Inference and Learning of Boolean Networks using Answer Set Programming

Alexandre Rocca, Tony Ribeiro, and Katsumi Inoue
National Institute of Informatics.
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan

Abstract. A Boolean Network is a compact mathematical representation of biological systems widely used in bioinformatics. However, in practice, experiments are usually not sufficient to infer a Boolean network which represents the whole biological system. Previous works relied on inferring and learning techniques to complete those models, or to learn new networks satisfying experimental properties represented as temporal logic properties. In this work, we use the Answer Set Programming (ASP), a highly expressive declarative language with fast solvers, to provide an efficient, and easily adaptable approach to learn/complete Boolean networks. We use the fast generation-constraint approach of the ASP, with temporal logic specifications, to learn and infer a minimal transition model of a Boolean network.

Keywords: Answer Set Programming, Boolean Network, Inference, Learning, Temporal Logic

1 Introduction

A Boolean Network (BN) is a compact mathematical representation widely used in bioinformatics [13–15]. Initially introduced to represent gene regulatory networks by [13], Boolean Networks have been used in many research fields to represent other Boolean interaction system such as electronic circuits [4] or social interaction models [10]. In recent years, there is a growing interest in the development of techniques for analysis and learning of Boolean networks.

Some works like [7], focus on finding cycle, i.e. attractors, in the behaviour of the system. Detecting attractors and their basins of attraction are very important for analysts to ensure non-time dependant property of a system. In the case of electronic circuits, analysis techniques can also be used to perform model checking: ensure that the system behaviour is correct. Some other works develop methods to construct a BN. In [12], the authors proposed a framework to learn the dynamics of a BN from the interpretation of its states transitions, but not from general expression like with temporal logics. In bioinformatics, learning the dynamics of a biological systems helps in identifying the influence of genes and designing more efficient drugs.

In this paper, we propose a model checking framework based on Answer Set Programming dedicated to Boolean Network. Answer set programming (ASP)
Inference and Learning of Boolean Networks using Answer Set Programming

\[9\] is a form of declarative programming that has been successfully used in many model-checking problems \([11, 1, 17]\). This framework allows us to check temporal logic properties against a Boolean Network represented in ASP. The temporal logic is an extension of the propositional logic that can describe properties on dynamical behaviours. In particular, we provide an ASP translation of the Linear Time Logic (LTL) \([16]\) and the Computational Tree Logic (CTL) \([6]\). To check those properties, we use well known model checking techniques similar to the ones used in the model checkers of \([2, 5]\).

The novelty of our model checking framework is the possibility to analyse and infer Boolean Network using ASP. Many model checkers have already been proposed for the analysis of Boolean network. The most similar to our framework are \([2]\), and \([5]\). However, these model checker rely on SAT and/or BDD approaches to solve the problem. Like \([3]\), our framework can complete an existing Boolean Network by ensuring temporal logic properties; where other work like \([15]\) (focusing on the Consistency Problem), complete or learn a BN by satisfying experimental properties. But, again, our framework use ASP whereas the approach in \([3]\) uses SAT and BDD approaches. ASP takes advantage of the expressiveness of first order logic and high performance solvers like clasp \([8]\) make it an interesting alternative to SAT/BDD-based approaches\(^1\). If there are some previous work about model-checking using ASP, to the best of our knowledge, none of them consider both analysis, construction and completion of Boolean Networks over LTL/CTL properties.

2 Preliminary

2.1 Boolean Network

A Boolean network (BN) \([13]\) is a pair \((N,F)\) with \(N = \{n_1, \ldots, n_k\}\) a finite set of nodes and \(F = \{f_1, \ldots, f_k\}\) a corresponding set of Boolean functions. In the case of a gene regulatory network, nodes represent genes and Boolean function represent their relations. If \(n_i(t)\) represents the value of \(n_i\) at the time \(t\) of computation, then \(n_i\) takes either 1 (expressed) or 0 (not expressed). A vector (or state) \(s(t) = (n_1(t), \ldots, n_k(t))\) is the expression of the nodes in \(N\) at time step \(t\). There are \(2^k\) possible states for each time step. The state of a node \(n_i\) at the next time step \(t + 1\) is determined by \(n_i(t + 1) = f_i(n_{i_1}(t), \ldots, n_{i_p}(t))\), with \(n_{i_1}, \ldots, n_{i_p}\) the nodes directly influencing \(n_i\), and also called regulation nodes of \(n_i\). Boolean networks can be represented by three different ways: the interaction graph (see Fig. 1), the written diagram which represents the transitions between \(n_i(t)\) and \(n_i(t + 1)\), and the truth table. From the truth table we can create the state-transitions diagram. The value of nodes can be updated synchronously, or asynchronously. A Synchronous Boolean network (SBN) is a network where all the nodes are updated at the same time. The successive sequence of states during an execution, called trajectory of a BN, or path, is deterministic in a SBN. An Asynchronous Boolean network (ABN) is a network where one node may be

