Static Analysis and Stochastic Search for Reachability Problem

Xinwei Chai, Tony Ribeiro, Morgan Magnin, Olivier Roux[1,2]

Laboratoire des Sciences du Numérique de Nantes
1 Rue de la Noë, 44300 Nantes, France

Katsumi Inoue[3]

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan

Abstract

This paper focuses on a major improvement on the analysis of reachability properties in large-scale dynamical biological models. To tackle such models, where classical model checkers fail due to state space explosion led by exhaustive search. Alternative static analysis approaches have been proposed, but they may also fail in certain cases due to non-exhaustive search. In this paper, we introduce a hybrid approach ASPReach, which combines static analysis and stochastic search to break the limits of both approaches. We tackle this issue on a modeling framework we recently introduced, Asynchronous Binary Automata Network (ABAN). We show that ASPReach is able to analyze efficiently some reachability properties which could not be solved by existing methods. We studied also various cases from biological literature, emphasizing the merits of our approach in terms of conclusiveness and performance.

Keywords: Model checking, Reachability problem, Asynchronous Binary Automata Network, Local Causality Graph, Heuristics, Answer Set Programming.

1 Introduction

With increasing quantities of available data provided by new technologies, *e.g.* DNA microarray [22], there is a growing need for expressive modelings and their related high-performance analytic tools. Among them, works on concurrent systems have been of interest in systems biology for over a decade [4,5,35]. If model validation is a major concern, one of the main challenges nowadays is predicting the behavior of these systems.

Reachability problem on formal models is a critical challenge where both validation problems (whether the model satisfies the *a priori* knowledge) and prediction problems (properties to be discovered) meet. From a formal point of view, numerous biological properties in computational models can be transformed to reachability properties. For example, the reachability of state 0/1 of a could represent the activation/inhibition of certain gene or synthesis of a protein, while initial state could represent initial observation in an experiment. If the reachability of a certain state contradicts with the *a priori* knowledge, one can modify the model and/or design a new experiment to verify whether there is an error in the *a priori* knowledge or former observation. Also, reachability analysis is of help to medicine design: for example if one wants to prevent the carcinogenesis of a cell (target state), one possible solution is to find the critical pathways towards the target state and design a medicine to cut them in order to keep the cell healthy.

In the domain of model checking, reachability has been of great interest for over 30 years [10,11]. Various modeling frameworks and semantics in bioinformatics have been studied: Boolean network [2], Petri nets [23,14], timed-automata [12,36]. These approaches rely on global search and thus face state explosion problem as the state space grows exponentially with the number of variables. In [28], it has been shown that the reachability problem of Petri net is exponential time-hard and exponential space-hard, and this conclusion does not change even under some specific conditions [14]. For 1-safe Petri nets, the complexity of reachability analysis is generally PSPACE-complete [8]. Li *et al.* [20,21] investigated theoretically the stability, the controllability and the reachability of Switched Boolean Networks, but their method remains computationally expensive; Saadatpour *et al.* [30] researched only the reachability of fixed points.

To tackle the complexity issue, symbolic model checking [6] based on ordered binary decision diagrams (OBDDs) and SAT-solvers (satisfiability) [1] have been studied over years, but still fail to analyze big biological systems with more than 1000 variables. Bounded Model Checking (BMC) [9] is an efficient approach but generally not complete as its searching depth is limited to a given integer k.

Beside these approaches, abstraction is an efficient strategy to deal with such models of big scale. It aims at approximating the model while keeping the most important parts influencing the reachability. Abstract methods often have better time-memory performance but with a loss of information. They solve usually a simplified version of the original model, i.e. the results from these approaches are not necessarily compatible with all the properties of the original model. While studying reachability problems, the system dynamics is abstracted to static causalities between states and transitions.

We have designed a new discrete modeling framework for a concurrent system [7]: Asynchronous Binary Automata Network (ABAN). In ABAN, we applied the approach developed by Paulevé *et al.* [25,15,26] to address reachability problem. This approach refers to a static abstraction of the reachability (with an over-approximation and an under-approximation of the real dynamics). It is based on an abstract interpretation: Local Causality Graph (LCG). This interpretation drasti-

cally reduces the searching state-space thus avoids costly global search [27]. However, this pure static analysis is not complete as there are inconclusive cases which can not be decided reachable or not.

Many biological networks are encoded in Boolean style, *e.g.* [2,18], because BN is a simple formalism but with strong applicability: discretization in BN is a way to handle the imprecision of *a priori* knowledge on the model. However BN may be not expressive enough. As to the modeling of the dynamic behavior "$a \leftarrow 1$ at moment $t + 1$ if $b = 1$ at moment t", one has $a(t + 1) = b(t)$ in BN. a always follows the evolution of b but with a potentially unwanted behavior "$a \leftarrow 0$ when $b = 0$ at moment t". ABAN models this dynamics as *via* $\{b_1\} \rightarrow a_1$ without this redundancy. Besides, BNs are transformable to ABANs, and this property makes our approach applicable to a wider domain (Appendix A).

Our work shares similar concerns but we combine static analysis and bounded model checking. We have developed a heuristic approach PermReach to attack reachability problem [7] which is more conclusive than pure static analysis but time-consuming and still not able to solve reachability problems under certain conditions. In this paper, we propose a hybrid approach ASPReach based on the former LCG reasoning and a non-exhaustive search in the LCG to obtain a more conclusive solution of reachability problems. ASPReach allows one to solve the cases where other static methods fail. Furthermore, it can also solve the reachability of a set of states which to our knowledge has never been done in static way. We assess the value of our contribution using benchmarks on biological examples from the literature.

This paper is organized as follows: Section 2 introduces the formal background and the formalization necessary to the understanding of the work; Section 3 presents the concrete methods and algorithms composing the whole approach; Section 4 shows the benchmarks evaluating our approach and other alternatives; Section 5 concludes this paper.

2 Formalization

Notations:

a_i means automaton a is taking value i;

$x :: y$ is the sequential connector of entities x and y, where x appears just before y;

$a.next$ is the immediate successor of a; $a.pred$ is the immediate predecessor of a.

Boolean Network is a traditional and efficient modeling framework, with many biological networks encoded in BN [18]. To describe the dynamical properties more precisely, Automata Network is introduced [7,29]. It can be considered as a subset of communicating finite state machines or safe Petri Nets.

Asynchronous Binary Automata Network (ABAN) is a special case of Automata Network. "Asynchronous" implies the update scheme that no more than one automaton can change its value at a time. Asynchronous update scheme makes a

trade-off between the biological reality and complexity. From a given state, biological systems may evolve to multiple future states (e.g. cell differentiation), but it is costly to simulate generalized ANs where arbitrary subsets of automata can be updated simultaneously. "Binary" implies that every automaton has exactly two possible states $(0, 1)$. In Appendix A, we show how to transform a Boolean Network into ABAN.

Definition 2.1 [ABAN] An ABAN is a triplet $AB = (\Sigma, L, T)$, where:

- $\Sigma \triangleq \{a, b, \ldots\}$ is the finite set of automata with every component having a Boolean state;

- $LS \triangleq \underset{a \in \Sigma}{\cup} \{a_0, a_1\}$ is the set of all local states, $L \triangleq \underset{a \in \Sigma'}{\times} \{a_0, a_1\}$ is the set of joint states where $\Sigma' \subseteq \Sigma$. Particularly, if $\Sigma' = \Sigma$, L is the set of global states.

