BIOCHAM 2.9 Reference Manual

François Fages, Aurélien Rizk, Sylvain Soliman
INRIA Paris-Rocquencourt, France
`http://contraintes.inria.fr/BIOCHAM/`

June 2009

Contents

Chapter 1

Syntax of Reaction Rules and Temporal Properties

1.1 Biochemical objects

BIOCHAM manipulates formal objects which represent chemical or biochemical compounds, ranging from small molecules to macromolecules and genes. BIOCHAM objects can be used also to represent control variables and abstract processes. They are written with the following syntax:

$$object =\quad molecule \mid abstract$$

$$molecule =$$

$$\begin{array}{ll} name & \\ \mid\ molecule\text{--}molecule & \text{molecular complex} \\ \mid\ molecule\text{\textasciitilde}\{name,...,name\} & \text{modified molecule} \\ \mid\ molecule\text{::}name & \text{located molecule} \\ \mid\ gene & \\ \mid\ (\ molecule\ ) & \end{array}$$

$$gene =\quad \#name$$

$$abstract =\quad "@name \qquad\qquad process$$

A name is a word of alphanumerical and '_' characters beginning with a letter. The syntax of a molecule has five forms.

1. The first form is the simplest and the most flexible one. It concerns the case in which a molecule is simply given a (case sensitive) name.

2. The second form serves to denote multimolecular complexes with the linking operator -. This binary operator is assumed to be associative and commutative, hence the

5

order of the elements in a complex does not matter. In the cases where one would like to distinguish between different orders of association, one should give names to the different complexes.

3. The third form serves to write modified forms of molecules, by attaching the set of modified sites of the protein with the operator , like the set of phosphorylated sites for example. Several sets can be attached, the union is considered. The order of the elements is irrelevant.

   Example 1 `cdk1, cdk1-cycB` *and* `cdk1~{tyr15,thr161}-cycB` *are valid notations for, respectively, the cyclin dependent kinase 1, the complex* `cdk1 cyclin B`, *and the phosphorylated form at phosphorylation sites tyrosine 15 and threonine 161 of* `cdk1` *in the complex* `cdk1-cycB`.

   `(cdk1-cycB)~{tyr15,thr161}` *is another notation for the same phosphorylated form of the complex without specifying the constituent which is phosphorylated. Note that in this syntax, the complex* `(cdk1-cycB)~{tyr15,thr 161}` *is considered as formally different from* `cdk1~{tyr15,thr161 }-cycB`.

4. One can provide a location to a molecule, using the `::` operator, otherwise the molecule is supposed to be in the default location.

5. The fifth syntactical form is used to denote genes or gene promotors, with a name beginning with `#`. These objects are assumed to be unique, which has a consequence on the way reactions involving such objects are interpreted by BIOCHAM (see below).

Example 2 `DMP1-#p19ARF` *can be used to denote the binding of protein* `DMP1` *on the promotor of the gene producing protein* `p19ARF` *noted* `#p19ARF`.

The same assumption of uniqueness is made on abstract objects that are noted with a '@'. Abstract objects can be used to represent particular phases of a process, complete subsystems or abstract processes.

Example 3 `CycD1 =[@UbiPro]=> _` *can be used to denote the Ubiquitin/Proteasome system degrading* `CycD1`.

1.2 Reaction and transport rules

BIOCHAM reaction rules are used primarily to represent biochemical reactions and transport. They can be used also to represent state transitions involving control variables or abstract processes, or to represent the main effects of complete subsystems such as protein synthesis by DNA transcription without introducing RNAs in the model. They are written with the following syntax:

reaction =	*kinetics for basic_reaction*
	| basic_reaction
	| name : basic_reaction
	| name : kinetics for basic_reaction
basic_reaction =	*solution => solution.*
	| solution =[object]=> solution.
	| solution =[solution => solution]=> solution.
	| solution <=> solution.
	| solution <=[object]=> solution.
solution =	*_ | object | integer*object | solution + solution | (solution)*
kinetics =	*simple_kinetics*
	| (simple_kinetics , simple_kinetics)
	| if condition then simple_kinetics
	| if condition then simple_kinetics else simple_kinetics
	| if condition then simple_kinetics else (kinetics)
simple_kinetics =	*[molecule]*
	| float
	| name
	| simple_kinetics * simple_kinetics
	| simple_kinetics / simple_kinetics
	| simple_kinetics + simple_kinetics
	| simple_kinetics − simple_kinetics
	| simple_kinetics ^ simple_kinetics
	| log(simple_kinetics)
	| exp(simple_kinetics)
	| MA(simple_kinetics)
	| MM(simple_kinetics, simple_kinetics)
	| H(simple_kinetics, simple_kinetics, integer)
	| HN(simple_kinetics, simple_kinetics, integer)
	| (simple_kinetics)
condition =	*simple_kinetics < simple_kinetics*
	| simple_kinetics > simple_kinetics
	| simple_kinetics = simple_kinetics
	| simple_kinetics =< simple_kinetics
	| simple_kinetics >= simple_kinetics
	| condition and condition

A solution is a multiset of objects. The character _ denotes the empty solution. The order and multiplicity of molecules in a solution (1 if not explicitly written) will be ignored for all qualitative operations (like boolean model-checking), only the presence or absence of objects will then be considered. In such a boolean abstraction of stoichiometric models,

a reaction transforms one solution matching the left-hand side of the rule, into another solution in which the objects of the right-hand side have been added. The molecules in the left-hand side of the rule which do not appear in the right-hand side may be non-deterministically present or consumed in the resulting solution. This convention reflects the capability of BIOCHAM to reason about all possible behaviors of the system with unknown kinetic parameters [1, 2]. Following the uniqueness assumption, molecules marked as "genes" with the '#' notation, or any compound built on such a molecule (such as DMP1- #p19ARF for instance) are not multiplied. These objects remain unique and are deterministically consumed in the form in which they appear in the left-hand side of the rule. The same goes for control variables, noted with a '@', which are deterministically made absent.

Reactions involving a catalyst molecule (i.e. a molecule appearing in both the left and right-hand sides of the rule) are written with the catalyst molecule between square brackets in the arrow. Similarly a catalyst reaction can be written between square brackets in the arrow of the rule (see the last example below).

The kinetic expressions will only be used in numerical operations (like ODE-based simulation). If a rule is provided without kinetic expression, a mass action law kinetic with reaction rate 1 is assumed, i.e. MA(1). Pairs of kinetic rates are given for reversible reactions. The exponential notations 1e6 or 7.2E-4 are accepted.

Compound concentrations are allowed between square brackets in kinetic expressions, e.g. [cycB]. When a name is given in a kinetic expression, a corresponding parameter or macro will be looked for. A special parameter with name Time gives the current value of time during a numerical simulation.

The abbreviations MA, MM, H and HN represent respectively Mass Action law, Michaelis-Menten, Hill and negative Hill kinetics. In the first case, the parameter given as argument will be multiplied by all reactants' concentrations to provide the kinetic law. In the second case the two arguments represent the Vm and Km of the michaelian kinetics; the law will have the form: Vm*[S]/(Km+[S]), where S is the reactant (a warning is raised if the rules contains several reactants). In the third case, H(Vm,Km,n) = Vm*[S]\^n /(Km\^n+[S]\^n), the first argument Vm represents the maximum value, Km the threshold, n the order of Hill function and [S] is the concentration of the reactant. The fourth abbreviation is the negative Hill kinetics HN(Vm,Km,n) = Vm*Km\^n /(Km\^n+[S]\^n).

A kinetic expression may contain a conditional expression, where the condition is a conjunction of linear (in)equations. This can be used to restrict the use of some laws (see the last example below) and to represent hybrid systems with different continuous dynamics in a finite number of phases.

Examples: MA(0.001) for cdk1 + cycB => cdk1-cycB. is a complexation rule with mass action law kinetics and constant rate 0.001. The rule
cdk1-cycB =[Myt1]=> cdk1~thr14-cycB.
is a phosphorylation rule with catalyst Mytosine 1. This rule is equivalent to
cdk1-cycB + Myt1 => Myt1 + cdk1~{thr14}-cycB.

The rule `(1*[RAF]*[RAFK],0.4*[RAF-RAFK])` `for RAF + RAFK <=> RAF-RAFK.` is a reversible complexation rule given with a pair of kinetic rates: the product of the concentrations for the complexation, 0.4 times the complex concentration for the decomplexation, which is equivalent to the pair `(MA(1), MA(0.4))`.
`if [X] > 0.8 then k for X =[ATP => ADP]=> Y` represents a reaction, only able to proceed when there is enough X and such that X is transformed into Y by consuming energy from ATP. `cdk1::nucleus=>cdk1::cytoplasm` is a transport rule of cdk1 from the nucleus to the cytoplasm.

1.3   Boolean temporal properties

Propositional logic can be used to describe the states of the system. For instance, the formula `cdk1 & MPF & !(cdk7-cycH)` represent the states of the system where `cdk1` and `MPF` are present and `cdk7-cycH` is absent, the rest being undetermined. The temporal boolean properties of BIOCHAM models can be formalized in Computation Tree Logic (CTL). CTL is an extension of propositional logic with two path quantifiers for non-determinism: E, A, and several operators for time: F, G, X, U. The path quantifier E expresses the existence of a path, A means for all paths, F means at some time point (on the path), G means at all time points, X means at the next time point, U is a binary operator meaning that a formula is true until a second formula becomes true. Furthermore, since it is possible to have an initial state only partially defined (the molecules may be present, absent, or their presence is unspecified), a BIOCHAM temporal boolean formula is prefixed with an initial state quantifier. There are two such quantifiers: Ei and Ai, meaning respectively, "there exists an initial state" and "for all initial states". The syntax of CTL with initial state quantifiers is defined by the grammar *ictl* below:

ictl =	*Ei(ctl)	Ai(ctl)*	
ctl=	*object*	
	`|` *(ctl)*	
	`| EF`*(ctl)*	
	`| AF`*(ctl)*	
	`| EG`*(ctl)*	
	`| AG`*(ctl)*	
	`| E`*(ctl U ctl)*	
	`| A`*(ctl U ctl)*	
	`| E`*(ctl W ctl)*	
	`| A`*(ctl W ctl)*	
	`| EX`*(ctl)*	
	`| AX`*(ctl)*	
	`| !`*(ctl)*	*negation*
	`|` *ctl* `&` *ctl*	*conjunction*
	`|` *ctl* `|` *ctl*	*disjunction*
	`|` *ctl* `xor` *ctl*	*exclusive or*
	`|` *ctl* `->` *ctl*	*implication*
	`|` *ctl* `<->` *ctl*	*equivalence*
	`| reachable`*(ctl)*	*same as* `EF`*(ctl), i.e. on some path the formula can become true*
	`| stable`*(ctl)*	*same as* `AG`*(ctl), i.e. on all paths the formula remains always true*
	`| steady`*(ctl)*	*same as* `EG`*(ctl), i.e. on some path the formula is always true*
	`| checkpoint`*(ctl,ctl)*	*same as* `!(E(!(ctl1) U ctl2)))` *i.e. there is no path where the first formula is false until the second is true*
	`| loop`*(ctl,ctl)*	*same as* `AG((ctl1->EF(ctl2))\&(ctl2->EF(ctl1)))` *approximates the oscillation property where two formulae are alternatively true.*
	`| oscil`*(ctl)*	*same as* `loop(ctl,!(ctl))` *approximates the oscillation property where a formula is alternatively true and false*

