Scalable Enumeration of Trap Spaces in Boolean Networks via Answer Set Programming

Giang Trinh, Belaid Benhamou, Samuel Pastva, Sylvain Soliman

▶ **To cite this version:**

Giang Trinh, Belaid Benhamou, Samuel Pastva, Sylvain Soliman. Scalable Enumeration of Trap Spaces in Boolean Networks via Answer Set Programming. AAAI 2024 - The 38th Annual Conference on Artificial Intelligence, Feb 2024, Vancouver, Canada. pp.10714-10722, 10.1609/aaai.v38i9.28943 . hal-04523118

HAL Id: hal-04523118
https://amu.hal.science/hal-04523118

Submitted on 27 Mar 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Scalable Enumeration of Trap Spaces in Boolean Networks via Answer Set Programming

Giang Trinh[1], **Belaid Benhamou**[1], **Samuel Pastva**[2], **Sylvain Soliman**[3]
[1]LIRICA team, LIS, Aix-Marseille University, Marseille, France
[2]Institute of Science and Technology, Klosterneuburg, Austria
[3]Lifeware team, Inria Saclay, Palaiseau, France
trinh.van-giang@lis-lab.fr, belaid.benhamou@lis-lab.fr, samuel.pastva@ist.ac.at, Sylvain.Soliman@inria.fr

Abstract

Boolean Networks (BNs) are widely used as a modeling formalism in several domains, notably systems biology and computer science. A fundamental problem in BN analysis is the enumeration of trap spaces, which are hypercubes in the state space that cannot be escaped once entered. Several methods have been proposed for enumerating trap spaces, however they often suffer from scalability and efficiency issues, particularly for large and complex models. To our knowledge, the most efficient and recent methods for the trap space enumeration all rely on Answer Set Programming (ASP), which has been widely applied to the analysis of BNs. Motivated by these considerations, our work proposes a new method for enumerating trap spaces in BNs using ASP. We evaluate the method on a mix of 250+ real-world and 400+ randomly generated BNs, showing that it enables analysis of models beyond the capabilities of existing tools (namely `pyboolnet`, `mpbn`, `trappist`, and `trapmvn`).

Introduction

Boolean Networks (BNs) are a simple yet efficient mathematical formalism with many applications in various areas from science to engineering (Schwab et al. 2020; Rozum et al. 2021b). A BN includes n nodes, s.t. each node is associated with a Boolean *variable* and with a Boolean *function* that determines the value update of this node over time. In systems biology, BNs are used to model complex biological phenomena (Glass and Kauffman 1973; Thomas 1973). The utility of BNs has been demonstrated primarily in cases where not enough quantitative biological data is available (Wang, Saadatpour, and Albert 2012). Here, BNs provide explainable and executable abstractions of large-scale processes (including, but not limited to gene regulation and cell signalling). This leads to an ever-increasing complexity of network topology and Boolean functions in logical models *à la* Thomas (Aghamiri et al. 2020), as new qualitative genomics datasets are continuously being measured and integrated into existing models.

The main component of BN analysis is the computation of attractors: minimal sets of states that cannot be escaped once entered. These represent the long-term behaviour of a BN and are linked to observable biological *phenotypes* (Glass and Kauffman 1973; Wang, Saadatpour, and Albert 2012). Attractor analysis can provide insights into the origin of diseases (e.g., SARS-CoV-2, Alzheimer, and cancers) or play a role in systems medicine (Schwab et al. 2020; Park et al. 2021). However, exact attractor enumeration is challenging due to the complex dynamics of large-scale models.

Recently, the concept of trap spaces (Klarner, Bockmayr, and Siebert 2015) was introduced as a more computationally viable alternative to attractors. A trap space is a hypercube in the state space of a BN that cannot be escaped once entered. Every such minimal hypercube thus approximates one or more attractors. Not only are minimal trap spaces a good approximation of attractors in practice, but the much improved tractability of this problem also enabled analysis of a significantly larger class of Boolean models. This makes minimal (and other) trap spaces popular in many biological applications such as control (cell reprogramming) and model inference (Chevalier et al. 2019; Paulevé et al. 2020; Rozum et al. 2021b; Trinh et al. 2022; Richard and Tonello 2023; Trinh, Benhamou, and Soliman 2023).

To the best of our knowledge, the theoretical complexity of the different variants of this problem has not been fully explored yet. However, has been shown recently that even one of the simplest decision variants of the problem is coNP-complete in the general case (Moon, Lee, and Paulevé 2023). Several methods have been proposed for enumerating (minimal or maximal) trap spaces such as `pyboolnet` (Klarner, Streck, and Siebert 2017), `bioLQM` (Naldi 2018), `mpbn` (Paulevé et al. 2020), and `trappist` (Trinh et al. 2022; Trinh, Benhamou, and Soliman 2023), and `trapmvn` (Trinh et al. 2023). However, they still suffer from scalability and efficiency issues, particularly for very large and complex models. The main reason often being that they require intermediate representations of the original BN which may be expensive or even intractable to obtain, e.g., prime implicants, Petri Nets (PNs), Disjunctive Normal Forms (DNFs), or Binary Decision Diagrams (BDDs). The Related Work section gives a more detailed summary of these methods. In particular, the constantly increasing complexity of recently proposed models (Aghamiri et al. 2020) shows the limitations of these methods and motivates the search for a more scalable alternative.

Answer Set Programming (ASP) (Gelfond and Lifschitz 1988) has found widespread application in computational

Copyright © 2024, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

systems biology (Videla et al. 2015) due to its declarative nature and strong tooling support (Gebser et al. 2011a). ASP was employed early for modeling biological networks (Dworschak et al. 2008; Schaub and Thiele 2009). With the rising popularity of BNs, it was natural for ASP to be quickly adopted for both the modeling and analysis of BNs. The initial connection between ASP and BNs can be traced back to the theoretical work by Inoue 2011. Numerous authors have subsequently demonstrated successful applications of ASP to modeling and reasoning over biologically-motivated BNs. ASP has notably been used for tasks such as fixed-point enumeration (Klarner, Streck, and Siebert 2017; Abdallah et al. 2017; Paulevé et al. 2020), enumeration and approximation of attractors (Mushthofa et al. 2014; Klarner, Streck, and Siebert 2017; Abdallah et al. 2017; Paulevé et al. 2020; Khaled, Benhamou, and Trinh 2023), inference of BNs from biological data (Rocca et al. 2014; Videla et al. 2015, 2017; Chevalier et al. 2020; Ribeiro et al. 2021), model repair (Gebser et al. 2011b), and reprogramming (Kaminski et al. 2013; Videla et al. 2017). In particular, the most recent and efficient methods for trap space enumeration all rely on ASP (Klarner, Streck, and Siebert 2017; Paulevé et al. 2020; Trinh et al. 2022; Trinh, Benhamou, and Soliman 2023; Trinh et al. 2023).