\(^1\) If we compare with the work on qualitative models
updated at given time time. A ABN path can be non deterministic.
One of the interesting properties of the Boolean network is the attractors. Given

\[
p(t+1) = p(t) \lor q(t)
q(t+1) = \neg p(t)
\]

Fig. 1. Example of Boolean network

a set \( S = (s_1, ..., s_n) \), and a reachability function \( R \). Then \( R(s_i) \) are the reachable states from any path starting from \( s_i \). Then, \( S \) is an attractor if for any state \( s_i \in S \), \( R(s_i) = S \). Attractors represent the stable states of a Boolean network, and describe a stability in the behaviour.

### 2.2 Temporal Logic

In model checking, a model is described by a Kripke structure. A Kripke structure is \( M(S, I, T, L) \), with \( S \) a set of states, \( I \subseteq S \) a set of initial states, \( T \subseteq S \times S \) the transition relations, and \( L: S \rightarrow P(A) \) a labelling function, with \( A \) the set of atomic propositions and \( P(A) \) the powerset of \( A \). For each state \( s \in S \), \( L(s) \) is the set of atomic propositions which are true in \( s \). The behaviour of \( M \) is defined by paths. A path \( p \) of \( M \) is a succession of states \( (s_0, s_1, ...) \), where \( s_i \in S \) and \( T(s_i, s_{i+1}) \) holds for all \( i \geq 0 \). The \( i \)-th state of a path is denoted \( p(i) \).

The temporal logic is an extension of the propositional logic, to describe properties of a system. First the \textit{Linear Temporal Logic} (LTL) is defined as follow:

\[
\varphi ::= a \in A | \neg \varphi | \varphi_1 \land \varphi_2 | \varphi_1 \lor \varphi_2 | G \varphi | F \varphi | X \varphi | \varphi_1 U \varphi_2 | R \varphi_1 \varphi_2 | \Rightarrow
\]

\[
p \models a \text{ iff } a \in L(p(0))
\]

\[
p \models \neg \varphi \text{ iff } p \not\models \varphi
\]

\[
p \models \varphi_1 \land \varphi_2 \text{ iff } p \models \varphi_1 \text{ and } p \models \varphi_2
\]

\[
p \models G \varphi \text{ iff } p(i) \models \varphi \forall i \geq 0
\]

\[
p \models X \varphi \text{ iff } p(1) \models \varphi
\]

\[
p \models \varphi_1 U \varphi_2 \text{ iff } \exists i \geq 0 \text{ s.t. } p(i) \models \varphi_2 \text{ and } \forall 0 \leq k < i \text{ s.t. } p(k) \models \varphi_1
\]

From these formulas, we can build any other LTL formulas:

\[
p \models \varphi_1 \Rightarrow \varphi_2 \text{ iff } p \models \neg(\varphi_1 \land \neg \varphi_2)
\]

\[
p \models F \varphi \text{ iff } p \models P U \varphi, \text{ and } p \models \varphi_1 R \varphi_2 \text{ iff } p \models \neg \varphi_1 U \neg \varphi_2
\]

We note that verifying a property on a given path \( p \) is equivalent to verifying the property on the initial state of the path.