- $T \triangleq \{A \to b_i \mid b \in \Sigma \wedge A \in L\}$ is the set of transitions, where A (called head) is the set of required state(s) for transition $tr = A \to b_i$, which allows to flip b_{1-i} to b_i (called body). In other words, transition tr is said fireable iff $A \subseteq s$, where s is the current global state.

A local state represents the state of *one* automaton, e.g. a_1, while a global state represents the *joint state* of all the automata in the network, e.g. $\langle a_0, b_1, c_0 \rangle$ where $L = \{a, b, c\}$.

Definition 2.2 [Asynchronous Dynamics] From current global state s, the global state after firing transition $tr = A \to b_i$ is denoted $s \cdot tr = s \setminus \{b_{1-i}\} \cup \{b_i\}$, where $b_{1-i} \in s$. The state of a certain automaton a is noted $(s \cdot tr)[a]$. Particularly, if there is no fireable transition, the next state remains the same as the current state.

To describe the evolution in an ABAN, we use the notion of trajectory.

Definition 2.3 [Trajectory] Given an ABAN $AB = (\Sigma, L, T)$ and a global initial state $\alpha \in L$, a trajectory t from α is a sequence of transitions $t = tr_1 :: \cdots :: tr_i :: \cdots :: tr_n$ with $tr_i \in T$ and each tr_i is fireable in $(s \cdot tr_1 \cdot \ldots \cdot tr_{i-1})$. From α, the global state after firing all transitions of t is $(s \cdot tr_1 \cdot \ldots \cdot tr_n)$, denoted $s \cdot t$.

Definition 2.4 [State sequence] Given an ABAN $AB = (\Sigma, L, T)$ and a global initial state $\alpha \in L$ and trajectory t, the state sequence $seq = s_1 :: \cdots :: s_i :: \cdots :: s_n$ with $s_i \in LS$ is formed by the updated local states during the trajectory t.

Example 2.5 Fig. 1 shows an ABAN with initial state $\alpha = \langle a_0, b_0, c_0, d_0, e_0 \rangle$ and a possible trajectory is $t = \{d_0\} \to b_1 :: \{b_1\} \to d_1 :: \{d_1\} \to c_1 :: \{b_1, c_1\} \to a_1$. After firing the transitions in trajectory t, the state becomes $\Omega = s \cdot t = \langle a_1, b_1, c_1, d_1, e_0 \rangle$, and $\omega = a_1 = (\alpha \cdot t)[a]$. The state sequence is $seq = b_1 :: d_1 :: c_1 :: a_1$.

As to reachability problem, given an ABAN $AB = (\Sigma, L, T)$, the joint reachability $REACH(\alpha, \Omega)$ is formalized as: joint state Ω is reachable iff there exists a trajectory t s.t. $\alpha \cdot t = \Omega$. Partial reachability $reach(\alpha, \omega)$ is defined analogously: local state $\omega = a_i$ is reachable iff there exists a trajectory t s.t. $(\alpha \cdot t)[a] = a_i$. $REACH(\alpha, \Omega)$ and $reach(\alpha, \omega)$ take Boolean values **True**, **False** or **Inconclusive**
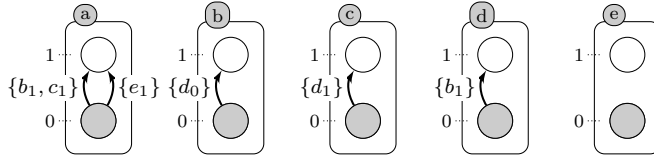
Fig. 1. An example of ABAN

if it cannot be decided. In Fig. 1, $\Omega = \langle a_1, b_1, c_1, d_1, e_0 \rangle$ or $\omega = a_1$ is reachable from initial state α via trajectory t, i.e. $reach(a_1) = \textbf{True}$ and $REACH(\alpha, \Omega) = \textbf{True}$. In fact, the reachability of a joint state even a global state is equivalent to that of one local state. Given a set of target states $\{a_i, \ldots z_j\}$, by adding a new automaton x to Σ, setting its initial state to x_0 and adding transition $\{a_i, \ldots z_j\} \to x_1$ to T, $REACH(\alpha, \{a_i, \ldots z_j\})$ is thus equivalent to $reach(\alpha, x_1)$. For convenience, we study partial reachabilities in this paper.

Paulevé *et al.* [26] have proposed Local Causality Graph (LCG) to analyze reachability problems statically. LCG abstracts the original problem through an over-approximation (necessary condition) and an under-approximation (sufficient condition). It is a very efficient tool as there is no global search and all the operations are bounded in polynomial complexity. However LCG does not guarantee to obtain a result, i.e. some inconclusive instances satisfy the necessary condition and fail sufficient conditions. In this paper, we make use of LCG by removing some elements needed only in multivalued networks, then we try to analyze it more deeply to solve inconclusive cases of binary valued systems. In fact, Didier *et al.* have shown a technique to transform a multivalued network to Boolean network [13], which provides us the applicability to multivalued situations.

Definition 2.6 [LCG] Given an ABAN $AB = (\Sigma, L, T)$, an initial state α and a target state ω, LCG $l = (V_{\text{state}}, V_{\text{solution}}, E)$ is the smallest recursive structure with $E \subseteq (V_{\text{state}} \times V_{\text{solution}}) \cap (V_{\text{solution}} \times V_{\text{state}})$ which satisfies:

$$\omega \in V_{\text{state}}$$
$$a_i \in V_{\text{state}} \Leftrightarrow \{(a_i, sol_{a_i}) | a_i \in \alpha\} \subseteq E$$
$$sol_{a_i} \in V_{\text{solution}} \Leftrightarrow \{(sol_{a_i}, \mathbf{V}_a(sol_{a_i})\} \subseteq E$$

where $V_{\text{state}} \subseteq LS$ is the set of local states, $V_{\text{solution}} \subseteq T$ is the set of solutions and $\mathbf{V}_a$ is the set of required local states of sol_{a_i}.

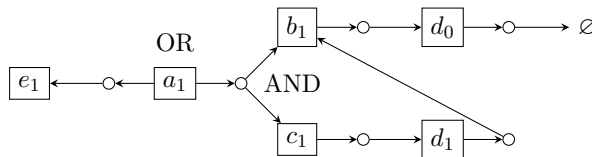**Example 2.7** Fig. 2 shows the LCG for analyzing $reach(a_1)$ in Fig. 1.

Fig. 2. Visualization of LCG, with the squares representing local states and small circles representing solution nodes. $\varnothing$ signifies that there is no need to link any transitions, i.e the former state d_0 is in the initial state.

Algorithm 1 describes how to construct an LCG from an ABAN $AB = (\Sigma, L, T)$.

Starting from a given target state $Ls = \omega$, one can find all the transitions $T_s \subseteq T$ reaching ω and add edges $\omega \to T_s$. Then we find all the heads A of T_s and add edges $T_s \to A$ and replace Ls with A (recursion). Finally, we update the structure until $Ls \subseteq \alpha$ or there is no transition with body in Ls.