Taking inspiration from the aforementioned works, we propose a new method that utilizes ASP to effectively enumerate minimal and maximal trap spaces of a BN. The key strength of the new method lies in its utilization of negative normal forms of Boolean functions, which are much easier to obtain than other intermediate representations used in the previous methods. In rare instances, the method still requires the use of BDDs, but the scope of their use is significantly reduced compared to the previous works, making it less problematic. To test the efficiency of our method, we conduct extensive evaluation using both real-world and random models. The experimental results unequivocally show superior performance of the new method over state-of-the-art techniques.

Related Work

The concept of *trap spaces* was introduced in Klarner, Bockmayr, and Siebert 2015. The authors proposed a method for enumerating minimal and maximal trap spaces of a BN using ASP, which has been implemented in `pyboolnet` (Klarner, Streck, and Siebert 2017). However, this approach requires the computation of prime implicants of a Boolean function, which is NP-complete. Furthermore, for complex functions, even the resulting number of prime implicants can be exponential in the number of function inputs, hence the problem often becomes impractical in this application beyond 10-20 inputs.

Subsequently, a method by Paulevé et al. 2020 has been proposed for enumerating minimal trap spaces that avoids enumeration of prime implicants. This approach has been implemented in the tool `mpbn` and demonstrated in Paulevé et al. 2020 as capable of handling medium-sized models from the literature and very large synthetic models. However, `mpbn` has some limitations that restrict its applicability. First, the BN must be locally-monotonic. This is a small subset of all possible BNs. Second, `mpbn` requires the DNF of a Boolean function, which is easier to construct than prime implicants, but still can be exponential in the number of function inputs.

Additionally, the `bioLQM` platform (Naldi 2018) also offers an alternative method for computing trap spaces using BDDs[1]. The method characterizes the set of generic trap spaces of a BN by a BDD, and then filters this set to obtain the set of all minimal (or maximal) trap spaces, without requiring the computation of prime implicants. However, it requires the computation of all solutions, while the methods based on ASP like `pyboolnet` or `mpbn` can enumerate solutions as they are found, making them potentially more efficient than the BDD-based method. Finally, `bioLQM` is limited by the fact that the number of generic trap spaces of a BN is often much greater than the number of minimal (or maximal) trap spaces.

Recently, a new method has been proposed for enumerating trap spaces of a BN based on its PN encoding (Trinh et al. 2022; Trinh, Benhamou, and Soliman 2023). A PN is a bipartite graph between a set of places and a set of transitions. A PN can encode a BN so that their dynamics are identical (see Trinh et al. 2022 and references therein). The authors have established a connection between trap spaces of a BN and conflict-free siphons of its PN encoding, allowing for an alternative approach to enumerating minimal (resp. maximal) trap spaces by enumerating maximal (resp. minimal) conflict-free siphons. It is implemented in the tool `trappist`, which has been shown to outperform `pyboolnet` and `bioLQM` for general BNs and is comparable or better than `mpbn` on locally-monotonic models. However, constructing the corresponding PN remains a bottleneck, as it generally requires obtaining two DNFs for each node v_i, one for $f_i \wedge \neg v_i$ and one for $\neg f_i \wedge v_i$. In `trapmvn` (Trinh et al. 2023), the authors iterate on `trappist` by adding heuristics to improve this PN encoding process, but the bottleneck still retains.

Preliminaries

Boolean Networks

A Boolean Network (BN) is a pair $\mathcal{N} = (V, F)$ where $V = \{v_1, \ldots, v_n\}$ is the set of nodes and $F = \{f_1, \ldots, f_n\}$ is the corresponding set of update Boolean functions. Each node v_i is associated with a Boolean variable (by an abuse of notation, we also use v_i to denote this variable) and a Boolean function f_i whose signature is $f_i \colon \mathbb{B}^{|IN(v_i)|} \mapsto \mathbb{B}$, where $\mathbb{B} = \{0, 1\}$ and $IN(v_i) \subseteq V$ denotes the set of input nodes of v_i. A Boolean function is *locally-monotonic* when it can be represented by a formula in DNF in which all occurrences of any variable are either positive or negative, but not a mix of both (Paulevé et al. 2020). A BN is said to be locally-monotonic if all its Boolean functions are locally-monotonic. The BN is non-locally-monotonic otherwise.

A state $s \in \mathbb{B}^n$ is a mapping $s \colon V \mapsto \mathbb{B}$ that assigns either 0 (inactive) or 1 (active) to each node. We denote the set of all possible states of a BN $\mathcal{N}$ by $\mathcal{S}_\mathcal{N} = \mathbb{B}^n$. At each time

[1] http://colomoto.org/biolqm/doc/tools-trapspace.html

step t, node v_i can update its state to $s'(v_i) = f_i(s)$, where s (resp. s') is the state of $\mathcal{N}$ at time t (resp. $t+1$). For simplicity, we write $f_i(s)$ even if $IN(v_i) \neq V$. An *update scheme* of a BN refers to how nodes update their states over (discrete) time (Schwab et al. 2020). Various update schemes exist, but the primary types are *synchronous*, where all nodes update simultaneously, and *fully asynchronous*, where a single node is non-deterministically chosen for updating. By adhering to the update scheme, the BN transitions from one state to another, which may or may not be the same. This transition is referred to as the *state transition* and denoted by $\rightarrow \subseteq \mathcal{S}_\mathcal{N} \times \mathcal{S}_\mathcal{N}$. More specifically, for the synchronous update scheme, we have $x \rightarrow y$ iff $y(v_i) = f_i(x)$ for all $v_i \in V$, whereas for the fully asynchronous update scheme, we have $x \rightarrow y$ iff there exists a node $v_i \in V$ such that $y(v_i) = f_i(x)$ and $y(v_j) = x(v_j)$ for all $v_j \neq v_i$. Then the dynamics of $\mathcal{N}$ is captured by the directed graph $(\mathcal{S}_\mathcal{N}, \rightarrow)$ referred to as the *State Transition Graph* (STG).

