨*boolean*⟩
| $UPDATE ⟨*var-name*⟩ ⟨*algebraic-expression*⟩
| $PLOTENTRY
| $PRINT ⟨*string-expression*⟩ > ⟨*string-expression*⟩
| $SPECIES_OFF ⟨*string-expression*⟩ ⟨*pattern*⟩ ⟨*boolean*⟩

⟨*string-expression*⟩ ::= ε
| `string` . ⟨*string-expression*⟩
| ⟨*algebraic-expression*⟩ . ⟨*string-expression*⟩

⟨*intervention*⟩ ::= %mod: (ε | alarm ⟨*float*⟩) ⟨*boolean-expression*⟩ do ⟨*effect-list*⟩ repeat ⟨*boolean-expression*⟩

C Counters

Counters are a special kind of site that can be used to store bounded non-negative integers and perform some simple tests on them, see Grammar 14. Importantly, the rate constant of a rule can refer to the counters that appear in it.

Grammar 14: Counters

⟨*counter*⟩	::= ⟨*counter-name*⟩ { (⟨*counter-expression*⟩ | ⟨*counter-var*⟩ | ⟨*counter-mod*⟩) }	
⟨*counter-name*⟩	::= ⟨*Name*⟩	
⟨*counter-expression*⟩	::= (= | >=) ⟨*integer*⟩	
⟨*counter-var*⟩	::= = ⟨*variable-name*⟩	
⟨*counter-mod*⟩	::= (− | +) = ⟨*integer*⟩	// only on the right of a rule
⟨*integer*⟩	::= $i \in \mathbb{Z}$	
⟨*variable-name*⟩	::= ⟨*Name*⟩	

Counters must be declared in the agent signature (section 2.4.2). Using counters thus forces a signature declaration, which is otherwise optional. Counters must be initialized (section 2.4.3):

```
// agent signature with a counter site c ranging from 2 to 6
%agent: A(x,y,c{=2 / += 6})
// initialization of 100 instances of agents of type A
%init: 100 A(x,y,c{=4})
```

The following examples illustrate the use of counters. (That makes them counter examples.)

- ```
  // test for equality
  A(x[.],c{=5}),A(y[.]) -> A(x[1]),A(y[1]) @ 0.001
  ```
  Here the dimerization can only happen if the `A` to be bound at site `x` has a value of 5 in counter `c`. Note that the counter is only tested and can be omitted from the right hand side of the rule.

- ```
  // test for inequality
  A(x[.],c{>=5}),A(y[.]) -> A(x[1]),A(y[1]) @ 0.001
  ```
  Here the dimerization happens if the `A` to be bound at site `x` has a value of at least 5 in counter `c`.

- ```
  // test for inequality and modify counter
  A(x[.],c{>=5}),A(y[.]) -> A(x[1],c{+= 2}),A(y[1]) @ 0.001
  ```
  As above, but upon dimerization the counter value in `c` is incremented by 2.

- ```
  // edit notation for counter test and modification
  A(x[./1],c{=4 / += -1}),A(y[./1]) @ 0.001
  ```
  In this edit notation of a rule, the dimerization happens if the agent binding at `x` has a counter value of 4. Upon dimerization the counter is *decremented* by 1.

- ```
  // rule with counter-dependent rates
  A(x[.],c{=var}),A(y[.]) -> A(x[1]),A(y[1]) @ 'var' * 0.001
  ```
  Rate constants can depend on counter values. The construct `c=var` declares a variable that can be used in the rate expression of the rule. Here, for example, the dimerization rate constant is the
  ```

counter value divided by 1,000. (Rates can be algebraic expressions, see section 2.4.1.)

Keep in mind that, in KaSim, counters are just syntactic sugar. Under the hood, each counter contributes a factor linear in the size of its range to the overall number of rules. Importantly, the simulator does *not* check whether a counter stays in its declared range. If it does not, the simulator aborts with an error. (The static analyzer can give certain assurances, see section **??**.) The syntax of counters is ugly at present and this might change.