The computational Tree Logic (CTL) is an extension of propositional logic to describe properties on a branching time behaviour. Like in the LTL description, we use a Kripke model to describe the system. We can separate the CTL operators in two classes: the Global operators with a \( A \), and the Existential operators...
with a $E$. If the LTL describes properties on paths, CTL does it on set of path. The CTL syntax is the following:

$$\varphi ::= a \in A | \neg \varphi | \varphi_1 \land \varphi_2 | \varphi_1 \lor \varphi_2 | EG \varphi | E \varphi_1 U \varphi_2 | AX \varphi | EF \varphi | AG \varphi | A \varphi_1 U \varphi_2 | AX \varphi | AF \varphi$$

For the description of the properties, the common part with the LTL ($p, \neg, \land, \lor$) will not be explicated again. For $M = (S, I, T, L)$ a Kripke model and $s \in S$:

$$(M, s) \models EG \varphi \text{ iff } \exists \text{ a path } p \mid p(0) = s \text{ and } \forall 0 \leq i (M, s_i = p(i)) \models \varphi$$

$$(M, s) \models E \varphi_1 U \varphi_2 \text{ iff } \exists \text{ a path } p \mid p(0) = s \text{ and } \exists i \geq 0 (M, s_i = p(i)) \models \varphi_1 \text{ and } \forall 0 \leq k \leq i (M, s_k = p(k)) \models \varphi_2$$

$$(M, s) \models EX \varphi \text{ iff } \exists \text{ a path } p \mid p(0) = s \text{ and } (M, s_1 = p(1)) \models \varphi$$

Same as before, the other CTL formulas can be defined from those three:

$$(M, s) \models AG \varphi \text{ iff } (M, s) \models \neg EF \neg \varphi$$

$$(M, s) \models A \varphi_1 U \varphi_2 \text{ iff } (M, s) \models \neg (E (\neg \varphi_1 U (\neg \varphi_1 \land \varphi_2)) \land \neg EG (\neg \varphi_2))$$

$$(M, s) \models AX \varphi \text{ iff } (M, s) \models (M, s) \models \neg EX \neg \varphi$$

$$(M, s) \models AF \varphi \text{ iff } (M, s) \models (M, s) \models \neg EG \neg \varphi$$

### 3 Inferring a non complete Boolean network

Boolean networks constructed from real life observations are often incomplete, especially in biology: there is often interactions between two genes (represented by two Boolean nodes) that are unknown, or ambiguous.

In this section, we first focus on how to complete a Boolean network thanks to some experimental data expressed as temporal logic formulas. The Boolean network given as input can be synchronous, or asynchronous, and the temporal logics used will be the CTL and the LTL.

The number of possible completed network of an incomplete Boolean network correspond to the number of possible behaviours of this network. There is at most $n$ ambiguous interactions (if there is $n$ nodes) per node and each ambiguous interactions can be either an activation, an inhibition or with no effect so that in the worst case there is $n^3$ possible behaviours. In the first BN of Figure 2, the influence of $x_2$ on $x_3$ is unknown so that there should be $3 \times 3^3 = 9$ possibilities. But here the influence of $x_1$ on $x_3$ and $x_3$ among itself is partially known and the number of possibilities is only 5. In the second BN of Figure 3, we know that $x_2$ has an influence on $x_3$ according to $x_1$, so that the number of possibilities is only 3. We can either choose to complete the interaction graph, or the boolean functions, in both case the reasoning is the same.

In the following sections, we propose a method which combines ASP with model-checking techniques to compute the complete models of an ambiguous
BN, keeping only the ones with a behaviour satisfying a set of LTL, or CTL, formulas. The techniques we use can be divided in two parts: bounded and non-bounded model-checking. The bounded model-checking consists on finite computations: the temporal logic formulas are checked for a limited number of steps and/or a limited run time. For the non-bounded model-checking, the temporal logic formulas are checked for a potentially infinite computation. The following, describes both techniques and their use for the inference: starting by the LTL (Section 3.1), followed by the CTL (Section 3.2).

### 3.1 Inference and LTL model-checking in ASP

LTL model-checking verifies properties on a linear path. This particularity gives an interesting property to the states:

**Property 1.** If \( s_1 \rightarrow \ldots \rightarrow s_n \) is a linear path, i.e., \( \{s_1, \ldots, s_n\} \) is a set of states, and \( s_i \) is the state generated at the step \( i \). If \( s_1, \ldots, s_n \) are all distinct (\( s_1 \neq s_2 \neq \ldots \neq s_n \)), then the time \( t \) of the state generation becomes an unique identifier of a state.

The principle of the LTL translation in ASP, is to use, at the maximum, this special case of equivalence between state and generation time. In fact, from a given initial state, we generate a path until we find a loop, or until the time bound is reached in bounded model-checking (proof in annexe). We use the wanted LTL properties as constraints on the path generation, so that each answer set is a possible combination of interactions that validates the LTL properties (to reduce the run time we can ask for a limited number of answer set).