Trap Spaces

A non-empty set $T \subseteq \mathcal{S}_\mathcal{N}$ is a *trap set* with respect to $\rightarrow$ if for every $x \in T$ and $y \in \mathcal{S}_\mathcal{N}$ we have that $x \rightarrow y$ implies $y \in T$ (Klarner, Bockmayr, and Siebert 2015). An *attractor* of $\mathcal{N}$ with respect to $\rightarrow$ is an inclusion-wise minimal trap set of $(\mathcal{S}_\mathcal{N}, \rightarrow)$. A *sub-space* m of a BN is a mapping $m\colon V \mapsto \mathbb{B} \cup \{\star\}$. Here, variable v_i is called *fixed* if $m(v_i) \in \mathbb{B}$ (the value of v_i is fixed in m). Meanwhile, if $m(v_i) = \star$, v_i is called a *free* (the value of v_i can be both 0 and 1 in m). Let us denote D_m the set of all fixed variables of m. A sub-space m is equivalent to the set of all states s (denoted by $\mathcal{S}_\mathcal{N}[m]$) such that $s(v) = m(v), \forall v \in D_m$. For example, $m = \star 11$ (for simplicity, we write sub-spaces and states as a sequence of values) means that $D_m = \{v_2, v_3\}$ and $m(v_2) = m(v_3) = 1$. This sub-space is equivalent to the set of states $\{011, 111\}$. We denote $\mathcal{S}^\star_\mathcal{N} = (\mathbb{B} \cup \{\star\})^n$ the set of all possible sub-spaces of $\mathcal{N}$.

If a sub-space is also a trap set, it is a *trap space*. Unlike attractors, trap spaces of a BN are independent of the update scheme (Klarner, Bockmayr, and Siebert 2015). We define a partial order $<$ on $\mathcal{S}^\star_\mathcal{N}$ as: $m < m'$ iff $\mathcal{S}_\mathcal{N}[m] \subsetneq \mathcal{S}_\mathcal{N}[m']$. Then a trap space m is minimal iff there is no other trap space $m' \in \mathcal{S}^\star_\mathcal{N}$ such that $m' < m$. It is easy to derive that a minimal trap space contains at least one attractor of the BN regardless of the update scheme. Furthermore, any two minimal trap spaces are disjoint. Thus the set of minimal trap spaces can be a good approximation for the set of attractors (Klarner, Streck, and Siebert 2017). The maximal trap spaces are defined analogously. Let ε be the special trap space of $\mathcal{N}$ where all the variables are free. The trap space m is called maximal if $m \neq \varepsilon$ and there is no other trap space m' such that $m' \neq \varepsilon$ and $m < m'$.

Indeed, $|\mathcal{S}^\star_\mathcal{N}| = 3^n$. For the case of general BNs (resp. locally-monotonic BNs), deciding whether a sub-space is a trap space is a coNP-complete (resp. PTIME) problem, whereas deciding whether it is a minimal trap space is a coNP$^{\text{NP}}$-complete (resp. coNP-complete) problem (Moon, Lee, and Paulevé 2023). Hence, naive approaches for finding (minimal or maximal) trap spaces will rapidly get infeasible.

For illustration, we give a BN $\mathcal{N} = (V, F)$, where $V = \{v_1, v_2, v_3\}$ and $F = \{f_1, f_2, f_3\}$ with $f_1 = \neg v_1$, $f_2 = v_3$, and $f_3 = (v_1 \wedge v_2) \vee (\neg v_1 \wedge v_2)$. Figure 1 shows its STGs under the synchronous and the fully asynchronous update schemes where thin and dashed lines denote synchronous state transitions, whereas thick and solid lines denote the fully asynchronous ones. It has three trap spaces (same in the both STGs): $m_1 = \star 11$, $m_2 = \star 00$, and $m_3 = \star\star\star$. By the definitions above, m_1 and m_2 are two minimal (but also maximal) trap spaces of the BN. We shall use this BN as a simple example throughout this article.

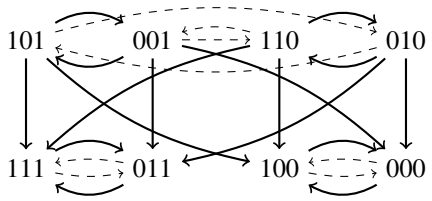

Figure 1: Synchronous and fully asynchronous STGs.

Answer Set Programming

In ASP, a logic program $\mathcal{L}$ is a set of (conjunctive) rules of the form

$$A \leftarrow A_1 \wedge \cdots \wedge A_m \wedge \neg A_{m+1} \wedge \cdots \wedge \neg A_k$$

where A and A_i are atoms ($k \geq m \geq 0$) and $\neg$ denotes the default negation. For any rule R of this form, A is called the *head* of R (denoted by $h(R)$) and the conjunction to the right of $\leftarrow$ is called the *body* of R. We denote $b^+(R)$ and $b^-(R)$ as the positive literals and the negative literals in the body of R, respectively. For simplicity, we assume in the rest of this work that all programs are ground, i.e., all atoms of every rule are variable-free. Let $\mathcal{A}$ be the set of all ground atoms of $\mathcal{L}$. A *Herbrand interpretation* I is a subset of $\mathcal{A}$, and is called a *Herbrand model* if for any rule R in $\mathcal{L}$, $b^+(R) \subseteq I$ and $b^-(R) \cap I = \emptyset$ imply $h(R) \in I$. Note that if $h(R) = \emptyset$, R is called a *constraint* and all Herbrand models satisfying the body of R are excluded from the model space.

A program $\mathcal{L}$ is *positive* if $b^-(R) = \emptyset$ for all R. In this case, $\mathcal{L}$ contains a single canonical Herbrand model expressed by the unique least Herbrand model. Then, an Herbrand interpretation I is a stable model (Gelfond and Lifschitz 1988) of a general program $\mathcal{L}$ if I is the least Herbrand model of the program $\mathcal{L}^I = \{(h(R) \leftarrow \bigwedge_{B \in b^+(R)} B) | R \in \mathcal{L}, b^-(R) \cap I = \emptyset\}$. It is known that if an atom is in a stable model, then it must appear in the head of at least one rule in the program. We can add to $\mathcal{L}$ a (*disjunctive*) rule in the form of $A_1 \vee \cdots \vee A_l \leftarrow$ making $\mathcal{L}$ become a disjunctive logic program. Then, I is a stable model of $\mathcal{L}$ if it is a minimal Herbrand model of $\mathcal{L}^I$. In addition, we can also add to $\mathcal{L}$ a (*choice*) rule in the form of $A_l \leftarrow \neg\neg A_l$, in which the stable semantics is still employed but in $\mathcal{L}^I$, if $b^-(R)$ contains $\neg A_l$, then $A_l \notin I$ implies $b^-(R) \cap I \neq \emptyset$.

ASP-Based Method

We first show the ASP encoding for enumerating minimal trap spaces of a BN. We then show how to modify this en-

coding to enumerate maximal trap spaces.