Input: an ABAN $AB = (\Sigma, L, T)$, an initial state α, a target state ω
Output: LCG $l = (V_{\text{state}}, V_{\text{solution}}, E)$
Initialization: $Ls \leftarrow \{\omega\}$, $V_{\text{state}} \leftarrow \varnothing$, $V_{\text{solution}} \leftarrow \varnothing$, $E \leftarrow \varnothing$
while $Ls \neq \varnothing$ **do**
    $Ls = Ls \backslash V_{\text{state}}$
    for $a_i \in Ls$ **do**
        $Ls \leftarrow Ls \backslash \{a_i\}$
        if $a_i \in \alpha$ **then**
            $E \leftarrow E \cup \{(a_i, \varnothing)\}$
        else
            // Choose the transitions reaching a_i, i.e. with body a_{1-i}
            for $sol = A \to a_{1-i} \in T$ **do**
                $V_{\text{solution}} \leftarrow V_{\text{solution}} \cup \{sol\}$
                $E \leftarrow E \cup \{(a_i, sol)\}$
                $V_{\text{state}} \leftarrow V_{\text{state}} \cup A$
                for $b_j \in A$ **do**
                    $E \leftarrow E \cup \{(sol, b_j)\}$
            $Ls \leftarrow Ls \cup A$
            $V_{\text{state}} \leftarrow V_{\text{state}} \cup Ls$
        $V_{\text{solution}} \leftarrow V_{\text{solution}} \cup a_i.\text{next}$
return $(V_{\text{state}}, V_{\text{solution}}, E)$

Algorithm 1. Construction of LCG

Intuitively, when the recursive construction is complete, SLCG is in fact a digraph with state nodes V_{state} and solution nodes V_{solution}. E consists of the edges between local state nodes and solution nodes. To access certain local states, at least one of its successor solutions (corresponding transitions from solution nodes) need to be fired; to make one solution node firable, all of its successor local states need to be satisfied. A recursive reasoning of reachability begins with a state node representing target local state, goes through $a_i \to sol_{a_i} \to b_j \cdots$ and ends with initial state (possibly reachable) or a local state without solution successor (unreachable).

With LCG, it is easy to verify whether their are potential pathways from the target state ω to the initial state α. If there does not exist such a pathway, one can ensure that ω is not reachable from α. [27] explains this local reasoning.

Definition 2.8 [Pseudo-reachability] Given an LCG $l = (V_{\text{state}}, V_{\text{solution}}, E)$ with

global initial state α, the pseudo-reachability of node $v \in V_{\text{state}}$ is defined as

$$
reach'(\alpha, v) = \begin{cases} \textbf{True} & \text{if } v \in \alpha \\ \textbf{False} & \text{if } v \notin \alpha \text{ and } \nexists(s, sol) \in E \\ \bigvee_{(s,sol) \in E}(\bigwedge_{(sol,s) \in E} reach'(\alpha, s)) & \text{otherwise} \end{cases}
$$

However, pseudo-reachability is named "pseudo" because it is only an over-approximation of reachability, i.e. it reveals a necessary condition of reachability.

Example 2.9 In Fig. 3, $reach'(l, c_1) = reach'(l, a_1) \wedge reach'(l, b_1) = reach'(l, a_0) \wedge reach'(l, b_0) = \textbf{True}$. Both a_1 and b_1 are reachable, but they can not be reached simultaneously. In such LCG, there are two branches, $a_1 \mapsto b_0$ and $b_1 \mapsto a_0$, the automata a and b involve themselves in different branches, the reachability of a_1 impedes the reachability of b_1 and *vice versa*.
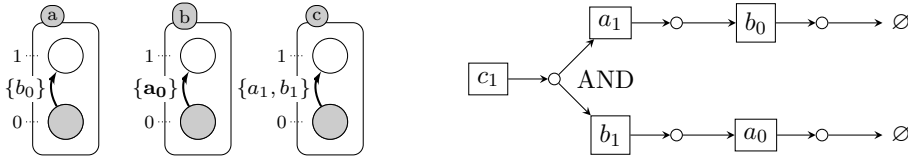

Fig. 3. $\Sigma = \{a, b, c\}$, $T = \{\{b_0\} \rightarrow a_1,\ \{a_0\} \rightarrow b_1,\ \{a_1, b_1\} \rightarrow c_1\}$, $\omega = c_1$

Also, the recursive reasoning does not terminate if there exists cycles in LCG. While computing the pseudo-reachability, self-dependent form $reach'(l, a_i) = \ldots = reach'(l, a_i)$ will appear. Dealing with cycles becomes inevitable.

Definition 2.10 [Cycle] In an LCG, a cycle is formed by a sequence of nodes linked as follows: $a_i \rightarrow \circ \rightarrow \cdots \rightarrow \circ \rightarrow a_i$

To identify cycles, we search instead Strongly Connected Components (SCC) of size greater than one. Because cycles may interlace and there is no such problem in SCC. In other words, a SCC may contain several nested cycles which connect to each other. [33] shows that the detection of SCCs can be done in $O(|V| + |E|)$ time, with $|V|$ the cardinality of the vertices and $|E|$ the cardinality of the edges. LCG is usually a sparse graph, as in biological systems, the components mostly interact with only a part of the system, hence the out-degree can be considered of $O(1)$ and the detection of SCCs [4] can be done in $O(|V|)$, *i.e.* linear time.

Theorem 2.11 *Given a cycle* $x \rightarrow \circ \rightarrow \cdots \rightarrow \circ \rightarrow x$ *in an LCG, if there is at most one incoming edge to the cycle, the cycle can be removed.*

Proof. If there is no incoming edge, the target state y must be in the cycle. The edge $y.pred \rightarrow \circ \rightarrow y$ can be removed, because the reachability of $y.pred$ requires y, but y is the target state, which is never reached before the other local states in the

---

LCG are reached. Thus the transition corresponding to this edge is never fired and the edge can be removed. Similarly, if there is an outside incoming edge $a \to \circ \to x$, a must be the successor of target state y or the target itself, $x.pred \to \circ \to x$ can hence be removed.   □
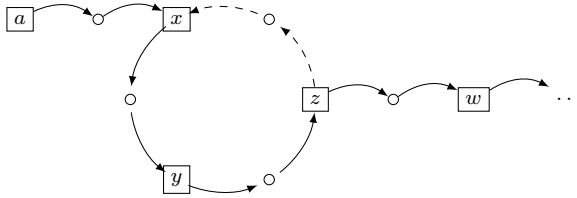

Fig. 4. LCG l containing cycle $x \to \circ \to y \to \circ \to z \to \circ \to x$

Example 2.12 In Fig. 4, the pseudo-reachability of a is

$$reach'(l, a) = reach'(l, x) = reach'(l, y) = reach'(l, z) = reach'(l, x) \vee reach'(l, w)$$

To reach x, we need to reach z, but z cannot depend on x as x is already to be reached. Self-dependence appears: x is reachable if x is reachable. Thus edge $z \to \circ \to x$ is deleted (dashed line).

Unfortunately, not all cycles are removable via Theorem 2.11. Example 2.13 explains the issue.

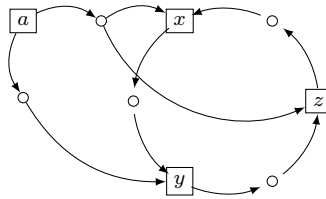Fig. 5. x, y, z all have external links, thus none of the links can be discarded

Example 2.13 Cycle $x \to \circ \to y \to \circ \to z \to \circ \to x$ is unbreakable according to Theorem 2.11, is possesses 3 incoming edges. But with Theorem 2.14, if **OR gates** are removed, the cycle can be dealt with. The removal of **OR gates** is stated in the next section.

