Exhaustive dynamic analysis of reachability and stable states of biological regulatory networks modeled in Process Hitting using Answer Set Programming

Emna Ben Abdallah
[1]*LUNAM Universit, École Centrale de Nantes,*
IRCCyN UMR CNRS 6597
(Institut de Recherche en Communications et Cybernétique de Nantes),
1 rue de la No, 44321 Nantes, France.
(*e-mail:* emna.ben-abdallah@irccyn.ec-nantes.fr),

Maxime Folschette[1,2]
[2]*School of Electrical Engineering and Computer Science,*
University of Kassel, Germany
(*e-mail:* maxime.folschette@uni-kassel.de),

Olivier Roux[1]
LUNAM Universit, École Centrale de Nantes, IRCCyN
(*e-mail:* olivier.roux@irccyn.ec-nantes.fr),

Morgan Magnin[1,3]
[3]*National Institute of Informatics,*
2-1-2, Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan.
(*e-mail:* morgan.magnin@irccyn.ec-nantes.fr),

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

In this paper, we propose a new logical approach to perform model-checking on a restriction of synchronous automata networks, namely Process Hitting. The Process Hitting framework is notably suitable, but not limited, to model and analyze efficiently biological regulatory networks. The originality of our model-checking approach relies in the use of Answer Set Programming to consider exhaustive enumeration of all possible solutions of a property with respect to a given execution length. Our work here focuses on state reachability properties (which are equivalent to the CTL EF operator with paths limited to a given length) on the one hand, on the identification of fixed points on the other hand. The merits of our methods is illustrated by applying them to biological examples of various sizes and comparing the results with some existing approaches. It turns out that our approach succeeds in processing large models with a high number of components and interactions.

KEYWORDS:Process Hitting, Answer Set Programming, stable states, fixed points, reachability

1 Introduction

For more than a decade, model-checking and SAT-solvers developments have been strongly connected (Biere et al. 1999). Among different reasons, the main ones are related to some of the drawbacks of Binary Decision Diagrams (BDD) -based symbolic approaches, that are:

- BDD-based structures require some canonical form to be given so that a total order can be defined;
- They often become too large;
- Variable ordering is crucial to the performances of the approaches.

Behind the use of SAT, respectively ASP, to perform model-checking, there is the intuitive idea that the existence of a trace satisfying a given property expressed in LTL (Biere et al. 1999) or CTL can be reduced to the satisfiability of an equivalent logic formula, respectively a logic program. But first it should be reduced to universal properties (Penczek et al. 2002), then extended to other properties. Although there have been some works in the field of unbounded model-checking (McMillan 2002), it generally requires to consider bounded traces, which gave birth to the concept of Bounded Model-Checking (BMC). Its principle is following: given a discrete model M (which, equipped with a semantics, leads to a transition system), a property φ and $k \in \mathbb{N}$, does M allow a counterexample to φ of k (or fewer) transitions?

Bounded model-checking has been the subject of numerous researches. On the one hand, it also suffers from some limitations. First, it leads to incomplete analysis: if the SAT solver proves that the problem given as input is unsatisfiable for a length k, it only proves there are no counterexamples of length k. Finding bounds on the length is difficult and worst case is exponential. On the other hand, SAT is a viable alternative to BDD-based symbolic model-checking. It is also an efficient approach for debugging, i.e., quickly find counterexamples of minimal length.

Recently, the rise of ASP has also given birth to some innovative approaches, taking profit of the performances of ASP to process efficient model-checking algorithms. In (Rocca et al. 2013), the authors proposed an approach to perform LTL and CTL model-checking through ASP, with the goal to infer and learn (synchronous) Boolean networks. This method has been refined in the context of (asynchronous) Thomas' biological regulatory networks in (Rocca et al. 2014).

As regulatory phenomena play a crucial role in biological systems, they need to be studied accurately. Biological Regulatory Networks (BRNs) consist in sets of either positive or negative mutual effects between the components. With the purpose of analyzing these systems, they are often modeled as graphs which makes it possible to determine the possible evolutions of all the interacting components of the system. Thus, in order to address the formal checking of dynamical properties within very large BRNs, we recently introduced a new formalism, named the Process Hitting (PH) (Paulevé et al. 2011), to model concurrent systems having components with a few qualitative levels. A PH describes, in an atomic manner, the possible evolutions of a process (representing one component at one level) triggered by the hit of at most one other process in the system. This particular structure makes the formal analysis of BRNs with hundreds of components tractable by using abstract methods computing approximations of the dynamics (Paulevé et al. 2012). Furthermore, PH is suitable, according to the precision of the available information, to model BRNs with different levels of abstraction by capturing the most general dynamics.

The objectives of the work presented in this paper are the following. Firstly, we show that starting from a PH model, it is possible to find all possible fixed points (also called stable states) which are specific attractors with only one stateFor this, we perform an exhaustive search of the possible states and then check which ones are fixed points. The second phase of our work consists in computing the dynamics by determining, from a given initial state, the possible next states of the PH model. This also permits to check if a goal is reachable for a given component or set of components in the system. The results are ensured to respect strictly the PH dynamics.