ASP Encoding

Let $\mathcal{N} = (V, F)$ be a BN. We construct a logic program $\mathcal{L}$ as follows. For each node $v_i \in V$, we introduce to $\mathcal{L}$ two (variable-free) atoms: p_i and n_i. Then, an Herbrand model I of $\mathcal{L}$ should correspond to a sub-space m such that for every node $v_i \in V$, $m(v_i) = 1$ iff $p_i \in I \wedge n_i \notin I$, $m(v_i) = 0$ iff $p_i \notin I \wedge n_i \in I$, and $m(v_i) = \star$ iff $p_i \in I \wedge n_i \in I$. To ensure this correspondence, we add to $\mathcal{L}$ the rule $p_i \vee n_i \leftarrow$ for each $v_i \in V$.

Next, we define $f[m]$ as a Boolean formula obtained by substituting fixed variables of m into Boolean formula f. For example, consider the straight example BN, we have if $m = \star 11$ then $f_1[m] = v_1$ but $f_3[m] = 1$. Recall that a trap space is also a trap set with respect to any update scheme of the BN. Hence, it must hold for any trap space m and for all $v_i \in V$ that if $f_i[m]$ can receive 1 then $m(v_i) = 1 \vee m(v_i) = \star$ and if $f_i[m]$ can receive 0 then $m(v_i) = 0 \vee m(v_i) = \star$. Note that $f_i[m]$ can receive both 0 and 1 (e.g., $f_1[m] = v_1$). By the correspondence between an Herbrand model and a sub-space, we have that the former condition can be characterized by $v_i \leftarrow f_i$, the latter one by $\neg v_i \leftarrow \neg f_i$. The conjunction of the two parts of all nodes v_i can characterize a trap space of the BN. For the part $v_i \leftarrow f_i$, to avoid the presence of negation, we convert f_i into its NNF, which can be obtained by recursively applying De Morgan laws until all negations that remain are on literals. Note that this procedure can be done purely syntactically. We now add to $\mathcal{L}$ ASP rules $\delta(v_i) \leftarrow \delta(\text{NNF}(f_i))$ for all $v_i \in V$ where we define function δ as

$$\delta(v_i) = p_i, \delta(\neg v_i) = n_i,$$

$$\delta(\bigwedge_{1 \le j \le J} \alpha_j) = \delta(\alpha_1) \wedge \ldots \wedge \delta(\alpha_J), \delta(\bigvee_{1 \le j \le J} \alpha_j) = aux_k,$$

where aux_k is a new atom and for each j we add the rule $aux_k \leftarrow \delta(\alpha_j)$ to $\mathcal{L}$. For convenience, we distinguish two types of atoms: *main* atoms (p_i or n_i) and *auxiliary* atoms (aux_k). Note that k is here a global counter starting from 1 and will be increased by 1 after a new auxiliary atom is created. For the part $\neg v_i \leftarrow \neg f_i$, we similarly apply the above process with $\neg v_i$ and $\neg f_i$ play the roles of v_i and f_i, respectively.

Correctness

Definition 1 (Safeness)**.** *A Boolean formula f is said to be* safe *if it does not contain any conjunction between two sub-formulae f_1 and f_2 such that there exists a variable a appearing in f_1 with $\neg a$ appearing in f_2.*

Intuitively, this allows us to identify formulae where we should not derive the contradiction from logical formulae when both atoms p_i and n_i are present, representing the case of a free variable in a trap space. Specifically, when NNF(f_j) is unsafe with respect to variable v_i and $m(v_i) = \star$, the presence of both p_i and n_i can lead to the presence of p_j whereas $f_j[m]$ can be evaluated as 0 and $m(v_j)$ should be 0, which is a contradiction. A detailed example for this issue shall be given later on. Formally, we have the below result whose formal proof is given in the supplementary material[2].

Theorem 1. *Let $\mathcal{N} = (V, F)$ be a BN and $\mathcal{L}$ be its encoded logic program. If NNF(f_i) and NNF($\neg f_i$) are safe for all $v_i \in V$, then the set of stable models of $\mathcal{L}$ one-to-one corresponds to the set of minimal trap spaces of $\mathcal{N}$.*

Next, we show how to deal with the case of unsafe formulae to ensure the correctness of the proposed encoding. When considering the rule in form of $\delta(v_i) \leftarrow \delta(f)$, if we detect f unsafe, we can convert f into its DNF. By the safeness definition, DNF(f) is safe. Furthermore, f and DNF(f) are still logically equivalent. Hence, the new resulting $\mathcal{L}$ still characterizes correctly all trap spaces of the BN. To obtain DNF(f), we construct a BDD for f from which we can easily derive each conjunction of DNF(f) as a satisfying path from the root node to the one node of the constructed BDD.

For illustration, we consider the straight example BN $\mathcal{N}$. Listing 1 (following `clingo`'s syntax) shows the encoded logic program of $\mathcal{N}$ with '`;`', '`,`', and '`:-`' denote operators $\vee$, $\wedge$, and $\leftarrow$, respectively. Line 1 represents the rules $p_i \vee n_i \leftarrow$ for the three nodes enforcing that at least one of p_i or n_i is true. Herein, NNF(f_1), NNF($\neg f_1$), NNF(f_2), NNF($\neg f_2$) are all safe. Then Line 2 represents the rules for $\delta(v_1) \leftarrow \delta(\text{NNF}(f_1))$ and $\delta(\neg v_1) \leftarrow \delta(\text{NNF}(\neg f_1))$. Similarly, Line 3 represents the rules for node v_2. The problem lies on node v_3 where NNF(f_3) is safe but NNF($\neg f_3$) is unsafe. Lines 4 and 5 represent the rules for $\delta(v_3) \leftarrow \delta(\text{NNF}(f_3))$. Since NNF($\neg f_3$) is unsafe, we need to use DNF(NNF($\neg f_3$)) that is $\neg v_2$. Note that NNF($\neg f_3$) = $(\neg v_1 \vee \neg v_2) \wedge (v_1 \vee \neg v_2)$ that should be simplified to $\neg v_2$, and the BDD construction implicitly makes the simplification. Finally, Line 6 represents the rules for $\delta(\neg v_3) \leftarrow$ DNF(NNF($\neg f_3$)). $\mathcal{L}$ has exactly two stable models (only considering main atoms): $\{p_1, n_1, p_2, p_3\}$ corresponding to minimal trap space $m_1 = \star 11$ and $\{p_1, n_1, n_2, n_3\}$ corresponding to minimal trap space $m_2 = \star 00$.