Theorem 2.14 *Given a cycle, if it contains no **OR gate**, all the local states in the cycles are unreachable.*

Proof. Suppose an arbitrary cycle $C = a_i \to \cdots b_j \to \cdots \to a_i$, with $\to$ an edge in the LCG. Note that $reach'(\alpha, a_i) \implies reach'(\alpha, b_j) \implies reach'(\alpha, b_j.next) \implies \cdots \implies reach'(\alpha, a_i)$. According to the definition of $reach'$, $reach'(\alpha, a) = $ **True** only if $\exists c_k \in C$ and $c_k \in \alpha$. If there exists such c_k, C should not exist as the reasoning stops at c_k and does not form a cycle, contradiction. $reach'(\alpha, a_i) = reach'(\alpha, b_j) = \cdots = $ **False**.   □

3   ASPReach Algorithm

In this section, we present the main contribution of this paper, our analyzer **AS-PReach**: an algorithm for checking the reachability of a target local state ω from a global initial state α (which can also be partial) in a given ABAN. However, exhaustive search leads to heavy computation and huge need of memory. The algorithm proposed below tries to overcome those shortcomings by combining static analysis and stochastic search into the following hybrid approach.

ASPReach:
- Input: An ABAN AB, an initial state α, a target state ω and a max number of iterations k
- Output: $reach(\omega) \in \{\textbf{False}, \textbf{True}, \textbf{Inconclusive}\}$

  (i) Construct the LCG $l = LCG(AB, \alpha, \omega)$

 (ii) Try to remove all cycles and prune useless edges from l

(iii) Try to prove unreachability of ω in l using pseudo-reachability $reach'(l, \omega)$ and return **False** if $reach'(l, \omega) = $ **False**

(iv) Try at most k times
  - $l' \leftarrow l$
  - Simplify each **OR gate** such that l' is a LCG with only **AND gates**
  - If there remain cycles:
    · Back to step (iv)
  - Generate all trajectory that starts with α in l' using ASP
    · If a trajectory t ending with ω is found, return **True**

 (v) return **Inconclusive**

    LCG illustrates the causality between necessary transitions to be fired to reach the target state; the tentative of removing cycles simplifies the LCG and keep the reachability unchanged; pseudo-reachability allows one to filter some unreachable cases based on the topology of LCG. The heuristic approach is the core of our algorithm. Stochastic choices avoid combinatorial explosion on different **OR Gates**. The ASP part searches thoroughly the result but does not traverse the whole state space (ASP solver starts from constraints, finds one consistent order and terminate the search).

    Algorithm 2 provides the detailed pseudocode of the algorithm taking an LCG l as input whose detailed construction is given in Algorithm 1. Lines 4-8 delete all cycles with at most one incoming edge. After removing cycles, the LCG may contain nodes without successor. Such nodes can be pruned since they do not lead to initial state (Line 9-18). This preprocessing reduces the search space of the stochastic search performed in step 4. Now l is pruned and might be cycle-free. Static analysis of l can then be used as heuristics to check pseudo-reachability (Definition 2.8) in order to detect some unreachability cases (Lines 19-21) which may conclude before searching. LCG shows the dependencies between local states and

1: Input: LCG $l = (V_{\text{state}}, V_{\text{solution}}, E)$, an integer k

2: Output: reachability r and a trajectory t

3: Compute SCCs, classify them into SCC1(l) with at most 1 incoming edge and SCC2(l) otherwise

4: // 1) Break all cycles and prune useless branches

5: **for** each $(V'_{\text{state}} \subseteq V_{\text{state}}, V'_{\text{solution}} \subseteq V_{\text{solution}}) \in$ SCC1(l) **do**

6:     **for** each $v \in V'_{\text{state}}$ **do**

7:         **if** $\exists (v, v') \in E, v' \in (V_{\text{solution}} \setminus V'_{\text{solution}})$ **then**

8:             $E \leftarrow E \setminus \{(v, v'') | v'' \in V'_{\text{solution}}, (v, v'') \in E\}$

9: // 2) remove useless nodes/edges

10: pruned = **True**

11: **while** pruned **do**

12:     pruned = **False**

13:     **for** $v \in V_{\text{state}}$ **do**

14:         **if** $\nexists (v, v') \in E$ **then**

15:             $V_{\text{state}} \leftarrow V_{\text{state}} \setminus \{v\}$; $E \leftarrow E \setminus \{(v'', v) \in E\}$

16:             $E \leftarrow E \setminus \{(v'', sol) \in E | sol \in \{sol = (A \rightarrow a) \in V_{\text{solution}} | v \in A\}\}$

17:             $V_{\text{solution}} \leftarrow V_{\text{solution}} \setminus \{sol = (A \rightarrow a) \in V_{\text{solution}} | v \in A\}$

18:             pruned = **True**

19: // 3) Check pseudo-reachability

20: **if** $pseudoReach(l) = $ **False then**

21:     **return** (**False**, $\varnothing$)

22: // 4) main search loop

23: **for** each i in $1 \ldots k$ **do**

24:     $l' = (V'_{\text{state}}, V'_{\text{solution}}, E') \leftarrow (V_{\text{state}}, V_{\text{solution}}, E)$

25:     **for** $v \in V'_{state}$ **do** // Treat each OR gates

26:         pick a random element $(v, v') \in E'$

27:         $E' \leftarrow E' \setminus \{(v, v'') \in E' | v'' \neq v'\}$ with $\nexists i \in$ SCC2(l) and $i \in E'$

28:     $(r, t) \leftarrow$ ASPsolve(l')

29:     **if** $r = $ **True then return** (**True**, t)

30: **return** (**Inconclusive**, $\varnothing$)

Algorithm 2. ASPReach

transitions. A pathway in LCG suggests a possible trajectory of reaching the target state. If $pseudoReach(l, \omega) = $ **False**, we can ensure that ω is unreachable, as pseudo-reachability checks a necessary condition of reachability. If $reach'(l, \omega) = $ **True**, static analysis is not sufficient for reachability analysis. When static analysis fails, a stochastic search is performed at most k times (line 22-29) to find a state sequence from the initial state α to target state ω. If there remain cycles with multiple incoming edges, according to Theorem 2.14, ω is unreachable. The value of k will be discussed later in the evaluation section. Random choices are made to fix a value for each **OR gate** of the LCG allowing to perform a reachability check by generating all possible variable assignment order using ASP. Keep in mind that every state node is an **OR gate**, we have to choose one of its successor solution

nodes to access the state. A set of **OR gate** choices is called an *assignment*.

3.1 Stochastic Search

As every **OR gate** has multiple choices, to avoid combinatorial explosion, we use a simple heuristic: choose randomly one assignment for each trial. Then we can construct a new LCG without **OR gate**, every state node has exactly one successor solution node.

After applying heuristics to delete **OR gates**, we use ASP (Answer Set Programming) [3] to analyze the newly obtained LCG with only **AND gates**. ASP is a prolog-like declarative programming paradigm. It uses description and constraints of the problem (called rule) instead of imperative orders. ASP solver tackles problems by generating all the possibilities respecting the constraints. We use Clingo[16] which is a combination of grounder Gringo and solver Clasp. Given an input program with first-order variables, grounder computes an equivalent ground (variable-free) program for an ASP program, while solver selects admissible solutions (answer sets) in the ground.

A rule is in the following form:

$$a_0 \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n.$$

where the element on the left of the arrow is called head and the ones on the right called body. a_0 is **True** if $a_1, \ldots, a_m$ are **True** and $a_{m+1}, \ldots, a_n$ are **False**. Some special rules are noteworthy. A rule where $m = n = 0$ is called a fact and is useful to represent data because the left-hand atom a_0 is thus always **True**. It is often written without the central arrow. On the other hand, a rule where $n > 0$ and $a_0 = \perp$ is called a constraint. As $\perp$ can never become **True**, if the right-hand side of a constraint is **True**, this invalidates the whole solution. Constraints are thus useful to filter out unwanted solutions. The symbol $\perp$ is usually omitted in a constraint.