The particularity of our contribution relies in the use of Answer Set Programming (ASP)to compute the result of these searches. This declarative programming framework has been proven efficient to tackle models with a large number of components and parameters. Our aim here is to assess its potential w.r.t. the computation of some dynamical properties of PH models. In this paper, we show that ASP turns out to be effective for these enumerative searches, which justifies its use. In some cases, the computed dynamics can also be bounded by a given number of evolution steps.

2 Preliminary definitions

2.1 Process Hitting

1 introduces the Process Hitting (PH) (Paulevé et al. 2011) which allows to model a finite number of local levels, called processes, grouped into a finite set of components, called sorts. A process is noted a_i, where a is the sort's name, and i is the process identifier within sort a. At any time, exactly one process of each sort is active, and the set of active processes is called a state.

The concurrent interactions between processes are defined by a set of actions. Each action is responsible for the replacement of one process by another of the same sort conditioned by the presence of at most one other process in the current state. An action is denoted by $a_i \rightarrow b_j \Rsh b_k$, which is read as a_i hits b_j to make it bounce to b_k, where a_i, b_j, b_k are processes of sorts a and b, called respectively hitter, target and bounce of the action. We also call a self-hit any action whose hitter and target sorts are the same, that is, of the form: $a_i \rightarrow a_i \Rsh a_k$.

The PH is therefore a restriction of synchronous automata, where each transition changes the local state of exactly one automaton, and is triggered by the local states of at most two distinct automata. This restriction in the form of the actions was chosen to permit the development of efficient static analysis methods based on abstract interpretation (Paulevé et al. 2012).

Definition 1 (*Process Hitting*)
A Process Hitting is a triple $(\Sigma, \mathcal{L}, \mathcal{H})$ where:

- $\Sigma = \{a, b, \dots\}$ is the finite set of sorts;
- $\mathcal{L} = \prod_{a \in \Sigma} \mathcal{L}_a$ is the set of states where $\mathcal{L}_a = \{a_0, \dots, a_{l_a}\}$ is the finite set of processes of sort $a \in \Sigma$ and l_a is a positive integer, with $a \neq b \Rightarrow \mathcal{L}_a \cap \mathcal{L}_b = \emptyset$;
- $\mathcal{H} = \{a_i \rightarrow b_j \Rsh b_k \in \mathcal{L}_a \times \mathcal{L}_b^2 \mid (a,b) \in \Sigma^2 \wedge b_j \neq b_k \wedge a = b \Rightarrow a_i = b_j\}$ is the finite set of actions.

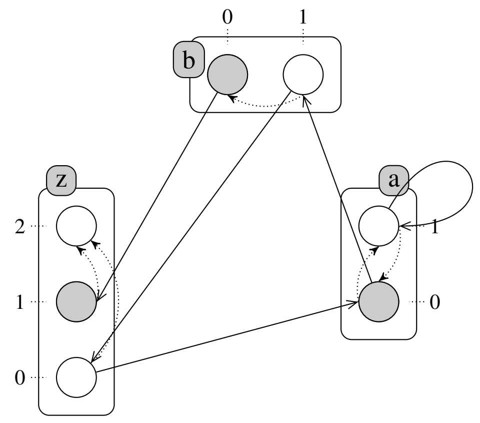

Figure 1. A PH model example with three sorts: a, b and z (a is either at level 0 or 1, b at either level 0 or 1 and z at either level 0, 1 or 2). Boxes represent the sorts (network components), circles represent the processes (component levels), and the 5 actions that model the dynamic behavior are depicted by pairs of arrows in solid and dotted lines. The grayed processes stand for the possible initial state: $\langle a_1, b_0, z_1 \rangle$.

Example 1
1 represents a $\mathcal{PH}$ $(\Sigma, \mathcal{L}, \mathcal{H})$ with three sorts ($\Sigma = \{a, b, c\}$) and: $\mathcal{L}_a = \{a_0, a_1\}$, $\mathcal{L}_b = \{b_0, b_1\}$, $\mathcal{L}_z = \{z_0, z_1, z_2\}$.

A state of the networks is a set of active processes containing a single process of each sort. The active process of a given sort $a \in \Sigma$ in a state $s \in \mathcal{L}$ is noted $s[a]$. For any given process a_i we also note: $a_i \in s$ if and only if $s[a] = a_i$.

Definition 2 (*Playable action*)
Let $\mathcal{PH} = (\Sigma, \mathcal{L}, \mathcal{H})$ be a Process Hitting and $s \in \mathcal{L}$ a state of PH. We say that the action $h = a_i \rightarrow b_j \Rsh b_k \in \mathcal{H}$ is playable in state s if and only if $a_i \in s$ and $b_j \in s$ (i.e., $s[a] = a_i$ and $s[b] = b_j$). The resulting state after playing h in s is called a successor of s and is denoted by $(s \cdot h)$, where $(s \cdot h)[b] = b_k$ and $\forall c \in \Sigma, c \neq b \Rightarrow (s \cdot h)[c] = s[c]$.

The PH was chosen for several reasons. First, it is a general framework that, although it was mainly used for biological networks, allows to represent any kind of dynamical model, and converters to several other representations are available (see section 5). Although an efficient dynamical analysis already exists for this framework, based on an approximation of the dynamics, it is interesting to identify its limits and compare them to the approached we present later in this paper. Finally, the particular form of the actions in a PH model allow to easily represent them in ASP, with one fact per action, as described in the next section. Other representations may have required supplementary complexity; for instance, a labeling would be required if actions could be triggered by a variable number of processes.