**Example 1.** An ASP program which uses our translation to check the possibilities of complete networks of the second BN of Figure 2. Here we add a constraint which states that starting from \( (110) \): \( x_3 \) should not be true in the future. The ASP program will output one answer set where \( x_2 \) inhibits \( x_3 \).

\[ \text{time limit} \]
Translation of LTL in ASP

See Section 2.2 for the formal definition.

%Definition of the atomic formulas.
phi(T) :- x_i(T).
not_phi(T) :- not x_i(T).

%phi1 and phi2 can also be two LTL sub-formulas.
phi1_and_phi2(T) :- phi1(T), phi2(T).
phi1_or_phi2(T) :- phi1(T).
phi1_or_phi2(T) :- phi2(T).
phi1_Imply_phi2(T) :- not phi1_and_not_phi2(T).

Xphi(T) :- phi(T+1), t(T+1).

%With tmax the bound of the steps.
Xphi(T) :- T==Tmax, loop(T_,Tmax), Xphi(T_).
phi1_U_phi2(T) :- phi2(T).
phi1_U_phi2(T) :- phi1(T), phi1_U_phi2(T+1), t(T+1).
phi1_U_phi2(T) :- t(T), T==Tmax, t(T_), loop(T_,Tmax), phi1_U_phi2(T_).

%The other formulas are given by:
Gphi(T) :- not Fnotphi(T).
Fphi(T) :- True_U_phi(T).
phi1_R_phi2(T) :- not not_varphi1_U_not_phi2(T).

t(0..8).
%initial state: variable(0|1,t), here (110)
x1(1,0).
x2(1,0).
x3(0,0).

%Incertitude on the interaction of x2 on x3
x2activateX3 :- not x2inhibitX3, not x2noeffectonX3.
x2inhibitX3 :- not x2activateX3, not x2noeffectonX3.
x2noeffectonX3 :- not x2inhibitX3, not x2activateX3.

% Transitions rules of the Boolean network
x3(1,T+1) :- x1(1,T), x2(1,T), x2activateX3, t(T).
x3(1,T+1) :- x1(1,T), x2(0,T), x2inhibitX3, t(T).
x3(1,T+1) :- x1(1,T), x2noeffectonX3, t(T).
x1(0,T) :- not x1(1,T), t(T).
x2(0,T) :- not x2(1,T), t(T).
x3(0,T) :- not x3(1,T), t(T).

loop(T,T_) :- t(T), t(T_), T<T_, x1(X1,T), x2(X2,T), x3(X3,T),
x1(X1,T_), x2(X2,T_), x3(X3,T_).

%constraints
fx3(T) :- x3(1,T).
fx3(T) :- fx3(T+1), t(T).
f3(T) :- t(T), T==8, t(T_), loop(T_,8), fx3(T_).
:- fx3(0).

3.2 Inference and CTL model-checking in ASP

For the CTL properties, unlike in LTL, there is no identification between a state and a step. For this reason, a translation similar to the LTL cannot be provided for all the CTL formulas.

The tree computation can be divided in n different paths from the initial state to a leaf. On each path, it is possible to check fully existential properties (no sub formulas Aϕ) as checking LTL formulas on a path (see proof in annexe): for example Fϕ becomes EFϕ. In fact, during the computation of the tree we equate an answer set to a path satisfying all the fully existential properties.

In the other case, we need the transition system to check the CTL properties.
In bounded model-checking, the program generates all the paths of fixed length k from the top to the leaves of the tree. Then it reconstructs the corresponding explored part of the state-transition model. Finally, it verifies if the model satisfies or not the CTL formulas. As we can see to find a complete model, there is one execution of the programs for each possible set of interactions.

In non bounded model-checking, we generate directly the whole states-transitions model of a given Boolean network in an answer set. Then we can directly constraint the generated model with the satisfaction of the CTL formulas. The generation cost of the transition system is exponential (we need to generate at least one transition by state so $2^n$).

### Translation of CTL in ASP

See Section 2.2 for the formal definition.