Programs can yield no answer set, one answer set, or several answer sets. For example, the program `b:- not c.  c:- not b.` produces two answer sets: $\{b\}$ and $\{c\}$. Indeed, the absence of c makes b true, and conversely absence of b makes c true. Cardinality constraints are another way to obtain multiple answer sets. The most usual way of using a cardinality is in place of a_0:

$$l\{q_1, \ldots, q_k\}u \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n.$$

where $k \geq 0$, l is an integer and u is an integer or ∞. Such cardinality means that under the condition that the body is satisfied, the answer set X must contain at least l and at most u atoms from the set $\{q_1, \ldots, q_m\}$, or, in other words: $l \leq |\{q_1, \ldots, q_m\} \cap X| \leq u$.

ASP Encoding
After deleting **OR gates**, to encode the reachability problem in ASP, we first describe the facts:

Predicate `init(a,i)` shows the automaton a is at initial state i. Predicate `node(a,i,n)` shows the node a_i in the LCG is numbered n, while `parent(n1,n2)` expresses node No.n_1 is the predecessor of No.n_2. The LCG in Fig. 3 is encoded as follows:

```
init(a,0). init(b,0). init(c,0).

node(a,1,1). node(b,1,2). node(c,1,3).
node(b,0,4). node(c,0,5).

parent(1,2). parent(1,3).
parent(2,5). parent(3,4).
```

After the facts, we want the nodes to appear in an order by which we can fire all the transitions sequentially from initial state to target state.

The rough idea is: If different states of one automaton a appear, e.g. a_0 and a_1. One of them must be in initial state (suppose a_0). The transitions with head a_0 have to be fired before a_0 flipping to a_1, otherwise there is no solution node in the LCG which allows a_1 return to a_0. In other words, the predecessor of a_0 must appear before a_1. `Core rule` describes this constraint.

Predicate `prior(N1,N2)` signifies node N_1 appears earlier than N_2 in the resulting state sequence. `seq(O,a,i)` shows that state node a_i appears in the O-th place in a trajectory. `reachable/unreachable` is the final result of the program.

```
%Rule 1, a node appears always earlier than its predecessor
prior(N1,N2) :- parent(N2,N1).
%Rule 2, transitivity
prior(N1,N3) :- prior(N1,N2), prior(N2,N3).
%Rule 3, Core rule
prior(N1,N2) :- node(P1,S1,N1), node(P2,S2,N2), node(P2,S3,N3),
                parent(N1,N3), init(P2,S3), S2!=S3, P1!=P2.
%target is unreachable if there is a conflict in order
unreachable :- prior(N1,N2), prior(N2,N1), N1<N2.
%One node appears once and at least once in a sequence
1seq(1..O,P,S)1 :- O=node(P1,S1,N1):node(P1,S1,N1),
                   node(P,S,N), not unreachable.
%Nodes in the sequence are consistent with the order
:- prior(N1,N2), node(P1,S1,N1), node(P2,S2,N2),
   seq(O1,P1,S1), seq(O2,P2,S2), O1>O2.
%One place in the sequence cannot be taken by multiple nodes
:- seq(O1,P1,S1), seq(O2,P2,S2), P1!=P2, O1=O2.
:- seq(O1,P1,S1), seq(O2,P2,S2), S1!=S2, O1=O2.
%--------output formatting, displaying initial states first
:- seq(O1,P1,S1), seq(O2,P2,S2), init(P1,S1),
   not init(P2,S2), O1>O2.
:- seq(O1,P1,S1), seq(O2,P2,S2), init(P1,S1), init(P2,S2),
```

```
    P1<P2, O1>O2.
reachable :- not unreachable.
```

Notation: $a \triangleright b$ means a appears before b.

When analyzing the LCG in Fig. 3, Rule 1 gives $b_0 \triangleright a_1$, $a_1 \triangleright c_1$, $a_0 \triangleright b_1$, $b_1 \triangleright c_1$; Rule 2 gives $a_0 \triangleright c_1$ and $b_0 \triangleright c_1$; Rule 3 gives $a_1 \triangleright b_1$ and $b_1 \triangleright a_1$ which is impossible, therefore there does not exist a state sequence to reach c_1 from initial state. c_1 is unreachable.

If we find a state sequence consistent with all the order constraints, we can obtain its corresponding trajectory, thus we are sure that the target state is reachable.
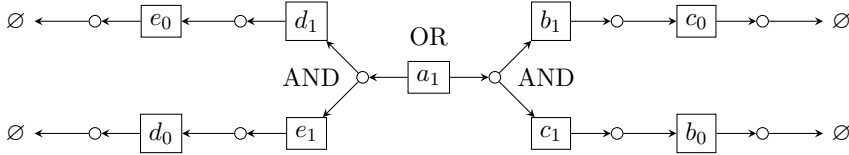

Fig. 6. If an LCG contains such structure, the result could be inconclusive. However the inconclusiveness requires a_1 does not possess other reachable branches.

Still, ASPReach is not complete. A counter-example is shown in Fig. 6, when there are multiple branches of one **OR gate** leading to unreachability, the result can be inconclusive. There is a tricky way to deal with this issue when $|\mathbf{ORgates}|$ is not big: we set a limit n, if $|\mathbf{ORgates}| < n$, we shift the heuristics on the assignment of **OR gates** to the enumeration of all possible assignments. This "hacking" can deal with some inconclusiveness. In the benchmarks in the next section, inconclusive instances appear neither in biological examples nor in random generated tests.

We also show some algorithmic properties of ASPReach:

Theorem 3.1 (ASPReach termination and correctness) *Let $l = (V_{\text{state}}, V_{\text{solution}}, E)$ be an LCG with initial state α and target local state ω and $k > 0$ be an integer. The call ASPReach(l,k) terminates.*
$ASPReach(l, k) = (\mathbf{False}, \varnothing)$ only if $\nexists t$ a trajectory in l from α to ω.
$ASPReach(l, k) = (\mathbf{True}, t)$ only if $\exists t$ a trajectory in l from α to ω. The proof is given in appendix B.

Theorem 3.2 (ASPReach complexity) *Let $l = (V_{\text{state}}, V_{\text{solution}}, E)$ be an LCG with initial state α and $k > 0$ be an integer. Let $s = |V_{\text{solution}}|$ be the number of target state of l. Let $v = |V_{\text{state}}|$ be the number of vertices of l. Let $e = |E|$ be the number of edges of l. The complexity of ASPReach(l,k) is $O(v + e + v/2 \times v \times e \times s + v^2 \times e + v \times e + k \times (v \times e^2 + 2^v))$ which is bound by $O(k \times 2^v)$. Proof is given in Appendix B.*

4 Evaluation

In this section we evaluate our approach through various experiments. All tests were run on a Intel Core i7-3770 CPU, 3.4GHz with 8Gb of RAM computer.