Example 4 `Ei(EF(cycB))` *expresses the existence of an initial state such that there exists a path on which at some time point* `cycB` *is present.* `Ai(AF(cycB))` *expresses that for all initial states and on all paths,* `cycB` *is finally present at some time point.*
`Ai(!(E(!(cdc25) U cdk1-cycB)))` *checks that cdc25 is a checkpoint for the activation of* MPF *(the unphosphorylated form of the cdk1 cycB complex) for all initial states, that is there does not exist a path on which* `cdc25` *is always absent until the complex cdk1-cycB is present.*

1.4   Numerical temporal properties

In BIOCHAM, temporal quantitative properties about concentrations and their derivatives can be formalized as well in Linear Time Logic with numerical constraints, noted LTL(R). In addition, quantifier free LTL(R) (QFLTL(R)) formulas can contain free real valued

variables. A word of alphanumerical characters beginning with a lowercase letter which is not already a kinetic parameter or a macro is interpreted as a free variable. The syntax of LTL(R) and QFLTL(R) formulae is given by the following grammar:

ltl =	*condition*	*(*)*
	| *(ltl)*	
	| `F`*(query)*	*finally*
	| `G`*(query)*	*globally*
	| `X`*(query)*	*next*
	| *(ltl)*`U`*(ltl)*	*until*
	| `!`*(ltl)*	*negation*
	| *ltl* `&` *ltl*	*conjunction*
	| *ltl* `|` *ltl*	*disjunction*
	| *ltl* `xor` *ltl*	*exclusive or*
	| *ltl* `->` *ltl*	*implication*
	| *ltl* `<->` *ltl*	*equivalence*
	| `oscil`*(molecule, int)*	*oscillations*
	| `oscil`*(molecule, int, float)*	
	| `period`*(molecule, float)*	*periodic oscillations*
	| `phase_shift`*(molecule, molecule, float)*	*phase delay*
	| `cross`*(molecule, molecule, int)*	*repetitive crossing*
	| `curve_fit`*(list_of_molecules,list_of_floats,list_of_names)*	*curve fitting*
qfltl =	*ltl*	*(**)*

(*): In the conditions here, the derivatives of some concentrations can also appear with the syntax: `d([Molecule])/dt`.

(**): the grammar of QFLTL(R) formulae differs from LTL(R) by allowing free variables in the conditions which are restricted in this case to linear inequalities.

Example 5 `G(([RAF-RAFK] >= [RAF~{p1}]) U (d([RAF])/dt < 0.3))` *expresses that all along the simulation trace, the concentration of the* `RAF-RAFK` *complex is greater than that of phosphorylated* `RAF`, *until the derivative of the concentration of* `RAF` *becomes lower than 0.3.*

The formula `oscil(M,n)` is an abbreviation for *n* successive alternations of the sign of `d([X])/dt`.

`oscil(M,n,V)` states that M oscillates, with a maximum value greater than V, at least n times.

Example 6 `oscil(cycB,3)` *expresses the fact that, along the trace, the concentration of cycB goes up and down twice.*

`period(M,p)` states that molecule M oscillates at least 3 times (with the above meaning) and has period p (with 4% error) for the last three oscillations. `phase_shift(M,N,s)` states that there is delay of s between the (last three) peaks of M and those of N.

`cross(M1,M2,n)` is an abbreviation for *n* successive repetitions of crossings between the concentration values of `[M1]` and `[M2]`.

`curve_fit(list_of_molecules,list_of_floats,list_of_names)` is an abbreviation for a curve fitting formula, stating that each molecule, at a given time, is equal to a particular value or variable. The three lists must be of the same length,

Example 7 `curve_fit([A,A],[10,50],[v1,v2])`.

stands for the QFLTL(R) formula `G((Time=10 -> [A]=v1) & (Time=50 -> [A]=v2))` *giving the values of A at time* 10 *and* 50 *in variables v1 and v2.*

1.5 Object and rule patterns

Patterns are used to define sets of objects and rules in a concise manner. Patterns can be used to specify the initial state, or the reaction rules or sets of objects, rules or temporal formulae passed as arguments in various BIOCHAM commands.

Patterns introduce the special character ? and variables noted with a name beginning with '$', to denote unspecified parts of a molecule. The parts of the molecules matched by ? or a variable can be empty.

$$
\begin{array}{ll}
object\_pattern = & molecule\_pattern \mid abstract \\[6pt]
variable = & ? \mid \$name \\[6pt]
simple\_pattern = & name \mid variable \\[6pt]
molecule\_pattern = & simple\_pattern \\
& \mid molecule\_pattern\text{--}molecule\_pattern \\
& \mid molecule\_pattern\text{\textasciitilde}\{simple\_pattern,...,simple\_pattern\} \\
& \mid molecule\_pattern\text{\textasciitilde}variable \\
& \mid gene \\
& \mid (\ molecule\_pattern\ )
\end{array}
$$

Example 8 `cdk1~?` *is a pattern representing* `cdk1` *itself and any phosphorylated form of it.* `cdk1~{tyr15,?}` *or equivalently* `cdk1~{tyr15}~?` *represents any form of* `cdk1` *phosphorylated on tyrosine 15 at least.* `cdk1-?` *represents* `cdk1` *itself and any complex containing* `cdk1`. `cdk1~?-?` *is a pattern representing all forms of* `cdk1`, *phosphorylated and/or complexed.*

When patterns are used to define the initial state or some reaction rules (e.g. with the command `add_rules`, see below), the variables that appear in the pattern have to be given a range of possible values, by using the `where` construct. When patterns are used to match rules, like in the `list_rules` or `delete_rules` command, the `where` construct cannot be used. The character ? can match any solution (even empty).

reaction_pattern =	*kinetics_pattern* `for` *basic_reaction_pattern* `where` *constraints*
	| *basic_reaction_pattern* `where` *constraints*
	| *reaction_shortcut*
basic_reaction_pattern =	*name* : *basic_reaction_pattern*
	| *solution_pattern* `=>` *solution_pattern*.
	| *solution_pattern* `=[`*object_pattern*`]=>` *solution_pattern*.
	| *solution_pattern* `=[`*solution_pattern* `=>` *solution_pattern*`]=>` *solution_pattern*.
	| *solution_pattern* `<=>` *solution_pattern*.
	| *solution_pattern* `<=[`*object_pattern*`]=>` *solution_pattern*.
solution_pattern =	*variable* | *object_pattern* | *integer* `*` *object_pattern*
	| *solution_pattern* `+` *solution_pattern* | *(solution_pattern)*
constraints =	*constraints* `and` *constraints*
	| *variable* `in` {*object_pattern*,...,*object_pattern*}
	| *variable* `not in` {*object_pattern*,..., *object_pattern*}
	| *variable* `in all`
	| *variable* `in all_simple`
	| *variable* `in parts_of`{ *name, name,...* }
	| *name* `not in` *variable*
	| *variable* `diff` *object_pattern*
	| *variable* `phos_form` *object_pattern*
	| *variable* `not phos_form` *object_pattern*
	| *variable* `more_phos_than` *object_pattern*
	| *variable* `not more_phos_than` *object_pattern*
	| *variable* `submol` *object_pattern*
	| *variable* `not submol` *object_pattern*
	| *variable* `has_simple_mol_in_common` *object_pattern*
	| *variable* `has_no_simple_mol_in_common` *object_pattern*
kinetics_pattern =	*same as 'kinetics' but with 'molecule_pattern' instead of 'molecule'*
reaction_shortcut =	`complexation`
	| `decomplexation`
	| `re_complexation`
	| `phosphorylation`
	| `dephosphorylation`
	| `re_phosphorylation`
	| `synthesis`
	| `degradation`
	| `elementary_interaction_rules`
	| `more_elementary_interaction_rules`
	| `more_elementary_interaction_rules(object)`

The `in` constraint is set membership. The set `all` (resp. `all_simple`) refers to all the

molecules already known by the system, i.e. (resp. limited to non-localized, non-complexed and non-phosphorylated forms). This means that using these constructs makes the semantics of the rule dependent on the context (order of rules, model parts loaded before, etc.), and is thus recommended only to advanced users.

The `diff` and `not in` have the opposite meaning (the variable cannot take the given value(s)). For instance `$A not in {$B,per}` means that `$A` represents a molecule which cannot be per, not have the same value as `$B`. The second case of `not in` corresponds to the absence of some phosphorylation site (the name) in a set of phosphorylated sites (the variable). See the example below.

The `phos_form` constraint, forces the variable and the `object_pattern` to differ only by some phosphorylations. `more_phos_than` forces the variable to be more phosphorylated than the `object_pattern`.

The `submol` constraint forces the variable to be a sub-molecule of the pattern, the `has_simple_mol_in_common` constraint imposing a common sub-molecule.

When using constraints relating a variable to an `object_pattern`, variables appearing in that `object_pattern` have to be constrained beforehand. Moreover, to define a variable range (for instance to add a rule), one has to use for each variable at least one positive constraint: in or `sub_mol`, and if the adequate declaration exists, `phos_form` or `more_phos_than`.

