KaSim & KaSa reference manual

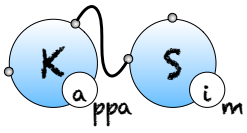
(release 3.90)

Pierre Boutillier, Jérôme Feret, Jean Krivine[1] and Lý Kim Quyên

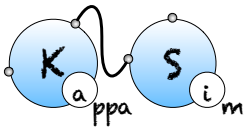
KappaLanguage.org

[1]corresponding author: jean.krivine@irif.fr

Contents

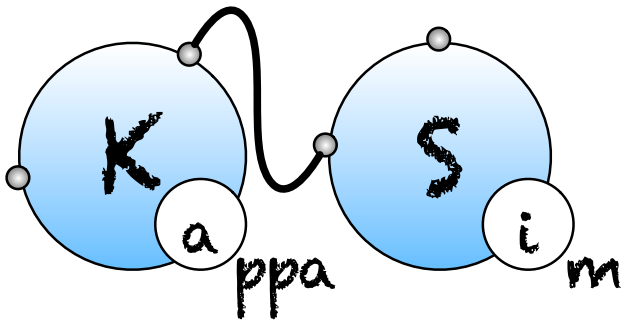


List of Tables


K
a
ppa
S
i
m

Chapter 1

Introduction



1.1 Preamble

This manual describes the usage of KaSim and KaSa, the latest implementation of Kappa, one member of the growing family of rule-based languages. Rule-based modelling has attracted recent attention in developing biological models that are concise, comprehensible, easily extensible, and allows one to deal with the combinatorial complexity of multi-state and multi-component biological molecules. Although this manual contains a self-contained description of Kappa, it is *not* intended as a tutorial on rule-based modeling.

To get an idea of how Kappa is used in a modeling context, the reader can consult the following note Agile modelling of cellular signalling (SOS'08). A longer article, expounding on causal analysis is also available: Rule-based modelling of cellular signalling (CONCUR'07). See also this tutorial: Modelling epigenetic information maintenance: a Kappa tutorial (CAV'09).

1.2 The KaSim engine

KaSim is an open source stochastic simulator of rule-based models [7, 6, 8] written in Kappa. The Kappa language describes site graphs and their local transformations. KaSim takes one or several Kappa files as input and generates stochastic trajectories of various observables. KaSim implements Danos *et al*'s implicit state simulation algorithm [4] which adapts Gillespie's algorithm [13, 14] to rule-based models.

A *simulation event* corresponds to the application of a rewriting rule, contained in the Kappa files, to the current graph (also called a *mixture*). At each step, the next event is selected with a probability which is proportional to the rate of the rule it is an event of. If there are no events, that is to say if none of the rules apply to the current state of the system, one has a *deadlock*. Note that a given rule will in general apply in many different ways; one says it has many instances. The *activity* of a rule is the number of its instances in the current mixture multiplied by its rate. The probability that the next event is associated to a given rule is therefore proportional to the activity of the rule. Rule activities are updated at each step (see Fig. 1.1). Importantly, the cost of a simulation event is bounded by a constant that is independent of the size of the graph it is applied to [4].

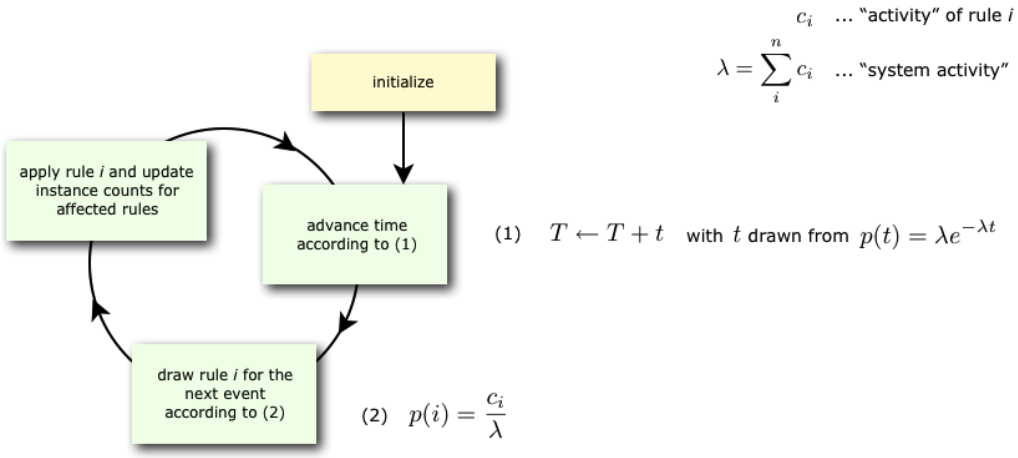
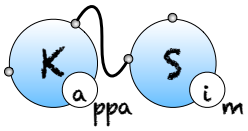


Figure 1.1: The event loop

Note that KaSim can only render curves in the svg format. However, data outputs given in a text format can be displayed using any standard plotting software such as gnuplot.



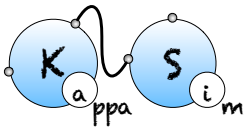
1.3 The KaSa static analyser

KaSa is an open source static analyser tool of rule-based models [7, 6, 8] written in Kappa. KaSa takes one or several Kappa files as input and some command line options to toggle on/off some specific static analysis. Currently, KaSa can compute the contact map and the influence map. It can perform reachability analysis [10, 5] as well. Other analyses including model reduction [11, 3, 1] will come soon.

A graphical interface is proposed to navigate through the various options and utilities of KaSa. The compilation of this interface requires lablTk and, in particular, tk-dev.

1.4 Support

- Kappa language tutorials and downloads: `http://kappalanguage.org`
- Bug reports should be posted on github: `https://github.com/Kappa-Dev/KaSim/issues`
- Questions and answers on the kappa-user mailing list: `http://groups.google.com/group/kappa-users`
- Want to contribute to the project? `jean.krivine@irif.fr`


K
a
ppa
S
i
m

Chapter 2

Installation

2.1 Using precompiled binaries

The easiest way to use KaSim and KaSa is to use pre-compiled versions available in the release section on the github repository (https://github.com/Kappa-Dev/KaSim/releases). Download the version that corresponds to your operating system (Windows, Linux or Mac OSX) and rename the downloaded file into KaSim and KaSa. Note that on Mac OSX or Linux, it might be necessary to give executable permissions to KaSim and KaSa. This can be done using the shell commands: `chmod u+x KaSim` and `chmod u+x KaSa`

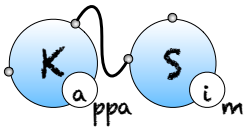
To test whether your program does work, simply type `./KaSim --version` on a terminal, from the directory that contains the binaries. If the version is displayed it means that the binaries are indeed compatible with your OS. Otherwise you may need to compile KaSim from the sources (see next Section).

2.2 Obtaining the sources

To obtain KaSim/KaSa you can either use pre-compiled binaries (see previous section) or compile the sources for your architecture by yourself.

To do so, download the source code from https://github.com/Kappa-Dev/KaSim, make sure you have a recent OCaml compiler (KaSim/KaSa currently requires Ocaml 4.02.3 to compile) as well as `ocamlbuild`, `findlib` and the `yojson` library installed.

You can check if it is the case from a terminal window by typing first `ocamlfind ocamlopt -v`. If it fails or prints a version number too old, then you need to install Ocaml Native



compiler that can be downloaded from http://caml.inria.fr/download.en.html and/or findlib available at http://projects.camlcity.org/projects/findlib.html.

Then, type `ocamlfind query yojson`. The answer should be a path. If it is not, install yojson using a package manager.

Ocamlbuild is hosted on https://github.com/ocaml/ocamlbuild.

2.3 Compilation

Once OCaml is safely installed, untar KaSim archive and compile following these few steps:

```
$ tar xzvf kasim.tar.gz -d Kappa
$ cd Kappa
$ make bin/KaSim
$ make bin/KaSa
```

At the end of these steps you should see, in the `bin` directory of the `Kappa` directory, an executable file named KaSim. In order to check the compilation went fine, simply type `bin/KaSim --version`.

If the tool `ocamlbuild` is not in your path, you may set the variable `OCAMLBINPATH` to point to the location of the compiler by doing `make OCAMLBINPATH='the_correct_dir' bin/KaSim`.

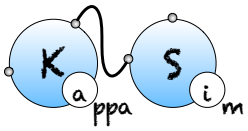
2.4 Compilation of KaSa graphical interface

The graphical interface of KaSa requires tk-dev and lablTk. By default, the graphical interface is not compiled. The compilation of this interface can be toggled on by using the following command: `make USE_TK=1 bin/KaSa`

Common compilation errors are the following:

1. The following error:

```
/usr/bin/ld: cannot find -ltk
collect2: error: ld returned 1 exit status
File "caml_startup", line 1:
Error: Error during linking
Command exited with code 2.
```



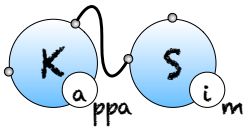
occurs when the module tk-dev is not installed.

2. The following error:

```
File "_none_", line 1:
Error: Cannot find file jpflib.cmxa
Command exited with code 2.
```

occurs when ocaml cannot link the lablTk library.

Please document the variable `LABLTKLIBREP` in the Makefile.


K
a
ppa
S
i
m

Chapter 3

The kappa language

3.1 General structure

A model is represented in Kappa by a set of *Kappa File*. We use KF to denote the union of the files that are given as input (to either KaSim or KaSa).

Each line of the KF is interpreted as a *declaration* except if the line ends by the ' \' character. Therefore, in order to write a declaration on several lines, ends every by the last of the lines with a \.

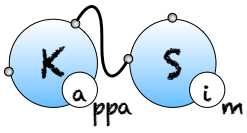
Declarations can be: agent and token *signatures* (Sec. 3.2), *rules* (Sec. 3.4), *variables* (Sec. 3.5), *initial conditions* (Sec. 3.6), *perturbations* (Sec. 6.1) and *parameter configurations* (Sec. 6.4).

The KF's structure is quite flexible. Neither dividing into several sub-files nor the order of declarations matters (to the exception of variable declarations, see Section 3.5 for details).

Comments can be used either by inserting the marker # that tells KaSim to ignore the rest of the line or by putting any text between the delimiters /* and */. The combined use of \ and # is an alternative way to write comments in the middle of a declaration.

3.2 Agent and token signatures

In Kappa there are two entities that can be used for representing biological elements: *agents* and *tokens*. Agents are used to represent complex molecules that may bind to other molecules on specific sites. Tokens are typically used to represent small particles such as



ions, ATP, etc. Tokens cannot bind to each other, they can only appear or disappear. In a given model, agents always have a discrete number of instances while tokens may have a continuous concentration.

In order to use agents or tokens in a model, one needs to declare them first. *Agent signatures* constitute a form of typing information about the agents that are used in the model. It contains information about the name and number of interaction sites the agent has, and about their possible internal states. A signature is declared in the KF by the following line:

`%agent:` *signature_expression*

according to the grammar given Table 3.1 where terminal symbols are written in (blue) typed font, and ε stands for the empty list. An identifier `Id` can be any string generated by a regular expression of the type [*a-z A-Z*][*a-z A-Z* 0-9 _ − +]*.

Table 3.1: Agent signatureexpression

signature_expression	::=	`Id(`*sig*`)`
sig	::=	`Id` *internal_state_list* `,` *sig* | ε
internal_state_list	::=	`~Id` *internal_state_list* | ε

For instance the line:

```
1 %agent: A(x,y~u~p,z~0~1~2) # Signature of agent A
```

will declare an agent `A` with 3 *(interaction) sites* `x,y` and `z` with the site `y` possessing two *internal states* `u` and `p` (for instance for the unphosphorylated and phosphorylated forms of `y`) and the site `z` having three possible states 0, 1 and 2. Note that internal states values are treated as untyped symbols by KaSim, so choosing a character or an integer as internal state is purely a matter of convention.

Token signatures are declared using a statement of the form:

```
1 %token: ca+ # Signature of calcium token
```

3.3 Sited-graph pattern: Kappa expression

The state of the system is represented in Kappa as a sited graph: a graph were edges specifies a site of the node they use. You must think as sites as resources. At most one edge of the graph can use a site of a node (representing an agent in our case). We call this concept the *rigidity of Kappa*.

3.3.1 Graph syntax

The ascii syntax we use to represent sited graphs follows the skeletons (describe formaly in fig 3.2):

- we write the type of the agent and then its interface (the comma separeted list of the state of its sites) between parenthesis.
- When the site is *free* (is not a member of a edge) you just write its name to represent its state. If it has a internal state, you write it after a '~'. For example, the graph TODO is written A(x,y~p,z~0)
- When a site is part of an edge, you assign an integer identifier n to this edge and you specify the appartenance of the site to this edge by writing site_name!n. The graph TODO can be reprensented as A(x!23,y~u!4,z~1), A(x!4,y~u!954,z~1), A(x!95,y~u!234,z~1).

Remark Each link identifier appears exactly twice of course. this is a consequence of the regidity of Kappa.

3.3.2 Pattern syntax

Kappa strength is to describe transformations by only mentionning (and storing) the relevant part of the subgraph required for that transfor;ation to be possible. It plays a key role in resisting combinatorial explosion when writing models. We use the *don't care, don't write* principle. If a transformation occurs independently of the state of a site of an agent, don't mention it in the *pattern* to match. The pattern A(x,z) represents an agent of type A whose sites x and z are free but the sites y and z can be in any internal state and the site y can be linked or not to anything.

If the link state of a site does not matter but the internal state does, an '?' has to be added after the site name (and internal state). An agent A whose sites x and z are free, y is in state u and z in state 2 is written A(x,y~u?4,z~2).

Table 3.2: Kappa expressions

kappa_expression	::=	*agent_expression* `,` *kappa_expression* $\mid \varepsilon$
agent_expression	::=	`Id(`*interface*`)`
interface	::=	`Id` *internal_state link_state* `,` *interface* $\mid \varepsilon$
internal_state	::=	ε $\mid$ `~Id`
link_state	::=	ε $\mid$ `!n` $\mid$ `!_` $\mid$ `?` $\mid$ `!Id.Id`

3.3.3 Link type

In standard kappa, in order to require a site to be bound for an interaction to occur, one may use the *semi-link* construct `!_` which does not specify who the partner of the bond is. For instance in the variable: `%var: \var{ab}~|A(x!_),B(y!_)|` will count the number of `A`s and `B`s connected to someone, including the limit case `A(x!1),B(y!1)`. It is sometimes convenient to specify the *type* of the semi-link, in order to restrict the choice of the binding partner. For instance the variable: `%var: \var{ab}~|A(x!y.B),B(y!x.A)|!` will count the number of `A`s whose site `x` is connected to a site `y` of `B`, plus the number of `B`s whose site `y` is connected to a site `x` of `A`. Note that this still includes the case `A(x!1),B(y!1)`.