To validate our approach, we first tested a small model, λ-phage model [34] to compare with an alternative reachability analyzer Pint [27] implementing solely

analysis using LCG [25,15,26]. In this model with 4 automata and 12 transitions (without taking consideration of the self-regulations), our result shows complete conclusiveness while Pint cannot (Fig. 7). PermReach [7] is not able either to handle some of the special cases where multiple states of one automaton appear in different branches (Fig. 8). Theses cases are solvable by ASPReach. The whole approach is implemented in Python3 [5]. The call of ASP is done by package *pyasp* [6].
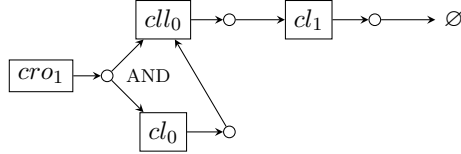

Fig. 7. One LCG in λ-phage model, automaton cl appears in both branches of the **AND gate**. Static analyzer Pint cannot decide whether in this case cro_1 is reachable or not, because it does not consider the order in the state sequence even though there exists a solution is of length 3: $cll_0 :: cl_0 :: cro_1$ corresponding to the trajectory $\{cl_1\} \to cll_0 :: \{cll_0\} \to cl_0 :: \{cll_0, cl_0\} \to cro_1$.
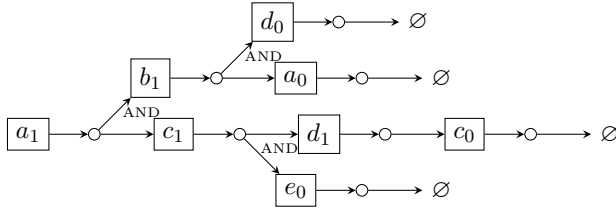

Fig. 8.   This counterexample shows former PermReach is inconclusive if multiple states of one automaton appear in the branches of one **AND gate** (d_0 and d_1 in this example). However there exists a consistent state sequence: $d_1 :: b_1 :: c_1 :: a_1$ corresponding to the trajectory $\{c_0\} \to d_1 :: \{d_0, a_0\} \to b_1 :: \{d_1, e_0\} \to c_1 :: \{b_1, c_1\} \to a_1$ which could be found by ASPReach.

To evaluate the scalability in *in silico* networks, we take T-cell Receptor model (TCR) [31] and epidermal growth factor receptor model (EGFR) [32] as examples, with the former one containing 95 automata and 206 transitions and the latter one containing 104 automata and 389 transitions respectively.

These models are originally Boolean networks. According to the approach in Appendix A, BNs are transformed into ABANs. Here, we ran the same test as in [15]. In the TCR model, we take 3 automata as input (`cd4 cd28 tcrlig`), varying exhaustively their initial states combinations (2^3), take the reachability of states of 5 automata (`sre ap1 nfkb nfat sigmab`) as output. Similarly we carried a bigger test on EGFR model with 13 automata as input and 12 automata as output. We first tested the performance of traditional model checkers, Mole [7] and NuSMV [8], in which Mole turns out to be memory-out for 6 in 12 outputs, and all memory-out for NuSMV in model EGFR. Due to the big state space, traditional model checkers are not applicable. In the TCR tests, our approach gives exactly the same result as Pint did. As for EGFR tests, ASPReach returned no inconclusive output.

---

[5] Code and testing data available at https://github.com/XinweiChai/LCG-in-ASP

[6] https://pypi.python.org/pypi/pyasp

[7] http://www.lsv.fr/~schwoon/tools/mole

[8] http://nusmv.fbk.eu

Model	λ-phage		TCR		EGFR	
Inputs	4		3		13	
Outputs	4		5		12	
Total tests	$2^4 \times 4 = 64$		$2^3 \times 5 = 40$		$2^{13} \times 12 = 98,304$	
Analyzer	Pint	**ASPReach**	Pint	**ASPReach**	Pint	**ASPReach**
Reachable	36(56%)	38(59%)	16(40%)		64,282(65.4%)	74,268(75.5%)
Inconclusive	**2(3%)**	**0(0%)**	0(0%)		**9,986(10.1%)**	**0(0%)**
Unreachable	26(41%)		24(60%)		24,036(24.5%)	24,036(24.5%)
Total time	< 1s	< 1s	7s	40s	**9h50min**	**3h46min**

Table 1
Results of the tests on small (λ-phage) and large (TCR,EGFR) examples from biology literature. Results of model-checkers using global search are memory-out so are not listed in the table. "Reachable", "Inconclusive" and "Unreachable" give respectively the number of different results of reachability. It is worth noticing that the inconclusive cases in Pint are caused by time-out [15].

As seen in Table 1, our approach can be more conclusive than Pint for ABANs. In the configuration of heuristics, we set a threshold for **OR gates**. If there are less than 10 **OR gates** after preprocessing, the computation will be shifted from heuristic to the enumeration of all combinations of **OR gates**. Here is the case for these three benchmarks. The experiments show the ability of ASPReach is already more conclusive than Pint in "simple" cases.

To test the global applicability of ASPReach, we ran two sets of tests on random models generated as follows: Given the number of transitions, for every transition tr, the head a_h is randomly chosen from LS, the first element of the body A_1 is chosen from $LS_1 = LS \setminus \{a_h, a_{1-h}\}$. For $i > 1$, if $A_{i-1} = b_x$ exists, we generate A_i with an 80% probability, choosing from $LS_i = LS_{i-1} \setminus \{b_x, b_{1-x}\}$.

Fig. 9(a) shows the average run time of ASPReach on randomly generated ABAN. In this experiment, we fixed the number of transitions to $|\Sigma| \times 3$ and each transition has a random number of heads from 1 to $|\Sigma - 1|$, with Σ the number of automata of the generated ABAN. For traditional model checkers like NuSMV and Mole, memory-out cases begin to appear when $|\Sigma| > 50$, so the runtime results are not displayable in Fig. 9(a). Also, Pint is not able to check all test sets of any $|\Sigma|$. Even though the curve of average runtime of our approach shows an exponential-like tendency, ASPReach is very fast for $|\Sigma| < 100$ (less than 0.03s) and can also perform on models with 1000 automata. Moreover, the longest runtime among the test sets is less than 20s. Because we stop the computation if one reachability check takes more than 20s and we note it as timeout. This result suggests that the heuristic approach of ASPReach has a good performance in conclusiveness in general as the testing models are generated randomly.

Another test is on the different density with the same number of automata. In Fig. 9(b), we fixed $|\Sigma| = 20$ and vary the number of the transitions per automaton

(density) from 1 to 12. The runtime peak is at density 8, a possible explanation is that even the topology of the network is more complex with the growth of density, more available transitions lead to more pathways from the initial state to the target state, thus the heuristics may end with less trials.
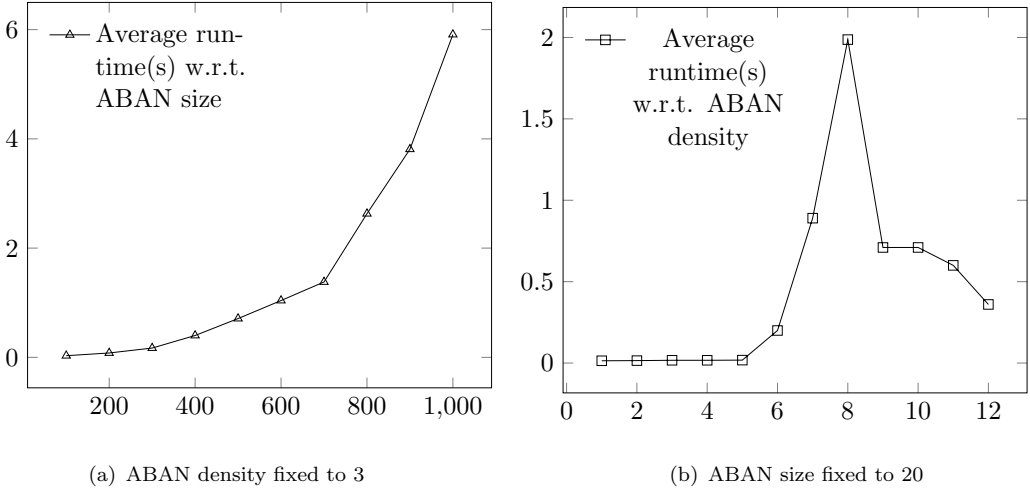

(a) ABAN density fixed to 3   (b) ABAN size fixed to 20

Fig. 9. Runtime of ASPReach on random ABANs

These experimental evaluations show that ASPReach has a better scalability on reachabilty analysis than traditional exhaustive model checkers and a better performance regarding conclusiveness than existing static analyzer.