D Continuous-time Monte-Carlo

Imagine a single agent `A(s[.])` and a single agent `B(s[.])` whose interaction produces a dimer:

```
'bind' A(s[.]),B(s[.]) -> A(s[1]),B(s[1]) @ 'γ'
```

The basic assumption in modeling stochastic chemical kinetics is that the past does not influence the present. This means that the *conditional* probability that `A(s[.])` and `B(s[.])` form a bond during the time interval between t and $t+dt$, *given that no bond was present at* t, is independent of t: γdt (where γ is a probability per time unit—a probability rate).

Suppose that at $t=0$ no bond has formed yet. The (unconditional) probability that `A` and `B` bind between t and $t+dt$ is given by $p(t)dt$ as

$$p(t) = \gamma \exp(-\gamma t). \tag{12}$$

$p(t)$ is called the exponential probability density and is the only possible form given the assumption that the past does not influence the future. (We shall abuse language and refer to a continuous probability density simply as a probability.) From (12) we infer that the cumulative probability for the bond occurring between 0 and t is $1-\exp(-\gamma t)$.

Suppose that the system now contains n_A and n_B agents of type `A` and `B`, respectively, all with identifiers. To simulate reaction events we need to know the probability that a particular choice of agents reacts at time t. It is operationally useful to split this into a probability that the next event occurs at t and a probability, conditioned on such occurrence, that the event is between specific agents. This way we can determine the time of an event by drawing a random number from one distribution and the specific reaction combination by drawing from a second distribution.

In our example, there are $n_A n_B$ potential reaction events, each occurring with probability $p(t)$ at time t. One of them must be the first to occur at time t. This happens with probability $p(t)$ only if none of the other $n_A n_B - 1$ reactions occurred before, which is the case with probability $\exp(-\gamma t)^{(n_A n_B)-1} = \exp(-n_A n_B \gamma t)\exp(\gamma t)$. Thus, the probability that a specific choice of `A` and `B` agents is the first pair to bind at time t is $p(t)\exp(-n_A n_B \gamma t)\exp(\gamma t)$, that is:

$$P\,[\text{a specific reaction occurs next at time } t] = \gamma \exp(-n_A n_B \gamma t). \tag{13}$$

It is more practical to split this into two probabilities:

$$\begin{aligned} &P\,[\text{a specific reaction occurs next at time } t] = \\ &\quad = P\,[\text{specific reaction} \mid \text{next reaction occurs at } t]\,P\,[\text{next reaction occurs at } t]. \end{aligned} \tag{14}$$

There are $n_A n_B$ choices for a specific reaction, which here have the same probability (13). Thus,

$$P\,[\text{next reaction occurs at } t] = n_A n_B \gamma \exp(-n_A n_B \gamma t). \tag{15}$$

The term $n_A n_B \gamma$ is called the *activity* of rule `'bind'`. The activity is the number of matchings that the pattern `A(s[.]),B(s[.])` has in the mixture. The `A` in the pattern can match any of the n_A instances of type `A` and the `B` can match any of the n_B instances of type `B`; together they can match in $n_A n_B$ ways.

The conditional probability that a specific pair of agents is chosen for reaction, given that some reaction occurs at t, is calculated by dividing (13) by (15):

$$P\,[\text{specific reaction} \mid \text{next reaction occurs at } t] = \frac{\gamma}{n_A n_B \gamma} = \frac{1}{n_A n_B}, \tag{16}$$

which, as expected in this case, is the uniform distribution.

To simulate one binding event, given the mixture, one proceeds in two steps: (1) Determine the time t at which a reaction occurs by drawing a random number distributed according to (15). (2) Determine which pair of agents reacts by drawing a random number according to (16).

To simulate the subsequent event, a third step is added: (3) update the system state. This means taking into account that one `A` and one `B` were consumed, $n_A \leftarrow n_A - 1$ and $n_B \leftarrow n_B - 1$, and moving forward the simulated wall-clock time T, $T \leftarrow T + t$, as the event that just occurred marked the advancement of time.