Remark Transformations on semi-links and links type indiuce side effects (effect on unmention agents/unmentionned site of agent) and can even don't make sense at all. What would mean to remove the link to A but not the link to B in the example above? Be carefull when you use them.

3.4 Rules

Once agents are declared, one may add to the KF the rules that describe their dynamics. A *pure rule* looks like:

`'my rule'` *kappa_expression* $\rightarrow$ *kappa_expression* `@` *rate*

where `'my rule'` can be any name. This rule name can be used to refer to the rule which follows immediately. A rule can be decomposed into a *left hand side* (LHS), a *right hand side* (RHS) kappa expressions, and a *rate expression*. One may also declare a *bi-directional rule* using the convention:

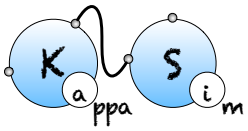
`'bi-rule'` *kappa_expression* $\leftrightarrow$ *kappa_expression* `@` $rate^{+}$,$rate^{-}$

Note that the above declaration is equivalent to writing, in addition of `'my-rule'`, another rule named `'my rule_op'` which swaps left and right hand sides, and has rate $rate^{-}$.

Rate expressions are given by the grammar in Table 3.3. Algebraic expressions are described later in Table 3.6 (but can be thought of for now as positive real numbers).

Table 3.3: Rate expressions

rate_expression	::=	*algebraic_expression*
		| *algebraic_expression* {*algebraic_expression*:*algebraic_expression*}



If pure rules induce reactions between agents, it is possible to mix agents and tokens in *hybrid rules* (which may also be bi-directional). A hybrid rule has the following form:

kappa_expression | *token_expression* → *kappa_expression* | *token_expression* `@` *rate*

Token expressions are also given by the grammar in Table 3.4.

Table 3.4: Token expressions

token_expression	::=	*algebraic_expression* `:` *token_name*
		| *token_expression* `+` *token_expression*
token_name	::=	`Id`

3.4.1 Pure rules

A simple rule

With the signature of `A` defined in the previous section, the line

```
1 'A␣dimerization' A(x),A(y~p) -> A(x!1),A(y~p!1) @ 'gamma'
```

declares a dimerization rule between two instances of agent `A` provided the second is phosphorylated (say that is here the meaning of `p`) on site `y`. Note that the bond between both `A`s is denoted by the identifier `!1` which uses an arbitrary integer (`!0` would denote the same bond). Note also the fact that site `z` of `A` is not mentioned in the expression which means that it has no influence on the triggering of this rule. This is the *don't care don't write convention* (DCDW) .

Adding and deleting agents

Sticking with `A`'s signature, the rule

```
1 'budding␣A' A(z) -> A(z!1),A(x!1) @ 'gamma'
```

indicates that an agent `A` free on site `z`, no matter what its internal state is, may beget a new copy of `A` bound to it via site `x`. Note that in the RHS, the interface of the new copy is not completely described. Following the DCDW convention, KaSim will then assume that the sites that are not mentioned are created in the *default state*, ie they appear free of any bond and their internal state (if any) is the first of the list shown in the signature (here state `u` for `y` and `0` for `z`).

Importantly, KaSim respects the *longest prefix convention* to determine which agent in the RHS stems from an agent in the LHS. In a word, from a rule of the form $a_1, \ldots, a_n \to b_1, \ldots, b_k$, with a_is and b_js being agents, one computes the largest $i \leq n, k$ such that the agents $a_1, \ldots, a_i$ are pairwise consistent with $b_1, \ldots, b_i$, ie the a_js and b_js have the same name and the same sites. In which case we say that for all $j \leq i$, a_j is *preserved* by the transition and for all $j > i$, a_j is *deleted* by the transition and b_j is *created* by the transition. This convention allows us to write a deletion rule as:

```
'deleting A' A(x!1),A(z!1)} -> A(x) @ 'gamma'
```

which will remove the `A` agent in the mixture that will match the second occurrence of `A` in this rule. Note that the rule:

```
'weird' A(x!1),A(z!1) -> A(z) @ 'gamma'
```

will delete both `A`s and create a new one with a free `z` site.

Side effects

It may happen that the application of a rule has some *side effects* on agents that are not mentioned explicitly in the rule. Consider for instance the previous rule:

```
'deleting A' A(x!1),A(z!1)} -> A(x) @ 'gamma'
```

The `A` in the graph that is matched to the second occurrence of `A` in the LHS will be deleted by the rule. As a consequence all its sites will disappear together with the bonds that were pointing to them. For instance, when applied to the graph

$G =$`A(x!1,y~p,z~2),A(x!2,y~u,z~0!1),C(t!2)`

the above rule will result in a new graph $G' =$ `A(x!1,y~p,z~2),C(t)` where the site `t` of `C` is now free as side effect.

Wildcard symbols for link state `?` (for bound or not), `!_` (for bound to someone), may also induce side effects when they are not preserved in the RHS of a rule, as in

```
'Disconnect A' A(x!\_)} -> \ttt{A(x) @ 'gamma'
```

or

```
'Force bind A' A(x?)} -> A(x!1),C(t!1) @ 'gamma'
```

Both these rule will cause KaSim to raise a warning at rule compilation time.

3.4.2 Hybrid rules

Using KaSim *hybrid rules*, one may declare that an action has effects on the concentration of some particles of the system. For instance a rule may consume atp, calcium ions etc. It would be a waste of memory and time to use discrete agents to represent such particles. Instead one may declare tokens using declarations of the form:

```
%token: atp
%token: adp
```

One may then use these tokens in conjunction with a classical rule using the hybrid format:

```
'hybrid␣rule' S(x~u!1),K(y!1) | 0.1:atp -> S(x~p),K(y) | 0.1
   :adp @ 'k'
```

When applied, the above rule will consume 0.1 `atp` token and produce 0.1 `adp` token. Note that as specified by the grammar given Table 3.4, the number of consumed (and produced) tokens can be given by a sum of the form:

$$\mathit{lhs} \;\texttt{|}\; \mathtt{a}_1\texttt{:}\mathtt{t}_1 \;\texttt{+}\; \ldots \;\texttt{+}\; \mathtt{a}_n\texttt{:}\mathtt{t}_n \;\rightarrow\; \mathit{rhs} \;\texttt{|}\; \mathtt{a}'_1\texttt{:}\mathtt{t}'_1 \;\texttt{+}\; \ldots \;\texttt{+}\; \mathtt{a}'_k\texttt{:}\mathtt{t}'_k \;\texttt{@}\; \mathtt{r}$$

where each a_i, a'_i is an arbitrary algebraic expression (see Table 3.6) and each t_i, t'_i is a declared token. In the above hybrid rule, calling n_i, n'_i the evaluation of a_i and a'_i, the concentration of token t_i will decrease from n_i and the concentration of token t'_i will increase from n'_i. Importantly, the activity of a hybrid rule like the above one is still defined by $|\mathit{lhs}|*\mathtt{r}$, where $|\mathit{lhs}|$ is the number of embeddings of the lhs of the rule in the mixture, and does not take into account the concentration of the tokens it mentions. As we will see in the next section, it is however possible to make its rate explicitly depend on the concentrations of the tokens using a *variable* rate.

3.4.3 Rates

As said earlier, Kappa rules are equipped with one or two *kinetic rate(s)*. A rate is a real number, or an algebraic expression evaluated as such, called the *individual-based or stochastic rate constant*, it is the rate at which the corresponding rule is applied per instance of the rule. Its dimension is the inverse of a time $[T^{-1}]$.

The stochastic rate is related to the *concentration-based rate constant* k of the rule of interest by the following relation:

$$k = \gamma(\mathcal{A}\,V)^{(a-1)} \tag{3.1}$$

where V is the volume where the model is considered, $\mathcal{A} = 6.022 \cdot 10^{23}$ is Avogadro' s number, $a \geq 0$ is the arity of the rule (ie 2 for a bimolecular rule).

In a modeling context, the constant k is typically expressed using *molars* $M := moles\, l^{-1}$ (or variants thereof such as μM, nM), and seconds or minutes. If we choose molars and seconds, k' s unit is $M^{1-a}s^{-1}$, as follows from the relation above.

Concentration-based rates are usually favored for measurements and/or deterministic models, so it is useful to know how to convert them into individual-based ones used by KaSim. Here are typical volumes used in modeling:

- Mammalian cell: $V = 2.25\ 10^{-12} l$ ($1l = 10^{-3} m^3$), and $\mathcal{A}V = 1.35\ 10^{12}$.

  A concentration of $1M$ in a mammalian cell volume corresponds to $1.35\ 10^{12}$ molecules; $1nM \approx 1350$ molecules per cell.

- Yeast cell (haploid): $V = 4\ 10^{-14} l$, and $\mathcal{A}V = 2.4\ 10^{10}$.

  A concentration of $1M$ in a yeast cell volume corresponds to $2.4\ 10^{10}$ molecules; $1nM \approx 24$ molecules per cell. The volume is doubled in a diploid cell.

- E. Coli cell: $V = 10^{-15} l$, and $\mathcal{A}V = 10^8$.

  A concentration of $1M$ in a yeast cell volume corresponds to 10^8 molecules; $10nM \approx 1$ molecule per cell.

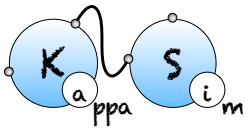
The table below lists typical ranges for deterministic rate constants and their stochastic counterparts assuming a mammalian cell volume.

Table 3.5: Example of kinetic rates.

process	k	γ
general binding	$10^7 - 10^9$	$10^{-5} - 10^{-3}$
general unbinding	$10^{-3} - 10^{-1}$	$10^{-3} - 10^{-1}$
dephosphorylation	1	1
phosphorylation	0.1	0.1
receptor dimerization	$2\ 10^6$	$1.6\ 10^{-6}$
receptor dissociation	$1.6\ 10^{-1}$	$1.6\ 10^{-1}$

3.4.4 Ambiguous molecularity

It is considered malpractice to use a Kappa rule of the form `A(x),B(y)`$\rightarrow$ `...` `@` γ in a model where this rule could be applied in a context where `A` and `B` are sometimes already connected and sometimes disconnected. Indeed, this would lead to an inconsistency in the



definition of the kinetic rate γ which should have a volume dependency in the former case and no volume dependency in the latter (see Section 3.4.3).

This sort of ambiguity should be resolved, if possible, by refining the ambiguous rule into cases that are either exclusively unary or binary. Each refinement having a kinetic rate that is consistent with its molecularity. Note that in practice, for models with a large number of agents, it is sufficient to assume that the rule `A(x),B(y)`$\rightarrow$ `...` `@` γ will have only binary instances. In this case it suffices to consider the approximate model:

```
'assumed binary AB' A(x),B(y) -> ... @ 'ga_2'
'unary AB' A(x,c!1),C(a!1,b!2),B(y,c!2) -> ... @ 'k_1'
```

There are however systems where even enumerating unary cases becomes impossible or the approximation on binary instances is wrong. As an alternative, one should use the kapma notation for ambiguous rules:

`'my rule'` *kappa_expression* $\rightarrow$ *kappa_expression* `@` $\gamma_2\{k_1\}$

which will tell KaSim to apply the above rule with a rate γ_2 for binary instances and a rate k_1 for unary instances. The obtained model will behave exactly as a model in which the ambiguous rule has been replaced by unambiguous refinements. However the usage of such rule *slowdowns simulation in a significant manner* depending on various parameters (such as the presence of large polymers in the model). We give below an example of a model utilizing binary/unary rates for rules[1].

```
%agent: A(b,c)
%agent: B(a,c)
%agent: C(b,a)
##
%var: 'V' 1
%var: 'k1' INF
%var: 'k2' 1.0E-4/'V'
%var: 'k_off' 0.1
##
'a.b' A(b),B(a) -> A(b!1),B(a!1) @ 'k2'{'k1'}
'a.c' A(c),C(a) -> A(c!1),C(a!1) @ 'k2'{'k1'}
'b.c' B(c),C(b) -> B(c!1),C(b!1) @ 'k2'{'k1'}
##
'a..b' A(b!a.B) -> A(b) @ 'k_off'
'a..c' A(c!a.C) -> A(c) @ 'k_off'
'b..c' B(c!b.C) -> B(c) @ 'k_off'
##
```

[1] This model is available in the source repository `models/poly.ka`.

```
18 %var: 'n'  1000
19 ##
20 %init: 'n'  A(),B(),C()
21 %mod: [E] = 10000 do $STOP "snap.dot"
```

Notice at lines 10-12 the use of binary/unary notation for rules. As a result binding between freely floating agents will occur at rate `'k2'` while binding between agents that are part of the same complex will occur at rate `'k1'`. Line 21 contains a *perturbation* that requires KaSim to stop the simulation after 10,000 events and output the list of molecular species present in the final mixture as a dot file (see Section 6.1) and that we give Figure 3.1.

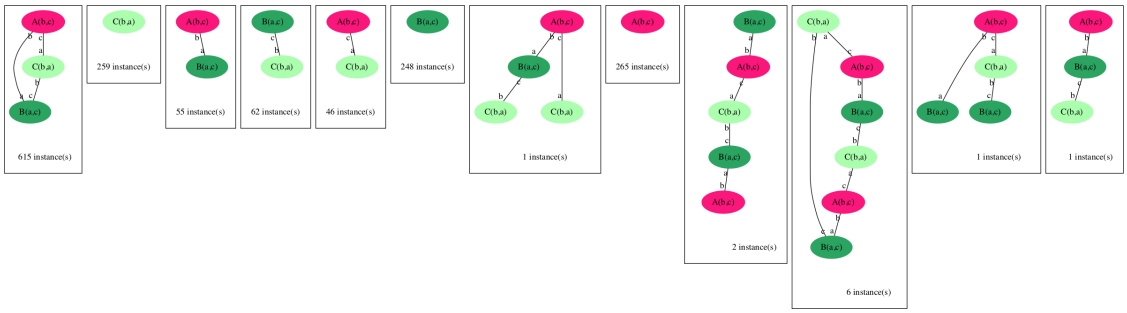


Figure 3.1: Final mixtureobtained after 10,000 events of simulation of the `poly.ka` model. The infinite rate for cycle closure allows one to obtain a large number of triangles.

For rules with unary rates, one can also specify an *horizon*. For example in the rule

```
1 'a.b' A(b),B(a) -> A(b!1),B(a!1) @ 'k_1'\{'k_2':5\}
```

the unary rate is applied only when the agents A and B are at an horizon 5 (or closer), of each other. Horizon is an algebraic expression but during simulation, it is always truncated to a positive integer. This feature *can change in the future*.

3.5 Variables

In the KF it is also possible to declare *variables* with the declaration:

`%var: 'var_name'` (*algebraic_expression*)

where `var_name` can be any string and *algebraic_expression* are defined Table 3.6 (available symbols for variable, constants and operators are given Table 3.7).