% $\phi$ is a CTL formulas.
% $\phi$ can be a atomic formulas $\phi$ as $\phi_i(S)$ or not_ $\phi_i(S)$, $i$ in [1..n]

\[ \text{phi}(S) :- \text{state}(x_1,..,x_i=1,..,x_n) = S. \]
\[ \text{notphi}(S) :- \text{state}(x_1,..,x_i=0,..,x_n) = S. \]
% $\phi_1$ and $\phi_2$ can also be two CTL sub-formulas.

\[ \text{phi1_and_phi2}(S) :- \phi_1(S), \phi_2(S). \]
\[ \phi_1_or_phi2(S) :- \phi_1(S). \]
\[ \phi_1_or_phi2(S) :- \phi_2(S). \]
\[ \text{not_phi}(S) :- \text{not phi}(S), \text{state}(S). \]
\[ \phi_1_{\text{Imply}}\phi_2(S) :- \text{not phi1_and_notphi_2}(S), \text{state}(S). \]

% transition($S$, $S'$) means there is a transition from $S$ to $S'$.

\[ \text{EXphi}(S) :- \phi(S'), \text{transition}(S,S'). \]
\[ \text{Ephi1_U_phi2}(S) :- \phi_2(S). \]
\[ \text{Ephi1_U_phi2}(S) :- \phi_1(S), \text{Ephi1_U_phi2}(S'), \text{transition}(S,S'). \]
% For $\text{EG}(S)$ we need to cut the loop inside the formulas.

\[ \text{A naive translation would be:} \]
\[ \text{EGphi}(S) :- \phi(S), \text{EGphi}(S'), \text{transition}(S,S'). \]
% However we can easily see that there is a circular dependence if there is a loop.

\[ \text{Then we add hypothesis on EGphi:} \]
\[ \text{hypoTrueEGphi}(S) :- \phi(S), \text{not hypoFalseEGphi}(S). \]
\[ \text{hypoFalseEGphi}(S) :- \text{not hypoTrueEGphi}(S). \]
\[ \text{EGphi}(S) :- \phi(S), \text{hypoTrueEGphi}(S'), \text{transition}(S,S'). \]
% Finally we must add constraints to eliminate the false hypothesis.
\[ :- \text{hypoTrueEGphi}(S), \text{not EGphi}(S), \text{state}(S). \]
% From those formulas any others CTL formulas can be define.

\[ \text{AXphi}(S) :- \text{not EXnotphi}(S). \]
\[ \text{AGphi}(S) :- \text{not EPnotphi}(S). \]
\[ \text{AYphi}(S) :- \text{not EDnotphi}(S). \]
\[ \text{EFphi}(S) :- \text{true_U_phi}(S). \]
\[ \text{Aphi1_or_phi2}(S) :- \text{not E(notphi1_U(not phi1_and_notphi2))}, \text{not EDnotphi2}(S). \]

## 4 Learning Boolean networks

In some case there is no existing models and we need to learn a model from experimental data, or constraints, given as temporal logic properties. In these section, we show how to construct from scratch a Boolean network satisfying
a given set of LTL properties. To learn this network, we will generate a possible execution of the Boolean network consistent with the LTL properties. The techniques is similar to the inference problem, but we only focus on generating correct trajectories. A valid execution is defined as follows:

**Definition 1.** If $p$ is a path of length $k$, i.e., a set of ordered states $s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_k$, $p$ is a valid execution of a synchronous deterministic Boolean network, if all the states $(s_1, \ldots, s_{k-1})$ are distinct. This means there is no loop between the step 1 and $k-1$, because the execution is synchronous and deterministic.

**Example 2.** If the length of the paths is $k = 4$, then the following paths are valid execution: (000) → (010) → (011) → (110) or (000) → (010) → (011) → (011)

However, the following path is not a valid execution because of the (010) → (010) loop: (000) → (010) → (010) → (110).

Here the problem is to find all valid executions satisfying the whole set of constraints. However, some set of constraints can be inconsistent. The inconsistency can be distinct in two case: two constraints can be inconsistent on a common path, we will call them path-inconsistent (resp. path-consistent), like in example 3. Or they can be inconsistent on the whole state-transition graph, like: the next state of (100) is (000) and the next state of (100) is (111). To determine this path-consistency, we can check all the possible combinations of the constraints until we find one with a satisfying model. In theory, we would generate all the possible combinations of the sets of constraints. In practice, we consider those sets of constraints given as input, because one set is the result of one experience.

**Example 3.** In the following example, we will learn a Boolean network with 3 nodes (p, q, r) from the following LTL properties: $FGqr(000)$: in the future of (000) there is a state $s$, with $Gqr(s)$. $Gqr(s)$ means that all states in the future of $s$ will verify q is true and r is true (so the states (011) or (111)).