The above arguments generalize verbatim one level up to the case in which we deal not only with one single type of rule that can apply to several agent instances, but a collection of different rules. With r rules whose activities at a given moment are $\alpha_i, i = 1, \ldots, r$, the total system activity is $\lambda = \sum_{i=1}^{r} \alpha_i$ and we obtain in complete analogy to (15) and (16):

$$P\,[\text{next event occurs at } t] = \lambda \exp(-\lambda t) \tag{17}$$

$$P\,[\text{rule } i \text{ fires} \mid \text{next event occurs at } t] = \frac{\alpha_i}{\lambda}. \tag{18}$$

E The symmetries of a rule

The activity of a rule is defined in terms of the set of embeddings of its left pattern into a mixture, equation (1). The idea being that an embedding constitutes a candidate physical event. Here we argue that if the action of a rule along two symmetrically related embeddings produces *identical* mixtures at the level of identifiers (microstate), the two embeddings should be viewed as expressing the same physical event, not distinct events. This degeneracy is due to symmetries on the left side of a rule that are preserved by the rule action.

As an example consider Figure 13. On the left panel of Figure 13A, the agents of the pattern $\mathscr{L}$ of the rule are given identifiers 1 and 2. These agents are put in correspondence with those in $\mathscr{R}$ by virtue of the agent mapping—slot `#1L` corresponds to slot `#1R`, etc.—which is a constitutive element of a rule (section 2.3.1). One match from $\mathscr{L}$ to the molecule $\mathscr{M}$ below (standing for a mixture) maps $\mathtt{A}_1$ of $\mathscr{L}$ to $\mathtt{A}_8$ of $\mathscr{M}$ and $\mathtt{A}_2$ of $\mathscr{L}$ to $\mathtt{A}_9$ of $\mathscr{M}$ (solid arrows). The transformation of the molecule-fragment matched by $\mathscr{L}$ then proceeds according to $\mathscr{R}$ with the embedding inherited from $\mathscr{L}$ via the agent mapping (dotted arrows). This yields mixture $\mathscr{M}_1$.

The symmetry of $\mathscr{L}$ enables a second embedding, shown in the right panel of Figure 13A. Here the agent with identifier 1 has been swapped with the agent with identifier 2. The agent mapping from $\mathscr{L}$ to $\mathscr{R}$ forced the same swap in pattern $\mathscr{R}$. Proceeding as before, yields mixture $\mathscr{M}_2$.