For instance the declarations

Table 3.6: Algebraic expressions

algebraic_expression ::= $x \in \mathbb{R}$ | `variable`
| *algebraic_expression* `binary_op` *algebraic_expression*
| `unary_op (`*algebraic_expression*`)`

Table 3.7: Symbol usable in algebraic expressions

`variable`	Interpretation
`[E]`	the total number of (productive) simulation events since the beginning of the simulation
`[E-]`	the total number of null events
`[Emax]`	the max (productive) event limit as set by `-l` (in `-u event` mode). Note that if unset `Emax`= ∞
`[T]`	the bio-time of the simulation
`[Tsim]`	the cpu-time since the beginning of the simulation
`[Tmax]`	the max (bio)-time limit as set by the option `-l`. Note that if unset `Tmax`= ∞
`[pp]`	the number of requested plotting interval set by the option `-p`.
`'v'`	the value of variable `'v'` (declared using the `%var:` statement)
`|t|`	the concentration of token `t`
`|`*kappa_expression*`|`	number of occurences of the pattern *kappa_expression*
`inf`	symbol for ∞

`unary/binary_op`	Interpretation
`[`f`]`	usual mathematical functions and constants with $f \in$ {`log,exp,sin,cos,tan,sqrt,pi`}
`[int]`	the floor function $x \in \mathbb{R} \mapsto \lfloor x \rfloor \in \mathbb{Z}$
`+,-,*,/,^`	basic mathematical operators (infix notation)
`[mod]`	the *modulo* operator (infix notation)
`[max]`	the *maximum* of two values
`[min]`	the *minimum* of two values

```
%var: 'homodimer' |A(x!1),A(x!1)|
%var: 'aa' 'homodimer'/2
```

define 2 variables, the first one tracking the number of embeddings of `A(x!1),A(x!1)` in the graph over time, while the second divides this value by 2: the number of automorphisms in `A(x!1),A(x!1)`. Note that variables that are used in the expression of another variable must be declared beforehand.

It is also possible to use algebraic expressions as kinetic rates as in

```
%var: 'k_on' 1.0E-6 # per molecule per second
'ab' A(x),A(x) -> A(x!1),A(x!1) @ 'k_on'/2
```

KaSim may output values of variables in the data file (see option `-p` in Chapter 4) using plot do:

```
%plot: 'var_name'
```

One may use the shortcut:

`%obs: 'var_name'` *algebraic_expression*

to declare a variable and at the same time require it to be outputted in the data file.

3.6 Initial conditions

The initial mixture to which rules in the KF will be applied are declared as

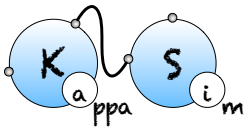
`%init:` *algebraic_expression kappa_expression*

or

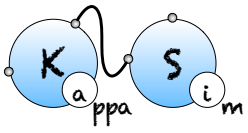
`%init:` *token_name* `<-` *algebraic_expression*

where *algebraic_expression* is evaluated before initialization of the simulation (hence all token and kappa expression values in the expression are evaluated to 0). This will add to the initial state of the model *mult* copies of the graph described by the kappa expression. Again the DCDW convention allows us not to write the complete interface of added agents (the remaining sites will be completed according to the agent's signature). For instance:

```
%var: 'n' 1000
%init: 'n' A(),A(y\intstate p
%init: ca2+ <- 0.39 #mM
```



will add 1000 instances of `A` in its default state `A(x,y~u,z~0)`, 1000 instances of `A` in state `A(x,y~p,z~0)` and a concentration of 0.39 mM of calcium ions. Recall that the concentration of calcium can be observed during simulation using `|ca2+|`. As any other declaration, `%init` can be used multiple times, and agents will add up to the initial state.


K
a
ppa
S
i
m

Chapter 4

The command line

4.1 General usage

From a terminal window, KaSim can be invoked by typing

```
$ KaSim file_1 ... file_n [option]
```

where `file_i` are the input Kappa files containing the rules, initial conditions and observables (see Chapter 3). Tables 4.1 and 4.2 summarize all the options that can be given to the simulator. Basically, one specifies an upper bound either in simulated or bio-time (arbitrary time unit), or in number of events. Note that bio-time is computed using Gillespie's formula for time advance (see Fig. 1.1) and should not be confused with CPU-time (it's not even proportional). In doubt, we recommend using a bound in number of events since the cost of one event is bounded (in CPU time) by a constant, so the CPU-time used for simulating n events is roughly k times lower than that used for simulating $k \times n$ events.

4.2 Main options

Table 4.1 summarizes the main options that are accessible through the command line. Options that expects an argument are preceded by a single dash, options that do not need any argument start with a double dash.

Two key options are the plot period `-p` (how often you want a line in the data file) and the limit `-l` of simulation. These quantities can expressed in simulated time (the default) or in number of event (using `-u event`).

Table 4.1: Command line: main options

Argument	Description
`-u` *unit*	unit of options (time/event)
`-l` *max*	Terminates simulation after $max \geq 0$ *unit*
`-initial` *min*	starts the simulation at *min unit* (data outputs convienience only)
`-p` x	plot a line in the data file every x *unit*
`-o` *file*	Set the name of data file to *file*
	Use the extension to determine the format ('.tsv', '.svg' or comma separeted value else)
`-i` *file*	Interpret *file* as an input filename
	(for compatibility with KaSim<= 3 and filenames starting by -)
`-d` *dir*	Output any produced file to the directory *dir*

4.3 Advanced options

Table 4.2 summarizes the advanced options that are accessible through the command line.

Table 4.2: Command line: advanced options

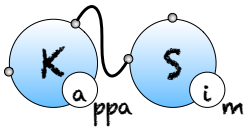
Argument	Description
`-seed` n	Seeds the pseudo-random number generator $n > 0$
`-rescale` r	Multiply each initial quantity by r
`-make-sim` *sim_file*	makes a simulation package out of the input kappa files
`-load-sim` *sim_file*	use simulation package *sim_file* as input
`--gluttony`	simulation mode that is memory intensive
	but that speeds up simulation time
`-mode batch`	Set non interactive mode (never halt waiting for an user
	action but assume default (data loosing) answer)
`-mode interactive`	Launch the toplevel just after model initialisation

4.4 Example

The command

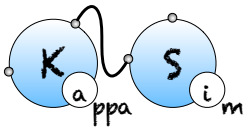
```
$ KaSim model.ka -u event -l 1000000 -p 1000 -o model.out
```

will generate a file `model.out` containing the trajectories of the observables defined in the kappa file `model.ka`. A measure will be taken every 1000 events in file `model.out`. The command




```
$ KaSim init.ka rules.ka obs.ka mod.ka -l 1.5 -p 0.0015
```

will generate a file `data.csv` (default name) containing 1000 data points of a simulation of 1.5 (arbitrary) time units of the model. Note that the input Kappa file is split in four files containing, for instance, the initial conditions, `init.ka`, the rule set, `rules.ka`, the observables, `obs.ka`, and the perturbations, `pert.ka` (see Chapter 3 for details).


K
a
ppa
S
i
m

Chapter 5

A simple model

We describe below the content of a simple Kappa model and give examples of some typical run[1].

5.1 ABC.ka

```
#### Signatures
%agent: A(x,c) # Declaration of agent A
%agent: B(x) # Declaration of B
%agent: C(x1~u~p,x2~u~p) # Declaration of C with 2 modifiable
   sites
#### Rules
'a.b' A(x),B(x) -> A(x!1),B(x!1) @ 'on_rate' #A binds B
'a..b' A(x!1),B(x!1) -> A(x),B(x) @ 'off_rate' #AB dissociation
'ab.c' A(x!_,c),C(x1~u) -> A(x!_,c!2),C(x1~u!2)  @ 'on_rate' #
   AB binds C
'mod␣x1' C(x1~u!1),A(c!1) -> C(x1~p),A(c)  @ 'mod_rate' #ABC
   modifies x1
'a.c' A(x,c),C(x1~p,x2~u) -> A(x,c!1),C(x1~p,x2~u!1) @ 'on_rate
   ' #A binds C on x2
'mod␣x2' A(x,c!1),C(x1~p,x2~u!1) -> A(x,c),C(x1~p,x2~p) @
   mod_rate #A modifies x2
#### Variables
%var: 'on_rate' 1.0E-4 # per molecule per second
```

[1]The corresponding kappa file is included in the distribution of KaSim, in the directory `models/`

```
14 %var: 'off_rate' 0.1 # per second
15 %var: 'mod_rate' 1 # per second
16 %obs: 'AB' |A(x!x.B)|
17 %obs: 'Cuu' |C(x1~u,x2~u)|
18 %obs: 'Cpu' |C(x1~p,x2~u)|
19 %obs: 'Cpp' |C(x1~p,x2~p)|
20 #### Initial conditions
21 %init: 1000 A(),B()
22 %init: 10000 C()
```

Line 1-4 of this KF contains signature declarations. Agents of type `C` have 2 sites `x1` and `x2` whose internal state may be `u`(nphosphorylated) or `p`(hosphorylated). Recall that the default state of these sites is `u` (the first one). Line 8, rule `'ab.c'` binds an `A` connected to someone on site `x` (link type `!_`) to a `C`. Note that the only rule that binds an agent to `x` of `A` is `'a.b'` at line 6. Hence the use of `!_` is a commodity and the rule could be replaced by

```
1 'alt_ab.c' A(x!1,c),B(x!1),C(x1~u) -> ...
```

There are two main points to notice about this model: `A` can modify both sites of `C` once it is bound to them. However, only an `A` bound to a `B` can connect on `x1` and only a free `A` can connect on `x2`. Note also that `x2` is available for connection only when `x1` is already modified.

5.2 Some runs

We try first a coarse simulation of 100,000 events (10 times the number of agents in the initial system).

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

Plotting the content of the `abc.csv` file one notices that nothing of significant interest happen to the observables after 250s. So we can now specify a meaningful time limit by running

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

which produces the data points whose rendering is given in Fig. 5.1. We will use this model as a running example for the next chapter, in order to illustrate various advanced concepts.

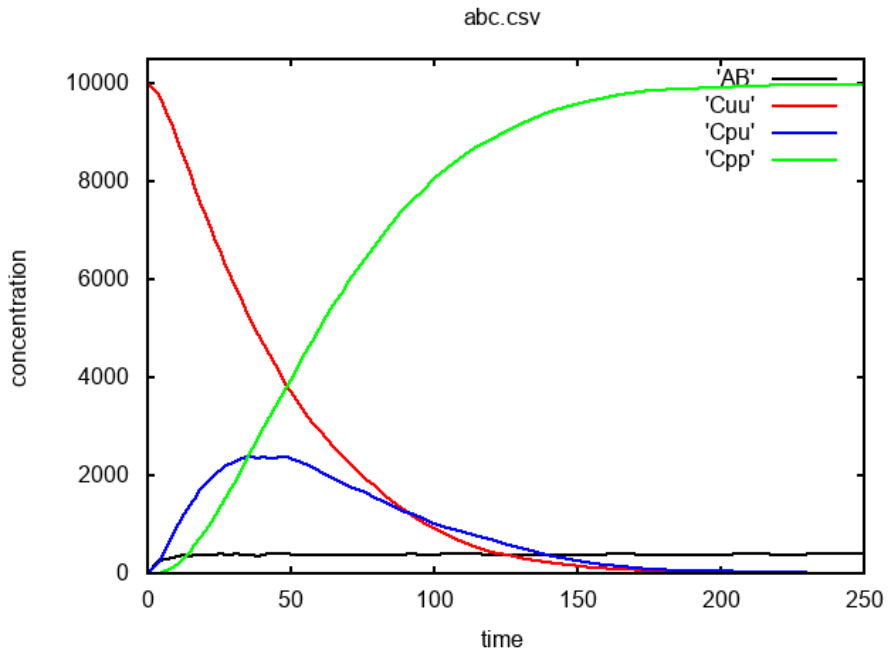
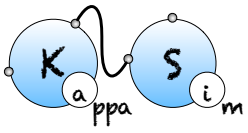


Figure 5.1: Simulation of the ABC model: population of unmodified `Cs` (observable `Cuu` in red) drops rapidly and is replaced, in a first step by simply modified `Cs` (observable `Cpu` in blue) which are in turn replaced by doubly modified `Cs` (observable `Cpp` in red). Note that the population of `AB` complexes (observable `AB` in black) stabilizes slightly below 400 individuals after about 20s.


K
a
ppa
S
i
m

Chapter 6

Advanced concepts

6.1 Perturbation language

It is possible to use variables of the model as precondition for triggering a *perturbation* of the simulation. Note that, by default, a perturbation is applied whenever its pre-condition is satisfied and then discarded. Such perturbation is called *"one shot"*. It is however possible to re-apply the same perturbation each time its pre-condition is satisfied and until a certain condition is met, using the `repeat ... until` constructors.

Basic perturbations are obtained using the declaration :

`%mod:` *boolean_expression* `do` *effect_list*

and may be applied repeatedly using:

`%mod: repeat` *boolean_expression* `do` *effect_list* `until` *boolean_expression*

where *boolean_expression* and *effect_list* are defined by the grammar given Table 6.1 (the operator `rel` can be any usual binary relation in $\{<, =, >\}$ and algebraic expressions are defined Table 3.6).

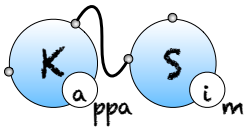
The boolean expression is used as a *precondition* that determines when the perturbation will be triggered, for instance a user writes

```
1 %mod: ([T]>10) && ('v1' /'v2') > 1 do ...
```

to indicate she wishes to trigger a perturbation whenever the simulation time has passed 10 time units and the ratio of variables `v1` over `v2` is above 1. Recall that the perturbations are "one shot" interventions on the simulation. Possible interventions are described in the following sections using examples.

Table 6.1: Perturbation expressions

perturbation_expression	::=	`%mod:` *perturbation*
		| `%mod: repeat` *perturbation* `until` *boolean_expression*
perturbation	::=	*boolean_expression* `do` *effect_list*
boolean_expression	::=	*algebraic_expression* `rel` *algebraic_expression*
		| (*boolean_expression* `||` *boolean_expression*)
		| (*boolean_expression* `&&` *boolean_expression*)
		| `[not]` *boolean_expression*
		| `[true]` | `[false]`
effect_list	::=	*effect* `;` *effect_list* | *effect*
effect	::=	`$ADD` *algebraic_expression agent_expression*
		| `$DEL` *algebraic_expression agent_expression*
		| *token_name* `<-` *algebraic_expression*
		| `$SNAPSHOT` *string_expression*
		| `$STOP` *string_expression*
		| `$FLUX` *string_expression boolean*
		| `$TRACK 'var_name'` *boolean*
		| `$UPDATE 'var_name'` *algebraic_expression*
		| `$PLOTENTRY`
		| `$PRINT` <*string_expression*>
		| `$PRINTF` *string_expression* <*string_expression*>
string_expression	::=	ε | `"string" .` *string_expression*
		| *algebraic_expression* `.` *string_expression*
boolean	::=	`[true]` | `[false]`



Note on time dependent preconditions:
Consider a perturbation of the form:

`%mod:` f`(t)=`x `do ...`

where $f(t)$ is an algebraic expression dependent on time and x an arbitrary algebraic expression. Having in mind the simulation algorithm implemented by KaSim, at the beginning of an event loop, both $f(t)$ and x will be evaluated. It is very unlikely (in general with a probability equal to 0) that both values coincide. Currently KaSim is not equipped with a solver able to detect that in the past of the current state, there was a t that made the precondition hold, unless the equation is trivial to solve. Therefore the only time dependent precondition with an equality test that is allowed in KaSim has to be of the form [T] $=n$ with $n \in \mathbb{N}$. For instance: `%mod: [T]=10 do $STOP` will interrupt the simulation after exactly 10 time units.