Example 9 *The reaction pattern* (`cdk1~?-? + ? => ?`) *will match all rules reacting with any form (phosphorylated or complexed) of* `cdk1`. *The pattern* (`? =[Myt1]=> ?`) *matches all rules involving the catalyst* `Myt1`. *This pattern will match all the rules having* `Myt1` *in their left and right-hand sides, even if they were not written with the catalyst notation. This pattern cannot be used to define reaction rules since it can match unconstrained molecules.*

Example 10 *The reaction pattern* `cdk46~$P + $cycD => cdk46~$P-$cycD`
where `$cycD in {DMP1~?-cycD~?, cycD~?}`. *will match all complexation rules of all phosphorylated forms of* `cdk46` *with all phosphorylated forms of* `cycD` *or* `DMP1-cycD`. *This pattern can be used to define reaction rules. All possible phosphorylated forms of molecules cdk46, DMP1 and cycD have to be declared however (see section below) in order to constrain the variable* `$P` *and the different occurrences of* `?`. *Note that if the three molecules in this pattern had three phosphorylation sites each, they would have* $2^3 = 8$ *forms each, thus the pattern would specify* $8 \times (8 \times 8 + 8) = 576$ *reaction rules! Reaction patterns must thus be used with care for specifying reaction rules and only the relevant phosphorylation sites should be declared for a molecule.*

Example 11 *The reaction pattern* `cdk1~$P-$cyc =[Wee1]=> cdk1~$P~p2-$cyc`
where `p2 not in $P and $cycA in {cycA, cks1-cycA}`
and `$cyc in { $cycA, cycB, cycB-cks1}`. *specifies the phosphorylation on site* `p2` *of several* `cdk1` *complexes not already phosphorylated on* `p2`. *This pattern can be used to*

define reaction rules. If `cdk1` *is declared with 3 phosphorylation sites* `{p1,p2,p3}`*, there are 4 forms not containing* `p2` *to combine with the 4 possibilities for the variable* `$cyc`*. This reaction pattern thus expands into 16 reaction rules.*

The reaction short-cuts stand for the following reaction patterns:
complexation :
```
$A + $B => $A-$B where $A in all and $B in all and $A diff $B
```
decomplexation :
```
$A-$B => $A+$B where $A in all and $B in all and $A diff $B
```
re_complexation :
```
$A + $B <=> $A-$B where $A in all and $B in all and $A diff $B
```
phosphorylation :
```
$A =[$C]=> $B where $A in all and $B in all and $C in all
                                  and $B more_phos_than $A and $A diff $B
```
dephosphorylation :
```
$A =[$C]=> $B where $A in all and $B in all and $C in all
                                  and $A more_phos_than $B and $B diff $A
```
re_phosphorylation :
```
$A <=[$C]=> $B where $A in all and $B in all and $C in all
                                  and $B more_phos_than $A and $A diff $B
```
synthesis :
```
_=[$G]=>$A where $A in all_simple and $G in all and $A diff $G
```
degradation :
```
$A =[$D]=>_ where $A in all and $D in all and $A diff $D
```
elementary_interaction_rules : either `complexation, decomplexation,`
   or `phosphorylation, dephosphorylation, synthesis, degradation`
more_elementary_interaction_rules : same as above plus combinations
more_elementary_interaction_rules(object) : forces the use of the given object in the rules.

1.6   Declarations

The set of all possible modified forms of a given molecule can be declared by associating to the molecule the set of sets of sites which can be modified (e.g. phosphorylated). In these declarations, the function `parts_of` can be used to denote all subsets of a set. These declarations are not mandatory, except for defining rules with patterns containing phosphorylation variables.

Example 12 *the declaration* `declare cdk2~{{},{p1},{p2},{p1,p2}}.` *specifies that cdk2 has two phosphorylation sites p1, p2 and that all combinations are possible. This declaration is equivalent to* `declare cdk2~parts_of({p1,p2}).`

The purpose of declarations is to constrain implicitly the domain of modification variables which appear in molecule patterns. Only modification sites can be declared. In a reaction pattern used for defining reaction rules, the other variables, such as complexation variables, must thus be explicitly constrained in the where part of the pattern.

Declarations can be entered also at top-level with the command declare, can be listed with the command `list_declarations` and can be cleared with the command `clear_rules`.

Note: Since the aim of declarations is to define the possible modification sites, any rule where a molecule appears in a form in contradiction with its declaration is ignored (with an error message). Since most molecules appear also in non-phosphorylated form, a warning is given when a declaration does not include the empty set.

Chapter 2

Commands at Top-level

The command `biocham` starts the BIOCHAM interpreter. Some BIOCHAM files can be passed as arguments to the command to be loaded immediately. Under the BIOCHAM prompt, the following commands are available, they must be terminated by a dot. The previous typed commands can be retrieved by pressing ctrl-p or up-arrow. The commands can be automatically completed by pressing on the tabulation key. A command can be interrupted by ctrl-c and a (abort).

Some commands take a list as argument. Lists are noted between square brackets, e.g. `[a,b,c]`. Sets are noted between braces, e.g. `{a,b,c}`. To quit the interpreter type quit.

- `quit`.
  quits the interpreter.

- `prolog('goal')`.
  The extra command executes a Prolog (the programming language in which BIOCHAM is implemented) goal passed as a string in the argument. This command is documented for the sake of completeness but should not be useful.

2.1 Loading and exporting files

The name of a BIOCHAM file must be suffixed by '.bc'. BIOCHAM files may contain:

- reaction rules and rule patterns;

- any command (including the definition of an initial state, parameter values, but also simulation or model-checking commands), which will be executed when read.

Usually, a BIOCHAM model is defined in a single file containing only declarations and reaction rules or rule patterns. Then different initial states, sets of parameters or test commands can be defined in different files.

- `load_biocham(file)`.

  the reaction rules of the BIOCHAM file are loaded and taken as current set of rules, deleting any previously loaded rule. The initial state is initialized according to the declarations in the file. The commands contained in the file are executed.

Example 13 `load_biocham('cell_cycle.bc')` *or* `load_biocham(cell_cycle)`. *Note that in the last form, the quotes are not necessary and the suffix is automatically added to the name.*

- `add_biocham(file)`.

  the rules of the given file are loaded and added to the current set of rules. The initial state is updated incrementally. The commands contained in the file are executed.

- `import_ode(file)`.

  an initial state and a set of reaction rules are inferred from the system of differential equations read in an xppaut file (see the inverse export_ode command). Both are loaded in BIOCHAM with the same differential semantics.

- `change_directory(directory)`.

  changes the current directory.

- `export_biocham(file)`.

  saves the current BIOCHAM set of rules, macros, parameters and initial state in a BIOCHAM file.

- `expand_biocham(file)`.

  saves the current BIOCHAM set of rule instances, macros, parameters and initial state in a BIOCHAM file where all reaction rule patterns are expanded.

- `export_init(file)`.

  saves the current initial state, macros and the value of parameters in a BIOCHAM file.

- `export_param(file)`.

  saves the current values of parameters (only) in a BIOCHAM file. This command might be useful when several parameter sets are associated to the same set of rules.

- `export_dot(file)`.

  exports the current BIOCHAM set of rules and initial state into a .dot file. That file can then be used to generate pictures of the interaction map, with for instance tools of the Graphviz suite.

  if you simply want to visualize the output and have both Graphviz and gv installed, see the dot command below.

Example 14 *The Unix command* `dot -Tpng file.dot > file.png` *will generate a PNG image of the map. For instance, the reaction graph of* `EXAMPLES/MAPK/mapk.bc` *will be depicted as follows*
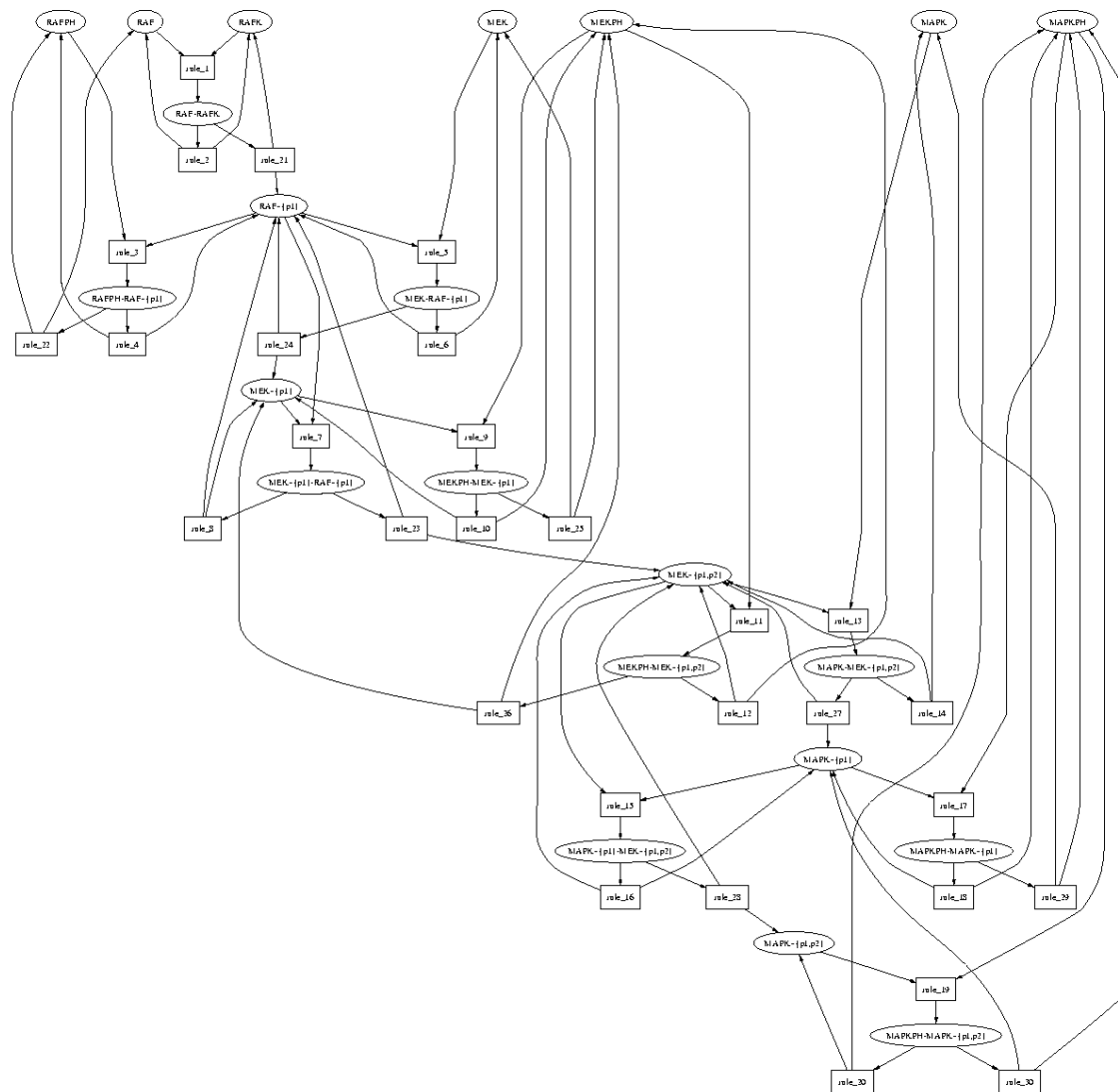
- `export_dot(file, list_of_options)`.

  exports the current BIOCHAM set of rules and initial state into a .dot file (see above). The list of options can contain: `init_up`, to force the molecules present in the initial state to be shown on the top of the image (see example above); `mod\_double`, to show enzymes catalyzing a reaction or modifiers of some kind (i.e. both input and output) with a double

arrow (input/output) instead of a simple dashed arrow; `col_path`, to color in red the latest pathway returned by a CTL query; `double\_size`, to produce A3 sized graphs instead of the default A4; state to produce the state graph instead of the reaction graph (be careful, it can be huge!).