Dynamical properties

The study of the dynamics of biological networks was the focus of many works, explaining the diversity of network modelings and the different methods developed in order to check dynamic properties. In this paper we focus on 2 main properties: the stable states and the reachability, which were already formalized and tackled by other methods in (Paulevé et al. 2011; Paulevé et al. 2012). In the following, we consider a PH model $(\Sigma, \mathcal{L}, \mathcal{H})$, and we formally define these properties and explain how they could be verified on such a network.

The notion of fixed point, also called stable state, is given in definition 3. A fixed point is a state which has no successor. Such states have a particular interest as they denote states in which the model stays indefinitely, and the existence of several of these states denotes a switch in the dynamics (Wuensche 1998).

Definition 3 (*Fixed point*)
A state $s \in \mathcal{L}$ is called a fixed point (or equivalently stable state) if and only if it has no successors. In other words, s is a fixed point if an only if no action is playable in this state:

$$\forall a_i \to b_j \Rsh b_k \in \mathcal{H}, a_i \notin s \vee b_j \notin s \ \ .$$

A finer and more interesting dynamical property consists in the notion of reachability property. Such a property, defined in definition 4, states that starting from a given initial state, it is possible to reach a given goal, that is, a state that contains a process or a set of processes. Checking such a dynamical property is considered difficult as in usual model-checking techniques, it is required to build (a part of) the state graph, which has an exponential complexity.

In the following, if $s \in \mathcal{L}$ is a state, we call scenario in s any sequence of actions that are successively playable in s. We also note $\mathbf{Sce}(s)$ the set of all scenarios in s. Moreover, we denote by $\mathbf{Proc} = \bigcup_{a \in \Sigma} \mathcal{L}_a$ the set of all process in $\mathcal{PH}$.

Definition 4 (*Reachability property*)
If $s \in \mathcal{L}$ is a state and $A \subseteq \mathbf{Proc}$ is a set of processes, we denote by $\mathcal{P}(s, A)$ the following reachability property:

$$\exists ?\delta \in \mathbf{Sce}(s), \forall a_i \in A, (s \cdot \delta)[a] = a_i \ \ .$$

The rest of the paper focuses on the resolution of the previous issues with the use of ASP. The enumeration of all fixed points of a PH model will be tackled in section 3 and the verification of a reachability property will be the subject of section 4.

3 Fixed point enumeration

The study of fixed points (and, more generally, basins of attraction) provides an important understanding of the different behaviors of a Biological Regulatory Network (BRN) (Wuensche 1998). Indeed, a system will always eventually end in a basin of attraction, and this may depend on biological switch or other complex phenomena. A fixed point is a state of the BRN in which it is not possible any more to have new changes; in other words, it is a basin of attraction that is composed of only one state.

In the following, we consider a Process Hitting $\mathcal{PH} = (\Sigma, \mathcal{L}, \mathcal{H})$. It has been shown that a state $s \in L$ is a fixed point (or stable state) of the Process Hitting model if and only if s has no next state, i.e., if there is no playable action in this state (Paulevé et al. 2011). Therefore, a stable state in a Process Hitting network is a state so that every process does not hit or is not hit by another process in the same state. We note that given this result, processes involved in a self-hit (an action whose hitter and target processes are the same) cannot be part of a stable state.

3.1 Process Hitting translation in ASP

Before analyzing the dynamics of the network, we first need to translate the concerned PH network in ASP [1].

To do this we use the following self-describing predicates: `"sort"` to define sorts, `"process"` for the processes and `"action"` for the network actions. We will see in example 2 how a PH network is defined with these predicates.

Example 2 (*Representation of a PH network in ASP*)
The representation of the PH network of figure 1 in ASP is the following:

```
1 process("a", 0..1). process("b", 0..1). process("z", 0..2).
2 sort(X) ← process(X,I)
3 action("a",0,"b",1,0). action("a",1,"a",1,0). action("b",1,"z",0,2).
4 action("b",0,"z",1,2). action("z",0,"a",0,1).
```

In line 1 we create the list of processes corresponding to each sort, for example the sort `"z"` has 3 processes numbered from `0` to `2`; this specific predicate will in fact expand into the three following predicates: `process("z", 0)`,`process("z", 1)`,`process("z", 2)`. Line 2 enumerates every sort of the network from the previous information. Finally, all the actions of the network are defined in lines 3 and 4; for example, the first predicate `action("a",0,"b",1,0)` represents the action $a_0 \rightarrow b_1 \Rsh b_0$.