6.1.1 Adding or deleting agents during a simulation

Continuing with the ABC model, the perturbation effect: `$ADD n C(x1~p)` will add $n \geq 0$ instances of `C` with `x1` already in state `p` (and the rest of its interface in the default state as specified line 4 of ABC.ka). Also the perturbation effect: `$DEL inf B(x!_)` will remove *all* `B`s connected to some agent from the mixture.

There are various ways one can use perturbations to study more deeply a given kappa model. A basic illustration is the use of a simple perturbation to let a system equilibrate before starting a real simulation. For instance, as can be seen from the curve given in Fig. 5.1, the number of AB complexes is arbitrarily set to 0 in the initial state (all `A`s are disconnected from `B`s in the initial mixture). In order to avoid this, one can modify the kappa file the following way: we set the initial concentration of `C` to 0 by deleting line 22. Now we introduce `C`s after 25 t.u using the perturbation: `%mod: [T]=25 do $ADD 10000 C()`

The modified kappa file is available in the source repository, in the `model/` directory (file `abc-pert.ka`). Running again a simulation (a bit longer) by entering in the command line:

```
$ KaSim ABC-pert.ka -l 300 -p 0.3 -o abc2.out
```

one obtains the curve given in Fig. 6.1.

6.1.2 Using snapshots to define a new initial state

In the previous example, we let the system evolve for some time without its main reactant `C` in order to let other reactants go to a less arbitrary initial state. One may object that

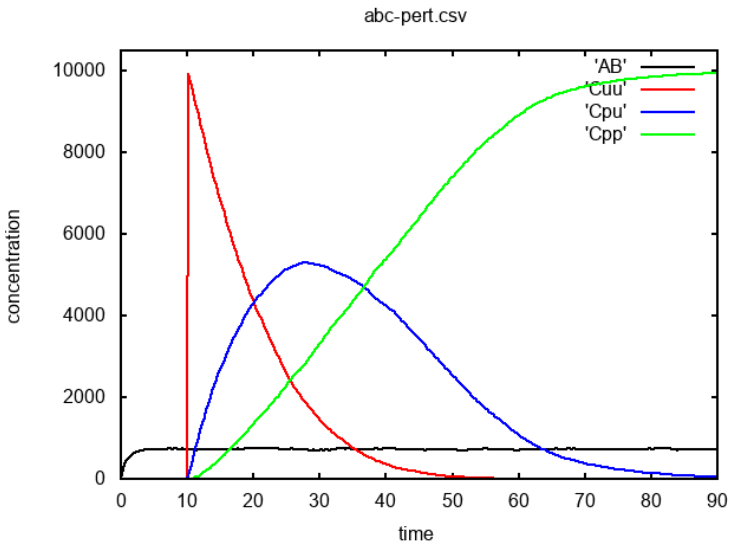


Figure 6.1: Simulation of the ABC model with a perturbation: for t<25s, only `'a.b'` and `'a..b'` rules may apply. This enables the concentration of `'AB'` complexes to go to steady state, before introducing fresh `C`s at t=25s.

this way of proceeding is CPU-time consuming if one has to do this at each simulation. An alternative is to use the `$SNAPSHOT` primitive that allows a user to export a snapshot of the mixture at a given time point as a new (piece of) kappa file. For instance, the declaration: `%mod: [E-]/([E]+[E-])>0.9 do $SNAPSHOT "prefix"` will ask KaSim to export the mixture the first time the percentage of null events reaches 90%. The exported file will be named `prefix_`n`.ka` where n is the event number at which the snapshot was taken. One may also use a *string_expression* to construct any prefix using local variables. Note that one may omit to define a prefix and simply type: `%mod: [E-]/([E]+[E-])>0.9 do $SNAPSHOT` in which case the default prefix `snap.ka` will be used for naming snapshots. If the name already exists a counter will be appended at the end of the file to prevent overwriting. Snapshots can be performed multiple times, for instance every 1000 events, using the declaration:

```
1 %mod: repeat ([E] [mod] 1000)=0 do $SNAPSHOT "abc.ka" until [
    false]
```

which results in KaSim producing a snapshot every 1000 (productive) events until the simulation ends. The perturbation `$STOP "final_state.ka"` will terminate the simulation whenever its precondition is satisfied and produce a snapshot of the last mixture. Note that instead of producing kappa files, one may use snapshot perturbations to produce an image

of the mixture in the dot/html format using the parameter by specifying the extention in the name skeleton (%mod: [E-]/([E]+[E-])>0.9 do $SNAPSHOT ‘‘snap.dot).

6.1.3 Changing the value of a token

The concentration of any token can be reset on the fly using a perturbation. For instance the declaration: %mod: repeat (|a|<100 do a <- |a|*2)until [false] will double the concentration of token a each time it gets below 100.

6.1.4 Causality analysis

In our ABC example, adding the instruction: %mod: [true] do $TRACK 'Cpp'[true] will ask KaSim to turn on causality analysis for the observable 'Cpp' since the beginning of the simulation, and display the causal explanation of every new occurrence of 'Cpp', until the end of the simulation. The explanation, that we call a *causal flow*, is a set of rule application ordered by causality and displayed as a graph using dot format. In this graph, an edge r$\longrightarrow$ r' between two rule applications r and 'r' indicates that the first rule application has used, in the simulation, some sites that were modified by the application of the former. We show Fig. 6.2 an example of such causal flow.

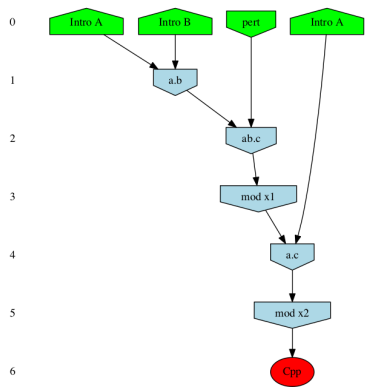


Figure 6.2: Causal flow for the observable 'Cpp' of the ABC model. Plain arrows represent causal dependency, dotted arrows show asymmetric conflict between rule occurrences. Here the 'ab.c' rule has to occur before the 'a.b' rule. Red observable indicate that the last rule allowed one to observe a new instance of 'Cpp'.

Causality analysis of the observable Cpp can be turned off at any time using a declaration of the form: %mod: [T]>25 do $TRACK 'Cpp'[false]

Each time KaSim detects a new occurrence of the observable that is being tracked, it will dump it's causal past as a graph using the dot format (see Fig. 6.2 above). The name of the file in which the causal flow is stored can be set using the `%def` instruction (see Section 6.4).

Compressing causal flows.

In general pure causal flows will contain a lot of information that modelers may not wish to consider. Indeed in classical flows, causality (represented by an edge between to rule applications in the graph) is purely local. Therefore a sequence $a \to b \to c$ only implies that an instance of rule a caused an instance of rule b which in turn created an instance of the observable c. However it does not imply that a was "necessary" for c to occur (for instance c might have been possible before a but not after, and b would be simply re-enabling c). It is possible to tell KaSim to retain only events that are more strongly related to the observable using two compression techniques (see Ref. [2] for formal details). Intuitively, in a *weakly* compressed causal flow one has the additional property that if an event e is a (possibly indirect) cause of the observable, then preventing e from occurring would have prevented the rest of the causal flow to occur (ie it is not possible to reconstruct a computation trace containing the observable with the events that remain in the causal flow). A *strongly* compressed causal flow enjoys the same property with an additional level of compression obtained by considering different instances of the same rule to be indistinguishable. Note that causal flow compressions may be memory and computation demanding. For large systems it may be safer to start with weak compressions only.

The type of compression can be set using the `%def` instruction (see Section 6.4). For instance: `%def: "displayCompression" "none" "weak" "strong"` will ask KaSim to output 3 versions of each computed causal flow, with all possible degrees of compressions. Each causal flow is outputted into a file `[filename][Type]_`n`.dot` where `filename` is the default name for causal flows which can be redefined using the parameter `cflowFileName`, `Type` is the type of compression (either nothing or `Strongly`, or `Weakly`) and n is the identifier of the causal flow. For each compression type a summary file, named `[filename][Type]Summary.dat`, is also produced. It allows to map each compressed causal flow to the identifier of its uncompressed version (row `#id`), together with the production time T and event number E at which the observable was produced. It also contains information about the size of the causal flow.

6.1.5 Flux maps

The *flux map* is a powerful observation that tracks, on the fly, the influence that rule applications have on each others. It is dynamically generated and tracks effective impacts (positive

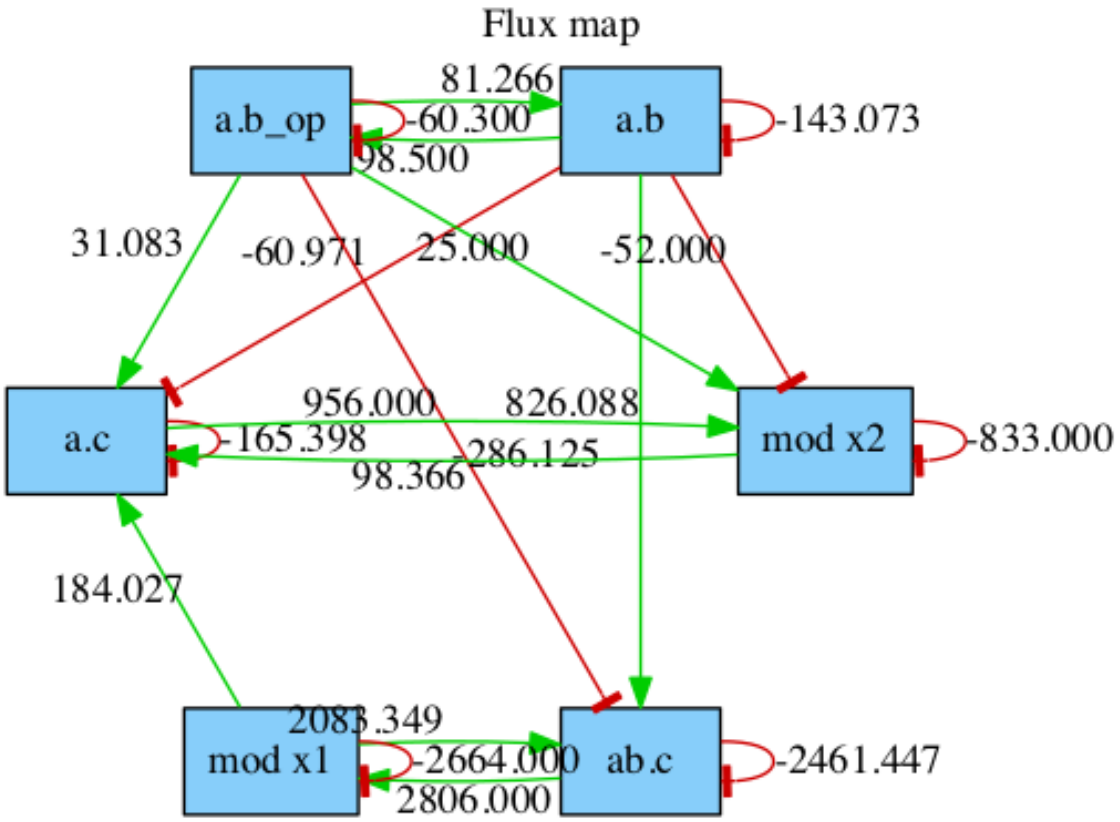


Figure 6.3: Flux map of the `abc.ka` model, taken from t=0 to t=20 time units. The `A` releasing rules `a..b` and `mod x2` are contributing very little to the activity of `a.c` which is a sign of an excess of free `A`s in the system at this time interval.

or negative) a every rule application. The flux map can be computed using declarations of the form:

```
%mod: [true] do $FLUX "flux.dot" [true]
%mod: [T]>20 do $FLUX "flux.dot" [false]
```

The resulting *flux map* is a graph where a positive edge between rules r and s (in green) indicates an overall positive contribution of r over s. Said otherwise, the sum of r applications increased the activity of s. Conversely, a negative edge (in red) will indicate that r had an overall negative impact on the activity of s. Note that the importance of the flux between two rules can be observed by looking at the label on the edges that indicate the overall activity transfer (positive or negative) between the rules. The above declaration produce a flux map that is shown Fig. 6.3. Note that flux may vary during time, therefore the time or event limit of the simulation is of importance and will likely change the aspect of the produced map.

6.1.6 Updating kinetic rates on the fly

Any variable between simple quotes can be updated during a simulation using a declaration of the form: `%mod: 'Cpp'> 500 do $UPDATE 'k_on'0.0`

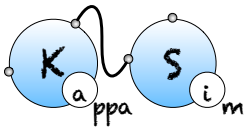
This perturbation will be applied whenever the observable `'Cpp'` will become greater than 500. Its effect will be to set the on rate of all binding rules to 0. Note that according to the grammar given Table 6.1, one may use any algebraic expression as the new value of the variable. For instance: `%mod: 'Cpp'> 500 do $UPDATE 'k_on''k_on'/100` will cause the on rate of all rules to decrease a hunderd fold. Note that it is possible to override the kinetic rate of a specific rule: in our ABC example, the declaration: `%mod: 'Cpp'> 500 do $UPDATE 'a.b'inf` will set the kinetic rate of rule `'a.b'` to infinity.

6.1.7 Combining several effects in a single perturbation

As an example, consider the computation of causal flows between $t = 10$ and $t = 20$ using the declarations:

```
%mod: [T]>10 do $TRACK 'Cpp' [true]
%mod: [T]>20 do $TRACK 'Cpp' [false]
```

The above declaration will ask KaSim to analyze each new occurrence of 'Cpp' in that time interval. If n new instances took place, then KaSim will have to compute n causal flows. One may want to bound the number of computed flows to a certain value, say 10. One may do so using the combination of perturbations and variables given below:



```
%var: 'x' 0
%mod: [T]>10 do ($TRACK 'Cpp' [true] ; $UPDATE 'x' 'Cpp')
%mod: [T]>20 || ('x' > 0 && 'Cpp' - 'x' > 9) do $TRACK 'Cpp' [
   false]
```

The first line is a declaration of an x variable that is initially set to 0. Note that the second line is a perturbation that contains two simultaneous effects, the first one triggering causality analysis and the second one updating the value of variable x to the current value of variable `'Cpp'`. The last line stops causality analysis whenever time is greater than 20 or when 10 new observables have been found (the difference between the current value of `'Cpp'` and x.