Example 15 `export_dot(test,[init_up,mod_double,col_path]).` *to use all of the options, and save the result in the file* `test.dot;`

- `export_dot(test,[init_up]).`
  to obtain an image like that of the above example (after using dot).

- `dot.` • `draw_reactions.`

  visualizes the graph resulting from an `export_dot` with no options, using both Graphviz and gv that have to be installed and in your PATH.

- `export_nusmv(file).`
  exports the current BIOCHAM set of rules and initial state in an SMV file. Notations for molecules are translated by replacing the characters `_`, `(`, `)`, `}`, `,`, and `~{` respectively by `__`, `_L`, `_R`, `_r`, `_c` and `_l`. Furthermore the names `A, AF, AG, AX, B, E, EF, EG, EX, F, G, H, O, S, U, T, V, W, X, Y, Z` and `xor` are prefixed by `_`.

- `export_lotos(file).`
  exports the current BIOCHAM set of rules and initial state in a LOTOS file. Molecules are translated with the same convention as for SMV.

- `export_ode(file).`
  exports the current BIOCHAM set of kinetic laws and initial state to an ASCII file in ODE format (humanly readable but also usable with xppaut). Molecules are translated by removing all `-` and `~{,}` and if necessary shortening the resulting name (see the inverse `import_ode` command).

- `export_ode_latex(file).`
  exports the current BIOCHAM set of kinetic laws and initial state to an ASCII file in La-TeX format (suitable for inclusion with the LaTeX command `\include{file}`). Molecules are translated by removing all `-` and `~{,}`.

- `export_prolog(file).`
  exports the current BIOCHAM set of rules, initial state and temporal specification in a Prolog file, where reactions are represented by a transition system between boolean states, and where CTL properties can be evaluated with a model checker implemented in Prolog in the file ctl.pl.

- `export_sbml(file).`

exports the current BIOCHAM set of rules (including kinetic laws) and initial state to an SBML file. Molecules are translated by removing all '-' and '~{',','}'.

- `load_sbml(file)`.

- `add_sbml(file)`.
  act as the corresponding `load_biocham`/`add_biocham` commands but importing reactions, parameters and initial state (and only that!) from an SBML file.

2.2 Listing and defining rules and events

2.2.1 Rules

- `list_rules`.
  lists the current set of rules.

- `list_rules(reaction_pattern)`.
  lists the current set of rules matching the given pattern.

Example 16 `list_rules(? => cycA~?-? + ?)` *will list all the rules containing any form of* `cycA` *in the right hand side.*

- `expand_rules`.
  lists all the instances of the current set of rules, with the associated rule number.

- `expand_rules(reaction_pattern)`.
  lists all the instances of the current set of rules matching the given pattern.

- `add_rules(reaction_pattern)`.

- `add_rules(set_of_reaction_patterns)`.
  adds reaction rules to the current set of rules.

- `delete_rules(reaction_pattern)`.

- `delete_rules(set_of_reaction_patterns)`.
  deletes all reaction rules matching one pattern from the current set of rules. Warning: currently, if a kinetics pattern is provided, it will be ignored.

- `clear_rules`.
  clears the current set of rules and all molecule declarations.

- `rule(integer)`.

- `rule(name)`.

  shows the n-th rule (after expansion), or the rule(s) with the corresponding name (i.e. of the form name : A =¿ B).

- `pathway(list_of_integers)`.

- `pathway`.

  show the rules of corresponding numbers (after expansion). If no list is given and the trace corresponding to a query has been generated, then use the list given by that trace as argument.

- `show_kinetics`.

  returns the set of ordinary differential equations and initial concentrations (one line per molecule).

- `show_kinetics(molecule)`.

  Same as above but only for the given molecule.

2.2.2   Events

- `add_event(condition,name,kinetics)`.

- `add_event(condition,list_of_names,list_of_kinetics)`.

  sets up an event that will be fired each time the condition given as first argument becomes true. This command is effective in numerical simulations only. Upon firing, the parameter(s) given as second argument receives a new value computed from the third argument. The parameters' initial value is restored after the simulation.

Example 17 `parameter(N,1). add_event([X]>=2.0,N,1-N)`. *Each time* `[X]` *goes above the threshold value 2.0, N will become 1-N. At the end of the simulation, N is reset to 1.*

- `delete_event(condition,name,kinetics)`.

- `delete_event(condition,list_of_names,list_of_kinetics)`.

  removes an existing event.

- `delete_events`.

  deletes all the declared events.

- `list_events`.

  lists all the declared events.

2.3 Listing and defining objects and locations' volumes

2.3.1 Objects

- `declare(molecule_declaration)`.
   declares the set of all phosphorylated forms of a molecule.

- `list_declarations`.
   lists all the declarations of phosphorylated forms of molecules.

- `list_molecules`.
   lists the molecules contained in all instances of the current set of rules.

- `list_molecules(object_pattern)`.

- `list_molecules(set_of_object_patterns)`.
   lists the objects (appearing in the instances of the current set of rules) and matching one of the given object patterns.

Example 18 `list_molecules(cycE-?)`.
```
cycE
cycE-cdk2
```

- `list_all_molecules`.

- `list_all_molecules(object_pattern)`.

- `list_all_molecules(set_of_object_patterns)`.
   same as above but also including phosphorylation sites declarations and initial state.

- `check_molecules`.
   checks lower/upper case errors in molecule names. Then, tries to find production and degradation rules for all known molecules. For each potential problem, displays a warning.

2.3.2 Locations' volumes

One can define a location or compartment with a name. All molecules are localized, either explicitly with the :: operator, or implicitly in a default location.

- `volume(name,simple_kinetics)`.
   one can provide a numerical expression defining the volume of the location given as first argument. It may depend on parameters or even concentrations of compounds. If no volume is provided for a location, it will supposed equal to 1, which is the volume of the default location. The rate of a reaction will be divided by the volume of the location

of a molecule before being derived as a term in the differential equation controlling its evolution.

- `volume(name)`.
  displays the current volume (formula) of the given location.

- `list_volumes`.
  lists all the defined locations with the corresponding volume (formula).

2.4   Listing and defining parameters, macros and initial state

2.4.1   Parameters

Parameters can be defined as either constants or macros and can be used in a number of places, including kinetic expressions, initial concentrations and events.

- `parameter(name,float)`.
  sets the value of a given parameter to the corresponding value.

- `parameter(name)`.
  shows the value of the given parameter.

- `list_parameters`.
  shows the values of all known parameters.

2.4.2   Macros

A macro gives a name to an expression and can be used in any place where the expression could be used.

- `macro(name,simple_kinetics)`.

- `macro(name,sq_wave(name,float,name,float))`.
  sets the value of a given macro to the corresponding value, which will be re-evaluated each time a kinetic law is to be computed.

  The special value `sq_wave` generates a square wave signal between the values of two parameters (given by the two names) with their respective duration (given by the two floats). Please note that some adaptive step integration methods will give a quite poor result when using square waves.

- `macro(name)`.
  shows the value of the given macro.

- `list_macros`.
  shows the values of all known macros.

2.4.3   Initial state

The initial state can be partially defined by giving the list of objects which are present in the initial state and the list of objects which are absent from the initial state. The other objects can be present or absent. When no precision is given, present objects are given a default concentration of 1.

- `show_initial_state`.
    lists the objects which are present (including their initial concentration) and absent from the initial state.

- `clear_initial_state`.
    makes undefined all objects possibly present or absent in the initial state. Also deletes all parameters and macros.

- `present(object_pattern)`.

- `present(set_of_object_patterns)`.
    all objects (appearing in the instances of the current set of rules) and matching one of the given object patterns are made present in the initial state.

Example 19 `present({cycA, cdk1, cycE~?})`. *makes cycA, cdk1 and all modified forms of cycE present in the initial state.*

- `present(object_pattern, float)`.

- `present(object_pattern, name)`.
    gives to the given object the given initial concentration, or sets it to be equal to the value of the given parameter.

Example 20 `present(MAPK,0.3)`. `present(CycE, k)`.
    makes present two molecules with concentration, respectively 0.3 *and the value of parameter k.*

- `absent(object_pattern)`.

- `absent(set_of_object_patterns)`.
    makes all objects (appearing in the instances of the current set of rules) and matching one of the given object patterns, absent from the initial state.

- `undefined(object_pattern)`.

- `undefined(set_of_object_patterns)`.

makes all objects (appearing in the instances of the current set of rules) and matching one of the given object patterns, possibly present or absent in the initial state.

- `make_present_not_absent`.

makes all objects (appearing in the instances of the current set of rules) which are not declared absent, present in the initial state.

- `make_absent_not_present`.

makes all objects (appearing in the instances of the current set of rules) which are not declared present, absent in the initial state.

2.5   Simulation

A BIOCHAM model can be interpreted at three levels of abstraction: either as an asynchronous boolean transition system, an ordinary differential equation, or a continuous-time Markov process. The second interpretation is deterministic and produces a unique simulation trace, while the other two interpretations produce sets of simulation traces.

2.5.1   Boolean simulator

For the boolean simulator, all the objects with an undefined initial state are considered as absent in the initial state. The simulator prints the objects present in the successive states of the simulation. The set of objects which are printed can be specified with patterns. By default all objects are printed.

- `boolean_simulation`.

- `boolean_simulation(integer)`.

performs a random simulation up to a given number of transitions (default is 30).

Note that because BIOCHAM boolean models are highly non-deterministic, random simulation is often impractical and not representative of all possible behaviors. It is therefore much more interesting to query the temporal properties of BIOCHAM models w.r.t. all possible behaviors in CTL logic.