The two actions resulting from the automorphisms of $\mathscr{L}$ yield not only the same (i.e. macroscopically indistinguishable) mixtures $\mathscr{M}_1$ and $\mathscr{M}_2$, but microscopically *identical* ones. Note that the rule is applied to a molecule with *no* symmetry (one site is phosphorylated, the other not), but the mechanism represented by the rule is blind to this difference.

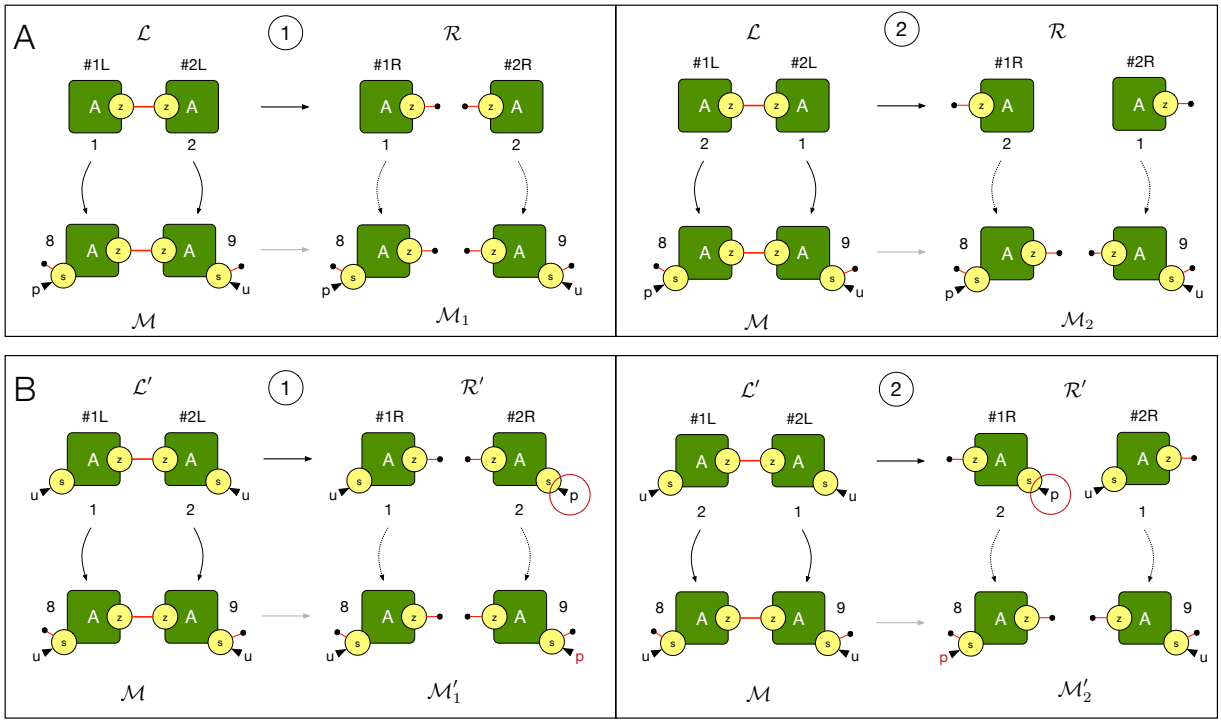


Figure 13: Rules and symmetries. The Figure illustrates the consequences on a mixture of applying a rule that preserves the symmetries of its left pattern (**A**) and and applying a rule that does not (**B**). The asymmetry of the right pattern in panel B is emphasized by a red circle. See text for details.

The rule considered in Figure 13B has a symmetric left pattern $\mathscr{L}'$ but an asymmetric right pattern $\mathscr{R}'$, in which the dissociation is accompanied by a simultaneous phosphorylation of the second A. This rule is applied to a symmetric molecule $\mathscr{M}$. As before, there are two embeddings of the left pattern. Swapping the agents on the left, which leaves $\mathscr{L}'$ unchanged, forces again a corresponding swap of the agents on the right. Unlike in Figure 13A (right panel), this swap is not an automorphism of $\mathscr{R}'$. As a consequence, the actions resulting from the two symmetric embeddings of $\mathscr{L}'$ differ: In $\mathscr{M}'_1$, the agent A_8 is phosphorylated, but in $\mathscr{M}'_2$ it is A_9. The two resulting mixtures $\mathscr{M}'_1$ and $\mathscr{M}'_2$ are isomorphic (i.e. they are the same macroscopically), but they are not identical (they differ microscopically): Unlike in the case of Figure 13A, they are distinct realizations of the same macrostate.

The difference between the rule of Figure 13A and the rule of Figure 13B is that the symmetry of the left pattern is preserved by the rule in the former case, whereas it is not preserved in the latter case. This example suggests *identifying* the two rule applications of $\mathscr{L}$ in Figure 13A as being one and the same physical event, as there is no way of distinguishing them given our defintion of state. The upshot is that in computing the activity of a rule, we should divide the number of embeddings by the number of symmetries on the left that are preserved by the rule. (This includes the always preserved trivial symmetry or identity.)