Listing 1: Encoded logic program of the example BN.

```
1 p1; n1.       p2; n2.   p3; n3.
2 p1 :- n1.       n1 :- p1.
3 p2 :- p3.       n2 :- n3.
4 p3 :- aux1.
5 aux1 :- p1, p2.  aux1 :- n1, p2.
6 n3 :- n2.
```

Let us explain why using an unsafe formula (i.e., NNF($\neg f_3$)) might lead to wrong results. It is easy to see that $m(v_1)$ is always $\star$ for any trap space m, which is characterized by the presence of both p_1 and n_1 in every Herbrand model of $\mathcal{L}$. Consider the case that $m(v_2) = 1$, i.e., p_2 is true and n_2 is false. Then we can derive that p_3 is true by the rules in Lines 4 and 5. The rules for $\delta(\neg v_3) \leftarrow$ NNF($\neg f_3$) should be $n_3 \leftarrow aux_2 \wedge aux_3$, $aux_2 \leftarrow n_1$, $aux_2 \leftarrow n_2$, $aux_3 \leftarrow p_1$, and $aux_3 \leftarrow n_2$ that lead to the presence of n_3. By the rule $n_2 \leftarrow n_3$, we have the presence of n_2, which is a contradiction. The root cause is that when $m(v_2) = 1$, $\neg f_3[m]$ is evaluated to 0 (equivalently $f_3[m]$ to 1), enforcing that only p_1 is true. All of these make the actual minimal

[2] https://zenodo.org/doi/10.5281/zenodo.10405520

trap space $m_1 = \star 11$ disappear. Note also that if there is no contradiction, a wrong stable model might be introduced, leading to wrong trap spaces, and then wrong minimal ones.

Enhancements

We here propose two technical enhancements as follows. First, we can see that if there is a rule $a \leftarrow aux_k$, we can remove this rule and atom aux_k, and replace aux_k by a wherever it appears in $\mathcal{L}$. For example, in Listing 1, the rules of Lines 4 and 5 can be replaced by $p_3 \leftarrow p_1 \wedge p_2$ and $p_3 \leftarrow n_1 \wedge p_2$. Second, some auxiliary atoms can correspond to the same sub-formula. In this case, we can use only one representative atom for all such atoms. The two above enhancements can reduce the number of atoms in $\mathcal{L}$, hoping likely to reduce the solving time, and clearly preserve the correctness of the encoding.

Theorem 2. *Let $\mathcal{N} = (V, F)$ be a BN and $\mathcal{L}$ be its encoded logic program. Let $\mathcal{L}'$ be the program obtained by apply the two above enhancements to $\mathcal{L}$. If NNF(f_i) and NNF$(\neg f_i)$ are safe for all $v_i \in V$, then the set of stable models of $\mathcal{L}'$ one-to-one corresponds to that of $\mathcal{L}'$.*

Maximal Trap Space Enumeration

The research discussed in (Rozum et al. 2021b) employs the concept of *stable motifs* to construct a *succession diagram* for a BN. It proves valuable for analyzing (e.g., enumerating attractors under the fully asynchronous update scheme) and controlling BNs (e.g., enumerating control interventions that drive any initial state into a given minimal trap space that might correspond to a desirable phenotype) (Rozum et al. 2021b,a). However, the enumeration of stable motifs, as presented in (Rozum et al. 2021b,a), is indeed a computational bottleneck. It also pointed out that a stable motif is identical to a maximal trap space (Rozum et al. 2021b). Hence, there is a need to develop an efficient technique for enumerating maximal trap spaces in a BN.

With this motivation, we adapt the core ASP encoding to enumerate maximal trap spaces. Recall that any trap space of $\mathcal{N}$ is represented by an Herbrand model of $\mathcal{L}$, but multiple ones might represent the same trap space due to the presence of auxiliary atoms. First, we need to exclude all the Herbrand models corresponding to the special trap space ϵ where all variables are free. To do this, we add to $\mathcal{L}$ the constraint $\leftarrow p_1 \wedge n_1 \wedge \cdots \wedge p_n \wedge n_n$ ensuring that every Herbrand model of $\mathcal{L}$ cannot contain all main atoms. Second, we add to $\mathcal{L}$ the choice rules for all main atoms, i.e., $p_i \leftarrow \neg\neg p_i$ and $n_i \leftarrow \neg\neg n_i$ for all $v_i \in V$. This ensures that any trap space is represented by a stable model of $\mathcal{L}$. In addition, since there are no choice rules for auxiliary atoms, a trap space is represented by exactly one stable model, leading to the one-to-one correspondence between trap spaces of $\mathcal{N}$ and stable models of $\mathcal{L}$. Now, we just need to consider the subset-maximal stable models of $\mathcal{L}$ (hence, the maximal trap spaces of $\mathcal{N}$) thanks to the built-in feature[3] of some ASP solvers such as `clingo` for this task (Gebser et al. 2011a).

[3]We used `clingo`'s configuration `-heuristic=Domain -enum-mod=domRec -dom-mod=3` (subset maximality).

Advantages

We here discuss the advantages of our new ASP-based method (named `tsconj` for convenience) as compared to other ASP-based methods (`pyboolnet`, `mpbn`, `trappist`) for enumerating minimal and maximal trap spaces in BNs. These advantages shall be justified by the experimental results shown in the next section. First, `tsconj`, along with `pyboolnet` and `trappist`, is applicable for general models, whereas `mpbn` is only applicable for locally-monotonic models. Second, regarding the encoding construction, all `tsconj`, `mpbn`, and `trappist` avoid the need of computing prime implicants, which is the bottleneck of `pyboolnet` and also the most demanding task. For every node v_i, `trappist` needs two BDDs ($f_i \wedge \neg v_i$ and $\neg f_i \wedge v_i$), `mpbn` needs a DNF of f_i. On the other hand, `tsconj` only needs two NNFs (NNF(f_i) and NNF$(\neg f_i)$), which are much easier to obtain than the BDDs or DNFs. Although `tsconj` still needs a BDD for an unsafe formula, it is still superior to `mpbn` and `trappist`. More specifically, the safeness identification of a formula can be done syntactically, thus quite efficient; `tsconj` may only need one BDD, whereas `trappist` always needs two BDDs; and if NNF(f_i) or NNF$(\neg f_i)$ is unsafe, then DNF(f_i) is also detected non-locally-monotonic by `mpbn` without simplification. For the last case, `tsconj` needs one or two BDDs, but `mpbn` gives up as it cannot handle non-locally-monotonic models. Note that f_i might be locally-monotonic after full simplification (e.g., $f_3 = (v_1 \wedge v_2) \vee (\neg v_1 \wedge v_2)$ can be simplified to v_2), but the full simplification is even a much more demanding task than the BDD construction.