3.2 Search of fixed points

The enumeration of fixed points requires to translate the definition of a stable state in a set of ASP rules. The first step consists of eliminating all processes involved in a self-hit; the others are recorded by the predicate `shownProcess`:

```
5 hiddenProcess(A,I) ← action(A,I,B,J,K), A=B, process(A,I),process(B,J),
6                       process(B,K).
7 shownProcess(A,I) ← not hiddenProcess(A,I), process(A,I).
```

Then, we have to browse all remaining processes of this graph in order to generate all possible states, that is, all possible combinations of processes by choosing one process from each sort:

```
8 1 { selectedProcess(A,I) : showProcess(A,I) } 1 ← sort(A).
```

The previous line creates as many potential answer sets as there are possible states to take into account. Finally, the last step consists of filtering any state that is not a fixed point, or, in other words, eliminating all candidate answer sets in which an action could be played. For this, we use a constraint: any solution that satisfy the body of this constraint will be removed from the answer set. Regarding our problem, a state is eliminated if there exists an action linking two selected processes:

```
9 ← hit(A,I,B,J), selectedProcess(A, I), selectedProcess(B, J), A != B.
```

[1] All programs, including this translation and the methods describes in the following, are available online at: `https://github.com/EmnaBenAbdallah/verification-of-dynamical-properties_PH`

Example 3 (*Fixed points enumeration*)
The PH model of 1 contains 3 sorts: a and b have 2 processes and z has 3; therefore, the whole model has $2 * 2 * 3 = 12$ states (whether they can be reached or not from a given initial state). We can check that this model contains only one fixed point: $\langle b_0, z_2, a_0 \rangle$. Indeed, there is no action between each two of the processes contained is this state so no execution is possible from this state. In this example, no other state verifies this property.

If we execute the ASP program detailed below, alongside with the description of the PH model given in example 2, we obtain one answer set, which matches the expected result:

```
Answer 1 : fixProcess(a, 0), fixProcess(b, 0), fixProcess(z, 2)
```

4 Dynamical analysis

In this section, we present at first how to determine the possible behaviour in a PH model after a finite number of steps with an ASP program. Then we tackle the reachability question: are there scenarios from a given initial state that allow to reach a given goal?

4.1 Future states identification

In the previous section, enumerating the fixed points did not require to encode the full dynamics of PH, but only a condition. In this section, we thus implement the dynamics of the PH into ASP, which will then permit to search for the paths allowing to reach the goals.

Firstly, we focus on the evolutions of models in a limited number of steps. We therefore define the predicate `time(0..n)` which sets the number of steps we want to play. The value of `n` can be arbitrarily chosen; for example, `time(0..10)` will be used to compute the 11 first steps, including the initial state. In order to specify such an initial state, we add several facts of the following form to list the processes included in this state:

```
11  init(activeProcess("a",0)).
```

where `"a"` is the name of the sort and `"0"` the index of the active process. The dynamics of a network is described by its actions, identifying the future states requires to first identify the playable actions for each state. We remind that an action is playable in a state when both its hitter and target are active in this state (see Definition 2). Therefore, we define an ASP predicate `playableAction(A,I,B,J,K,T)` that is true when the processes $\mathtt{A}_\mathtt{I}$ and $\mathtt{B}_\mathtt{J}$ are active at step `T`. It is also needed to enforce the strictly asynchronous dynamic which state that exactly one process can change between two steps. We thus represent the change of the active process of a sort by the predicate `activeFromTo(B,J,K,T)` which means that in sort `B`, the active process changes from index $\mathtt{B}_\mathtt{J}$ to $\mathtt{B}_\mathtt{K}$ between steps `T` and `T+1`. The cardinality rule of line 12 - 14 creates the set of the predicates as many as there are possible evolutions from the current step, thus recreating all possible branchings in the possible evolutions of the model so many answer sets. This allows to filter all scenarios where two actions have been played between two steps, by using the constraint of line 15. Thus, the remaining scenarios of the answer sets all strictly follow the asynchronous dynamics of the PH.

```
12  {activeFromTo(B,J,K,T)} ←  playableAction(A,I,B,J,K,T),
13           instate(activeProcess(A,I),T), instate(activeProcess(B,J),T),
14           J!=K, time(T).
15  ← 2 {activeFromTo(B,J,K,T)}, time(T).
```

Finally, the active processes at step `T+1`, that represent the next state depending on the chosen bounce, can be computed by the following rules:

```
16 instate(activeProcess(B,K),T+1) ← activeFromTo(B,J,K,T), time(T).
17 instate(activeProcess(A,I),T+1) ← instate(activeProcess(A,I),T),
18          activeFromTo(B,J,K,T), A!=B, time(T).
```

In other words, the state of step `T+1` contains one new active process resulting from the predicate `activeFromTo` (line 16) as well as all the unchanged processes that correspond to the other sorts (line 17 and 18).

The overall result of the pieces of program presented in this subsection is an answer set containing one answer for each possible evolution in `n` time steps, and starting from a given initial state.

4.2 Reachability verification

In this section, we focus on the reachability of a set of processes which corresponds to the reachability property (see definition 4): Is it possible, starting from a given initial state, to play a number of actions so that a set of given processes are active in the resulting state? We now want to adapt the code of the dynamics computation of previous section in order to resolve this reachability problem. For this, we first use a predicate to list the processes we want to reach, called `goal`, and we add as many rules of the following form as there are objective processes:

```
19 goal(activeProcess("a",1)).
```

Then, the literal `satisfiable(F, T)` checks if a given process `F` of the goal is contained in the state of step `T`, as defined in the rule of line 20. Else the answer will be eliminated by a constraint (not detailed here) which verifies if all processes of the goal are satisfied.

```
20 satisfiable(F, T) ← goal(F), instate(F, T).
```

However, the limitation of the method above is that the user has to decide upstream the number of computed steps that should be sufficient. It is a main disadvantage because a search in N steps will find no solution if the shortest path to the goal requires K steps with $K > N$. It may also needlessly lengthen the resolution if the shortest path requires n steps with $n << N$. One solution is then to use an incremental computation mode, which is especially tackled by the incremental solver ICLINGO (Gebser et al. 2008). The syntax of ICLINGO separates the program in 3 parts. The `#base` part contains only non-incremental elements and is thus used to declare general rules that do not depend on the time step (such as the data related to the model). The body iteration is written in the `#cumulative` and `#volatile` parts which are computed at each incremental step, the first part comprising rules depending on the time step, and the second containing a constraint that stops the iteration when needed. The step number is not given by a variable but by a constant placeholder called `t` in the following.