6.1.8 Printing values during a simulation

The effect `$PRINT` *<string_expression>* enables one to output values during a computation to standard output, or to a specific file when using `$PRINTF`. For instance:

```
%mod: repeat \
|A|<0 do $PRINTF "token_".[E].".dat" <"Token A is: " . |A| . "
   at time=". [T]>\
until [false]
```

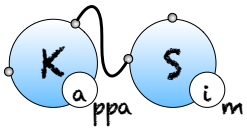
will ask KaSim to output the value of token `A` in a file "token_n.dat" which changes at each new productive event, each time its value gets below 0.

6.1.9 Add an entry in the output data

The effect `$PLOTENTRY` outputs a line with the current value of observables in the data file. For example, `%mod: repeat [E] [mod] 10 = 0 do $PLOTENTRY until [false]` will store the value of observables every 10 productive events.

6.2 Implicit signature

KaSim permits users in a hurry to avoid writing agent signatures explicitly using the option `--implicit-signature` of the command line. The signature is then deduced using information gathered in the KF. Note that it is not recommended to use the DCDW convention for introduced agents in conjunction with the `--implicit-signature` option unless the default state of all sites is mentioned in the `%init` declarations or in the rules that create agents.



6.3 Simulation packages

The simulation algorithm that is implemented in KaSim requires an initialization phase whose complexity is proportional to $R*G$ where R is the cardinal of the rule set and G the size of the initial mixture. Thus for large systems, initialization may take a while. Whenever a user wishes to run several simulations of the *same* kappa model, it is possible to skip this initialization phase by creating a *simulation package*. For instance:

`KaSim abc.ka -l` n `-make-sim abc.kasim`

will generate a standard simulation of the `abc.ka` model, but in addition, will create the simulation package `abc.kasim` (.kasim extension is not mandatory). This package is a binary file, ie not human readable, that can be used as input of a new simulation using the command:

`KaSim -load-sim abc.kasim -l` k

Note that this simulation is now run for k time units instead of n. Importantly, simulation packages can only be given as input to the *same* KaSim that produced it. As a consequence, recompiling the code, or obtaining different binaries, will cause the simulation package to become useless.

6.4 Simulation parameters configuration

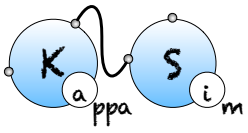
In the KF (usually in a dedicated file) one may use expressions of the form:

`%def: "`*parameter_name*`" "`*parameter_value*`"`

where tunable parameters are described table 6.2 (default values are given first in the possible values column).

Table 6.2: User defined parameters

parameter	possible values	description
Causality analysis		
`"displayCompression"`	any combination of `"none", "strong", "weak"`	type of compression
`"cflowFileName"`	`"cflow"`, any string	file name prefix for causal flows
`"dotCflows"`	`"no", "html"`	generate causal flows in html
	`"yes", "dot"`	generate causal flows in dot
	`"json"`	generate causal flows in json
Pretty printing		
`"colorDot"`	`"no", "yes"`	use colors in dot format files
`"progressBarSymbol"`	`"#"` or any character	symbol for the progress bar
`"progressBarSize"`	`"60"` or any integer	length of the progress bar
Simulation options		
`"dumpIfDeadlocked"`	`"no","yes"`	Snapshot when simulation is stalled
`"maxConsecutiveClash"`	`"2"` or any integer	number of consecutive clashes before giving up
		square approximation
`"storeUnaryHorizon"`	`"true", "false"`	Record distance between connected components when unary rule applies


K
a
ppa
S
i
m

Chapter 7

The KaSa static analyser

7.1 General usage

From a terminal window, KaSa can be invoked by typing

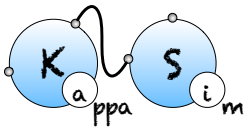
```
$ KaSa file_1 ... file_n [option]
```

where `file_i` are the input Kappa files containing the rules, initial conditions and observables (see Chapter 3).

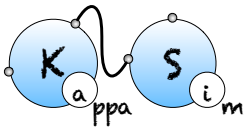
All the options are summarised as follows:

```
General options
  --help            Verbose help
   -h               Short help
  --version         Show version number
  --gui             GUI to select
  --(no-)expert     Expert mode (more options)

0_Actions
  --do-all
      launch everything
  --reset-all
      launch nothing
  --(no-)compute-contact-map    (default: enabled)
      compute the contact map
  --(no-)compute-influence-map    (default: enabled)
      compute the influence map
```

```
  --(no-)compute-ODE-flow-of-information    (default: disabled)
      Compute an approximation of the flow of information in the ODE
      semantics
  --(no-)compute-stochastic-flow-of-information    (default: disabled)
      Compute an approximation of the flow of information in the stochastic
      semantics
  --(no-)compute-reachability-analysis    (default: enabled)
      Compute an approximation of the states of agent sites
  --(no-)views-domain    (default: enabled)
      enable local views analysis
  --(no-)double-bonds-domain    (default: enabled)
      enable double bonds analysis
  --(no-)sites-accross-bonds-domain    (default: enabled)
      enable the analysis of the relation amond the states of sites in
      connected agents
  --(no-)compute-local-traces    (default: disabled)
      Compute the local traces of interesting parts of agent interfaces

1_Output
  --output-directory <value>
      Default repository for outputs
  --output-contact-map-directory <name>   (default: output)
      put the contact map file in this directory
  --output-influence-map-directory <name>   (default: output)
      put the influence map file in this directory
  --output-local-traces-directory <name>   (default: output)
      put the files about local traces in this directory
  --output-log-directory <name>   (default: output)
      put the log files in this directory
  --contact-map-format DOT    (default: DOT)
      Tune the output format for the contact map
  --influence-map-format DOT | HTML    (default: DOT)
      Tune the output format for the influence map
  --local-traces-format DOT | HTML    (default: DOT)
      Tune the output format for the local transition systems
  --output-contact-map <name>   (default: contact)
      file name for the contact map output
  --output-influence-map <name>   (default: influence)
      file name for the influence map

2_Reachability_analysis
```




```
  --(no-)compute-reachability-analysis    (default: enabled)
      Compute an approximation of the states of agent sites
  --enable-every-domain
      enable every abstract domain
  --disable-every-domain
      disable every abstract domain
  --contact-map-domain static | dynamic    (default: dynamic)
      contact map domain is used to over-approximate side-effects
  --(no-)views-domain    (default: enabled)
      enable local views analysis
  --(no-)double-bonds-domain    (default: enabled)
      enable double bonds analysis
  --(no-)sites-accross-bonds-domain    (default: enabled)
      enable the analysis of the relation amond the states of sites in
      connected agents
  --verbosity-level-for-reachability-analysis Mute | Low | Medium | High |
                                              Full    (default: Low)
      Tune the verbosity level for the reachability analysis
  --output-mode-for-reachability-analysis raw | kappa | english    (default: kappa)
      post-process relation and output the result in the chosen format

3_Trace_analysis
  --(no-)compute-local-traces    (default: disabled)
      Compute the local traces of interesting parts of agent interfaces
  --(no-)show-rule-names-in-local-traces    (default: enabled)
      Annotate each transition with the name of the rules in trace
      abstraction
  --(no-)use-macrotransitions-in-local-traces    (default: disabled)
      Use macrotransitions to get a compact trace up to change of the
      interleaving order of commuting microtransitions
  --(no-)ignore-trivial-losanges    (default: disabled)
      Do not use macrotransitions for simplifying trivial losanges
  --output-local-traces-directory <name>   (default: output)
      put the files about local traces in this directory
  --local-traces-format DOT | HTML    (default: DOT)
      Tune the output format for the local transition systems

4_Contact_map
  --(no-)compute-contact-map    (default: enabled)
      compute the contact map
  --output-contact-map-directory <name>   (default: output)
```



```
      put the contact map file in this directory
  --contact-map-format DOT    (default: DOT)
      Tune the output format for the contact map
  --contact-map-accuracy-level Low | High    (default: Low)
      Tune the accuracy level of the contact map
  --(no-)pure-contact    (default: disabled)
      show in the contact map only the sites with a binding state
  --output-contact-map <name>   (default: contact)
      file name for the contact map output

5_Influence_map
  --(no-)compute-influence-map    (default: enabled)
      compute the influence map
  --influence-map-accuracy-level Low | Medium    (default: Medium)
      Tune the accuracy level of the influence map
  --output-influence-map-directory <name>   (default: output)
      put the influence map file in this directory
  --influence-map-format DOT | HTML    (default: DOT)
      Tune the output format for the influence map
  --output-influence-map <name>   (default: influence)
      file name for the influence map

6_FLow_of_information
  --(no-)compute-ODE-flow-of-information    (default: disabled)
      Compute an approximation of the flow of information in the ODE
      semantics
  --(no-)compute-stochastic-flow-of-information    (default: disabled)
      Compute an approximation of the flow of information in the stochastic
      semantics

7_Debugging_info
  --output-log-directory <name>   (default: output)
      put the log files in this directory
  --(no-)debugging-mode    (default: disabled)
      dump debugging information
  --(no-)unsafe-mode    (default: enabled)
      Exceptions are gathered at the end of the computation, instead of
      halting it

(57 options)
```

Orders in option matter, since they can be used to toggle on/off some functionalities or to assign a value to some environment variables. The options are interpreted from left to right.

More options are available in the OCaml file `KaSa_rep/config/config.ml` and can be tuned before compilation.

7.2 Graphical interface

7.2.1 Launching the interface

The graphical interface can be launched by typing

```
$ KaSa
```

without any option.

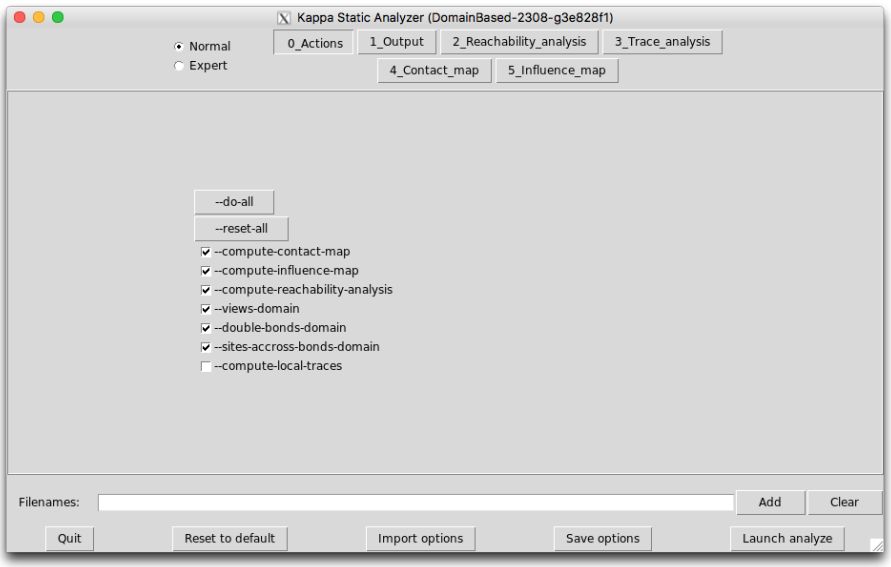
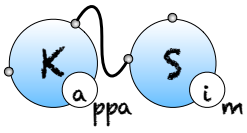


Figure 7.1: KaSa graphical interface - sub-tab `0_Actions`

7.2.2 The areas of interests

There are five different areas of importance in the graphical interface:



1. On the top left of the window, a button allows for the selection between the Normal and the Expert mode (other modes may be available if activated at compilation). In expert modes, more options are available in the graphical interface.
2. On the top center/right, some button allows for the selection of the tab. There are currently six sub-tabs available: `0_Actions`, `1_Output`, `2_Reachability_analysis`, `3_Trace_analysis`, `4_Contact_map`, `5_Influence_map`.
3. Center: The options of the selected sub-tab are displayed and can be tuned.

   Contextual help is provided when the mouse is hovered over an element.

   The interface will store the options that are checked or filled and the order in which they have been selected. When launched, the analysis interprets these options in the order they have been entered.

   Some options appear in several sub-tabs. They denote the same option and share the same value.
4. File selector: The file selector can be used to upload as many kappa files as desired. The button '`Clear`' can be used to reset the selection of files.
5. Bottom: Some buttons are available. The button '`Quit`' can be used to leave the interface. The button '`Reset to default`' tune all the options to their default value. The button '`Import options`' can be used to restore the value of the options as saved during a previous session of the graphical interfaces. The button '`Save options`' can be used to save the value of the options for a further session. The button '`Launch analyze`' launch KaSa with the current options.

   Importantly, options are saved automatically under various occasions. Thus, it is possible to restore the value of the options before the last reset, before the last quit, or before the last analysis.

7.2.3 The sub-tab `0_Actions`

The sub-tab `0_Actions` (see Fig. 7.1) contains the main actions which can be performed.

- The button `-do-all` activates all the functionalities.
- The button `-reset-all` inactivates all the functionalities.
- The option `-compute-contact-map` can be used to (des)activate the computation of the contact map.
- The option `-compute-influence-map` can be used to (des)activate the computation of the influence map.

- The option `-compute-reachability-analysis` can be used to (des)activate the computation of the reachability analysis.
- The option `-compute-local-traces` can be used to (des)activate the computation of the trace analysis.

7.2.4 The sub-tab `1_Output`

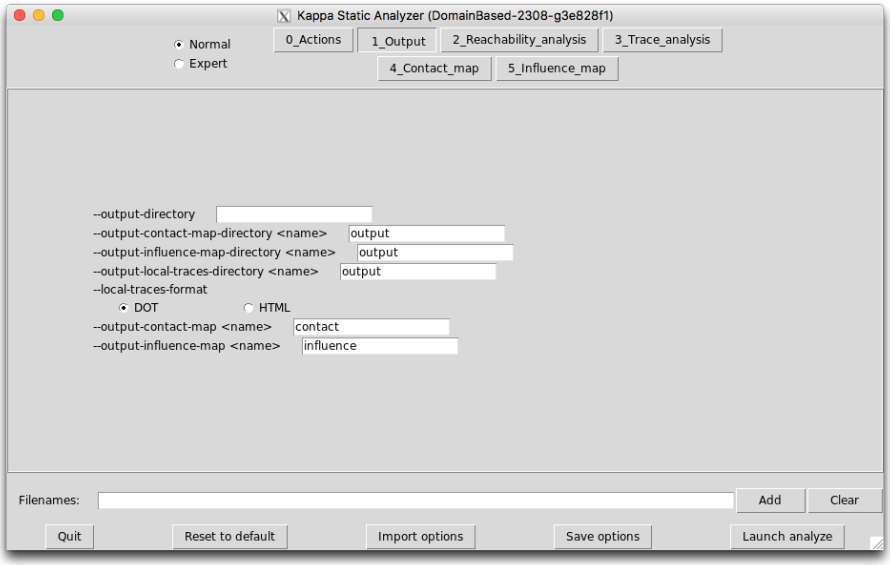


Figure 7.2: KaSa graphical interface - sub-tab `1_output`

The sub-tab `1_Ouput` (see Fig. 7.2) contains the names of the output files.

