HAL
open science

Using Reachability Properties of Logic Program for Revising Biological Models

Xinwei Chai, Tony Ribeiro, Morgan Magnin, Olivier Roux, Katsumi Inoue

To cite this version:

Xinwei Chai, Tony Ribeiro, Morgan Magnin, Olivier Roux, Katsumi Inoue. Using Reachability Properties of Logic Program for Revising Biological Models. 2018. hal-01951247

HAL Id: hal-01951247

https://hal.science/hal-01951247

Preprint submitted on 11 Dec 2018

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.

Using Reachability Properties of Logic Program for Revising Biological Models

Xinwei Chai, Tony Ribeiro, Morgan Magnin, Olivier Roux

Laboratoire des Sciences du Numérique de Nantes, 1 rue de la Noë, 44321 Nantes, France

Katsumi Inoue

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

Abstract

Learning system dynamics from the observations of state transitions has many applications in bioinformatics. It can correspond to the identification of the mutual influence of genes and can help to understand their interactions. A model can be automatically learned from time series data by using methods like Learning from interpretation transition *(LFIT)*. This method learns an exact model if all transitions of the systems are used as input. However, in real biological data, such complete data sets are usually not accessible and we have to learn a system with partial observations. Usually, biologists also provide with *a priori* knowledge about the system dynamics in the form of temporal properties. When building models, keeping critical properties valid is one of the major concerns and model checking plays a role in the verification of such desired properties. Our research aims at providing a model checking approach to revise logic programs thanks to temporal properties. In this paper, as a first step, we propose a method that can exploit reachability properties to fit such a model.

keywords: Model Checking, Learning From Interpretation Transition, Dynamical Systems, Temporal Properties, Local Causality Graph

1 Introduction

When modeling a real system, it is usually demanded to assess the correctness of a Boolean network with the concrete system by checking if the observed configurations are indeed reachable in the Boolean network. Whenever it is not the case, it typically means that the designed Boolean functions do not model the given system correctly, and thus should be revised before further model analysis. In [4], it has been shown that Boolean networks can be represented by logic

programs. In this paper, we provide a method to revise a logic program to fit temporal properties regarding reachability of partial states. Such logic program can be learned from observations of state transition using LFIT algorithm in [7], but the approach restricts the model to only synchronous update scheme. One of the benefits of synchronous modeling is computational tractability, while classical state space exploration algorithms fail on asynchronous ones. Yet the synchronous modeling relies on quite heavy assumptions: all genes can make a transition simultaneously and need an equivalent amount of time to change their expression level. Even if this is not realistic from a biological point of view, it is usually sufficient as the exact kinetics and order of transformations are generally unknown. However, the asynchronous semantics helps one to capture more realistic behaviors [1]. At a given time point, at most one single gene can change its expression level. Non-deterministic behaviors are often observed in biological systems, e.g. cell differentiation. From a given state, several possible behaviors can be expected as future states. Asynchronous update scheme results in a potential combinatorial explosion to the number of states. The first contribution of this paper is a simple adaptation of the LFIT algorithm for learning asynchronous dynamics. The main contribution is a method to revise a logic program in order to fit given reachability properties. Reachability problem on formal models is a critical challenge where both validation (whether the model satisfies *a priori* knowledge) and prediction (properties to be discovered) problems meet. From a formal point of view, numerous biological properties can be expressed in computation models as reachability properties [3]. Existing approaches usually rely on global search and thus face state space explosion problem as the state space grows exponentially with the number of components of model. Abstraction is an efficient strategy to deal with such systems. In [6], local properties of the model are exploited based on an abstract interpretation: Local Causality Graph (LCG). [2] provides a hybrid reachability analyzer based on LCG, with which one can verify the model is consistent with given reachability information with good runtime and conclusiveness. LFIT framework so far can only capture finite dynamical properties, i.e. relation at T-1 or T-k and the system has to be synchronous deterministic. In asynchronous systems, non-determinism can lead to loops for several times before taking a path to a certain state. In this paper, we adapt the algorithms of [7, 5] to capture asynchronous dynamics and extend upon this method to propose an approach allowing to fit a logic program to reachability properties. By modifying rules of the program using logic generalization/specialization operations, we iteratively revise the program to fit a set of reachability/unreachability constraints while keeping the observation and learned rules consistent.

2 Formalization

Boolean asynchronous systems can be non-deterministic, thus from the same state a variable can take both value 0 or 1. To encode this dynamics, one requires to have explicit rules for each value of a variable and the modeling of [7] is not suitable. In [5], we proposed a modeling of multi-valued synchronous systems as annotated logic program. This modeling can be applied to repre-

sent Boolean asynchronous systems and is recalled in the following section. In order to represent multi-valued variables, all atoms of a logic program are now restricted to the form var^{val}. The intuition behind this form is that var represents some variable of the system and val represents the value of this variable. In annotated logics, the atom var is said to be annotated by the constant val. We consider a *multi-valued logic program* as a set of *rules* of the form

$$var^{val} \leftarrow var_1^{val_1} \wedge \cdots \wedge var_n^{val_n} \tag{1}$$

where var^{val} and $var_i^{val_i}$'s are atoms ($n \geq 1$). For any rule R of the form (1), left part of $\leftarrow$ is called the *head* of R and is denoted as $h(R)$, and the conjunction to the right of $\leftarrow$ is called the *body* of R. We represent the set of literals in the body of R of the form (1) as $b(R) = \{var_1^{val_1}, \ldots, var_n^{val_n}\}$. A rule R of the form (1) is interpreted as follows: the variable var takes the value val in the next state if all variables var_i have the value val_i in the current state. A state of a multi-valued program provides the value of each variable of the system and a transitions is a pair of states. The value of a variable in a state is called a local state. The set of all local state is denoted **LS**. The subset of state is called a partial state. A rule R matches a state s when $b(R) \subseteq s$. A rule R subsumes a rule R' when $h(R) = h(R'), b(R) \subseteq b(R')$. A Boolean Asynchronous system can be represented by a multi-valued logic program. This section provides the necessary additional formalization to interpret asynchronous dynamics by such program and to learn from state transitions.