Third, regarding the ASP encoding characteristics, we divide two cases as follows. For minimal trap space enumeration, `tsconj` does not use choice rules, whereas all `pyboolnet`, `mpbn`, `trappist` do. Note however that `tsconj` does still contain disjunctive rules (i.e., $p_i \vee n_i \leftarrow$) and it remains open whether using choice rules is good or bad w.r.t. the presence of disjunctive rules. Upon closer inspection, `tsconj` and `trappist` use only variable-free atoms, whereas `pyboolnet` and `mpbn` use variable atoms (predicates), i.e., the ASP solver needs to perform the grounding step for the encoding of `pyboolnet` or `mpbn`. Furthermore, `tsconj`, `pyboolnet`, and `mpbn` use mostly conjunctive rules, whereas `trappist` uses mostly disjunctive rules. For maximal trap space enumeration, `tsconj` uses choice rules for main atoms, but all the remaining characteristics are the same as those of the minimal case. The ASP characteristics of `pyboolnet`, `mpbn`, and `trappist` for the maximal case are the same as theirs for the minimal case. It is worth noting that choice and disjunctive rules are sources for the inefficiency of most ASP solvers such as `clingo` (Gebser et al. 2011a).

One might wonder if it is possible to adapt other resolution techniques such as SAT, Constraint Programming (CP), and Integer Linear Programming (ILP). Indeed, there are the ILP version of `pyboolnet` (Klarner, Bockmayr, and Siebert 2015) and the SAT, CP, and ILP versions of `trappist` (Trinh, Benhamou, and Soliman 2023). They all employ an iterative manner, i.e., finding a minimum or maximum solution first, then adding new constraints to

force the solver to find new solutions. These versions were shown to be much worse than the corresponding ASP versions (Klarner, Bockmayr, and Siebert 2015; Trinh, Benhamou, and Soliman 2023). Similarly, it is possible to propose SAT, CP, and ILP versions for `tsconj`. However, due to the presence of auxiliary variables, the main issue of these versions is the enumeration of redundant models that encode the same trap space. A step to eliminate redundant models is therefore necessary to guarantee the correctness and this would add more complexity to the SAT/CP/ILP approach. In contrast, the ASP approach can avoid the above issue because of only searching for stable models. Note however that the superiority of the ASP-based approach against the SAT-based approach is mostly a question of "declarativity" and of current practice (in terms of enumerating multiple solutions) rather than the latter being less suitable in principle.

Experiments

The goal of this section is two-fold: First, to demonstrate that our proposed method is capable of handling realistic biological models, and second, to demonstrate that on such models, it significantly outperforms existing state of the art methods. We implemented the proposed method as a Python package `tsconj`[4]. All data and source code, addition technical details about the measurement process, and detailed analysis of results is available in the supplementary material[5]. Here, we only present a summary of our main observations.

Experiment Setup

Hardware and benchmark harness We test using a desktop computer equipped with Ryzen 5800X CPU. Each experiment is limited to 64GB of RAM and utilizes a single CPU core. The runtime is measured internally by each process. It includes all computation steps, including loading input data and any necessary encoding or transformations.

Tested software All tested software is distributed in the form of Python packages. As such, our tests use a single Python virtual environment with all the packages installed. Specifically, we consider the tools `pyboolnet` (Klarner, Streck, and Siebert 2017), `mpbn` (Paulevé et al. 2020), `trappist` (Trinh, Hiraishi, and Benhamou 2022), and `trapmvn` (Trinh et al. 2023). Where applicable, the employed tools use the ASP solver `clingo` (Gebser et al. 2011a). To the best of our knowledge, these represent the state-of-the-art methods for enumerating trap spaces of BNs.

Tested models We evaluate the tools on four separate datasets of different properties: First, the *Biodivine Boolean Models* (BBM) dataset (Pastva et al. 2023) which contains 212 real-world published BNs, ranging up to 321 variables. Second, a smaller dataset of 39 *selected* real-world models, collected by the authors of this paper, which do not appear in BBM, ranging up to 3158 variables. Third, a dataset of *Very Large Boolean Networks* (VLBN)[6] consisting of 28 randomly generated BNs ranging up to 100.000 variables.

[4]https://github.com/daemontus/tsconj

[5]https://zenodo.org/doi/10.5281/zenodo.10405520

[6]https://doi.org/10.5281/zenodo.3714875

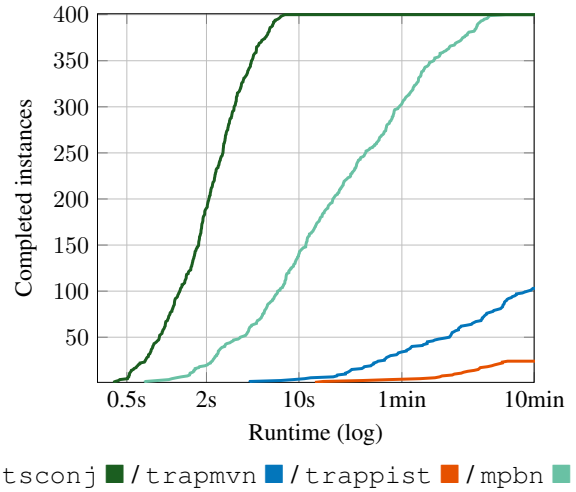

Figure 2: Cumulative experiments completed (y-axis) until a specific time point (x-axis, logarithmic). Concerns the 400 randomly generated models with 1.000-5.000 variables.

And fourth, a dataset of 400 models with 1.000-5.000 variables, sampled using the generator provided in (Benes et al. 2021). To best of our knowledge, this is a highly representative of biologically relevant BNs appearing in literature.

Tested problems We primarily consider detection of *minimal* trap spaces. However, some BNs admit a large number of minimal trap spaces. In such instances, a benchmark is often testing the technical ability of the implementation to enumerate the results quickly (especially in Python), instead of the actual problem-solving. To eliminate this issue, we only measure the time to compute the *first* result, which we believe to represent the most fair comparison between the underlying techniques. In the the supplementary material, we also provide a limited set of results for detection of *maximal* trap spaces. However, these tests do not reveal any conclusions not covered by the minimal trap space problem, and as such, we do not discuss them in detail here.

Results and Discussion

The main summary of results is given in Table 1. Here, we show the number of problems completed by every tool for each group of BNs. Then, Figure 2 shows a more detailed analysis of the completed problem instances in the dataset of 400 randomly generated BNs. Finally, Figure 3 plots the ratio between the runtime of the competing tools and `tsconj` on *all* benchmark instances where both tools computed a valid result.

Discussion

First, let us note that while the performance of `pyboolnet` is not terrible, it is clearly lacking compared to all the other tools: It did not complete a single random model benchmark within the timeout, and even for real-world models, it is certainly the slowest method. As such, it mostly serves as a baseline for the other tools.