When using this new syntax, the obtained program is almost identical to what was presented before, except that step numbers `T` are replaced by the constant placeholder `t`. In each step `t`, the program computes the playable actions `playableAction(A,I,B,J,K,t)`, the possible bounces `activeFromTo(B,J,K,t-1)` and the new states `instate(activeProcess(A,I),t+1)` in the `#cumulative` part the same way than previously, but only for the current step. The solver then compares its current answer sets with the t-dependent constraint given in the #volatile part. Regarding our implementation, this constraint is given in line 22 and simply states that all goals have to be met. If this constraint invalidates all current answer sets, the computation continues in the next iteration in order to reach a valid answer set. As soon as some answer sets meet the constraints, they are returned and the computation stops.

```
21  notSatisfiable(t) ←  goal(F), not instate(F,t).
22  ← notSatisfiable(t).
```

5 Comparative performance analysis

In this section, we show the effectiveness of our approach on some examples, and compare it to other existing approaches. All computations except one (ASP-THOMAS) were performed on a Pentium V, 3.2 GHz with 4 GB RAM.

5.1 Evaluation

To assess the efficiency of our new approach, we position ourselves with respect to existing methods dealing with different biological models. We have chosen the following tools, that are detailed below: GINSIM[2] (Gene Interaction Network Simulation) (Gonzalez et al. 2006; Naldi et al. 2009; Naldi et al. 2007), LIBDDD[3] (Library of Data Decision Diagrams) (Thierry-Mieg et al. 2009; Colange et al. 2013), PINT[4] (Paulevé et al. 2012) and the method for CTL model-checking proposed by Rocca *et al.* in (Rocca et al. 2014), which was developed also in ASP but for states transitions networks. Each method uses a specific kind of representation[5]: Thomas models (a particular kind of logical regulatory networks) for GINSIM, instantiable transition systems for LIBDDD, state transition networks for the method of Rocca *et al.* and Process Hitting (PH) for PINT as well as for our method.

For this comparative study, we focus on biological network of different sizes: a tadpole tail resorption (TTR) model with 12 biological components (Khalis et al. 2009), an ERBB receptor-regulated G1/S transition (ERBB) model with 20 components (Samaga et al. 2009) and a T-cell receptor (TCR) signaling network of 40 components (Klamt et al. 2006). These models were chosen to be of different sizes: from small (12 components) to large (40 components). We note however that the considered PH models may contain more sorts than the original number of biological components, due to the use of cooperative sorts, which allow to model Boolean gates but do not necessarily have a biological meaning. The different model representations that are required for performing these benchmarks have been obtained by translations from the PH ensuring the conservation of the dynamical properties. All results alongside with more detailed specifications of the models are given in table 2. The methods and the results provided by each of them are detailed in the following. The overall results show that our method is efficient in computing reachability from a given initial state; furthermore, it sometimes provides more information than the other existing ones.

[2] GINSIM http://ginsim.org/
[3] LIBDDD http://move.lip6.fr/software/DDD/
[4] PINT http://loicpauleve.name/pint/
[5] When available, we used the converters included into PINT for these translations.

Models		PH representation			Fixed points enumeration	
Name	Components	Sorts	Processes	States	computation time	Nbr of results
TTR	8	12	42	2^{19}	0.004s	0
ERBB	20	42	152	2^{70}	0.017s	3
TCR	40	54	156	2^{73}	0.021s	1

Table 1. *Description of the models used in our tests and results of our fixed point enumeration. Each model is referred to by its short name, where TTR stands for the tadpole tail resorption modelERBB for the receptor-regulated G1/S transition of the same name- and TCR for the T-cell receptor signaling network For each of them, this table gives the number of biological components in the original representation, and the number of sorts, the number of processes and the number of states in the PH model. Finally, the last column gives the computation time for the enumeration of all fixed points and the number of results returned.*

Experiments			Results				
	Model	Target	ASP-THOMAS	PINT	LIBDDD	GINSIM	ASP-PH
#1	TTR	full state	0m2.30s	0m0.97s	0m1.15s	0m2.05s	0m1.90s
#2	ERBB	full state	0m2.45s	out	1m55.38s	2m31.64s	0m11.84s
#3	ERBB	partial state	0m2.61s	0m0.03s	1m54.96s	-	0m5.02s
#4	TCR	full state	0m7.61s	Inconc	out	out	6m27.93s
#5	TCR	partial state	0m2.12s	0m0.02s	out	-	1m35.08s