In general, the number of symmetries $\omega_{\mathscr{P}}$ of a pattern $\mathscr{P}$ is calculated as follows. Let $C(\mathscr{P})$ be the number of distinct classes of connected components in $\mathscr{P}$ ("component classes"), $n_c, c = 1,\ldots,C(\mathscr{P})$ the number of isomorphic instances of component c, and ω_c the number of automorphisms of component c (Figure 14). Then

$$\omega_{\mathscr{P}} = \prod_{c=1}^{C(\mathscr{P})} n_c! \prod_{i=1}^{C(\mathscr{P})} (\omega_c)^{n_c}. \tag{19}$$

An instance of a component class (i.e. a component) has a set of identifiers associated with it. Each factor

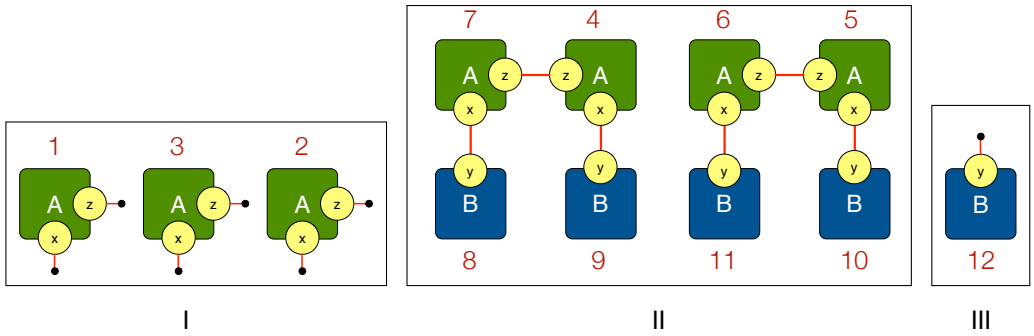


Figure 14: Automorphisms. The pattern shown consists of 3 component classes, I, II, and III. Component class I has 3 instances. Instances from class I have 1 automorphism (the identity). Component class II has 2 instances, each with 2 automorphisms, and class III has 1 instance with 1 automorphism. In class II, as in any other class, we can swap the set of identifiers of each instance for those of another instance of the same class yielding the identical pattern. In addition we can apply automorphisms within each instance and combine them freely with any other automorphisms in other classes to again produce an identical pattern. In total, the patter shown has $3!\,2!\,1!\,1^3\,2^2\,1^1 = 48$ symmetries.

in the first product results from the ways of reassigning the sets of identifiers (as a whole) to different components within a component class. Such a reassignment produces an identical $\mathscr{P}$ (Figure 14), so it is a symmetry. We can freely combine these within-class permutations of identifier sets across classes to obtain an automorphism of $\mathscr{P}$. This explains the first product. Each factor in the second product is the number of symmetries of the component class, which can be freely combined for each instance in that class (thus the power) and, in turn, across classes to yield the second product term. Figure 14 illustrates this reasoning.

It is useful to distinguish between embeddings of a pattern into the mixture and matching "locations" at which these embeddings occur, as illustrated in Figure 15. Locations differ in the set of identifiers of the host graph that are involved in the embedding of the pattern. In the example of Figure 15, two embeddings occur at location $\{8,9\}$ and two at location $\{2,4\}$. Recall that the activity of a rule i is defined as

$$\alpha_i = |\{\mathscr{L}_i \curvearrowright \mathscr{M}\}|\,\Omega_i k_i, \tag{20}$$

where $\{\mathscr{L}_i \curvearrowright \mathscr{M}\}$ is the set of embeddings from $\mathscr{L}_i$ into $\mathscr{M}$ and Ω_i is a correction factor to be determined. Let $\{\mathscr{L}_i \curvearrowright \mathscr{M}\}_{\text{loc}}$ denote the locations of embeddings. Clearly, there are $\omega_{\mathscr{L}_i}$ embeddings per location:

$$|\{\mathscr{L}_i \curvearrowright \mathscr{M}\}| = |\{\mathscr{L}_i \curvearrowright \mathscr{M}\}_{\text{loc}}|\,\omega_{\mathscr{L}_i}. \tag{21}$$

Following the reasoning developed in the example of Figure 13, we can distinguish the consequences of only as many rule applications as there are symmetries of $\omega_{\mathscr{L}_i}$ that the rule preserves, which we denote with $\omega_{\mathscr{L}_i \to \mathscr{R}_i}$. Thus,

$$\alpha_i = |\{\mathscr{L}_i \curvearrowright \mathscr{M}\}|\,\Omega_i k_i = |\{\mathscr{L}_i \curvearrowright \mathscr{M}\}_{\text{loc}}|\,\omega_{\mathscr{L}_i}\frac{1}{\omega_{\mathscr{L}_i \to \mathscr{R}_i}} k_i. \tag{22}$$

In this line of reasoning, only $\omega_{\mathscr{L}_i \to \mathscr{R}_i}$ of the possible applications of a rule in a given mixture are considered to be distinguishable and thus "physical" events:

$$\Omega_i = \frac{1}{\omega_{\mathscr{L}_i \to \mathscr{R}_i}}. \tag{23}$$

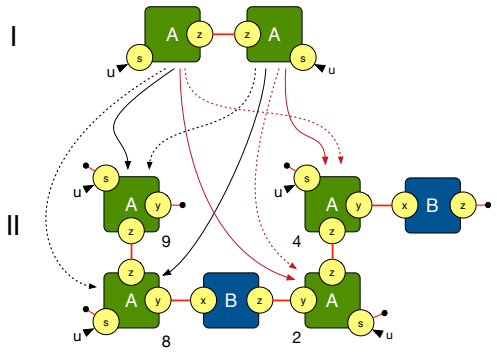


Figure 15: Embedding location. The diagram exhibits four embeddings of the pattern I into the host graph II. The locations of the embeddings are given by the set of identifiers involved in the embedding, here two: $\{8,9\}$ (black embedding arrows) and $\{2,4\}$ (red embedding arrows). Because of the twofold symmetry of the pattern, there are two embeddings at each location, shown as solid and dotted arrows.