Method	<1s	<1min	<1h
Biodivine Boolean Models			
pyboolnet	161/212	194/212	203/212
mpbn	182/212	185/212	185/212
trappist	207/212	208/212	208/212
trapmvn	208/212	211/212	211/212
tsconj	208/212	212/212	**212**/212
Manually selected models			
pyboolnet	17/39	26/39	33/39
mpbn	26/39	27/39	27/39
trappist	35/39	36/39	36/39
trapmvn	35/39	36/39	36/39
tsconj	35/39	36/39	36/39
Very Large Boolean Networks			
pyboolnet	0/28	0/28	0/28
mpbn	0/28	7/28	18/28
trappist	0/28	1/28	1/28
trapmvn	0/28	9/28	20/28
tsconj	4/28	19/28	**28**/28
Manually generated models			
pyboolnet	0/400	0/400	0/400
mpbn	3/400	303/400	**400**/400
trappist	0/400	4/400	24/400
trapmvn	0/400	34/400	103/400
tsconj	67/400	400/400	**400**/400

Table 1: Summary of tool performance for computing the *first* solution. Rows <1s, <1min and <1h give the number of models completed within the respective time limit.

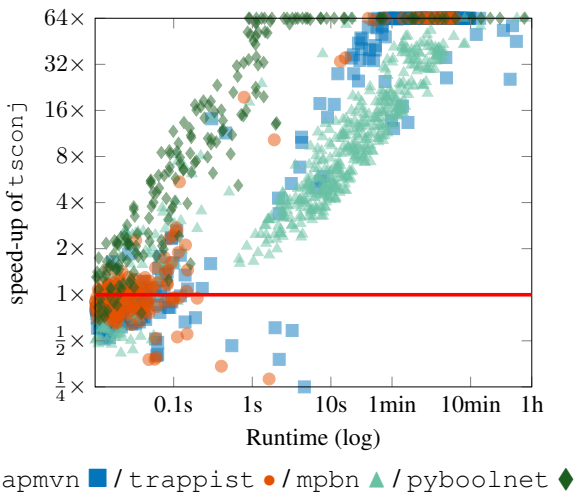

Figure 3: The speed-up of tsconj compared to the other tools, truncated to the interval $[\frac{1}{4}, 64]$. Points below the red line represent a slow-down instead of a speed-up.

Then, we should compare trapmvn and trappist. Both use the same underlying ASP encoding based on implicants and BDDs. However, trapmvn implements several improvements which allow it to either generate the logic program faster, or to simplify the program itself. For real-world models, these methods are largely comparable: trapmvn solved three more BBM models due to a technical limitation of trappist, but otherwise provides the same results. However, for the larger random models, we observe that the modifications implemented in trapmvn significantly improve the performance of the tool. For the VLBN dataset, trapmvn is the second-best tool, and in the 400 random models, it still significantly improves on the results obtained by trappist. Nevertheless, in all non-trivial instances (as evidenced by Figures 2 and 3), tsconj still significantly outperforms both methods.

Finally, comparing tsconj to mpbn, the performance on real-world models is similar once we disregard models with non-locally-monotonic functions, which are not supported by this version of mpbn. Even on the 400 random models those are all locally-monotonic, mpbn and tsconj are the only two tools that were able to complete all benchmarks (however, Figure 2 shows that tsconj still provides a non-trivial performance improvement). Nevertheless, on the VLBN dataset, we see that mpbn struggles with models of more than 10.000 variables, while tsconj completed the whole dataset.

Conclusion

In this work, we proposed a new method named tsconj based on ASP for enumerating minimal and maximal trap spaces in BNs. These are not only crucial in the analysis and control of biological systems, but also have applications in various other areas. We have shown that the proposed method has many advantages over the previous methods for the same task. Then, the efficiency of our method has been verified by rigorous experiments on many real-world and randomly generated models. In particular, it vastly outperforms the four state-of-the-art methods for enumerating minimal and maximal trap spaces in BNs and is able to handle very large and complex models.

It is likely that the SAT/CP/ILP variant of tsconj will be less efficient than the ASP approach. Nevertheless, it would be interesting to implement the SAT, CP, and ILP versions of tsconj and compare their performance on the set of models used in this work. Moreover, there are some "hard" models that none of the existing methods could handle. These models are not particularly typical in the systems biology community, but an improvement is needed to make tsconj able to handle such cases. A possible direction is to find a more "permissive" safeness condition where complicated unsafe formulas can be efficiently (ideally syntactically) detected as safe ones. In this case, we do not need to compute BDDs of those formulas. Finally, we plan to test the scalability of tsconj on other types of random models such as N-K models (Glass and Kauffman 1973; Rozum et al. 2021b) and scale-free models (Aldana 2003; Drossel and Greil 2009), which have been widely used in other fields such as theoretical physics, social modelling, and neural networks. This can broaden the applicable range of our method.

Acknowledgments

This work was supported by Institut Carnot STAR, Marseille, France and by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant Agreement No. 101034413.

References

Abdallah, E. B.; Folschette, M.; Roux, O. F.; and Magnin, M. 2017. ASP-based method for the enumeration of attractors in non-deterministic synchronous and asynchronous multi-valued networks. *Algorithms Mol. Biol.*, 12(1): 20:1–20:23.

Aghamiri, S. S.; Singh, V.; Naldi, A.; Helikar, T.; Soliman, S.; Niarakis, A.; and Xu, J. 2020. Automated inference of Boolean models from molecular interaction maps using CaSQ. *Bioinform.*, 36(16): 4473–4482.

Aldana, M. 2003. Boolean dynamics of networks with scale-free topology. *Physica D*, 185(1): 45–66.

Benes, N.; Brim, L.; Pastva, S.; and Safránek, D. 2021. Computing Bottom SCCs Symbolically Using Transition Guided Reduction. In *International Conference on Computer Aided Verification*, 505–528. Springer.

Chevalier, S.; Froidevaux, C.; Paulevé, L.; and Zinovyev, A. Y. 2019. Synthesis of Boolean Networks from Biological Dynamical Constraints using Answer-Set Programming. In *International Conference on Tools with Artificial Intelligence*, 34–41. IEEE.

Chevalier, S.; Noël, V.; Calzone, L.; Zinovyev, A. Y.; and Paulevé, L. 2020. Synthesis and Simulation of Ensembles of Boolean Networks for Cell Fate Decision. In *International Conference on Computational Methods in Systems Biology*, 193–209. Springer.

Drossel, B.; and Greil, F. 2009. Critical Boolean networks with scale-free in-degree distribution. *Phys. Rev. E*, 80(2): 026102.

Dworschak, S.; Grell, S.; Nikiforova, V. J.; Schaub, T.; and Selbig, J. 2008. Modeling Biological Networks by Action Languages via Answer Set Programming. *Constraints An Int. J.*, 13(1-2): 21–65.