Table 2. *Compared performances of several methods to compute reachability analyses: The method of Rocca et al. (denoted by* ASP-THOMAS*),* PINT*,* LIBDDD*,* GINSIM *and our new method presented in this paper, called* ASP-PH. *For each test, this table gives the short name of the considered model, as given in table 1, the type of goal (either a whole state or a sub-state) and the computation time of the different methods used for the tests, where out marks an execution taking too much time or memory, - indicates that is not possible to do the test, and Inconc states that the method terminates without a response.*

- **ASP-Thomas**[6] offers the possibility to model-check CTL properties of Thomas networks. There is no automatic way that allows the modelisation of Thomas networks in ASP. That is why it is necessary to study the entire network than to present it in ASP. But this modelisation takes so much time to be done regarding the complex representation of Thomas networks. Comparing with our method, we use the PH model which its actions are more easy to be presented in ASP, one fact per action. In addition, the method "ASP-Thomas" requires to provide a maximum number of steps for which the dynamics will be computed, which mays be difficult to be predicted. However it is clear that this approach gives a very quick result when compared with others. Indeed it also show that ASP is a good choice to run the dynamics of a model and check reachability properties. Furthermore, this method is able to check any kind

[6] These tests have been performed on another computer: dual core , 2.13 GHZ with 1.8 GB RAM. The authors wish to thank Laurent Trilling for his help.

of CTL formula (and not only the EF form that we focused on in this paper). (see the discussion in section 5.2).

- **GINSIM** is a software for the edition, simulation and analysis of gene interaction networks. It allows to compute all reachable stable states from a given initial state instantly; however, it is not possible to compute all stable states independently from the initial state. Regarding the reachability problem, GINSIM only allows to check the reachability of full states, because its approach consists in computing all the state-transition graph and then search for a path between the two given states. Therefore, it was not possible to perform reachability checks on partial states (experiments #3 & #5). Small state-transition graphs can also be displayed by this tool.
- **LIBDDD** is a library for symbolic model-checking of CTL & LTL properties. It can thus especially be used to check reachability properties; however, as opposed to our method, it does not output an execution path solving this reachability. In addition, it relies on the construction of the state-transition graph which is then stored under the form of a binary decision diagram for a more efficient analysis. This computation explains why LIBDDD takes more time to respond, and gets out of memory in about 12 minutes for the biggest example which contains 2^{73} states (experiments #4 & #5). Finally, LIBDDD is not able to compute the stable states of a network.
- **PINT** is a library gathering tools and converters related to the PH. It should be noted that PINT contains the only reachability analysis developed so far natively for the Process Hitting, before the method proposed in this paper. It consists in an approximation that avoids to compute the state-transition graph; it is thus ensured to be really efficient, which explains the fastest results, but at the cost of possibly terminating without being conclusive. However, it is not designed for goals consisting of many processes, which are more likely to trigger an inconclusive response (such as for experiment #4), or an exponential research in sub-solutions (such as for experiment #2). This explains the high computation times for some tests in table 2. Moreover, in the case of a positive answer, it currently does not return the execution of the path achieving the desired reachability, but only outputs its conclusion.

5.2 Strengths and limitations of our method

In the previous sections, we developed new methods in ASP to check dynamical properties, namely identifying stable states and finding all possible paths to reach a given goal. Compared to some other methods describes above (GINSIM, LIBDDD and the method ASP-THOMAS) our method is relatively faster and also permits to study larger networks (up to 2^{73} states in our tests). We study the networks which are modeled in Process Hitting. This new fomalisim for network modelisation is a restriction of synchronous automata so it allows to represent any kind of dynamical model. Moreover it is easy to implement the dynamics of the PH into ASP. However, our ASP program may still be non-conclusive in the cases where the given goal is not reachable and the model contains loops in its dynamics (which happens in almost every biological model). In this case, the program will compute all infinite paths in these loops, and never reach a goal or a fixed point. It is still possible, however, to limit the number of iterations to an arbitrary maximum which will be eventually reached in the case of an endless loop. This is possible with the option `"--imax=n"` of ICLINGO, where `n` is the maximum number of steps.

Several values can be given to this parameter. For example, the total number of states is an obvious maximum, as it will never be exceeded by a minimum path, but it is too hight to be very interesting. The total number of sorts is a more interesting value, under the hypothesis that each one will change its active process at most once, which is often the case for Boolean networks; or, with a similar reasoning, the total number of processes can be chosen. In these cases, however, a termination with no solution cannot be considered as a formal negative answer, unless one can prove that the chosen value `n` is bigger than the longest possible path in the state-transition graph. Given our implementation, if the step `n` is reached, the computation stops and there is no path to be displayed because there is no path which verifies the goals after `n` changes. Indeed this permits to get a final result, which is `unsatisfable` property after `n` steps so with this way we eliminate the unlimitaded execution time.

6 Conclusion and future directions

In this paper, we gave a new method to compute some dynamical properties on Process Hitting models, a subclass of asynchronous automata. The main interest of our method is the use of ASP, a declarative programming paradigm which benefits from powerful solvers. We first focused on the enumeration of the fixed points of a model, which is tackled simply on such models given their particular form. We also considered the reachability problem, that is, checking if it is possible to reach a state with a given property from a given initial state, which thus corresponds to an EF operator in CTL logic. Our analysis is thus exhaustive, but can be limited to a number of steps, for which the dynamics of the model from the given initial state is computed. We gave an implementation of these problems into ASP, and applied them to several biological examples of various sizes, up to 40 biological components. Our results showed that our implementation is faster and deals with bigger models than other approaches, especially LIBDDD which is a symbolic model-checker.