5 Conclusion

In this paper, we present the ABAN modeling framework and its model-related reachability analyzer ASPReach. Facing the two critical challenges: complexity and conclusiveness, we combine static analysis and Answer Set Programming for a good performance on both criteria. ASPReach performs normally on models with 1000 automata while traditional model checkers fail to compute and static analyzer Pint also fails to give conclusive results on certain instances.

We are now considering an incorporation of heuristics in the preprocessing phase, like the work of [19], to improve the performance of our approach regarding runtime. The development of dedicated heuristic for the orientation of the random search in ASPReach remains also a future work.

As to biological application, one path of research is the interfacing of our approach with model inference approaches. Biological models can be inferred from experimental data through machine learning techniques (e.g. [17]), but obtained models are not necessarily consistent with the *a priori* background knowledge about the dynamics of system. Thus it is important to check these obtained models and revise them to make them consistent with such background knowledge. In the case of reachability properties, our approach could be use as a model checker and a mean

to enforce such properties in a model under construction.

References

[1] Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén. Symbolic reachability analysis based on sat-solvers. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 411–425. Springer, 2000.

[2] Tatsuya Akutsu, Morihiro Hayashida, Wai-Ki Ching, and Michael K Ng. Control of boolean networks: Hardness results and algorithms for tree structured networks. *Journal of theoretical biology*, 244(4):670–679, 2007.

[3] Chitta Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge university press, 2003.

[4] Alexander Bockmayr and Arnaud Courtois. Using hybrid concurrent constraint programming to model dynamic biological systems. In *International Conference on Logic Programming*, pages 85–99. Springer, 2002.

[5] Luca Bortolussi and Alberto Policriti. Modeling biological systems in stochastic concurrent constraint programming. *Constraints*, 13(1):66–90, 2008.

[6] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and computation*, 98(2):142–170, 1992.

[7] Xinwei Chai, Morgan Magnin, and Olivier Roux. A Heuristic for Reachability Problem in Asynchronous Binary Automata Networks, 2018. arXiv:1804.07543v1.

[8] Allan Cheng, Javier Esparza, and Jens Palsberg. Complexity results for 1-safe nets. *Theoretical Computer Science*, 147(1-2):117–136, 1995.

[9] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal methods in system design*, 19(1):7–34, 2001.

[10] Edmund M Clarke. The birth of model checking. In *25 Years of Model Checking*, pages 1–26. Springer, 2008.

[11] Edmund M Clarke and Qinsi Wang. 2^5 years of model checking. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 26–40. Springer, 2014.

[12] Conrado Daws and Stavros Tripakis. Model checking of real-time reachability properties using abstractions. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 313–329, 1998.

[13] Gilles Didier, Elisabeth Remy, and Claudine Chaouiya. Mapping multivalued onto boolean dynamics. *Journal of theoretical biology*, 270(1):177–184, 2011.

[14] Javier Esparza. Reachability in live and safe free-choice Petri nets is NP-complete. *Theoretical Computer Science*, 198(1-2):211–224, 1998.

[15] Maxime Folschette, Loïc Paulevé, Morgan Magnin, and Olivier Roux. Sufficient conditions for reachability in automata networks with priorities. *Theoretical Computer Science*, 608:66–83, 2015.

[16] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5. In *OASIcs-OpenAccess Series in Informatics*, volume 52. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[17] Katsumi Inoue, Tony Ribeiro, and Chiaki Sakama. Learning from interpretation transition. *Machine Learning*, 94(1):51–79, 2014.

[18] Stuart Kauffman. Homeostasis and differentiation in random genetic control networks. *Nature*, 224:177–178, 1969.

[19] Juraj Kolčák, David Šafránek, Stefan Haar, and Loïc Paulevé. Parameter Space Abstraction and Unfolding Semantics of Discrete Regulatory Networks. *Theoretical Computer Science*, 2018. In press.

[20] Haitao Li and Yuzhen Wang. On reachability and controllability of switched boolean control networks. *Automatica*, 48(11):2917–2922, 2012.

[21] Haitao Li, Yuzhen Wang, and Zhenbin Liu. Stability analysis for switched boolean networks under arbitrary switching signals. *IEEE Transactions on Automatic Control*, 59(7):1978–1982, 2014.

[22] Vivien Marx. Biology: The big challenges of big data. *Nature*, 498(7453):255–260, 2013.

[23] Ernst W Mayr. An algorithm for the general Petri net reachability problem. *SIAM Journal on computing*, 13(3):441–460, 1984.

[24] Peter Bro Miltersen, Jaikumar Radhakrishnan, and Ingo Wegener. On converting CNF to DNF. *Theoretical computer science*, 347(1-2):325–335, 2005.

[25] Loïc Paulevé. Reduction of qualitative models of biological networks for transient dynamics analysis. *IEEE/ACM transactions on computational biology and bioinformatics*, 2017.

[26] Loïc Paulevé, Morgan Magnin, and Olivier Roux. Refining dynamics of gene regulatory networks in a stochastic π-calculus framework. In *Transactions on computational systems biology xiii*, pages 171–191. Springer, 2011.

[27] Loïc Paulevé, Morgan Magnin, and Olivier Roux. Static analysis of biological regulatory networks dynamics using abstract interpretation. *Mathematical Structures in Computer Science*, 22(04):651–685, 2012.

[28] James L Peterson. Petri nets. *ACM Computing Surveys (CSUR)*, 9(3):223–252, 1977.

[29] Brigitte Plateau and Karim Atif. Stochastic automata network of modeling parallel systems. *IEEE transactions on software engineering*, 17(10):1093–1108, 1991.

[30] Assieh Saadatpour, István Albert, and Réka Albert. Attractor analysis of asynchronous boolean models of signal transduction networks. *Journal of theoretical biology*, 266(4):641–656, 2010.

[31] Julio Saez-Rodriguez, Luca Simeoni, Jonathan A Lindquist, Rebecca Hemenway, Ursula Bommhardt, Boerge Arndt, Utz-Uwe Haus, Robert Weismantel, Ernst D Gilles, Steffen Klamt, et al. A logical model provides insights into T cell receptor signaling. *PLoS computational biology*, 3(8):e163, 2007.

[32] Regina Samaga, Julio Saez-Rodriguez, Leonidas G Alexopoulos, Peter K Sorger, and Steffen Klamt. The logic of EGFR/ErbB signaling: theoretical properties and analysis of high-throughput data. *PLoS computational biology*, 5(8):e1000438, 2009.

[33] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[34] Denis Thieffry and René Thomas. Dynamical behaviour of biological regulatory networks—ii. immunity control in bacteriophage lambda. *Bulletin of mathematical biology*, 57(2):277–297, 1995.