$X(110)(110)$: (110) is the immediate next state of (110).

$X(101)(001)$: (101) is the immediate next state of (001).

$F(111)(001)$: (111) is reachable in the future of (001).

We can see that $FGqr(000)$ and $X(110)(110)$ are not path-consistent, because $X(110)(110)$ imply a loop on (110) and $FGqr$ imply a loop with (111) and/or (011): this is not possible in one valid execution, we can divide the constraint in two set.

Arbitrary we use the following division: $\{FGqr(000), X(101)(001), F(111)(001)\}$ on one part, and $\{X(110)(110)\}$ on the other part.

Then the program gives a minimal valid execution for the first set of constraints: (001) → (101) → (000) → (111) → (111).

And for the other set: (110) → (110).

The transitions described by these two valid executions are consistent. But they do not represent all the possible transitions of the system: only the ones useful to
satisfy the two sets of constraints. To retrieve the rules of the Boolean function, the valid executions can given as input to the algorithm described in [12]. This algorithm will compute the rules defining the Boolean network and will complete the Boolean function by putting default transition to (000) for the states without explicit transition. The state transitions corresponding to the learned Boolean network can be seen in Figure 3.

Example 4. An ASP program which uses our translation to solve the problem of Example 3. Here we search for a path of length at most 4. This program will output multiple answer sets, and one of them corresponds to the state-transitions graph of Figure 3.

\[
\begin{align*}
\text{t(0..4).} \\
\text{p(1,T) :- not p(0,T), t(T).} \\
p(0,T) :- not p(1,T), t(T). \\
q(1,T) :- not q(0,T), t(T). \\
q(0,T) :- not q(1,T), t(T). \\
r(1,T) :- not r(0,T), t(T). \\
r(0,T) :- not r(1,T), t(T). \\
\text{xp qr(T)} :- r(1,T), p(1,T), q(0,T), t(T). \\
qr(T) :- q(1,T), r(1,T). \\
not qr(T) :- not qr(T). \\
f not qr(T) :- f not qr(T), t(T). \\
g qr(T) :- not f qr(T), t(T). \\
f not qr(T) :- f not qr(T). \\
p qr(T) :- p(1,T), q(1,T), r(1,T). \\
f qr(T) :- f qr(T), t(T). \\
\text{properties} \\
\text{property 1 :- fG qr(T), p(0,T), q(0,T), r(0,T).} \\
\text{property 2 :- f pq qr(T), p(0,T), q(0,T), r(1,T).} \\
\text{property 3 :- xp qr(T), p(0,T), q(0,T), r(1,T).} \\
\text{property 4 :- p(X,T), q(Y,T), r(Z,T), p(X,4), q(Y,4), r(Z,4), t(T), T < 4.} \\
\text{inside loop} \\
\text{fail :- p(X,T), q(Y,T), r(Z,T), p(X,4), q(Y,4), r(Z,4), t(T), T < 4.} \\
\text{succeed :- property 1, property 2, property 3, property 4.} \\
\text{fail :- not succeed.} \\
\end{align*}
\]

5 Experiments

In this section we evaluate the performance of our method on some Boolean Networks benchmarks from the bioinformatics literature. These benchmarks are Boolean networks taken from Dubrova and Teslenko [7], which include those networks for control of over morphogenesis in Arabidopsis thaliana, budding yeast cell cycle regulation, fission yeast cell cycle regulation and mammalian cell cycle regulation. The experiments were done on a processor intel core i7 720QM 1.6GHZ, with 4Gb of RAM. The ASP solver used in these experiments is clingo 3.05 win64 [8].
The main cost of the inferring method is the model-checking of the possible networks. The tested networks are complete biological Boolean networks where some interactions are made ambiguous. We used 6 models of 10, 15, 19, 23, and 40 nodes. The experiments have been divided in two parts: the model-checking of asynchronous and synchronous version of the Boolean networks.

**Asynchronous Boolean network**

Table 1 shows the run time for the creation of the full transition system of a Boolean network and its non bounded model-checking. The runtime (grounding + solving) is exponential: according to the definition of asynchrony there is \( n^{2^n} \) possible state transitions for a complete asynchronous BN of \( n \) nodes. However, once this generation has been done, we can easily check any other CTL formulas.