Example 21 *Boolean simulation of the MAPK signaling cascade visualized with the* `plot` *command (see model EXAMPLES/MAPK/mapk.bc):*
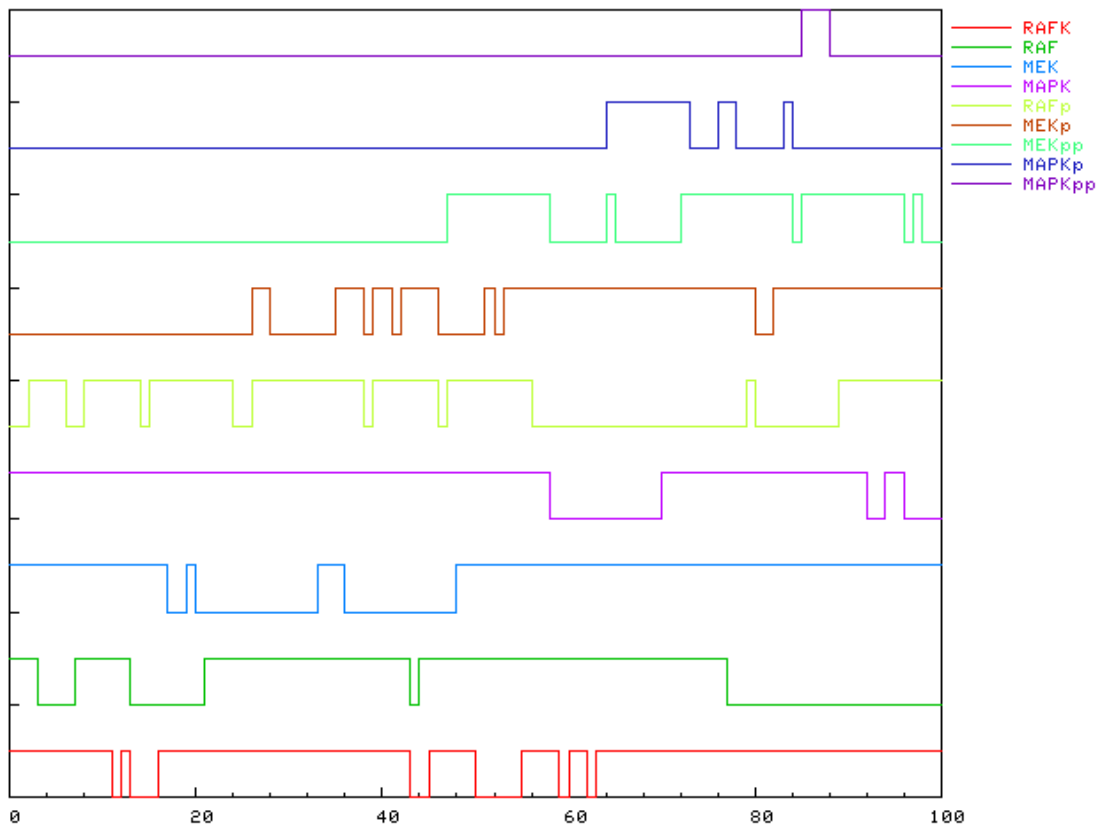
- `boolean_enumeration`.

  performs a step by step simulation by enumerating all possible behaviors of the system from the initial state. At each step you can either continue the simulation by typing return, backtrack to another transition by typing ¡, or stop the simulation by typing 'q'. The depth of the current derivation is printed.
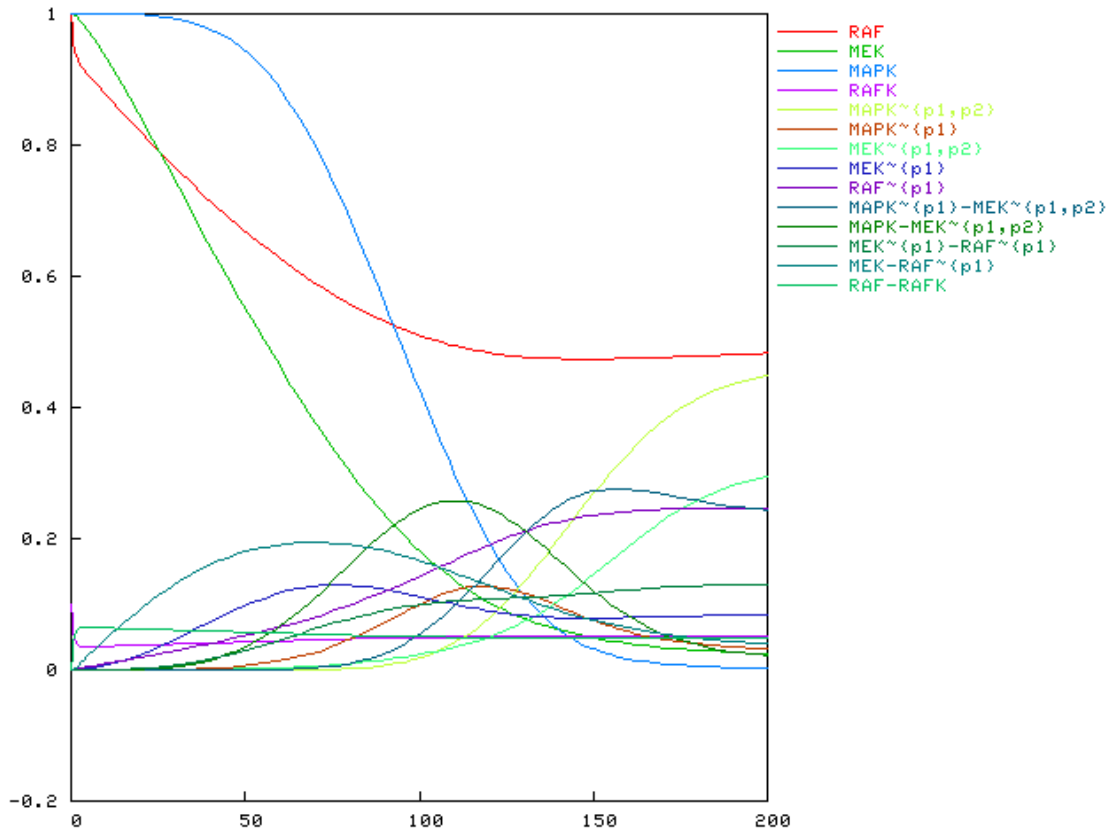
2.5.2 ODE and stochastic simulators

- `numerical_simulation`.

- `numerical_simulation(number)`.

  performs a numerical simulation (using one of the methods detailed below) up to a given number of time units (default is 20).

Example 22 *MAPK cascade numerical simulation visualized with the* `plot` *command:*

- `continue(number)`.

  performs a numerical simulation (using one of the methods detailed below) starting from the last point of the current simulation, up to a given number of time units.

- `numerical_method`.

- `numerical_method(name)`.

  shows the current numerical simulation method and the corresponding details (error, step_size, etc.), and if an argument is given, sets the desired simulation method. The currently available choices are: `rk` (Runge-Kutta), `stiff` (Rosenbrock), `ssa` (Gillespie) and `tl` (tau-lipping). The first one is a fourth-order Runge-Kutta method, with or without step-doubling (see below), the second one a Rosenbrock method (implicit), with variable step-size (this is the default choice), the third one is an implementation of Gillespie's algorithm and the last one of the tau-leaping method (these two stochastic methods rely on a conversion factor and a critical reaction threshold, see below).

- `step_size`.

- `step_size(float)`.
  sets the initial step-size (default is 0.01) for the Runge-Kutta method.

- `step_doubling`.

- `step_doubling(float)`.

- `no_step_doubling`.
  uses step-doubling method (default) to adapt the step-size to a specified maximum error (default is 0.0001). Step adaptation cannot be turned off for Rosenbrock method.

- `conversion_factor`.

- `conversion_factor(integer)`.
  Each initial concentration will be multiplied by this number, defaulting to 10000, to give the corresponding initial number of molecules of each kind. This amounts to say that the volume considered is actually this factor divided by Avogadro's Number.

- `critical_reaction_threshold`.

- `critical_reaction_threshold(integer)`.
  This integer (defaulting to 20) is used in the tau-leaping algorithm to determine for which reactions a tau leap is possible.

2.5.3 Conservations laws and P-invariants

It is possible to impose some invariance relation, or (mass) conservation law, to the system (which amounts to reduce the dimension of the ODE system). Such a law can be checked from the rules (i.e. no reaction can make it false), from the kinetics (i.e. the derivative is formally proven equal to zero), or trusted from the user.

Through the computation of the P-invariants of the underlying Petri-net, conservation laws can also be (partially) extracted without user input from the reaction model. This relies only on the reactions and not on the kinetics and will thus provide an incomplete list of invariants, however in most practical cases, using these invariants as conservation laws will already allow to reduce the dimension of the system.

Note also that the computed P-invariants point out some kind of modules describing the life cycle of some compounds.

- `conservation(set_of_object_patterns)`.

- `conservation(list_of_molecules_with_stoichiometry)`.
  Declares a new mass conservation law for all the molecules matched by the argument if it is a set, or for all molecules given with the corresponding weight. During a numerical simulation, one of those variables will be eliminated thanks to this conservation law. Be

careful if you declare conservation laws and then plot the result of a previous simulation, the caption might not be correct. When added, the conservation law will be checked against the rules (i.e. purely from stoichiometry), if that fails against the kinetics. Since these checks are not complete, even a failure will be accepted.

Example 23 `conservation({cycE-?})`.
  Will keep the sum of the numbers of molecules of all forms of cycE constant.

Example 24 `conservation([A-A, 2*A])`.
  Will keep the sum of the numbers of molecules of the dimer plus twice that of the monomer constant.

- `check_conservations`.
  Will check all conservation laws against rules, and if necessary kinetics (see above).

- `delete_conservation(set_of_object_patterns)`.

- `delete_conservation(list_of_molecules_with_stoichiometry)`.

- `delete_conservations`.
  Removes the given mass conservation law, or respectively all of them.

- `list_conservations`.
  Prints out all the mass conservation laws.

- `find_pinvar`.

- `find_pinvar(integer)`.
  Computes the P-invariants of the system, and thus conservation laws that are independent from the precise kinetics. One can give a limit on the highest value found in a given P-invariant (the default is 4), but note that giving a too high value might lead to a very long computation. Extra command

2.5.4 Plotting the result of simulations

- `plot`.
  plots the result of the last simulation, either boolean or numerical.

- `plot(object, object)`.
  plots the result of the last numerical simulation, as a trajectory in the phase space of the two given objects. The first object will be plotted on the x-axis, the second on the y-axis.