We refer to equation 23 as the "chemical" correction.

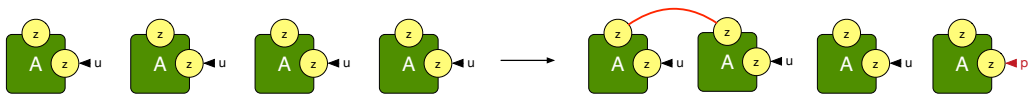


Figure 16: Symmetry correction. The pattern on the left has $4! = 24$ automorphisms and the pattern on the right has 2 automorphisms. In this case $\Omega_i = 2$.

In the (contrived) example of Figure 16, only 12 of 24 possible embeddings per matching location in the host graph lead to microscopically distinguishable outcomes.

Extending this line of thought to deal with agent creation and removal is straightforward. On the right hand side of a rule, any agent that appears anew and is not in a connected component with a modified agent is ignored. If a newly created agent is connected to agents already existing on the left side, one checks whether a given automorphism on the left can be extended. Any such extension would be unique by virtue of the rigidity of site graphs (section 2.2). If no such extension exists, the particular automorphism on the left is not preserved by the rule.

- `'remove two agents and create two different ones anew'`
  `A(), A() -> B(), B()` $\quad \omega_{\mathscr{L}_i \to \mathscr{R}_i} = 2$

  Two automorphisms (including the identity) on the left; the agents disappear and fresh ones are created. This preserves the non-trivial symmetry of the left.

- `'add two new agents'`
  `A(), A() -> A(), A(), B(), B()` $\quad \omega_{\mathscr{L}_i \to \mathscr{R}_i} = 2$

  Two automorphisms on the left; the agents are kept and fresh ones are created with no connection to the old ones. This preserves the non-trivial symmetry of the left.

- `'add two new agents connected to old'`
  `A(x[1]), A(x[1]) -> A(x[2]), A(x[3]), B(x[2]), B(x[3])` $\quad \omega_{\mathscr{L}_i \to \mathscr{R}_i} = 2$

  Two automorphisms on the left; the agents, bound to each other on the left, dissociate and associate with two new agents. The two `A` are treated equally, which preserves the non-trivial symmetry of the left.

- `'add two new agents connected to each other'`
  `A(x[1]), A(x[1]) -> A(x[1]), A(x[1]), B(x[2]), B(x[2])` $\omega_{\mathscr{L}_i \to \mathscr{R}_i} = 2$

  Two automorphisms on the left. Again, no difference from the point of view of the two `A` agents on the left; the non-trivial symmetry is preserved.

- `'add one new agent connected to one old'`
  `A(x[.]), A(x[.]) -> A(x[1]), A(x[.]), B(x[1])` $\omega_{\mathscr{L}_i \to \mathscr{R}_i} = 1$

  Two automorphisms on the left. Since the new agent is bound to one of the old agents, it creates a difference between them and the non-trivial symmetry on the left is lost.

⚠ Careful: As indicated at the end of sections 3.2 and 3.3.1, at present `KaSim` *does not* automatically apply symmetry corrections, i.e. $\Omega_i = 1$ in equation (1). Any such corrections are the responsibility of the user.