- The field `-output-directory` can be used to set the repository where output file are written. KaSa will create this repository, if it does not exist.
- The field `-output-contact-map-directory` can be used to set the reporitory where the output file for the contact map is written, if a contact map is requested. KaSa will create this repository, if it does not exist.
- The field `-output-influence-map-directory` can be used to set the reporitory where the output file for the influence map is written, if an influence map is requested. KaSa will create this repository, if it does not exist.
- The field `-output-local-traces-directory` can be used to set the reporitory where the output file for the result of trace analysis is written, if this analysis is requested. KaSa will create this repository, if it does not exist.

- The field `-output-contact-map` contains the name of the file for the contact map.
- The field `-output-influence-map` contains the name of the file for the influence map.

When a file already exists, it is overwritten without any warning.

7.3 Reachability analysis

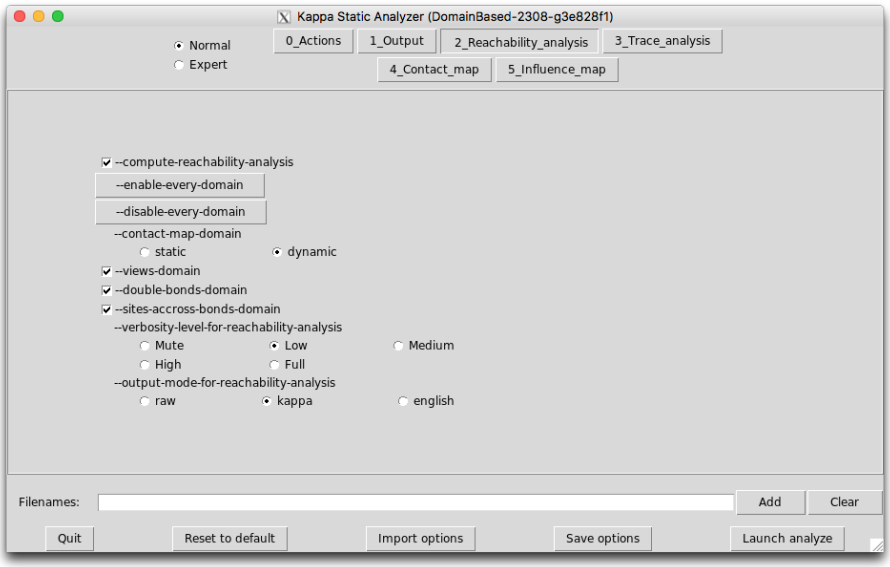
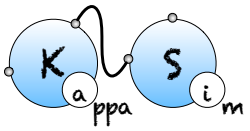


Figure 7.3: KaSa graphical interface - sub-tab `2_Reachability_analysis`

Reachability analysis aimed at detecting statically properties about the bio-molecular species that can be formed in a model. Knowing whether, or not, a given bio-molecular species, can be formed in a model is an undecidable problem [15]. Thus, our analysis is approximate. Indeed, it computes an over-approximation of the set of the bio-molecular species that can be reached from the initial state of the model, by applying an unbounded number of computation steps. As formalized in [5, **?**], the abstraction consists in:

1. firstly, ignoring the number of occurrences of bio-molecular species (we assume that whenever a bio-molecular species can be formed, then it can be formed as many time as it could be necessary),
2. secondly, abstracting a bio-molecular species by the set of its properties.

The classes of properties of interest are encoded in so called abstract domains, which can be independently enabled/disabled. The whole analysis can be understood as a mutual recur-



sion between smaller analyses (one per abstract domain), that communicates information between each other at each step of the analysis. We took the same scheme of collaboration between abstract domains as in [?].

As an example, we consider the following model:

```
%agent: E(x)
%agent: R(x,c,cr,n)

%init: 1 E()
%init: 1 R()

'E.R' E(x),R(x) -> E(x!1),R(x!1) @1
'E/R' E(x!1),R(x!1,c) -> E(x),R(x,c) @1
'R.R' R(x!_,c),R(x!_,c) -> R(x!_,c!1),R(x!_,c!1) @1
'R/R' R(c!1,cr,n),R(c!1,cr,n) -> R(c,cr,n),R(c,cr,n) @1
'R.int' R(c!1,cr,n),R(c!1,cr,n) -> R(c!1,cr!2,n),R(c!1,cr,n!2)
   @1
'R/int' R(cr!1),R(n!1) -> R(cr),R(n) @1
'obs' R(x,c,cr!_,n!_) -> R(x,c,cr,n) @1
```

Typing the following instruction:

```
KaSa reachability.ka --reset-all --compute-reachability-analysis
```

will perform the reachability analysis on the model `reachability.ka`.

We obtain the following result:

```
Kappa Static Analyzer (DomainBased-2343-gec98fbf) (with Tk
interface)
Analysis launched at 2016/12/02 09:42:09 (GMT+1) on
dhcp195.dmi.ens.fr
Parsing ../kappa/reachability.ka...
done
Compiling...
Reachability analysis...

------------------------------------------------------------
* There are some non applyable rules
------------------------------------------------------------
rule 6: obs will never be applied.
------------------------------------------------------------
every agent may occur in the model
```

```
------------------------------------------------------------
* Non relational properties:
------------------------------------------------------------
E() => [ E(x) v E(x!R.x) ]
R() => [ R(c) v R(c!R.c) ]
R() => [ R(n) v R(n!R.cr) ]
R() => [ R(cr) v R(cr!R.n) ]
R() => [ R(x) v R(x!E.x) ]

------------------------------------------------------------
* Relational properties:
------------------------------------------------------------
R() =>
   [
  R(c,cr,n,x!E.x)
v R(c!R.c,cr!R.n,n,x!E.x)
v R(c!R.c,cr,n,x!E.x)
v R(c!R.c,cr,n!R.cr,x!E.x)
v R(c,cr,n,x)
   ]
------------------------------------------------------------
* Properties in connected agents
------------------------------------------------------------
R(c!1),R(c!1) =>
[
  R(c!1,cr!R.n),R(c!1,cr)
v R(c!1,cr),R(c!1,cr)
v R(c!1,cr),R(c!1,cr!R.n)
]
R(c!1),R(c!1) =>
[
  R(c!1,cr!R.n),R(c!1,n!R.cr)
v R(c!1,cr),R(c!1,n)
]
R(c!1),R(c!1) =>
[
  R(c!1,n!R.cr),R(c!1,n)
v R(c!1,n),R(c!1,n)
v R(c!1,n),R(c!1,n!R.cr)
]
```

```
------------------------------------------------------------
* Properties of pairs of bonds
------------------------------------------------------------
R(c!R.c,cr!R.n) => R(c!1,cr!2),R(c!1,n!2)
R(c!R.c,n!R.cr) => R(c!1,n!2),R(c!1,cr!2)
execution finished without any exception
```

This result is displayed in the standard output, and it is made of six parts.

The first two parts provide an enumeration of dead rules and dead agents. The next parts display what we call refinement lemmas. A refinement lemma is made of a precondition (on the left of the implication symbol) that is a site graph, and a postcondition (on the right of the implication symbol) that is a list of site graphs. Each site graph in the post-condition is a refinement of the precondition (the position of agent matters: the n-th agent in the precondition corresponds to the n-th agent in each site graph in the postcondition, but site graphs in a postcondition may have more agents than the site graph in the corresponding precondition). The meaning of a refinement lemma is that every embedding between its precondition into a reachable state can be refined/extended into an embedding from one site graph in its postcondition into the same reachable state. This way, a refinement lemma provides an enumeration of all the potential contexts for the precondition.

We now detail the six different parts:

- **Detection of dead rules.** A rule is called dead, if there is no trace starting from the initial state in which this rule is applied. The analysis reports the list of the rules it has detected to be dead. Due to the over-approximation, it may happen that a dead rule is not discovered by the analysis. Yet, every rule that is reported as dead, is dead indeed.

  In our example, we notice that the rule '`obs`' can never be trigered.

- **Detection of dead agents.** An agent is called dead, if there is no trace starting from the initial state with at least one state in which this agent occurs. The analysis reports the list of the agents it has detected to be dead. Due to the over-approximation, it may happen that a dead agent is not discovered by the analysis. Yet, every agent that is reported as dead, is dead indeed.

  In our example, there are no dead agent.

- **Non-relational properties.** The analysis detects for each kind of site, the set of states this site can take. Due to the over-approximation, the analysis reports a super-set of the set of the potential states. Yet, we are sure that a given site only take states within this set.

  In our example, the site `cr` of `R` may be free, or bound to the site `n` of an agent `R`.

```
Kappa Static Analyzer (DomainBased-2343-gec98fbf) (with Tk
interface)
Analysis launched at 2016/12/02 09:42:09 (GMT+1) on
dhcp195.dmi.ens.fr
Parsing ../kappa/reachability.ka...
done
Compiling...
Reachability analysis...
execution finished without any exception
```

Figure 7.4: Reachability analysis of the model `reachbility.ka` with verbosity level "Mute".

- **Relational properties.** The analysis detects some relationships among the states of packs of sites within each agent, hence capturing potential valuations for local views [10, 5]. Due to the over-approximation of the analysis, the analysis may fail in discovering a relationship. But each relationship that is found by the analysis, is satisfied.

  In our example, the states of the sites `c`, `cr`, `n`, and `x` of `R` are entangled with a relational property (othewise, we would have 5^2 elements in the post-condition).

- **Properties in connected agent.** When two agents are connected, there may be a relation among the states of theirs respective sites. This abstraction [12] collects for each kind of bonds, the relation between the state of one site in the first agent and the state of one site in the second agent. Due to the over-approximation, the analysis reports a super-set of the set of the potential pairs of states.

  This abstraction aimed at capturing information about protein transportation. It is quite common to model the location of a protein as the internal state of a fictitious site. With such an encoding, it might be important to ensure that two connected proteins, are always located in the same location. This abstraction focuses on this kind of properties.

- **Properties of pairs of bonds.**

  It might be interesting to know whether a protein can be bound to another protein twice simultaneously, and whether a protein can be bound to two instances of a same protein simultaneously. This abstraction [12] captures this kind of constraint. It can be used to prove that some proteins do not polymerize.

  In our example, when a `R` has its sites `cr` and `c` bound, they are necessarily bound to the same instance of `R`. The same statement holds for the sites `cr` and `n`.

Now we describe the options that are available on this sub-tab.

```
Applying rule 6: obs:
the precondition is not satisfied yet
```

Figure 7.5: Reachability analysis: one rule that cannot be applied yet, according to the bio-molecular species already constructed.

```
Applying rule 0: E.R:
the precondition is satisfied
```

Figure 7.6: Reachability analysis: one rule successfully applied

The option `--compute-reachability-analysis` can be used to switch on/off then reachability analysis.

The option `--enable-every-domain` can be used to switch on every abstract domain, whereas the option `--disable-every-domain` can be used to switch off every abstract domain.

The option `--contact-map-domain` impacts the way side-effects are handled with during the analysis. In `static` mode, we consider that every bond that occurs syntactically in initial state, in the rhs of a rule, or in a introduction directive of a perturbation, may be released by side-effects. In `dynamic` mode, only the bond that has been encountered so far during the analysis are considered.

The option `--views-domain` can be used to switch on/off the views domains that combine the non-relational analysis and the relational analysis.

The option `--double-bonds-domain` can be used to switch on/off the analysis of potential double bonds between between proteins.

The option `--site-accross-bonds-domain` can be used to switch on/off the analysis of the relations among the states of the sites in connected proteins.

It is possible to get more details about the computation of the analysis by tuning the verbosity level of the view analysis:

- With the option `--verbosity-level-for-reachability-analysis Mute`, nothing is displayed. Even the result of the analysis is omitted (eg. see Fig. 7.4).
- With the option `--verbosity-level-for-reachability-analysis Low`, only the result of the analysis is displayed (by default).
- With the option `--verbosity-level-for-reachability-analysis Medium`, the analysis also describes which rules are applied and in which order.

  When trying to apply a rule, the analysis may detect that the rule cannot be applied

```
Views in initial state:

E(x!free)
--
Views in initial state:

R(x!free,c!free,cr!free,n!free)
```

Figure 7.7: Reachability analysis: extensional description of initial states.

```
Applying rule 0: E.R:
the precondition is satisfied

rule 0: E.R is applied for the first time

Updating the views for E(x!)

E(x!R@x)


Updating the views for R(x!,c!,cr!,n!)

R(x!E@x,c!free,cr!free,n!free)
```

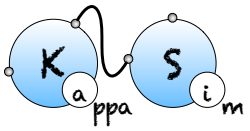
Figure 7.8: Reachability analysis: extensional description of the new patterns created when applying a rule.

yet because the precondition is not satisfied at the current state of the iteration (eg. see Fig. 7.5). Otherwise, the analysis can apply the rule and update the state of the iteration accordingly (eg. see Fig. 7.6).

- With the option `--verbosity-level-for-reachability-analysis High`, the analysis also describes which patterns are discovered.

  In particular, at the beginning of the iteration, the analysis prompts the patterns of interest that occur in initial state (eg. see Fig. 7.7). Then, each time a rule is applied successfully, the analysis shows which new patterns have been discovered (eg. see Fig. 7.8).

- When new patterns are discovered, then, it is necessary to apply again any rule that may operate over these patterns.



```
Applying rule 0: E.R:
the precondition is satisfied

rule 0: E.R is applied for the first time
(rule 1: E/R) should be investigated

(rule 1: E/R) should be investigated


Updating the views for E(x!)

E(x!R@x)


Updating the views for R(x!,c!,cr!,n!)

R(x!E@x,c!free,cr!free,n!free)

Update information about potential sites accross domain

R(c!1,x!E.x),R(c!1,x!E.x)

(rule 0: E.R) should be investigated
```

Figure 7.9: Reachability analysis: discovering new patterns force the analysis to apply some rules again, until reaching a fix-point.

With the option `--verbosity-level-for-reachability-analysis Full`, the analysis also describes which rules are awaken by the discovery of a new pattern (see Fig. 7.9).