- `plot(object, object, object)`.

plots the result of the last numerical simulation, as a trajectory in the phase space of the three given objects. The first object will be plotted on the x-axis, the second on the y-axis and the third on the z-axis.

- `export_plot(file_template).`

saves the result of the last simulation into two files: `file_template.csv` and `.plot`, so that you can produce afterwards the plot of a given simulation by giving to GNUplot the command `load file_template.plot`.

- `show_molecules(object_pattern).`

- `show_molecules(set_of_object_patterns).`

adds the molecules matched by the pattern to the molecules printed by the simulator. Initially all molecules are shown.

- `hide_molecules(object_pattern).`

- `hide_molecules(set_of_object_patterns).`

removes the molecules matched by the pattern from the molecules printed by the simulator.

- `show_macros.`

will plot all macros for the numerical simulator.

- `show_macros(set_of_names).`

will plot at least the given macros for the numerical simulator.

- `hide_macros.`

will hide macros for the numerical simulator.

- `show_hide.`

lists which molecules and macros are hidden or shown.

- `keep_plot.`

keeps the current plot window open and asks future plots to use another window (for comparison).

- `set_color(object, integer).`

- `set_color(name, integer).`

sets the color used for the corresponding object/macro when plotting. The colors can be chosen from the list provided by the next command: test_plot.

- `test_plot.`

opens a special plot window executing the test command of GNUplot, and thus showing the available colors and the corresponding integer.

- set_xmin(float).

- set_xmax(float).

- set_ymin(float).

- set_ymax(float).

    sets the range shown on a numerical plot. These settings are reset (to fit the plot) each time a simulation is run.

- fit_xmin.

- fit_xmax.

- fit_x.

- fit_ymin.

- fit_ymax.

- fit_y.

    sets the range shown on a numerical plot so that the given coordinate fits the simulation. fit_x is the same as fit_xmin followed by fit_xmax, same for y.

    Some secondary commands are also supplied to get information out of numerical traces.

- get_max_from_trace(molecule).

- get_min_from_trace(molecule).

    prints out the maximal/minimal value of the concentration of the given molecule for the current trace, and the corresponding time value.

- get_period_from_trace(molecule).

    prints out the value of the period of oscillation of the given molecule for the current trace. The molecule must oscillate at least 3 times. If periods appear not to be constant, the last two periods are printed out.

- set_init_from_trace(float).

    takes the closest calculated time point in the current simulation trace, and sets the initial state according to values at that time point.

2.6 Boolean temporal properties

Formulae in Computation Tree Logic (CTL) express the temporal properties of a model that are true in *all possible* boolean simulations. CTL formulae can thus be used to query, or constrain, the boolean temporal properties of a model.

2.6.1 Checking CTL properties

CTL formulae can be evaluated with the model-checker NuSMV by using the following commands. Note that other model-checkers than NuSMV can be used by exporting BIOCHAM rules and initial state in an appropriate SMV, LOTOS or PROLOG file using the commands export_nusmv or export_lotos or export_prolog.

- `nusmv(ictl).`
  evaluates a temporal query using the model-checker NuSMV. The first use of this command may take a while as it will compile the rules into an ordered binary decison diagram (OBDD).

Example 25 `nusmv(Ei(EF(cdk1))).`

`why.`
  explains the result of the last query by producing a witness pathway when this is possible.

- `nusmv_why(ictl).`
  like `nusmv(ictl),why.` except that if the query is false, the model-checker does not compute the query twice.

- `fairness_path.`

- `no_fairness_path.`
  when evaluating a specification, forces the model-checker to consider path quantifiers to apply only to fair paths (resp. all paths). This is especially useful for loop properties. Warning: the time to compute queries can be bigger.

- `nusmv_dynamic_reordering.`

- `nusmv_disable_dynamic_reordering`
  reorders BDD variables dynamically, using NuSMV's group sift converge method. For a better efficiency in building the BDD, this command should be used before the first NuSMV query, even if it still works afterwards.

- `nusmv_direct.`

- `nusmv_non_direct`

changes the mode of evaluating NuSMV queries. In non-direct mode, which is the default, Ei and Ai queries are distinguished. In direct mode, which is the most efficient one, all Ei queries are transformed into Ai queries. When the initial state is completely defined, the direct mode should be used for better performance.

Furthermore,a set of CTL properties can be given as a specification representing the expected behavior of the system, as observed for instance by wet lab experiments, and that need be satisfied by the model.

- `add_spec(ictl)`.

- `add_specs(set_of_biocham_queries)`.
  adds some CTL temporal properties to the specification.

- `delete_spec(ictl)`.

- `delete_specs(set_of_biocham_queries)`.
  deletes some CTL temporal properties from the specification.

- `list_spec`.
  lists the current set of CTL formulae in the specification.

- `clear_spec`.
  clears the CTL specification.

- `check`.
  tests the adequacy of the model w.r.t. its specification, for each unsatisfied CTL property, computes the result of why.

- `check_why`.
  tests the adequacy of the model w.r.t. its specification, and for each CTL property, computes the result of why.

- `check_all`.
  tests the adequacy of the model w.r.t. its specification, summarizes the result with the first unsatisfied property if there is one.

2.6.2 Inferring CTL properties

The set of all CTL properties of some simple pattern that are true in the model can also be automatically generated with the following commands :

- `genCTL`.

- `genCTL(filename)`.

writes some simple CTL properties (reachable, oscil, steady, checkpoint for each molecule) that are true in the model.

- `add_genCTL`.
  adds to the current specification those simple CTL properties (reachable, oscil, steady, checkpoint for each molecule) that are true in the model.

2.6.3   Model reductions preserving CTL properties

A CTL specification can also be used to reduce a model with the following commands :

- `reduce_model`.

- `reduce_model(reaction_pattern)`.

- `reduce_model(set_of_reaction_patterns)`.
  reduces the model by deleting rules, upto a minimal model that satisfies the whole specification. Iterates the command `learn_one_deletion(bias)` where the default bias is the rule pattern `?=>?` to consider all the rules of the model

2.6.4   Learning rules from a CTL specification

Since the rule language and property language have both been formalized, one can use Machine Learning techniques to try and find completions or modifications of a model such that the specification is satisfied. This is the very place where `reaction_shortcuts` are useful to define the reaction patterns of interest.

- `learn_one_addition(reaction_pattern)`.

- `learn_one_addition(set_of_reaction_patterns)`.
  finds all the single rules coming from the expansion of the given pattern(s) and such that adding them to the model makes it comply with the current specification.

- `learn_one_deletion(basic_reaction_pattern)`.

- `learn_one_deletion(set_of_basic_reaction_patterns)`.

- `learn_one_deletion`.
  tries to find a single rule coming from the expansion of the given pattern(s) and such that deleting it from the model makes it comply with the specification. If no pattern is given, the rules tried are the ones coming from the path of negative false specifications.

- `revise_model`.

- `revise_model(reaction_pattern)`.

- `revise_model(set_of_reaction_patterns)`.
    tries to correct the model using a theory revision algorithm. Takes all specification formulae one after the other, starting with positive (ECTL) ones, then undefined ones, then negative (ACTL) ones, in order to satisfy them:

    - positive formula: add a rule from the given reaction pattern, called bias (by default elementary_interaction_rules) such that all formulae already treated remain true;

    - negative formula: retract one or more rules from the path of the counter-example and if some positive or undefined formulae become false, treat them again;

    - undefined formula: try to retract some rules or to add a rule such that all formulae already treated remain true.

The algorithm stops as soon as one solution is found.

- `revise_model_interactive`.

- `revise_model_interactive(reaction_pattern)`.

- `revise_model_interactive(set_of_reaction_patterns)`.
    same as above, but instead of stopping after a solution is found, the user gets the choice to stop or to continue looking for another solution.

2.7   Temporal properties with numerical constraints

The properties formalized in Linear Time Logic with numerical constraints, LTL(R), can be used in numerical models either to check their correctness, to infer parameter values for satisfying them, or to provide robustness measures.

2.7.1   Checking LTL(R) properties

An LTL(R) specification can be created and checked against the last simulation with the following commands. It is worth noticing that the notion of next state, using the X operator of LTL(R), refers to the following state as computed by the (variable step-size) simulation, and thus does not necessarily imply real-time neighborhood, but a "calculation" neigborhood. Equality of values are checked according to the time discretization using Rolle's theorem : an equality A=B holds in a time point if the sign of A-B is zero or changes in the next time point.

- `trace_check(ltl)`.
    evaluates an LTL(R) query on the latest trace produced (if none exists, one will be generated by numerical_simulation).

- `add_ltl(ltl)`.

- `add_ltl(set_of_ltl_queries)`.
   adds the given formula(e) to the LTL(R) specification.

- `delete_ltl(ltl)`.

- `add_ltl(set_of_ltl_queries)`.
   removes the given formula(e) to the LTL(R) specification.

- `list_ltl`.
   prints out the current LTL(R) specification.

- `clear_ltl`.
   removes completely the current LTL(R) specification.

- `check_ltl(number)`.

- `check_ltl`.
   checks each formula of the LTL(R) specification against a simulation of the given duration. If no duration is provided, tests against the latest simulation.

2.7.2 Instantiating variables in QFLTL(R) formulas

QFLTL(R) formulas are quantifier-free first-order LTL(R) formulas possibly containing real valued variables. A QFLTL(R) formula can be checked on a numerical trace by computing the free variables' domains that make the formula true on the trace. This is done by the command `trace_analyze`. The trace analyzed can be the trace of a simulation or an external trace of numerical data time series obtained by some biological experiment.

- `load_trace(file)`.
   loads a numerical trace from a .csv file.

- `trace_analyze(qfltl)`.
   computes domains of variables contained in (qfltl) that make the formula true on a numerical trace. The trace used is the last simulated or loaded trace.

Example 26 `trace_analyze(F([A]>=v))`.

`(v =< 0.5)`

The domains are described in the output by a disjunction of conjonction of inequality constraints over the variables.

2.8   Learning parameters from an LTL(R) specification

In the same spirit as what is done for rule learning w.r.t. CTL specifications, one can use the above LTL(R) model-checking techniques to automatize the search for parameter values (i.e. kinetic parameters, initial value or control parameters) satisfying an LTL(R) specification with numerical constraints. The first commands below use a simple scanning of the parameter space and are limited to searching two or three parameters. The commands suffixed by cmaes are much more efficient. They apply to QFLTL(R) formulae containing variables and use the state-of-the-art optimization method CMAES (covariance matrix adaptive evolution strategy of N. Hansen [3]) for searching several tenths of parameter values in one run. The found parameter values are printed in a form that can be copied and pasted in the command line to adopt them.

