SMBioNet user manual

Laboratoire I3S - CNRS & Université de Nice

March 2010

SMBioNet[1] is a tool for modeling biological regulatory systems. It is based on (an extension of) the *multivalued logical formalism of René Thomas* [1, 2, 3] and the *Computational Tree Logic* (CTL) [4].

Input: A discrete regulatory network (with partially specified logical parameters) and a CTL formula (that expresses the biological observations or hypothesis on the behaviors of the system to model).

Output: All the parameterizations of the network that lead to a dynamical model satisfying the given CTL formula. The verification step is performed with the NuSMV symbolic model checker [5].

1 Input file

The input file is divided in four sections. The first two sections (VAR and REG) describe what we call a discrete regulatory network. In the third section (PARA) some constraints on the logical parameters associated to the network are given. The last section (CTL) contains a CTL formula.

1.1 Discrete regulatory network (sections VAR and REG)

1.1.1 Syntax

The input file starts by describing a *discrete regulatory network*. Such a network consists in a set of *discrete variables*, which interact each other *via regulatory processes*. The section VAR describes variables and their domains. These domains are always finite intervals of integers. For instance, with

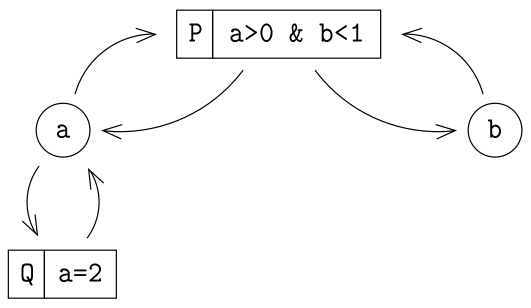
```
VAR
a = 0 2 ;
b = 0 1 ;
```

[1]Selection of Models of Biological Networks

we declare two variables: a variable `a` whose domain is `{0,1,2}`, and a variable `b`, which is a Boolean variable. The section `REG` describes the *regulatory processes* of the network. A regulatory process consists in: (1) a propositional formula, constructed from simple propositions of the form `a>1` or `b<a`, and connected with the logical operators `&` (and), `|` (or) and `->` (implication); and (2) a set of target variables. For instance, by writing

```
REG
P [a>0 & b<1]=> a b ;
Q [a=2]=> a ;
```

we specify that the network has two regulatory precesses, `P` and `Q`. The formula of `P` is `[a>0 & b<1]`, and its targets are `a` and `b`. The formula of `Q` is `[a=1]`, and its unique target is `a`. A discrete regulatory network can be seen as a (labeled) bipartite graph, in which vertices are either variables or regulatory processes (the predecessors of a process are the variables involves in its formula, and its successors are its target variables):



In the following, we say that a process is a process *of a given variable* if this variable is a target of the process (the processes of `a` are `P` and `Q`, and `P` is the unique process of `b`).

1.1.2 Semantic

The semantic of a discrete regulatory network is given by the *set* of dynamics that the network can display. To define these possible dynamics, we need additional notions, the first one being the notion of a state.

A *state* is the assignment of a value to each variable of the network. For the network described in the previous section, there are six possible states:

a	b
0	0
0	1
1	0
1	1
2	0
2	1

At a given state, we say that a process is *present* (1) when its formula is true, and *absent* (0) when its formula is false.

a	b	P	Q
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	0
2	0	1	1
2	1	0	1

From a dynamic point of view, at a given state, each variable evolves in the direction of a *target value* that only depends on the set of processes of the variable that are present. This target value is given by a so called *logical parameter*. For the running example, the logical parameters are

K_a	K_b
K_a+P	K_b+P
K_a+Q	
K_a+P+Q	

For instance, the logical parameter K_a+P gives the value toward which the variable a evolves when P is present and Q absent, and K_a gives the value toward which the variable a evolves when the regulatory processes P and Q are absent. The target value of each variable at each state is thus given by the following table:

a	b	Target of a	Target of b
0	0	K_a	K_b
0	1	K_a	K_b
1	0	K_a+P	K_b+P
1	1	K_a	K_b
2	0	K_a+P+Q	K_b+P
2	1	K_a+Q	K_b

Logical parameters satisfy *monotonicity conditions*: if the set of processes associated to a given parameter is included in the set of processes associated to another parameter, then the value of the former parameter is at most the value of the latter. In other words, the more the processes of a given variable are present, the more the target value toward which the variable evolves is great. In that sense, *the presence of a process favors the increase of its targets*. For the running example, the monotonicity conditions are:

K_a $\leq$ K_a+P	K_b $\leq$ K_b+P
K_a $\leq$ K_a+Q	
K_a+P $\leq$ K_a+P+Q	
K_a+Q $\leq$ K_a+P+Q	

A *parameterization* is the assignment of a value to each logical parameter. A parameterization satisfying the monotonicity conditions is a *monotonous parameterization*. The following parameterization is a monotonous parameterization:

```
K_a=0              K_b=0
K_a+P=2            K_b+P=1
K_a+Q=1
K_a+P+Q=2
```