The option `--output-mode-for-reachability-analysis` can be used to tune the output of the analysis. The default mode is `kappa`. In mode, `raw`, patterns of interest are displayed extensionally. In mode, `english`, properties of interest are explained in English. The option `--use-natural-language` can be used to switch on/off the translation of properties in natural language: when the option is disabled, each relationship is described in extension.

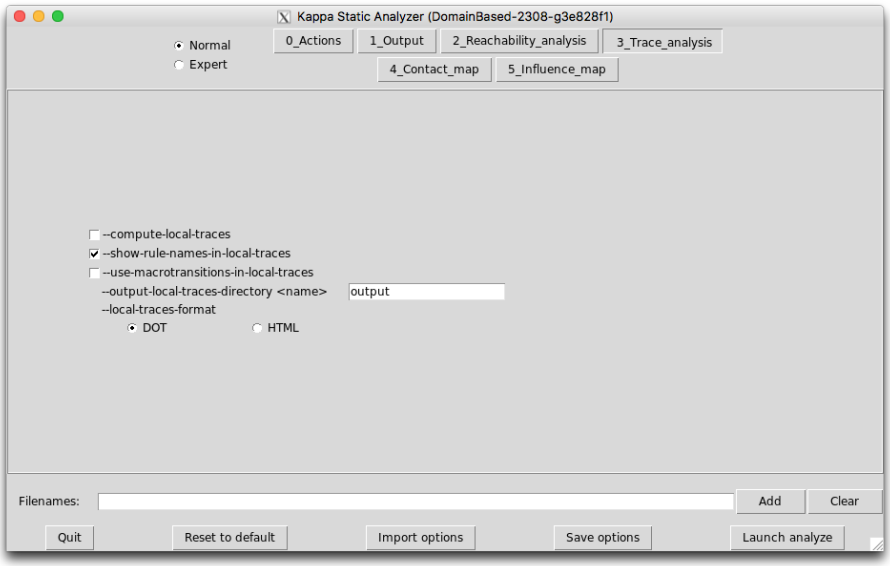


Figure 7.10: KaSa graphical interface - sub-tab `3_Trace_analysis`

7.4 Local traces

Trace analysis is a refinement of reachability analysis that additionaly explains how one agent can go from a given view to another one, following a path that we call a local trace. Thus the set of the local traces for a given agent can be described as a transition system among the views for a given agent: in this transition system, the nodes are local views; introduction arrows correspond to either initial states, or creation rules; transitions denote a potential conformation change of an agent, from one local views to another one, due to the application of a given rule.

We consider the following example:

```
1   %init: 1 P()
2   %init: 1 K()
3
4   'a1+' P(a1~u) -> P(a1~p) @1
5   'b1+' P(a1~p,b1~u) -> P(a1~p,b1~p) @1
6   'a1-' P(a1~p,b1~u) -> P(a1~u,b1~u) @1
7   'b1-' P(b1~p,g) -> P(b1~u,g) @1
8   'a2+' P(tab:siga2~u) -> P(a2~p) @1
9   'a2-' P(a2~p,g) -> P(a2~u,g) @1
10  'b2+' P(a2~p,b2~u) -> P(a2~p,b2~p) @1
```

```
11   'b2-' P(b2~p,g) -> P(b2~u,g) @1
12   'P.K' P(a1~p,a2~p,b1~p,b2~p,g),K(x) -> P(a1~p,a2~p,b1~p,b2~
        p,g!1),K(x!1) @1
13   'P/K' P(a1~p,a2~p,b1~p,b2~p,g!1),K(x!1) -> P(a1~p,a2~p,b1~
        p,b2~p,g),K(x) @1
```

Typing the following instruction:

`KaSa protein2x2.ka --reset-all --compute-local-traces`

will perform the trace analysis on the model `protein2x2ka`, and produce two dot format files `Agent_trace_K_x^.dot` and `Agent_trace.P.a1_.a2_.b1_.b2_.g^.dot`. The output repository can be changed thanks to the command line options `-output-directory` and `-output-local-trace-directory`. Moreover, file names is made of the prefix `Agent_trace`, followed by the kind of protein and the list of the sites of interest (the symbol '`^`' denotes a binding state, and the symbol '`_`' an internal state).

The transition system that describes the local traces for the agents of kind P is descrided in Figure 7.11. We notice that the nodes of this transition system are labelled with the states of the sites of P. The internal state of a site x is denoted as $x{\sim}u$ (meaning that the site x has state u, whereas the binding state of a site x is denoted as $x!\mathit{free}$, when the site is free, and as $x!K@x$ when the site x is bound to the site x of a given agent of kind K.

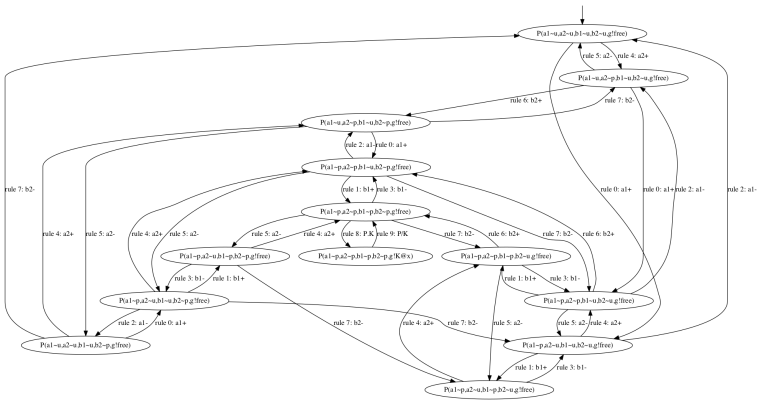


Figure 7.11: Local traces for the `protein2x2.ka` model defined in Section 7.4

We notice that the transition system that is given in Fig. 7.11 contains too many nodes. We can coarse-grain this transition system thanks to the following option:

`-use-macrotransitions-in-local-traces`.

Typing the following instruction:

```
KaSa protein2x2.ka --reset-all --compute-local-traces
                   --use-macrotransitions-in-local-traces
```

will perform the trace analysis on the model `protein2x2ka`, and produce two dot format files `Agent_trace_K_x^.dot` and `Agent_trace.P.a1_.a2_.b1_.b2_.g^.dot`. The name of the output repository can be changed thanks to the command line options `-output-directory` and `-output-local-trace-directory`. This time, the files describe a coarse-graining of the corresponding transition systems.

For instance, the coarse-grained transition system for the local traces of the proteins of kind P, is given in Figure 7.12. This coarse-grained transition system is a compact implicit encoding of the transition system in Figure 7.11. It is obtained by exploiting the fact that locally, the behavior of the pair of states a_1 and b_1 is independent from the behavior of the pair of states a_2 and b_2, until these four sites are phosphorylated, so that the site g can get bound.

More formally, in that transition system, some states are microstates (in a microstate, the state of each site is documented); some others are macrostates: (in a macrostate, the states of only a subset of site is documented). Thus a macrostate $v^\sharp$ can be seen intensionally as a part of a local view, but also extensionnaly as the set $\gamma(v^\sharp)$ of the local views they are a subpart of. A microstate v can be described by any sequence $(v_i^\sharp)$ of macrostates prodiding that the intersection $\bigcap \gamma(v_i^\sharp)$ of the extensional denotation $\gamma(v_i^\sharp)$ of these macrostates $v_i^\sharp$, is equal to the singleton $\{v\}$; moreover a transition between two microstates v and v' can be described by any transition between one macro state $v^\sharp$ and another one $v'^\sharp$, provided that there exists a sequence of macrostate $(v_i^\sharp)$ such that the sequence $(v^\sharp, (v_i^\sharp))$ denotes the microstate v and the sequence $(v'^\sharp, (v_i^\sharp))$ denotes the microstate v'.

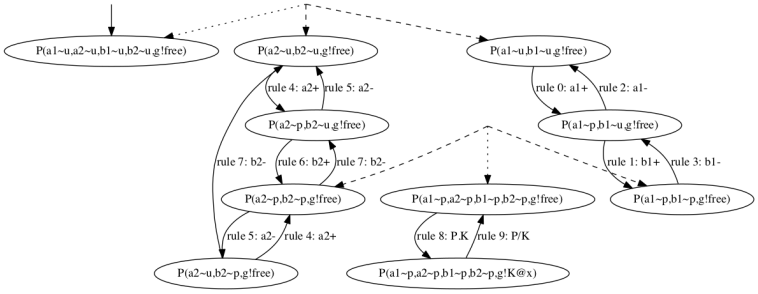


Figure 7.12: Local traces for the `protein2x2.ka` model defined in Section 7.4

Such coarse-grained transition system can be geometrically interpreted as a simplicial complex [9].

As a microstate could be decomposed into several sequences of macrostates (including the trivial sequence containing only the microstate itself), the system may jump sponta-

neously (by using a ε transition) from one representation to another representation. This corresponds to the intersection between several simplexes in the corresponding simplificial complex.

Although the semantics of a coarse-grainged transition system is fully defined by its labelled transitions, it is useful to annotate the graph by some information about the relation between the denotation of each macrostate. By default, we use hypertlinks to relate each macrostate v (including each microstate) to the set of its immediate subparts v'. In such a hyperlink, v is connected via a dotted arrow, whereas each immediate subpart is connected via a dashed arrow.

More options are available in expert mode, but they are not documented yet.

7.5 Contact map

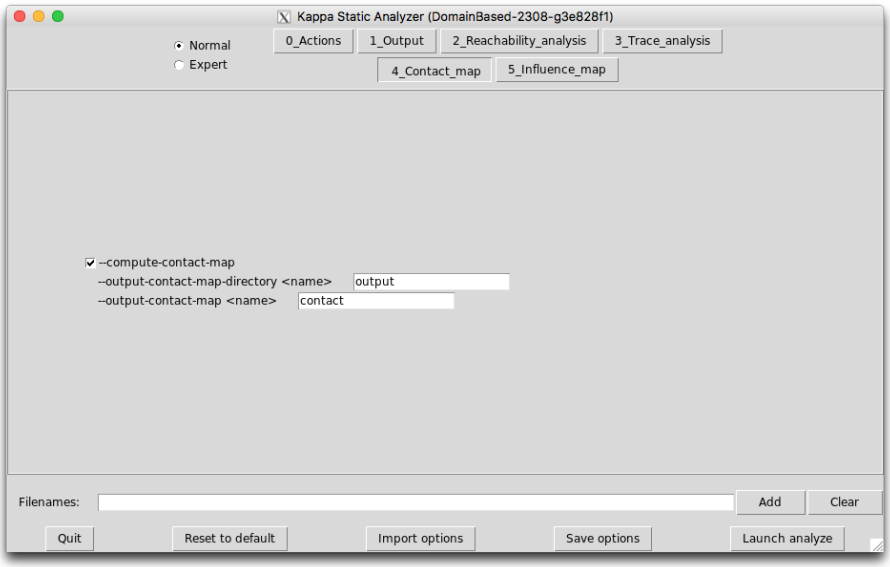


Figure 7.13: KaSa graphical interface - sub-tab `4_Contact_map`

The contact map of a model is an object that may help modelers checking the consistency of the rule set they use. The contact map is *statically* computed and does not depend on kinetic rates nor initial conditions.

Typing the following instruction:

```
KaSa abc.ka -reset-all -compute-contact-map
```

will produce a dot format file named `contact.dot`. The name of the output file and the directory can be changed to the command line options `-output-contact-map` and `-output-directory`. The directory is assumed to exist. The file will be overwritten if it exists. All the options related to the computation of the contact map can be accessed on the sub-tab `4_Contact_map` of the graphical interface (see Fig. 7.15).

The contact map summarises the different types of agent, their interface and the potential binding between sites. It is an over approximation, thus if the contact map indicates a potential bond, it does not mean that it is always possible to reach a state in which two sites of these kinds are bound, but if the contact map indicates no bond between two sites, it means that it is NOT possible to reach a state in which two sites of these kinds are bound together.

The contact map for the `abc.ka` model defined in Chapter 5 is given in Figure 7.14. On this map, we notice that there are three kinds of agent, namely A, B, and C. Agents of kind A have two sites x and c, that bear no internal state (they appear in yellow only), agents of kind B have one site x, that bears no internal state (they appear in yellow only), and agents of kind C have two sites x_1 and x_2 with both a binding state and an internal state (they appear both in yellow and in green). We notice that when a site can bear both an internal state and a binding state, they are considered as two different sites in the contact map. Additionally, the contact map indicates that sites x of the agents of kind A can be bound to the site x of an agent of kind B and that sites c of the agents of kind A can be bound to the agents of kind C either on the site x_1, or on the site x_2.

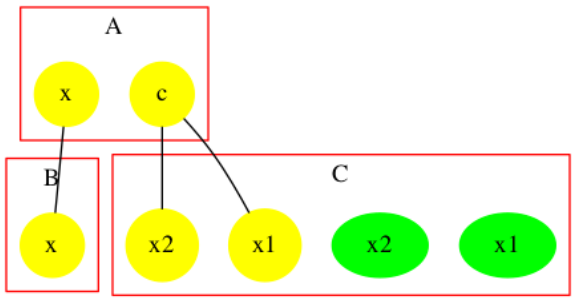


Figure 7.14: Contact Map for the `abc.ka` model defined in Chapter 5

7.6 Influence map

The influence map of a model is an object that may help modelers checking the consistency of the rule set they use.

Typing the following instruction:

```
KaSa abc.ka -reset-all -compute-influence-map
```

will produce a dot format file named `influence.dot`. The name of the output file and the directory can be changed to the command line options `-output-influence-map` and `-output-directory`. The directory is assumed to exist. The file will be overwritten if it exists. All the options related to the computation of the influence map can be accessed on the sub-tab `4_Influence_map` of the graphical interface (see Fig. 7.15).