2.8.1   Simple search methods with an LTL(R) specification

- `search_parameters(list_of_name,list_of_pairs_of_floats,int,ltl,number)`.

- `search_parameters(list_of_name,list_of_pairs_of_floats,int,number)`.
   returns the first values found for the parameters of given names, between the min and max values given, with the given number of tries for each parameter, such that the `ltl_query` representing a specification of the system is true for simulations up to the given time horizon number. If no LTL(R) query is given, the current LTL(R) specification will be used. This command is usually used for searching two or three parameter values at a time, as its time complexity is exponential in the number of parameters (and polynomial in the number of tries), i.e. in $O(n^p)$ where n is the number of tries for each parameter and p is the number of parameters.

Example 27 *The command*

`search_parameters([k],[(0,10)],100,F(([X] > 0.3) \& F(d([X])/dt =< 0)),20).`

   scans the values of parameter 'k' between 0 and 10 and returns the first value such that, before time 20, [X] gets greater than 0.3 and later [X] decreases. The total number of tries will not be more than 100.

- `search_all_parameters(list_of_name,list_of_pairs_of_floats,int,ltl,number)`.

- `search_all_parameters(list_of_name,list_of_pairs_of_floats,int,number)`.
   returns all the values found for the parameters satisfying the LTL(R) specification.

- `search_random_parameters(list_of_name,list_of_pairs_of_floats,int,ltl,number)`.

- `search_random_parameters(list_of_name,list_of_pairs_of_floats,int,number)`.

  returns the first values found for the parameters given in the first argument satisfying the specification. by making random choices of values in the given list of intervals for each parameter. The third argument is the maximum number of iterations. The last argument is the time horizon. A greater number of parameter values can be searched with this command for LTL(R) specification containing many solutions. Different runs of this command may give different answers.

- `search_random_all_parameters(pair_of_floats,int,ltl,number)`.

- `search_random_all_parameters(pair_of_floats,int,number)`.

  searches the values for all parameters of the current model, in a given interval of possible values. Returns the first value found satisfying the specification.

2.8.2 Continuous optimization methods with a QFLTL(R) specification

The following much more efficient parameter search commands use the non-linear optimization method CMAES with the continuous satisfaction degree of a temporal specification as fitness function. This satisfaction degree is defined by the distance between the validity domain of a QFLTL(R) formula with free variables (as given by trace_analyze) and the objective values given for some of the free variables.

- `search_parameters_cmaes(list_of_names,list_of_pair_of_floats,qfltl,`
`list_of_names,list_of_floats,number)`.

- `search_parameters_log_cmaes(list_of_names,list_of_pair_of_floats,qfltl,`
`list_of_names,list_of_floats,number)`.

  uses the violation degree of the given temporal specification as a fitness function for the non-linear optimization tool cmaes to guide the search. The temporal specification is composed of a QFLTL(R) formula given with a list of variables and a list of ojective values for these variables. Search command with log uses lognormals distributions of parameter values instead of normal distributions to explore their neighborhoods

Example 28 `search_parameters_cmaes([k],[(0,20)],F([A]>=v),[v],[100],50)`.
  Searches for a value of 'k' between 0 and 20 such that, before time 50, [A] gets greater than 100 (i.e. greater than v with v=100 as objective).

- `cmaes_params(int,number)`.

  set CMAES stop criteria concerning the number of calls to the fitness function (default 300) and the value of the fitness function (default 0.01).

2.8.3   Multi-trace conditions

Sometimes the temporal specification of the behavior of the systems does not refer to a single set of parameter values and initial conditions, but to several sets. The following commands make it possible to search parameter values with multi-trace conditions.

- first_search_condition(list_of_names,list_of_intervals,qfltl,
list_of_names,list_of_float,number).

- add_search_condition(qfltl,list_of_names,list_of_parameter_pairs,
list_of_nums).

- cmaes_multi_conditions.
    searches for parameter values satisfying different specifications in multiple conditions. Commands first_search_condition and add_search_condition define the search problem and specification for each condition while cmaes_multi_conditions starts the search with cmaes.

Example 29
first_search_condition([k1,k2], [(0,20),(0,20)], F([A]>v),[v],[100],50).

add_search_condition(F([A]>v), [v], [200], [(k_mutant,k_wildtype)]).

cmaes_multi_conditions.
    searches for values of 'k1' and 'k2' between 0 and 20 such that [A] gets greater than 100 in the first condition (for example the wild type) and gets greater than 200 in the second condition (for example a mutant) where parameter 'k_wildtype' is replaced by 'k_mutant'. The list of pairs of parameters defines changes between first condition and mutant conditions. These parameters can also be included in the search to search for conditions producing given specifications. The number of total conditions is not limited.

- robustness(list_of_names,list_of_floats,qfltl,list_of_names,
list_of_floats,int,number).

- robustness_log(list_of_names,list_of_floats,qfltl,list_of_names,
list_of_floats,int,number).
    computes the robustness and relative robustness of the system with respect to the specification given as QFLTL(R) formula, for normally distributed parameters perturbations around their current value with given coefficient of variation. Command suffixed by log uses lognormally perturbed parameters.

Example 30 robustness([k1,k2], [0.1,0.05], F([A]>=v), [v], [100], 500, 50).
evaluates the robustness, and relative robustness, of F([A]>=100 *in time horizon 50, i.e. "A*

reaches value 100 before time 50" by computing the satisfaction degree of this specification for 500 samples of normally distributed parameters, k1 and k2, with coefficient of variation 0.1 and 0.05.

- `landscape(list_of_names,list_of_floats,qfltl,list_of_names,`
`list_of_floats,int,number).`

- `landscape_log(list_of_names,list_of_floats,qfltl,list_of_names,`
`list_of_floats,int,number).`
  displays the satisfaction degree landscape of a QFLTL(R) formula on a 2D parameter grid. Command suffixed by log computes the satisfaction degree on a log scaled grid.

Example 31 `landscape([k1,k2], [(0,100),(50,100)], F([A]>=v), [v], [100], 20, 50).`
  *Displays the satisfaction degree landscape of 'A reaches value 100 before time 50' for parameters k1,k2 ranging in (0,100) and (50,100). Satisfaction degree values are computed on a grid of size 20*20.*

2.9 Types

A reaction model can be abstracted in many ways to get partial information on the objects of the model. The relationship between a reaction model and an abstraction of it is called a typing, by analogy to the use of types in programming languages.

Currently three simple typings are implemented in BIOCHAM for infering from the reaction rules, respectively, the influence graph (of activation and inhibition) between objects, the function (kinase or phosphatase) of objects, the neighborhood relation between locations.

2.9.1 Influence graph

A graph of positive or negative influences between molecular species can be inferred from the reaction rules. Under very general condition on the kinetic expressions in the rules [4], this graph is identical to the influence graph defined by the signs of the coefficient in the Jacobian matrix of the ODE interpretation of the rules. Thomas's necessary conditions for multistability (existence of a positive circuit in the influence graph) and for oscillations (existence of a non-tricial negative circuit) apply on this graph.

- `show_influences`.
  infers the influence graph and prints the activation and inhibition relations between objects.

- `export_influences_dot(file).`

infers the influence graph in terms of activations and inhibitions from the BIOCHAM rules, and exports it in a .dot file. That file can then be used to generate pictures of the influence graph, with for instance tools such as Graphviz.

- `export_influences_ginml(file)`.
  Same as above but for the GINsim tool.

- `draw_influences`.
  infers the influence graph and visualizes it using Graphviz and gv (that have to be installed and in your PATH).

2.9.2   Object functions

- `show_functions`.
  infers the kinase and phosphatase functions of molecules from the rules.

2.9.3   Location neighborhood

- `show_neighborhood`.
  infers the neighborhood relation between locations.

- `draw_neighborhood`.
  infers the neighborhood graph and visualizes it using Graphviz and gv (that have to be installed and in your PATH).

- `export_neighborhood_dot(file)`.
  infers the neighborhood relation between locations, and exports it in a .dot file. That file can then be used to generate pictures of the neighborhood graph, with for instance tools such as Graphviz.

Chapter 3

Graphical User Interface

A Graphical User Interface is provided to access in a easier way to BIOCHAM commands. It can be started with the command: `biocham_gui`. If you have not installed BIOCHAM with the make install command, then you must be in BIOCHAM's installation directory to use the above command. Another solution is to call directly `java -jar` BIOCHAM's installation directory`/gui/biocham.jar` BIOCHAM's arguments This invocation can be followed by .bc file names indicating files to load on startup.

Here is a screenshot of the interface:
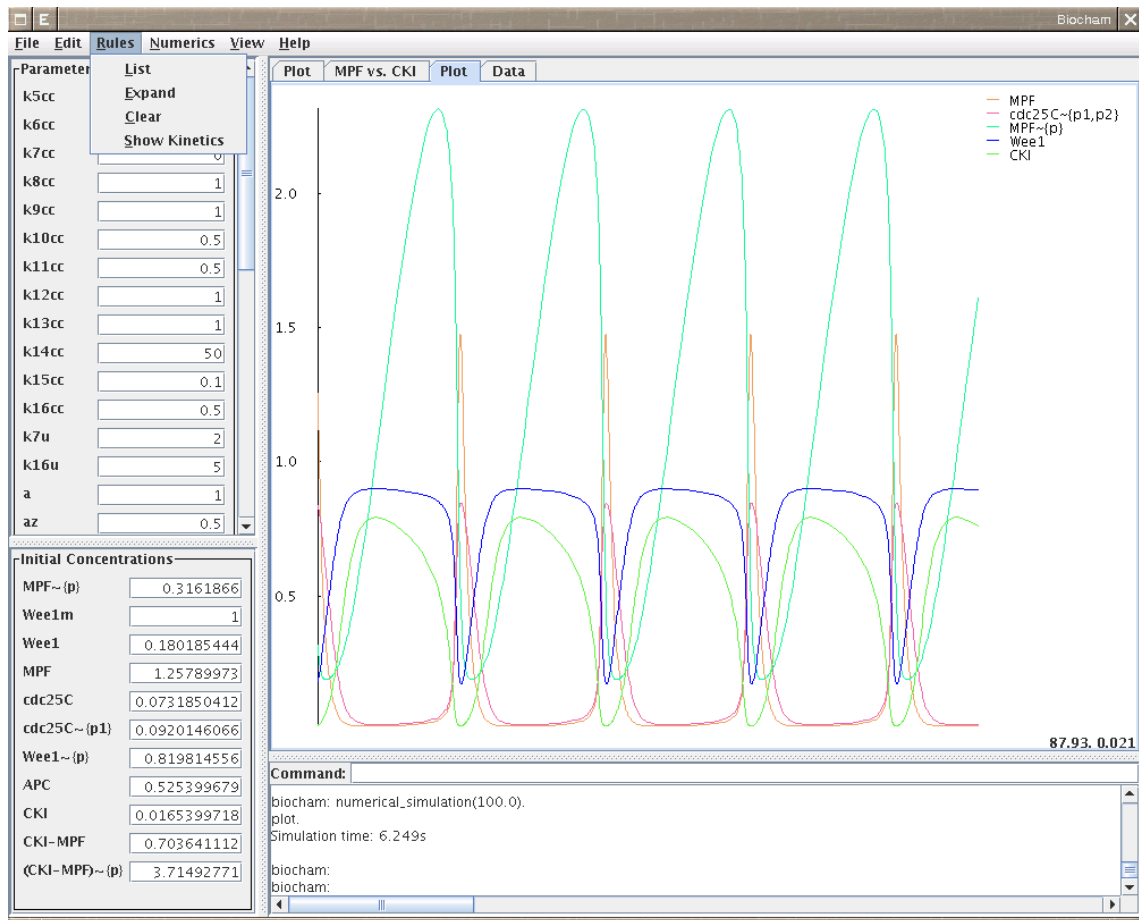
The upper left part of the window contains a list of the parameters of the model and their current value. The values can be modified by clicking in the white box. Similarly, the lower left part contains the biochemical compounds and their initial concentration, which can be changed in the same way.