Gebser, M.; Kaufmann, B.; Kaminski, R.; Ostrowski, M.; Schaub, T.; and Schneider, M. 2011a. Potassco: The Potsdam Answer Set Solving Collection. *AI Commun.*, 24(2): 107–124.

Gebser, M.; Schaub, T.; Thiele, S.; and Veber, P. 2011b. Detecting inconsistencies in large biological networks with answer set programming. *Theory Pract. Log. Program.*, 11(2-3): 323–360.

Gelfond, M.; and Lifschitz, V. 1988. The Stable Model Semantics for Logic Programming. In *International Conference and Symposium on Logic Programming*, 1070–1080. MIT Press.

Glass, L.; and Kauffman, S. A. 1973. The logical analysis of continuous, non-linear biochemical control networks. *J. Theor. Biol.*, 39(1): 103–129.

Inoue, K. 2011. Logic Programming for Boolean Networks. In *International Joint Conference on Artificial Intelligence*, 924–930. IJCAI/AAAI.

Kaminski, R.; Schaub, T.; Siegel, A.; and Videla, S. 2013. Minimal intervention strategies in logical signaling networks with ASP. *Theory Pract. Log. Program.*, 13(4-5): 675–690.

Khaled, T.; Benhamou, B.; and Trinh, V.-G. 2023. Using answer set programming to deal with Boolean networks and attractor computation: application to gene regulatory networks of cells. *Ann. Math. Artif. Intell.*, 1–38.

Klarner, H.; Bockmayr, A.; and Siebert, H. 2015. Computing maximal and minimal trap spaces of Boolean networks. *Nat. Comput.*, 14(4): 535–544.

Klarner, H.; Streck, A.; and Siebert, H. 2017. PyBoolNet: a python package for the generation, analysis and visualization of Boolean networks. *Bioinform.*, 33(5): 770–772.

Moon, K.; Lee, K.; and Paulevé, L. 2023. Computational complexity of minimal trap spaces in Boolean networks. arXiv:2212.12756.

Mushthofa, M.; Torres, G.; de Peer, Y. V.; Marchal, K.; and Cock, M. D. 2014. ASP-G: an ASP-based method for finding attractors in genetic regulatory networks. *Bioinform.*, 30(21): 3086–3092.

Naldi, A. 2018. BioLQM: a Java toolkit for the manipulation and conversion of logical qualitative models of biological networks. *Front. Physiol.*, 9: 1605.

Park, J.-C.; Jang, S.-Y.; Lee, D.; Lee, J.; Kang, U.; Chang, H.; Kim, H. J.; Han, S.-H.; Seo, J.; Choi, M.; et al. 2021. A logical network-based drug-screening platform for Alzheimer's disease representing pathological features of human brain organoids. *Nat. Commun.*, 12(1): 280.

Pastva, S.; Safranek, D.; Benes, N.; Brim, L.; and Henzinger, T. 2023. Repository of logically consistent real-world Boolean network models. *bioRxiv*.

Paulevé, L.; Kolčák, J.; Chatain, T.; and Haar, S. 2020. Reconciling qualitative, abstract, and scalable modeling of biological networks. *Nat. Commun.*, 11(1): 1–7.

Ribeiro, T.; Folschette, M.; Magnin, M.; and Inoue, K. 2021. Learning any memory-less discrete semantics for dynamical systems represented by logic programs. *Mach. Learn.*, 111(10): 3593–3670.

Richard, A.; and Tonello, E. 2023. Attractor separation and signed cycles in asynchronous Boolean networks. *Theor. Comput. Sci.*, 947: 113706.

Rocca, A.; Mobilia, N.; Fanchon, E.; Ribeiro, T.; Trilling, L.; and Inoue, K. 2014. ASP for construction and validation of regulatory biological networks. *Logical Modeling of Biological Systems*, 167–206.

Rozum, J. C.; Deritei, D.; Park, K. H.; Gómez Tejeda Zañudo, J.; and Albert, R. 2021a. pystablemotifs: Python library for attractor identification and control in Boolean networks. *Bioinform.*, 38(5): 1465–1466.

Rozum, J. C.; Gómez Tejeda Zañudo, J.; Gan, X.; Deritei, D.; and Albert, R. 2021b. Parity and time reversal elucidate

both decision-making in empirical models and attractor scaling in critical Boolean networks. *Sci. Adv.*, 7(29): eabf8124.

Schaub, T.; and Thiele, S. 2009. Metabolic Network Expansion with Answer Set Programming. In *International Conference on Logic Programming*, 312–326. Springer.

Schwab, J. D.; Kühlwein, S. D.; Ikonomi, N.; Kühl, M.; and Kestler, H. A. 2020. Concepts in Boolean network modeling: What do they all mean? *Comput. Struct. Biotechnol. J.*, 18: 571–582.

Thomas, R. 1973. Boolean formalisation of genetic control circuits. *J. Theor. Biol.*, 42: 565–583.

Trinh, V.; Benhamou, B.; Hiraishi, K.; and Soliman, S. 2022. Minimal Trap Spaces of Logical Models are Maximal Siphons of Their Petri Net Encoding. In *International Conference on Computational Methods in Systems Biology*, 158–176. Springer.

Trinh, V.; Hiraishi, K.; and Benhamou, B. 2022. Computing attractors of large-scale asynchronous Boolean networks using minimal trap spaces. In *ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, 13:1–13:10. ACM.

Trinh, V.-G.; Benhamou, B.; Henzinger, T.; and Pastva, S. 2023. Trap spaces of multi-valued networks: definition, computation, and applications. *Bioinf.*, 39(Supplement_1): i513–i522.

Trinh, V.-G.; Benhamou, B.; and Soliman, S. 2023. Trap spaces of Boolean networks are conflict-free siphons of their Petri net encoding. *Theor. Comput. Sci.*, 971: 114073.

Videla, S.; Guziolowski, C.; Eduati, F.; Thiele, S.; Gebser, M.; Nicolas, J.; Saez-Rodriguez, J.; Schaub, T.; and Siegel, A. 2015. Learning Boolean logic models of signaling networks with ASP. *Theor. Comput. Sci.*, 599: 79–101.

Videla, S.; Saez-Rodriguez, J.; Guziolowski, C.; and Siegel, A. 2017. caspo: a toolbox for automated reasoning on the response of logical signaling networks families. *Bioinform.*, 33(6): 947–950.

Wang, R.-S.; Saadatpour, A.; and Albert, R. 2012. Boolean modeling in systems biology: an overview of methodology and applications. *Phys. Biol.*, 9(5): 055001.