Our work could benefit from several extensions. Of course, the set of applicable models can be extended, for example with the addition of priorities or neutralizing edges, or by considering synchronous dynamics or other representations such as Thomas modeling (Bernot et al. 2007). However, the range of the analysis can also be extended, by searching instead the set of initial states allowing to reach a given goal, or extending the method to universal properties (like the AF operator). Finally, the research of attractors in a more general fashion (such as cyclic or complex attractors) would be of major interest to fully understand the behavior of models.

Acknowledgement The European Research Council has provided financial support under the European Community's Seventh Framework Programme (FP7/2007–2013) / ERC grant agreement no. 259267.

References

ANGER, C., KONCZAK, K., LINKE, T., AND SCHAUB, T. 2005. A glimpse of answer set programming. KI 19, 1, 12.

BARAL, C. 2003. Knowledge representation, reasoning and declarative problem solving. Cambridge university press.

BERNOT, G., CASSEZ, F., COMET, J.-P., DELAPLACE, F., MÜLLER, C., AND ROUX, O. 2007. Semantics of biological regulatory networks. Electronic Notes in Theoretical Computer Science 180, 3, 3 – 14.

BIERE, A., CIMATTI, A., CLARKE, E., AND ZHU, Y. 1999. Symbolic model checking without BDDs. Springer.

Colange, M., Baarir, S., Kordon, F., and Thierry-Mieg, Y. 2013. Towards distributed software model-checking using decision diagrams. In Computer Aided Verification. Springer, 830–845.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., and Thiele, S. 2008. A users guide to gringo, clasp, clingo, and iclingo.

Gonzalez, A. G., Naldi, A., Sanchez, L., Thieffry, D., and Chaouiya, C. 2006. Ginsim: a software suite for the qualitative modelling, simulation and analysis of regulatory networks. Biosystems 84, 2, 91–100.

Ibbotson, R. 2010. Extending and continuing the development of an integrated development environment for answer set programming. M.S. thesis, University of Bath.

Khalis, Z., Comet, J.-P., Richard, A., and Bernot, G. 2009. The smbionet method for discovering models of gene regulatory networks. Genes, Genomes and Genomics 3, 1, 15–22.

Klamt, S., Saez-Rodriguez, J., Lindquist, J., Simeoni, L., and Gilles, E. 2006. A methodology for the structural and functional analysis of signaling and regulatory networks. BMC Bioinformatics 7, 1, 56.

Lifschitz, V. 2002. Answer set programming and plan generation. Artificial Intelligence 138, 1, 39–54.

McMillan, K. L. 2002. Applying sat methods in unbounded symbolic model checking. In Computer Aided Verification. Springer, 250–264.

Naldi, A., Berenguier, D., Fauré, A., Lopez, F., Thieffry, D., and Chaouiya, C. 2009. Logical modelling of regulatory networks with ginsim 2.3. Biosystems 97, 2, 134–139.

Naldi, A., Thieffry, D., and Chaouiya, C. 2007. Decision diagrams for the representation and analysis of logical models of genetic networks. In Computational Methods in Systems Biology. Springer, 233–247.

Paulevé, L., Magnin, M., and Roux, O. 2011. Refining dynamics of gene regulatory networks in a stochastic π-calculus framework. In Transactions on Computational Systems Biology XIII. Springer, 171–191.

Paulevé, L., Magnin, M., and Roux, O. 2012. Static analysis of biological regulatory networks dynamics using abstract interpretation. Mathematical Structures in Computer Science 22, 04, 651–685.

Penczek, W., Woźna, B., and Zbrzezny, A. 2002. Bounded model checking for the universal fragment of ctl. Fundamenta Informaticae 51, 1, 135–156.

Rocca, A., Mobilia, N., Fanchon, É., Ribeiro, T., Trilling, L., and Inoue, K. 2014. ASP for construction and validation of regulatory biological networks. In Logical Modeling of Biological Systems, L. F. del Cerro and K. Inoue, Eds. Wiley-ISTE, 167–206.

Rocca, A., Ribeiro, T., and Inoue, K. 2013. Inference and learning of boolean networks using answer set programming. LNMR 2013, 27.

Samaga, R., Saez-Rodriguez, J., Alexopoulos, L. G., Sorger, P. K., and Klamt, S. 2009. The logic of egfr/erbb signaling: Theoretical properties and analysis of high-throughput data. PLoS Computational Biology 5, 8, e1000438.

Sureshkumar, A. 2006. Ansprolog* programming environment (ape): Investigating software tools for answer set programming through the implementation of an integrated development environment.

Thierry-Mieg, Y., Poitrenaud, D., Hamez, A., and Kordon, F. 2009. Hierarchical set decision diagrams and regular models. In Tools and Algorithms for the Construction and Analysis of Systems. Springer, 1–15.

Wuensche, A. 1998. Genomic regulation modeled as a network with basins of attraction. In Pacific Symposium on Biocomputing, R. B. Altman, A. K. Dunker, L. Hunter, and T. E. Klien, Eds. Vol. 3. World Scientific, 89–102.

