Instructions to use BioReSolve, the interpreter of Reaction Systems authored by Linda Brodo, Roberto Bruni, and Moreno Falaschi.

The interpreter extends a previous implementation defined by Moreno Falaschi and Giulia Palma for the execution of basic Reaction Systems with many additional features, like non-deterministic and recursive contexts, LTS semantics, BioHML verification and biosimilarity, along the theory developed in the paper:

SOS Rules for Equivalences of Reaction Systems
by *Linda Brodo, Roberto Bruni and Moreno Falaschi*
CoRR abs/2008.13016, https://arxiv.org/abs/2008.13016

BioReSolve has been developed under SWI-Prolog (https://www.swi-prolog.org/), exploiting the libraries `lists`, `ordsets`, `assoc` and `'dcg/basics'`.

Installation and usage instructions:

1. Choose a local folder in your file system and let *<workspace>* be it absolute path.
2. Download the main file BioReSolve.pl and save it into the folder *<workspace>* .
3. Write the RS specification and save it into the file *<workspace>*/spec-myRS.pl (for detailed instructions for writing RS specifications see next section).
4. Edit the file BioReSolve.pl to add the directives

```
:- ["<workspace>/spec-myRS.pl"]. % import RS specification
wdpath("<workspace>/"). % set working directory to save output
```

5. Launch SWI-Prolog and consult the file *<workspace>/BioReSolve.pl* .
6. Query the interpreter using the predicate `main/2`:

```
?- main(option,T).
```

where *option* can be one of the following atoms (more options are also available):
`stat` : computes some general information about the RS.
`target` : computes the terminal result set of the RS (requires a terminating context).
`run` : computes the result sequence of the RS (requires a terminating context).
`rundigraph` : draw the result sequence as an LTS (requires a terminating context).
`digraph` : computes the LTS of the Reaction System.
`advdigraph` : computes the LTS of the adversary Reaction System.
`biohml` : checks if the Reaction System satisfy a BioHML formula.
`biosim` : checks biosimilarity of a Reaction Systems and its adversary.

The Prolog interpreter will respond the query with

```
T = <execution time>
```

and save the result in (a newly created file in) the folder *<workspace>/* .

Each file is assigned a temporary name of the form

```
tmp-<keyword>-<timestamp>.<suffix>
```

Writing a custom RS specification:

An RS specification requires the definition of the following predicates:
myentities/1 : the initial set of entities of the custom RS.
myreactions/1 : the list of reactions of the custom RS.
myenvironment/1 : the list of constant declarations (for context processes) of the RS.
mycontext/1 : the list of context processes of the custom RS.
myexperiment/1 : (advanced usage) a special kind of context.
mybhml/1 : the BioHML formula to check.
myassert/1 : the assertion F for checking F-biosimulation ot the adversary RS.
adventities/1 : the initial set of entities of the adversary RS.
advreactions/1 : the list of reactions of the adversary RS.
advcontext/1 : the list of context processes of the adversary RS.
Important: The custom RS and its adversary share the same environment.

To ease the writing of custom RS specifications, a template with detailed instructions is available: you are warmly suggested to download the file spec-template.pl and edit it.

Below you find the syntax to use for each predicate.

myentities/1 and adventities/1 take a list of entities. Entities (and process constants) can be any sequence of letters, _ and digits that starts with a small cap letter. For example, you can set:

```
myentities([a,b]).
adventities([a,c]).
```

myreactions/1 and advreactions/1 take a list of reactions.
Each reaction is a term of the form react(R,I,P) where: R is the list of reactants, I is the list of inhibitors, P is the list of products. For example, you can set:

```
myreactions([react([a,b],[c],[b])]).
% an adversary with the same reactions as the custom RS
advreactions(Rs) :- myreactions(Rs).
```

mycontext/1 and advcontext/1 take a string the defines the list of context processes. Each context process K follows the grammar:

```
K ::= nil | X | {C}.K | (K1 + … + Kn) | <N,K1>.K2
```

where nil stops the computation, X invokes the context process associated with the process constant X according to the environment, {C}.K makes available the entities C in the current step and then behaves as K at the next step, (K1 + … + Kn) denotes a nondeterministic choice between K1,…,Kn and <N,K1>.K2 performs N steps as K1 and then behaves as K2. For example, you can set:

```
mycontext("[ ({a,c}.x + {a}.{b}.nil) ]").
advcontext("[ ({a,b}.y + {a}.{c}.nil) ]").
```

`myenvironment/1` takes a string that defines a list of (possibly mutual recursive) process constant declarations `X=K`. For example, you can set:

```
myenvironment("[ x = {a}.y , y = {a,b}.x ]").
```

`myassert/1` takes a string that defines the BioHML formula `F` to be used for checking biosimulation of the custom RS against the adversary RS. The assertion F must be given according to the grammar:

```
F ::= ? inW | ? inR | ? inI | ? inP
    | C inW | C inR | C inI | C inP
    | (F1 /\\ … /\\ Fn) | (F1 \\/ … \\/ Fn) | (F1^F2) | -F
```

where the productions in the first line represent non-emptyness tests for label components, the productions in the second line represent inclusion tests for label components (`C` can be any set of entities), the productions in the third line represent conjunction, inclusive disjunction, exclusive disjunction and negation, respectively. For example you can set:

```
myassert("-(? inP /\\ - {a,b,c} inR /\\ ? inW)").
```

`mybhml/1` takes a string that defines a BioHML formula `G`, defined over some fixed assertion `F`. The BioHML formulas must be given according to the grammar:

```
G ::= true | false | (G1 /\\ … /\\ Gn) | (G1 \\/ … \\/ Gn)
    | <F>G | [F]G
```

where the productions in the first line represent the obvious logical predicates, `<F>G` is the diamond modality and `[F]G` is the box modality. For example you can set:

```
mybhml("<-{c} inW> [-{c} inW] <-{c} inW> true").
```

`myexperiment/2` takes two lists that define a special kind of context that can be used to experiment with some advanced features (not to be discussed here). The special context has the form

```
Q1. … Q1.Q2. … Q2.Q3. … Q(n-1).Qn … Qn.nil
```

i.e. it provides a certain set of entities `Q1` for some fixed number of steps `W1`, then `Q2` for `W2` steps, and so on. Each `Qi` is possibly empty. The first argument is the list of numbers `[W1,…,Wn]` and the second list is just `[Q1,…,Qn]`. For example you can set

```
myexperiment([3,2,5] , [ [a,b] , [a,c] , [] ]).
```

Output files:

Depending on the query, results are saved in `.txt` files or `.dot` files. Both kinds of files can be opened and edited with any text editor.

The suffix `.dot` denotes a text-like representations in .dot format of graphs describing Reaction Systems computations. A file in `.dot` format roughly consists of the list of nodes and arcs of the graph.

Graphs in `.dot` format can be visualized using , e.g.,

GraphViz (https://graphviz.org)
Gephi (https://gephi.org)
Vis.js (https://visjs.github.io/vis-network/docs/network/)
graphviz-visual-editor (http://magjac.com/graphviz-visual-editor/)

The script `dottoxml.py` (https://github.com/dirkbaechle/dottoxml) can be used for conversions of `.dot` files to other formats (Graphml|GML|GDF). In particular, graphs in `.graphml` format can be easily manipulated using the graph editor

yEd (https://www.yworks.com/products/yed)