From this parameterization and the previous table, we obtain:

a	b	Target of a	Target of b
0	0	0	0
0	1	0	0
1	0	2	1
1	1	0	0
2	0	2	1
2	1	1	0

This table gives the target level of each variable at each state. This information is sufficient to explicitly describe a possible dynamics for the network under the form of a state transition graph. As in the multivalued logical formalism of Thomas, this state transition graph is constructed by assuming that variables evolve asynchronously and by unit:

> During a transition, there is a unique variable that evolves, and it evolves by unit (+1/-1) in the direction of its target level. In addition, for each variable that is not equal to its target level, there exists a transition allowing this variable to evolve.

For the running example, this updating rule leads to the following asynchronous state transition graph:

$$\begin{tikzcd}
0\,1 \arrow[d] & 1\,1 \arrow[l] \arrow[d, shift right] & 2\,1 \arrow[l] \arrow[d, shift right] \\
0\,0 \arrow[loop below] & 1\,0 \arrow[r] \arrow[u, shift right] & 2\,0 \arrow[u, shift right]
\end{tikzcd}$$

For instance, at state `1 0`, one can see that both variables are less than their target levels, which are `2` and `1` respectively. So we have two transitions: a transition from `1 0` to `2 0` that allows the variable `a` to increase, and a transition from `1 0` to `1 1` that allows the variable `b` to increase. Note that at state `0 0` both variables are equals to their respective target levels: `0 0` is a stable state of the network, and (by convention), there exists a unique transition starting from `0 0`, which goes to `0 0` itself.

Summary. The evolution of the network is driven by logical parameters, and each monotonous parameterizations describes a possible dynamics for the network under the form of an asynchronous state transition graph.

1.2 Constraints on parameters (section PARA)

By default, a logical parameter associated with a given variable takes a value in the domain of this variable. The (optional) section `PARA`, gives restrictions on the size of the intervals in which logical parameters take values. For instance, with

```
PARA
K_a+P = 1 2 ;
K_b = 0 ;
K_b+P = 1 ;
```

we say that the parameter `K_a+P` take a value inside `{1,2}` (instead of `{0,1,2}`) and that the values of the parameters `K_b` and `K_b+P` are `0` and `1` respectively.

1.3 CTL formula (section CTL)

The last section of the input file is the section `CTL`, which contains a CTL formula. This formula expresses a dynamical property that will be tested on all the asynchronous state transition graphs resulting from a monotonous parameterizations of the network (and satisfying the constraints of the section `PARA`). For instance, with

```
CTL
((a>0)->EG(a>0)) & ((a=0)->AG(a=0))
```

the considered property is "if `a>0`, then there exists an evolution of the network for which `a>0` is always true, and if `a=0`, then for all the possible evolutions, `a=0` is always true". Note that this property is satisfied by the asynchronous state graph given as example in section 1.1.2.

The syntax of CTL formula is the following:

```
id := variable_name | integer

atom := id "<" id  | id ">" id  |
         id "<=" id | id ">=" id | id "=" id

ctl := atom |
       "!" ctl |          # Logical NOT
```

```
    ctl "&" ctl |       # Logical AND
    ctl "|" ctl |       # Logical OR
    ctl "->" ctl |      # Logical implication
    "EX" ctl |          # Exists neXt state
    "AX" ctl |          # For All neXt state
    "EF" ctl |          # Exists Finally
    "AF" ctl |          # For All Finally
    "EG" ctl |          # Exists Globally
    "AG" ctl |          # For All Globally
    "E" ctl "U" ctl |   # Exists Until
    "A" ctl "U" ctl |   # For All Until
```

Define the semantic consists in defining the satisfactory relation between a state graph and a CTL formula. We say that the state graph satisfies a formula when every state of the graph satisfies the formula; the satisfactory relation between a state and a formula being defined as follows. First, if the formula is simply a propositional formula (that is, does not contain temporal operators such as `EX` or `AG`), then the satisfactory relation is usual. Second, a state `s` satisfies a formula of the form:

> `EX ctl` if there exists a transition starting from `s` that leads to a state satisfying the formula `ctl`.
>
> `AF ctl` if every transition starting from `s` leads to a state satisfying the formula `ctl`.
>
> `EF ctl` if there exists a path starting from `s` that leads to a state satisfying the formula `ctl`.
>
> `AF ctl` if every path starting from `s` leads to a state satisfying the formula `ctl`.
>
> `EG ctl` if there exists an infinite path starting from `s` that contains only states satisfying the formula `ctl`.
>
> `AG ctl` if every infinite path starting from `s` contains only states satisfying the formula `ctl`.
>
> `E ctl U ctl'` if there exists a path starting from `s` that contains a state `s'` satisfying the formula `ctl'` and such that every state before `s'` in the path satisfies the formula `ctl`.
>
> `A ctl U ctl'` if every path starting from `s` contains a state `s'` satisfying the formula `ctl'` and such that every state before `s'` in the path satisfies the formula `ctl`.