[35] H Steven Wiley, Stanislav Y Shvartsman, and Douglas A Lauffenburger. Computational modeling of the EGF-receptor system: a paradigm for systems biology. *Trends in cell biology*, 13(1):43–50, 2003.

[36] Bożena Woźna, Andrzej Zbrzezny, and Wojciech Penczek. Checking reachability properties for timed automata via SAT. *Fundamenta Informaticae*, 55(2):223–241, 2003.

A   Transformation from BNs to ABANs

Given Boolean functions $v_i(t+1) = f_i(\mathbf{V}_i)$, with $\mathbf{V}_i$ the set of participating variables among $v_1(t), \cdots, v_n(t)$. Boolean functions could be transformed to equivalent CNF (conjunctive normal form) and DNF (disjunctive normal form) if the length of Boolean functions is limited to $O(1)$ [24] which is often the case.

Proposition A.1 (Transformation from BN to ABAN) *Given a BN $G_B = (V, F)$, with its functions in CNF form $v^i(t + 1) = A_1 \wedge \ldots A_j \ldots \wedge A_n$ and DNF form $v^i(t + 1) = A'_1 \vee \ldots A_k \ldots \vee A'_m$, an equivalent ABAN AB has transitions $A_j \rightarrow v^i_1$ and $\neg A_k \rightarrow v^i_0$ where A_j are disjunctions and A'_K are conjunctions.*

Example A.2 Let $G_B = (V, F)$ a BN with $V = \{a, b, c, d, e\}$, and has only one Boolean function, $F = \{f(a) = (b \vee c) \wedge (d \vee e)\}$, we have $f(a) = (b \wedge d) \vee (b \wedge e) \vee (c \wedge d) \vee (c \wedge e)$, and $\neg f(a) = (\neg b \wedge \neg c) \vee (\neg d \wedge \neg e)$. The equivalent ABAN is then constructed: 5 automata $\Sigma = \{a, b, c, d, e\}$, with transitions: $\mathbf{T} = \{\{b_1, d_1\} \rightarrow a_1, \{b_1, e_1\} \rightarrow a_1, \{c_1, d_1\} \rightarrow a_1, \{c_1, e_1\} \rightarrow a_1, \{b_0, c_0\} \rightarrow a_0, \{d_0, e_0\} \rightarrow a_0\}$.

B   Proofs

Theorem B.1 (ASPReach termination and correctness) *Let l $=$ $(V_{\text{state}}, V_{\text{solution}}, E)$ be an LCG with initial state α and target local state ω and $k > 0$ be an integer. The call ASPReach(l,k) terminates.*
*ASPReach(l, k)=(**False**, $\varnothing$) only if $\nexists t$ a trajectory in l from α to ω.*
*ASPReach(l, k)=(**True**, t) only if $\exists t$ a trajectory in l from α to ω.*

Proof.

1: The algorithm starts by breaking all cycles of the LCG and according to Theorem 2.11 it terminates and does not affect the reachability of α in l.

2: Then all nodes of V_{state} and (resp. solution) with no (resp. missing) outgoing edges are removed. Such nodes cannot be part of a trajectory leading to initial state α and thus this operation does not affect the reachability of α in l. The internal for loop of this operation iterates over V_{state} which is finite. To continue looping, it requires one state deletion thus this operation will terminate atleast when V_{state} becomes $\varnothing$.

*Conclusion 1: ASPReach$(l, k) = $ **False** only if $\nexists t$ a trajectory in l from α to $v \in V_{\text{solution}}$.*

Conclusion 2: the call ASPReach(l,k) terminates.

*Conclusion 3: After this pre-processing, pseudo reachability is checked and according to [27], it terminates and is correct. It is the only possibility for ASPReach to output **False**.*

Conclusion 4: Stochastic search follows by randomly reducing each OR gate of l to one of its edges to form l'. This operation is run a finite time k and iterates over V_{state} which is finite and thus it terminates. This operation does not create new edges, i.e. $E' \subseteq E$. ASPsolve(l') generates all possible trajectories of l' leading to α. The number of possible trajectory is finite and thus ASPsolve(l') terminates.

*Furthermore when ASPsolve$(l') = ($**True**, $t)$, t is a trajectory of l proving reachability of α in l and it is the only possibility for ASPReach to output **True**.*

*Conclusion 3: ASPReach$(l, k) = ($**True**, $t)$ only if $\exists t$ a trajectory in l from α to $v \in V_{\text{solution}}$.* □

Theorem B.2 (ASPReach complexity) *Let $l = (V_{\text{state}}, V_{\text{solution}}, E)$ be an LCG with initial state α and $k > 0$ be an integer. Let $s = |V_{\text{solution}}|$ be the number of target state of l. Let $v = |V_{\text{state}}|$ be the number of vertices of l. Let $e = |E|$ be the number of edges of l. The complexity of ASPReach(l, k) is $O(v + s + e + (v + s)/2 \times v \times e \times s + v^2 \times e + v \times e + k \times (v \times e^2 + 2^v))$ which is bound by $O(k \times 2^v)$.*

Proof.

1: computing $SCC(l)$ has a complexity of $O(v + s + e)$. In worst case $|SCC(l)| = (v + s)/2$ and breaking one cycle of $SCC(l)$ is $O(v \times e \times s)$, thus complexity of removing cycle is $op1 = O(v + e + s + (v + s)/2 \times v \times e \times s)$

2: To remove useless nodes ASPReach iterates over all states and checking if one state has no successor in l requires to iterates over all edges. In worst case

all states will be removed one by one and thus the complexity of this operation is $op2 = O(v \times (v + s) \times e)$.

3: Computing pseudo reachability over l which have no loop correspond to perform a depth first search on all branch of a tree and thus bound to $op3 = O((v + s) \times e)$.

4: the stochastic search iterates atmost k times. Treating each OR gate to form l' *have a cost of* $O(v \times e \times e)$ $ASPsolve(l')$ *generates trajectories that can prove reachability of* α *in* l'. *Each trajectory is a sequence where each element of* $|V_{\text{state}}|$ *appears exactly once. It correspond to the number of total order of* $|V_{\text{state}}|$ *which is* 2^v. *Thus* $ASPsolve(l')$ *is bound by* $O(2^v)$ *and the whole stochastic search by* $op4 = O(k \times (v \times e^2 + 2^v))$.

Conclusion 1: The complexity of $ASPReach(l, k)$ *is* $O(op1 + op2 + op3 + op4) = O(v + e + s + (v + s)/2 \times v \times e \times s + v \times (v + s) \times e + v \times e + k \times (v \times e^2 + 2^v))$.

Conclusion 2: The complexity of $ASPReach(l, k)$ *is bounded by* $O(k \times 2^v)$ □

C   Pseudo Code of Pseudo-reachability

Input: an LCG $l = (V_{\text{state}}, V_{\text{solution}}, E)$, an initial state α, a target state ω
Output: a Boolean $reach'$
procedure PSEUDOREACH(s)
    // If ω is in initial state, it is already reached
    if $\omega \in \alpha$ **then return True**
    // The reachability of s depends on its successor solution nodes
    if $\nexists (s, sol) \in E$ **then return False**

    for each $(s, sol) \in E$ **do**
        if FIREABLE(sol) **then return True**
    return False
procedure FIREABLE(sol)
    for each $(sol, s') \in E$ **do**
        if PSEUDOREACH(s') **then return False**
    return True

Algorithm 3. Pseudo-reachability $reach'$