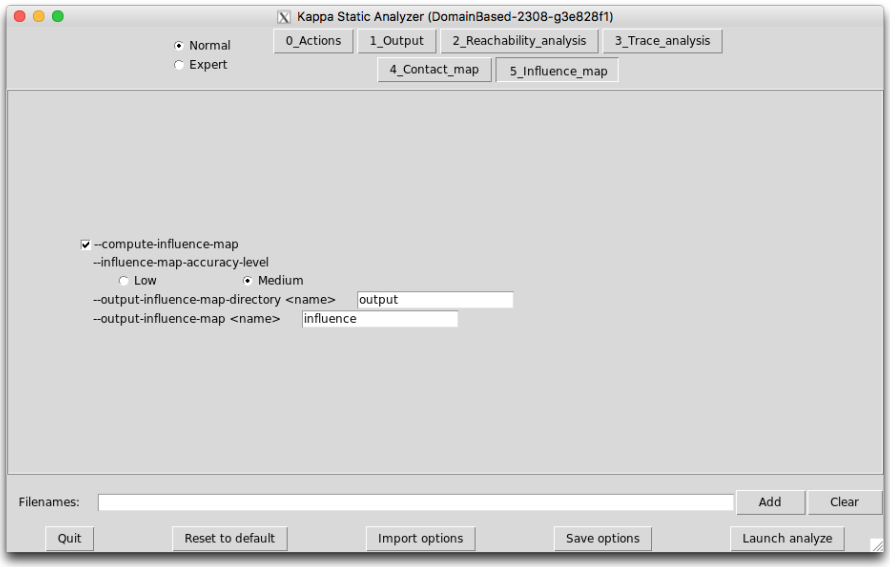
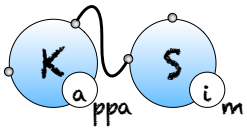


Figure 7.15: KaSa graphical interface - sub-tab `5_Influence_map`

Unlike the flux map, the influence map is *statically* computed and does not depend on kinetic rates nor initial conditions. It describes how rules may potentially influence each other during a simulation. KaSa will produce a dot format file containing the influence relation over all rules and observables of the model. The produced graph visualised using a circular rendering[1] is given in Figure 7.16. Observables are represented as circular nodes and rules as rectangular nodes. The labels of the nodes are either the label of the rule or of the observable (if available), otherwise it is made of a unique identifier allocated by KaSa followed by the kappa definition of the rule/observable. Edges are decorated with the list of embeddings (separated by a semi-colon) allowing the identification of agents in both rules's right hand sides/left hand sides. More precisely, for positive influences, the notation $[i \to j]$ denotes a pair of embeddings from the agent number i of the origin's right hand side and from the agent number j of the target's left hand side and the notation $[i\star \to j]$ denotes

[1] One may use for instance the `circo` program that is part of the *graphviz* suite.



a pair of embeddings from an agent attached to the agent number i of the origin's left hand side, which have been freed by side effect and from the agent number j of the target's left hand side; for negative influences, the notation $[i \to j]$ denotes a pair of embeddings from the agent number i of the origin's left hand side and from the agent number j of the target's left hand side and the notation $[i\star \to j]$ denotes a pair of embeddings from an agent attached to the agent number i of the origin's left hand side, which have been freed by side effect and from the agent number j of the target's left hand side; Observables have no influence, but they can be influenced by rules, if the rule can increase or decrease their value.

More formally, consider the rules $r : L \to R$ and $s : L' \to R'$. One wishes to know whether it is possible that the application of rule r over a graph G creates a new instance of rule s (which is called a positive influence and that is described by green arrows in the influence map), or destroy a previous instance of rule s (which is called negative influence and that is described by red arrows in the influence map). In Fig. 7.17, we illustrate the construction of positive influences due to overlap of the left hand side of a rule and the right hand side of another rule on some sites that are modified by the former one.

The current implementation has the following limitations:

- Currently, only observables that are defined as patterns are taken into account.
- Not atomic observables which are defined as algebraic expressions are not taken into account yet. The observables are ignored.
- The influence map does not take into account indirect influences due to perturbations (which could arises when the application of a rule triggers a perturbation which would create some agents or increase/decrease the value of some variables).
- Token are not taken into yet. They are currently ignored.
- Positive/negative influence of time is not taken into account either.

Lastly, KaSa computes an over-approximation of the influence map. They may show an influence despite the fact that there can be no actual one. But if it shows no influence it means that either there are NO such influence, or that we are in a case that is not covered yet as itemised previously.

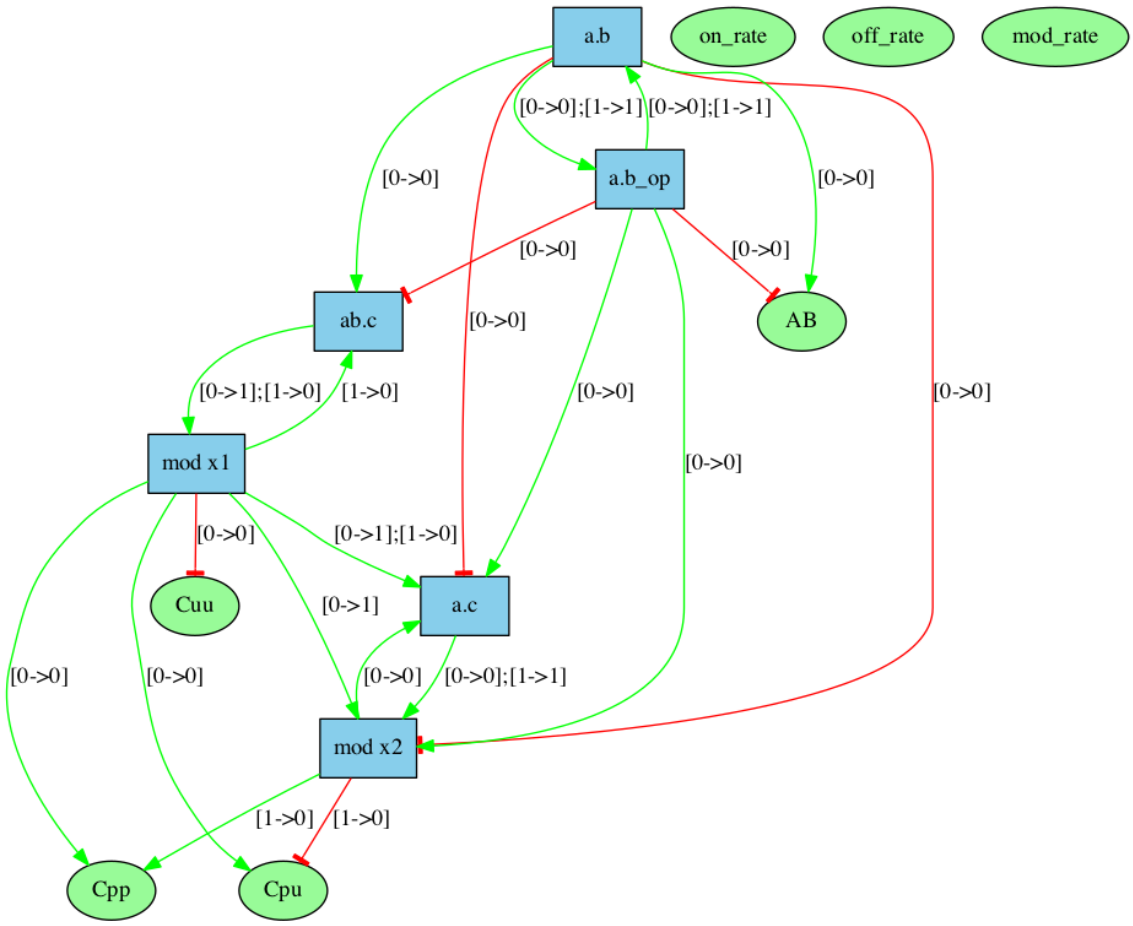


Figure 7.16: The influence map of the `abc.ka` model defined in Chapter 5. Edge labels denote embeddings with the convention that the notation $[i \to j]$, in a positive influence, denotes a pair of embeddings from the agent number i of the origin's right hand side and from the agent number j of the target's left hand side; the notation $[i \to j]$, in negative influence, denotes a pair of embeddings from the agent number i of the origin's left hand side and from the agent number j of the target's left hand side; the notation $[i\star \to j]$, whatever the influence is positive of negative, denotes a pair of embeddings from an agent attached to the agent number i of the origin's left hand side, which have been freed by side effect and from the agent number j of the target's left hand side.

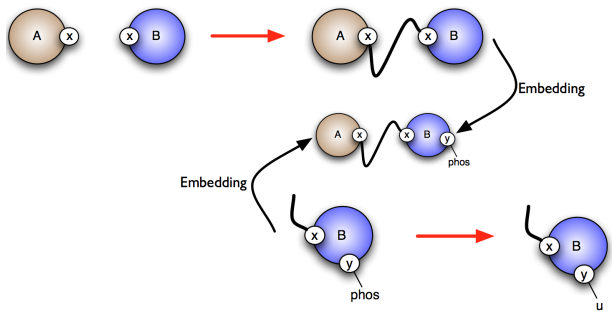


Figure 7.17: Computation of the influence of the top rule on the rule below: the right hand side of the first rules embeds in a common term with the left hand side of the second rule. It results that the first rule has a positive influence on the second.

Chapter 8

Frequently asked questions

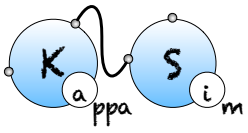
Simulation hangs after a while

If the progress bar seems stalled, it does not necessarily mean that the simulation is blocked. In particular when a simulation is triggered with a *time* limit (`-l` option of the command line) it might only indicate that the bio clock is stalled while computation events still occur. Recall that the average (bio) time one has to wait in order to apply a rule is $1/A$, where A is the sum of all the rule activities (which is equal to the number of instances that a rule has, times its kinetic rate). Whenever the number of occurrences of a rule grows too fast (if new agents are created during the simulation for instance), or if the kinetic rate of a rule is defined by a function that grows rapidly, the average time increment might tend to 0 and if it remains so for a while, it will block the progress bar whose advance is proportional to the bio time `[T]`.

In order to make sure that KaSim is not incorrectly blocked you may wish to plot the event clock against time clock using the observable `%obs: 'events' [E]` or run the simulation using an event limit (`-e` option of the command line) instead of a time limit.

What do null events mean, why do I have any?

Null events is a way for KaSim to compensate for some over approximation it is doing, in order deal with large simulations more efficiently. They usually do not impact significantly the performances of the simulator, unless the model contains rules using the special notation to deal with ambiguous molecularity (see Section 3.4.4). With pure Kappa rules, the ratio r of null event over productive ones (that you can track using the observable `%obs: 'r' [E-]/[E]`) should tend to 0 when models have a lot of agents.



No data points are generated

Make sure you have `%obs` or `%plot` instructions in your KF. Also make sure to use a reasonable value for the `-p` option in the command line to tell KaSim how often you wish to have points on your curves.

Too many instances of an observable

The value of a kappa expression E is equal to the number of embeddings it has in the current mixture M. Embeddings are maps from agents in E to agents in M. If E has symmetries then every permutation of E will be counted as a new embedding. For instance let $E =$`A(x!1),A(x!1)` and let $M =$`A(x!1,y~p),A(x!1,y~u)`. KaSim will count two instances of E in M: the one mapping the first `A` of E to the first `A` of M and the one mapping the first `A` of E to the second `A` of M.

The computed influence map is incorrect, it misses some activation or has too much of them

The influence map computed by KaSim contains relations that are computed on side effect free rules only. It is likely that a missing activation is due to a side effect that is not taken into account. If the influence map shows an activation between rule r and s that is never possible with a given model, just remember that activation computation implies that *there exists* a context in which applying rule r will create a new instance of rule s. This context might simply never be realized with the given rules or initial conditions.

Value `nan` in the data file at the end of the simulation

The value `nan` means "Not a Number". It is generated when a plotted variable is infinite. Make sure this variable is not divided by zero at some point.

Bibliography

[1] Ferdinanda Camporesi and Jérôme Feret. Formal reduction of rule-based models. In *Postproceedings of the Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics, MFPS XXVII*, volume 276C of *Electonic Notes in Theoretical Computer Science*, pages 31–61, Pittsburg, USA, 25–28 May 2011. Elsevier Science Publishers.

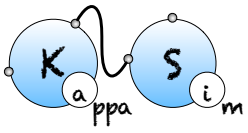
[2] Vincent Danos, Jérôme Feret, Walter Fontana, Russell Harmer, Jonathan Hayman, Jean Krivine, Chris Thompson-Walsh, and Glynn Winskel. Graphs, rewriting and pathway reconstruction for rule-based models. In Schloss Dagstuhl Leibniz-Zentrum fuer Informatik, editor, *FSTTCS 2012 - IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 18 of *LIPIcs*, pages 276–288, 2012.

[3] Vincent Danos, Jérôme Feret, Walter Fontana, Russell Harmer, and Jean Krivine. Abstracting the differential semantics of rule-based models: exact and automated model reduction. In Jean-Pierre Jouannaud, editor, *Proceedings of the Twenty-Fifth Annual IEEE Symposium on Logic in Computer Science, LICS '2010*, volume 0, pages 362–381, Edinburgh, UK, 11–14 July 2010. IEEE Computer Society.

[4] Vincent Danos, Jérôme Feret, Walter Fontana, and Jean Krivine. Scalable simulation of cellular signaling networks. In *Proc. APLAS'07*, volume 4807 of *LNCS*, pages 139–157, 2007.

[5] Vincent Danos, Jérôme Feret, Walter Fontana, and Jean Krivine. Abstract interpretation of cellular signalling networks. In Francesco Logozzo, Doron A. Peled, and Lenore D. Zuck, editors, *Proceedings of the Ninth International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI '2008*, volume 4905 of *Lecture Notes in Computer Science*, pages 83–97, San Francisco, USA, 7–9 January 2008. Springer, Berlin, Germany.

[6] Vincent Danos, Jérôme Feret, Walter Fontanta, Russ Harmer, and Jean Krivine. Rule based modeling of biological signaling. In Luís Caires and Vasco Thudichum Vas-



concelos, editors, *Proceedings of CONCUR 2007*, volume 4703 of *LNCS*, pages 17–41. Springer, 2007.

[7] Vincent Danos and Cosimo Laneve. Formal molecular biology. *Theoretical Computer Science*, 325, 2004.

[8] James R. Faeder, Mickael L. Blinov, and William S. Hlavacek. Rule based modeling of biochemical networks. *Complexity*, pages 22–41, 2005.

[9] Lisbeth Fajstrup, Eric Goubault, and Martin Raußen. Detecting deadlocks in concurrent systems. In *Proc. CONCUR '98*, volume 1466 of *LNCS*, 1998.

[10] Jérôme Feret. Reachability analysis of biological signalling pathways by abstract interpretation. In T.E. Simos, editor, *Proceedings of the International Conference of Computational Methods in Sciences and Engineering, ICCMSE'2007, Corfu, Greece*, number 963.(2) in American Institute of Physics Conference Proceedings, pages 619–622, Corfu, Greece, 25–30 September 2007. American Institute of Physics.

[11] Jérôme Feret, Vincent Danos, Jean Krivine, Russ Harmer, and Walter Fontana. Internal coarse-graining of molecular systems. *Proceedings of the National Academy of Sciences of the United States of America*, April 3 2009.

[12] Jérôme Feret and Kim Quyên Lý. Reachability analysis via orthogonal sets of patterns. In *Seventeenth International Workshop on Static Analysis and Systems Biology (SASB'16)*, ENTCS. elsevier. to appear.

[13] Daniel T. Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of Computational Physics*, 22(4):403–434, 1976.

[14] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[15] Peter Kreyßig. *Chemical Organisation Theory Beyond Classical Models: Discrete Dynamics and Rule-based Models*. PhD thesis, 2015.

Index