<table>
<thead>
<tr>
<th>nodes</th>
<th>10</th>
<th>15</th>
<th>19</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime(s)</td>
<td>1.1s</td>
<td>181s</td>
<td>out of memory</td>
<td>out of memory</td>
</tr>
</tbody>
</table>

**Table 1.** Not bounded model-checking of Asynchronous Boolean network

For the bounded model-checking (see Fig. 4) we evaluate the maximum depth of exploration in one minute. Bounded model-checking is often used for searching counter example, and in our case for validating existential constraints. Those reachability problems mainly depend of the maximum depth of the search.

**Synchronous Boolean network**

![Fig. 4. Model-checking of synchronous Boolean network](image)

\(^2\) The 19 nodes model is the test example small_cell_cycle used by BIOCHAM
In the synchronous case, non bounded and bounded model-checking differ only by the loop detection. We place ourself in the worst case, and we will check the loop of size 1 (unary attractor). We can see that the max depth search (see Fig 5) decreases with the number of nodes in most of the cases. However, for the 40 nodes Boolean network, the depth is far greater. In this case the rules describing the network are very simple (often one atom in the body), and the computation greatly depends on it.

![Fig. 5. Bounded model-checking of Asynchronous Boolean network](image)

Finally, the computation of the Boolean networks can be optimised to use the full potential of the ASP. Moreover, the latest ASP syntax used in gringo 4.01 has shown far greater performance on the transition generation of the 15 nodes asynchronous network (75% less memory consumption, and runtime divided by 9). An optimised version of this work for new ASP syntax should approach the classical model checker performances, with greater expression power and adaptability.

6 Conclusion

In this work we developed techniques to infer and learn, in ASP, a Boolean network from temporal logic constraints. We gave a translation of CTL and LTL in ASP for Boolean network. These translations allow to use the expression power of the temporal logic to complete a Boolean network. The translations also allow to analyse a Boolean network with model-checking techniques. If the current learning and inferring experiments do not compete with the SAT/BDD softwares, results suggest that an ASP implementation using the new ASP definition, can reach the similar performance than the current software, with a greater adaptability and expression power. Moreover, the easy description of BN analysis, and the adaptable implementation of other methods, or temporal logics, confirms the ASP as a good programming language for the analysis of the fast evolving domain of biological models.
References


Annexes

Annexe A: Validity of the LTL translation in ASP

Given a n nodes \((x_1, \ldots, x_n)\) Boolean network generated on \(k\) steps, \(k \in \mathbb{N}^*\). Then \(s(t)=(x_1(t), \ldots, x_n(t))\) is the state of the Boolean network at step \(t\), and the generated path \(p\), from \(t=0\) to \(t=k\), can be written \(p=(s(0),\ldots,s(k))\).

Remark 1. If \(\varphi\) is a LTL property, then \(p \models \varphi\) is equivalent to \(p(0) \models \varphi\) (see Section 2.2). By the same way, we note that for \(0 \leq i \leq k\), \(p(i) \models \varphi\) means there is a path \(p_i\) with \(p(i)=p_i(0)\) and \(p_i \models \varphi\).

As explained in Section 3.1, we compute the network until we find a loop, or until a given maximum runtime if there is no loop. The result of the computation can be divided in two possibility: there is no loop, and there is a loop with the last state.

In the first case, all the states of the path are distinct, and the property. 1 (see Section 3.1) can be applied: the state \(s(t)\) can be identified by the generation step \(t\). Thanks to the remark 1 we can write: for each state \(p(i) \in p\), \(0 \leq i \leq k\) \(p(i) \models \varphi\) means \(i \models \varphi\), which can be contracted in \(\varphi(i)\).

With those notations, a state \(p(i) \models X\varphi\) iff \(p(i+1) \models \varphi\) can be written \(X\varphi(i)\) iff \(\varphi(i+1)\). It is exactly the ASP translation \(X\varphi(t) :- \varphi(t+1)\). By the same way, the operator \(U\) is correctly translated in ASP.

In the second case, we need to manage the loop and the non equivalence step/state. The loop only appears at the last step \(k\). For \(0 \leq i \leq k-1\) we have \(s(0) \neq \ldots \neq s(i) \neq \ldots \neq s(k-1)\): We can apply the property 1 and the asp translation. For \(s(k)\) we transmit the property since the behaviour have already been verified in the \(k\) previous steps. This transmission is given by the following rule in ASP, for example with \(X\varphi\): \(X\phi(T) :- T==T_{\text{max}}, \text{loop}(T_{\text{min}},T_{\text{max}}), X\phi(T_{\text{min}})\), which means that \(T_{\text{max}} \models X\varphi\) if there is loop between \(T_{\text{min}}\) and \(T_{\text{max}}\) (\(s(T_{\text{min}})=s(T_{\text{max}})\)) and \(T_{\text{min}} \models X\varphi\).