In order to develop these methods we have used two main frameworks. The first is both a programming paradigm and a declarative programming language: the Answer Set Programming (ASP). The second framework is the Process Hitting (PH) a previously introduced formalism for the representation of biological regulatory networks.

.1 Answer Set Programming

Answer Set Programming (ASP) is a declarative programming paradigm with semantics known as the semantics of answer sets. This paradigm allows the programmer to specify what the problem to be solved is, and not how to solve it. ASP programs are written in AnsProlog* (Sureshkumar 2006) (short for Answer Set Programming in Logic). These programs are composed of a set of facts and a set of rules, as defined below, from which other facts can be derived. A consistent set of facts that can be derived from a program using the rules is called answer set for the program. The sets of possible responses to an AnsProlog* program are calculated with a program called a solver.

.1.1 Syntax and semantics of ASP

1. **The alphabet**
   According to (Baral 2003), the alphabet of ASP is composed of seven classes of symbols:
   - constants, symbols of functions and symbols of predicates are composed of letters and start with a lowercase letter,
   - variables follow the same rule but start with a capital letter,
   - connectors that are $\leftarrow$, **not** and ,,
   - punctuation that can be (,) and .,
   - the special symbols $\bot$ and $\top$.

   The notion of predicate allows to qualify constants in this grammar. For example, the property *Tweety is a bird* can be expressed by the predicate name *bird* around the constant *tweety* into the following literal: *bird(tweety).* Such literal can then be used as premise or conclusion in a rule as defined below.
2. **The rules:** A rule has the following form:

$$head \leftarrow body. \tag{1}$$

   which is equivalent to:

$$l_0 \leftarrow l_1, ..., l_m, \textbf{not } l_{m+1}, ..., \textbf{not } l_n. \tag{2}$$

   where l_i are literals for all $i \in [\![1; n]\!]$, and $0 \leq m \leq n$. This rule states:

$$\left.\begin{array}{lll}\text{If} & l_1, ..., l_m & \textbf{can} \text{ be proved true} \\ \text{and} & l_{m+1}, ..., l_n & \textbf{cannot} \text{ be proved true}\end{array}\right\} \text{Then } l_0 \text{ is } \textbf{true}.$$

   Therefore, $head$ is also called the conclusion of the rule, and $body$ is its premise. There are some special cases of rules (Lifschitz 2002; Baral 2003):
   - A ground rule is a rule where all the literals are constants, and thus contains no variables;
   - A fact, also called reality, is a rule with an empty body. It can be written without

the central arrow ($\leftarrow$) or with a *body* equal to $\top$:

$$l_0 \leftarrow \top. \qquad \text{or} \qquad l_0. \tag{3}$$

- A constraint is a rule where the head equals $\bot$. In this case, the head is often not represented and the constraint will be written generally as:

$$\leftarrow l_1, ..., l_m, \textbf{not } l_{m+1}, ..., \textbf{not } l_n. \tag{4}$$

  A set of literals X violates the constraint (4) if $\{l_1, ..., l_m\} \subseteq X$ and $\{l_{m+1}, ..., l_n\} \nsubseteq X$. Thus, X cannot be an answer set of any program containing this constraint.
- A cardinality constraint is a rule of the following form:

$$l\ \{q_1,\ ...,\ q_m\}\ u \leftarrow body. \tag{5}$$

  with $m \geq 1$, l an integer and u an integer or the infinity (∞). Such a cardinality means that the answer set X contains at least l and at most u atoms in the set $\{q_1, ..., q_m\}$, or, in other words:

$$l \leq |\{q_1, ..., q_m\} \cap X| \leq u \tag{6}$$

  where $\cap$ is the symbol of sets intersection and $|A|$ denotes the cardinality of set A. Thus, if such a cardinality is found in the head of a rule, then it directly constrains the resulting answer set. If it is found in the body of a rule, then it is simply true if the above condition is satisfied. In the following, if they are not explicitly given, we consider that l defaults to 0 and u defaults to ∞.

.1.2 Modeling and solving of a problem with ASP

The construction of models is one of the fundamental components of the scientific process. It concerns all systems we seek to control. A model has two main features (Anger et al. 2005): it is a simplification of a given system and it allows an action on the system like resolving problems, searching for a path, identifying states and verifying properties... This concept of model can address another angle issues related to the representation process. the programs written in ASP follow the strategy of generate and test. Indeed, the modeling process can be regarded as a special form of representation:

- First, the system should be presented with facts.
- Second, properties of the model can be explained with rules.
- Third, all candidate answers can be generated, usually using cardinalities.
- Finally, constraints allow to filter the answers and only the sets of predicates satisfying all these conditions will be returned which and considered as answer sets.

Therefore, in practice, the solving of ASP programs relies on a grounder and a solver that usually work together: first the grounder is used to remove variables in order to achieve an equivalent but constant program, then the solver computes all answer sets for this stabilized program (Lifschitz 2002; Ibbotson 2010). Amongst all available grounders for ASP, we can name GRINGO, DLV and LPARSE, and the solvers include SMODELS, DLV, CMODELS, CLASP...

For the work presented in this paper, we worked with CLINGO (version 3) which is a combination of the grounder GRINGO and the solver CLASP.