The upper right part contains a set of tabs. If a simulation is run and plotted, it will appear in one tab (use of the `keep_plot` command of BIOCHAM will allow to keep a tab for future reference). Clicking in the caption allows to change the colors of the plotted compounds. The coordinates are displayed below the caption. Note that the GUI can not plot the result of boolean simulations yet. Using the left button, one can select a rectangular area of a plot for zooming. Double-clicking will bring back the previous zoom.

Plots of the phase space will also appear in the upper right part, they will be updated if the corresponding simulation plot is modified. Finally the data browser will also use tabs, but those will remain static (no update). The data browser offers a search feature that will look for a value in the selected column, from the selected cell downwards.

To get rid of an old tab, double-click on it.

Below the tab area is a zone for inputting commands. It will complete command names if ¡TAB¿ is pressed, and keeps an history of commands that can be consulted with the up and down arrows.

The lower right panel shows the usual output of BIOCHAM.

The menu items are self-explanatory, except that running a simulation will automatically plot its result afterwards.

Chapter 4

Differences with Earlier Versions

New features of BIOCHAM 2.9 (Mar. 2009)

- GINsim ginml export of the influence graph;

New features of BIOCHAM 2.8 (Jan. 2009)

- Parameter optimization w.r.t. QFLTL(R) properties using CMAES for searching several tenths of parameter values in one run;

- Robustness analysis w.r.t. LTL(R) properties;

- Much improved SMBL support.

New features of BIOCHAM 2.7 (Apr. 2008)

- Stochastic simulation methods (Gillespie and tau leaping);

- Syntax for conservation laws with stoichiometry. Automatic computation through P-invariants;

- Instantiation of variables in LTL(R) formulas out of numerical traces;

- new search algorithms for the command `learn_parameter` renamed in `search_parameter`

- `check_ltl` and `search_parameter` now default to using the current ltl specification;

- Several improvements to the Java GUI;

- Commands to delete events;

- Bugfix conservation laws accross locations of different volumes.

- Windows version now runs completely without Cygwin

New features of BIOCHAM 2.6 (Feb. 2007)

- Abbreviations for Hill kinetics; MA and MM arguments generalized to expressions;

- Default mass action law MA(1) kinetics for rules written without kinetic expression;

- Shortcuts for temporal properties integrated as formulas and suppressed as commands;

- The dot command displays its result in the GUI;

- Export to Prolog; CTL model checker in Prolog; Prolog call command;

New features of BIOCHAM 2.5 (June 2006)

- Locations, i.e. SBML like compartments;

- Basic typing;

- Plot fitting commands;

- LTL(R) specifications can now be added to a model and checked in the same way as CTL ones;

- Mass conservation laws (algebraic invariants);

New features of BIOCHAM 2.4 (Oct. 2005)

- More options for LTL(R) model checking on numerical traces, especially about period of oscillations.

- Plotting of numerical simulations as trajectories in the phase space.

- Abbreviations for Mass Action Law and Michaelian kinetics.

- A preliminary Java-based Graphical User Interface.

- A possibility to select which macros to plot.

- Some syntax changes in command names.

- Event handling.

New features of BIOCHAM 2.3 (June 2005)

- Learning of (interaction) rules.

- Export to xppaut's .ode format.

- LTL(R) model checking on numerical traces, and learning of numerical parameters.

- A new, more versatile parser, thanks to Daniel de Rauglaudre.

- More shortcuts for NuSMV queries. Fairness (weak) or dynamic reordering BDD's variables. Support of NuSMV 2.2.2 and 2.2.3.

- The vim mode that was developed externally is now merged in the distribution, an emacs mode is available too.

New features of BIOCHAM 2.2 (Mar. 2005)

- BIOCHAM 2.2 brings more flexibility for plotting simulation results. The simulation can use (and that's the default) an implicit method, and thus handles stiff equations.

- Any BIOCHAM command can now appear in a BIOCHAM file.

- SBML (resp. BIOCHAM) parameters are now imported (resp. exported) to BIOCHAM (resp. SBML) parameters.

- Lots of minor bugs were fixed.

New features of BIOCHAM 2.1 (Oct. 2004)

- BIOCHAM 2.1 implements the adaptive step size method of step doubling for Runge-Kutta integration method.

- Bug fixed on the parsing of arithmetic expressions (implies to add parentheses around complexes given with stoichiometric coefficients).

New features of BIOCHAM 2.0 (Aug. 2004)

- BIOCHAM 2 introduces the ability to use quantitative models, with stoichiometric coefficients and kinetic laws (examples are available in the EXAMPLES/kinetics directory). These models can be numerically simulated, or used as before for model-checking via a boolean abstraction. Parameters and macros can also be defined, used in rate laws and saved in a separate file.

- An import/export function to/from SBML has also been introduced.

- Some shortcuts for common CTL queries are now available.

New features of BIOCHAM 1.0 (Feb. 2004)

- BIOCHAM 1.0 introduces a rich pattern language for defining reaction rules in a concise manner. The complexation operator is now supposed to be associative and commutative. The operators Ei and Ai have been introduced in the query language to quantify over the initial states.

- BIOCHAM 1.1 exports the interaction map to Graphviz dot format.

Features of BIOCHAM 0.0 (July 2003)

- Reaction rule based modeling language without patterns.

- Interface to NuSMV model checker for CTL queries.

Acknowledgements

Bibliography

[1] François Fages and Sylvain Soliman. Formal cell biology in BIOCHAM. In M. Bernardo, P. Degano, and G. Zavattaro, editors, *8th Int. School on Formal Methods for the Design of Computer, Communication and Software Systems: Computational Systems Biology SFM'08*, volume 5016 of *Lecture Notes in Computer Science*, pages 54–80, Bertinoro, Italy, February 2008. Springer-Verlag.

[2] Nathalie Chabrier and François Fages. Symbolic model checking of biochemical networks. In Corrado Priami, editor, *CMSB'03: Proceedings of the first workshop on Computational Methods in Systems Biology*, volume 2602 of *Lecture Notes in Computer Science*, pages 149–162, Rovereto, Italy, March 2003. Springer-Verlag.

[3] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.

[4] François Fages and Sylvain Soliman. From reaction models to influence graphs and back: a theorem. In *Proceedings of Formal Methods in Systems Biology FMSB'08*, number 5054 in Lecture Notes in Computer Science. Springer-Verlag, February 2008.

[5] François Fages, Sylvain Soliman, and Nathalie Chabrier-Rivier. Modelling and querying interaction networks in the biochemical abstract machine BIOCHAM. *Journal of Biological Physics and Chemistry*, 4(2):64–73, October 2004.

[6] François Fages. From syntax to semantics in systems biology - towards automated reasoning tools. *Transactions on Computational Systems Biology IV*, 3939:68–70, December 2006.

[7] Laurence Calzone, Nathalie Chabrier-Rivier, François Fages, and Sylvain Soliman. Machine learning biochemical networks from temporal logic properties. In Gordon Plotkin, editor, *Transactions on Computational Systems Biology VI*, volume 4220 of *Lecture Notes in BioInformatics*, pages 68–94. Springer-Verlag, November 2006. CMSB'05 Special Issue.

[8] Aurélien Rizk, Grégory Batt, François Fages, and Sylvain Soliman. On a continuous degree of satisfaction of temporal logic formulae with applications to systems biology.

In Monika Heiner and Adeline Uhrmacher, editors, *CMSB'08: Proceedings of the fourth international conference on Computational Methods in Systems Biology*, volume 5307 of *Lecture Notes in Computer Science*, pages 251–268. Springer-Verlag, October 2008.

[9] Aurélien Rizk, Grégory Batt, François Fages, and Sylvain Soliman. A general computational method for robustness analysis with applications to synthetic gene networks. *Bioinformatics*, 12(25):il69–il78, June 2009.

[10] François Fages and Sylvain Soliman. Abstract interpretation and types for systems biology. *Theoretical Computer Science*, 403(1):52–70, 2008.

[11] Nathalie Chabrier-Rivier, Marc Chiaverini, Vincent Danos, François Fages, and Vincent Schächter. Modeling and querying biochemical interaction networks. *Theoretical Computer Science*, 325(1):25–44, September 2004.

[5] is an easy-to-read introductory paper on BIOCHAM. [6] is a position paper and [1] an in-depth tutorial on the main concepts of BIOCHAM.

[7] describes the machine learning features of BIOCHAM (commands `revise_model` and `search_parameters`). The new methods currently used for parameter search and robustness analysis are described in [8] and [9] respectively.

[4] is a study on the formal relationship between reaction graphs and influence graphs (command `show_influences`). [10] introduces a general setting for defining abstractions on reaction models (commands `show_functions`, `show_neighborhood`).

[2] is the first paper on symbolic model-checking of biochemical networks (command `nusmv`) with evaluation on Kohn's map of the mammalian cell cycle control [11].

Index