Annexe B: Validity of the CTL translation in ASP

As seen in Section 2.2, all the CTL formulas can be describe with only \(EX\varphi\), \(E\varphi_1 U \varphi_2\) and \(EG\varphi\).

If \(s\) is a state, and \(M\) a Kripke model of the Boolean network as defined in Section 2.2. The prefix \(E\) means there are existential properties: there exits a path were the property will holds. The existence of a path \(p=(s,s_1,\ldots,s_n)\) is equivalent to the existence of transitions \(T(s_i,s_{i+1})\) with \(0 \leq i \leq n-1\). From those observation we can directly translate the 3 properties:

\((M,s) \models EX\varphi\) iff \(\exists\) a path \(p\mid p(0)=s\) and \((M,s_1=p(1)) \models \varphi\). If the Kripke model become implicit, this properties can be written \(EX\varphi(s)\) iff \(\exists\) a transition \(T(s,s_1)\) and \(\varphi(s_1)\) which is equivalent to the ASP translation \(EX\phi(S) :- \neg\phi(S'), \text{transition}(S,S')\), with \(S\), and \(S'\) two states.
\[(M, s) \models E\varphi_1 U\varphi_2 \text{ iff } \exists \text{ a path } p \mid p(0) = s \text{ and } \exists i \geq 0 \mid (M, s_i = p(i)) \models \varphi_2 \text{ and } \forall 0 \leq k \leq i \ (M, s_k = p(k)) \models \varphi_1. \]

Again we can contract the formulas in:

\[E\varphi_1 U\varphi_2(s) \text{ iff } \exists \text{ a set of transition } T = \{(s, s_1), \ldots, (s_k, s_{k+1}), \ldots, (s_{i-1}, s_i) \mid \varphi_2(s_i), \text{ and } \forall 0 \leq k \leq i, \varphi_1(s_k) \}.\]

Finally, \((M, s) \models EG\varphi \text{ iff } \exists \text{ a path } p \mid p(0) = s \text{ and } \forall 0 \leq i \ (M, s_i = p(i)) \models \varphi \text{ can be reduced in:}\]

\[EG\varphi(s) \text{ iff } \exists \text{ a set of transition } T = \{(s, s_1), \ldots, (s_k, s_{k+1}), \ldots, (s_{i-1}, s_i) \mid \forall 0 \leq k \leq i, \varphi(s_k) \text{ is true.}\]

In a finite path \(p=(s_1, \ldots, s_n), \forall s_i \in p, i < n : EG\varphi(s_i) \text{ is true if } \varphi(s_i) \text{ is true and } \exists T(s_i, s_{i+1}) \text{ with EG}\varphi(s_{i+1}). \text{ Then } EG\varphi(s_n) \text{ is true if } \varphi(s_n) \text{ is true.}\]

In a non finite path \(p=(s_1, \ldots), \text{ The property } EG\varphi(s) \text{ holds iff there is a loop in the path, with } \varphi(s_i) \text{ true for all } i \in [1..k], \text{ and for this reason we cannot translate directly the property. To check this property we make the assumption that if } \varphi \text{ holds then } EG\varphi \text{ can be true. However, we are sure that if } \varphi \text{ is false then } EG\varphi \text{ will be false. This will create a two possible answer sets for each state where } \varphi \text{ holds. We define } EG\varphi(s) \text{ true if } \varphi(s) \text{ is true and there is a transition } t(s, s') \text{ with hypEG}\varphi(s') \text{ true. All we need now is to add a constraint to eliminate the wrong assumption: } :- hyp\text{TrueEGphi}(S), \text{ not } EG\text{phi}(S), \text{ state}(S).\]

If the property is true, only remains the answer set with hypTrueEGphi(s) and EGphi(S) true for all states: this is the case were all the states verify \(\varphi\), this is what we wanted.

If the property is false, there is only the answer set with hypfalseEGphi(s) for all states, and then no state with EGphi(S), this what we wanted.