2 Output file

The output file is obtained with the command `smbionet input_file`. It contains all the monotonous parameterizations of the network satisfying the constraints of section `PARA` and leading to an asynchronous state transition graph satisfying the given CTL formula. For instance, with the input file

```
VAR
a = 0 2 ;
b = 0 1 ;

REG
P [a>0 & b<1]=> a b ;
Q [a=2]=> a ;

PARA
K_a+P = 1 2 ;
K_b = 0 ;
K_b+P = 1 ;

CTL
((a>0)->EG(a>0)) & ((a=0)->AG(a=0))
```

the output file (named `input_file.out` by default) is the following:

```
VAR

a = 0 2 ;
b = 0 1 ;

REG

P [((a>0)&(b<1))]=> a b ;
Q [(a=2)]=> a ;

PARA

# Parameters for a

K_a = 0 2 ;
K_a+P = 1 2 ;
K_a+Q = 0 2 ;
K_a+P+Q = 0 2 ;
```

```
# Parameters for b

K_b = 0 ;
K_b+P = 1 ;

CTL

((a>0)->EG(a>0))&((a=0)->AG(a=0))

# MODEL 1

# K_a = 0
# K_a+P = 1
# K_a+Q = 0 1
# K_a+P+Q = 0 1

# K_b = 0
# K_b+P = 1

# MODEL 2

# K_a = 0
# K_a+P = 1
# K_a+Q = 0 1
# K_a+P+Q = 2

# K_b = 0
# K_b+P = 1

# MODEL 3

# K_a = 0
# K_a+P = 2
# K_a+Q = 0 1
# K_a+P+Q = 2

# K_b = 0
# K_b+P = 1

# MODEL 4

# K_a = 0
# K_a+P = 1
# K_a+Q = 2
# K_a+P+Q = 2

# K_b = 0
# K_b+P = 1

# MODEL 5

# K_a = 0
# K_a+P = 2
# K_a+Q = 2
# K_a+P+Q = 2

# K_b = 0
# K_b+P = 1

# SELECTED MODELS / CHECKED MODELS = 5 / 10 (126ms)
```

The output file begins with a summary of the specifications (following the syntax of the input file; actually, the output file can be taken as input file). Then it gives (in comment (`#`)) all the monotonous parameterizations that are consistent with the specifications.

To be more precise, first note that *several parameterizations can lead to the same asynchronous state graph*. These parameterizations are then *equivalent* from a dynamical point of view. `SMBioNet` proceeds as follows: it enumerates all the classes containing a monotonous parameterization satisfying the constraints of the section `PARA`. Then, for each class, the corresponding asynchronous state graph is symbolically described using the `NuSMV` language, and confronted to the CTL formula with the `NuSMV` symbolic model checker. If the state graph satisfies the formula, then the corresponding class is given as output.

For the running example, `10` classes (or `MODELS`) have been enumerated, and `5` classes have been selected. For instance `MODEL 3` corresponds to a class containing two parameterizations:

```
K_a=0                  K_a=0
K_a+P=2                K_a+P=2
K_a+Q=1                K_a+Q=0
K_a+P+Q=2     and      K_a+P+Q=2
K_b=0                  K_b=0
K_b+P=1                K_b+P=1
```

The left parameterization is the one given in example in Section 1, and both parameterizations lead to the asynchronous state graph given in Section 1.

3 Options

The syntax of the `smbionet` command is the following:

```
smbionet [-v int]
         [-comp]
         [-dynamic]
         [-o output_file]
         input_file
```

The option `-v int` increases the quantity of information on the network given in the output (the more `int` is great, the more the quantity of information is great). The option `-comp` stops the execution of `SMBioNet` after the reading of the `input_file`. This can be useful, for instance, to display the logical parameters associated with the network. The option `-dynamic` allows `NuSMV` to perform a dynamic reordering of variables. This often increases significantly the speed of the verification when the number of variables is large. Finally, the option `-o output_file` gives the name `output_file` to the output file!

References

[1] R. Thomas and M Kaufman. Multistationarity, the basis of cell differentiation and memory II, *Chaos* 11:170-195, 2001.

[2] G. Bernot, J.-P. Comet, A. Richard and J. Guespin. Application of formal methods to biological regulatory networks: extending Thomas' asynchronous logical approach with temporal logic, *Journal of Theoretical Biology* 229(3): 339-347, 2004.

[3] Z. Khalis, J.-P. Comet, A. Richard and G. Bernot. The SMBioNet method for discovering models of gene regulatory networks. *Genes, Genomes and Genomics* 3(1):15-22, 2009.

[4] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2000.

[5] `http://nusmv.irst.itc.it/`