2.1 Modeling and learning of asynchronous dynamics

Due to the non-deterministic nature of asynchronous systems and its restriction to atmost one variable change per transition, the notion of consistency, realization and successor has to be adapted as follows.

Definition 1 (Consistency)**.** Let R be a rule and E be a set of state transition (I, J). R is *consistent* with E iff $b(R) \subseteq I$ implies $\exists (I, J) \in E, h(R) \in J$. A logic program P is *consistent* with E if all rules of P are *consistent* with E.

Definition 2 (Program realization)**.** Let P be a logic program and E be a set of state transitions. P realizes E if $\forall (I, J) \in E, \exists R, b(R) \subseteq I, (I \setminus J) = \{h(R)\}$.

Definition 3 (Asynchronous successors)**.** Let I be the current state of an asynchronous system represented by a set of multi-valued rules S. Let $T_P(I, S) = \{h(R) | R \in S, b(R) \subseteq I\}$. The successors of I according to S is

$$T_P^{as}(I, S) = \{I \setminus \{v^{val'}\} \cup \{v^{val}\} | v^{val'} \in I, v^{val} \in T_P(I, S)\} \cup \{I \mid T_P(I, S) = \emptyset\}$$

We now adapt the **LFIT** algorithm of [7] to the learning of asynchronous systems. In synchronous case, the rules R learned by **LFIT** represent a necessity: $h(R)$ *will* be in the next state if R match the current state. In asynchronous case, the rules represent a possibility: $h(R)$ *can* be in next state if R match the current state. It allows the modeling of non-determinism: two rules R, R' can have the same head variables but different values and match the same state

which occurs in these case: $h(R) = var^{val}, h(R') = var^{val'}, val \neq val'$ and $var^{val''} \in b(R), var^{val'''} \in b(R') \implies val'' = val'''$.

Like in previous versions, **LFIT** takes a set of state transitions E as input and outputs a logic program P that realizes E. In [5] multi-valued least specialization was used to learn multi-valued **synchronous** systems dynamics. Starting from the most general rules, least specialization allows to learn the minimal rules of such system iteratively from its state transition $(I, J) \in E$. For every possible var^{val}, $var^{val} \notin J$ the most specific rule that is not consistent, with the transition, i.e. an anti-rule, was generated: $MSR := var^{val} \leftarrow I$. Here, for the **asynchronous** case, this anti-rule is generated and the revision occurs only if $\nexists (I, J') \in E, var^{val'} \in J'$, i.e. it is impossible to have a transition to var^{val} from I. Each rule of the currently learned program P that subsumes such an anti-rule are specialized using least specialization. The resulting program P' is consistent and realizes all previously treated transition plus (I, J). Doing so iteratively for each transitions, the algorithm output a program P which model the dynamics of the system observed in the transitions E.

Asynchronous LFIT

- INPUT: $\mathcal{B}$ a set of annotated atoms and E a set of transitions
- Initialize $P := \{var^{val} \leftarrow \emptyset \mid var^{val} \in \mathcal{B}\}$
- For each $(I, J) \in E$
  - For each $var^{val} \in \mathcal{B}$
    - If $\nexists (I, J') \in E, var^{val} \in J'$
    - $MSR := var^{val} \leftarrow I$
    - Extract each rule R of P that subsumes MSR: $MR := \{R \in P \mid h(R) = var^{val}, b(R) \subseteq I\}, P := P \setminus MR$
    - For each $R \in MR$
      - Compute its least specialization $P' = ls(R, MSR, \mathcal{B})$.
      - Remove all the rules in P' subsumed by a rule in P.
      - Remove all the rules in P subsumed by a rule in P'.
      - Add all remaining rules in P' to P.
- OUTPUT: P

2.2 Reachability analysis

In the following definitions α is a state and ω a local state.

Definition 4 (LCG)**.** Given a logic program P, an initial state α and a target state ω, $LCG(P, \alpha, \omega) = (V_{\text{state}}, V_{\text{rule}}, Edges)$ is the smallest recursive structure with $Edges \subseteq (V_{\text{state}} \times V_{\text{rule}}) \cup (V_{\text{rule}} \times V_{\text{state}})$ which satisfies:

$$\begin{aligned} \omega &\in V_{\text{state}} \\ a_i \in V_{\text{state}} &\Leftrightarrow \{(a_i, R) | a_i \in \alpha, h(R) = a_i\} \subseteq Edges \\ R \in V_{\text{rule}} &\Leftrightarrow \{(R, b(R))\} \subseteq Edges \end{aligned}$$

where $V_{\text{state}} \subseteq \mathbf{LS}$ and $V_{\text{rule}} \subseteq P$ are the vertices of LCG.

Definition 5 (Trajectory)**.** Given a logic program P and $s_0 = \alpha$, a trajectory t from α is a sequence of rule-state pairs $t = (R_1, s_1) :: \ldots :: (R_i, s_i) :: \ldots :: (R_n, s_n)$ s.t each $i > 0, s_i \in T_P^{as}(s_{i-1}, P), s_i = (s_{i-1} \setminus v^{val} \in s_{i-1}) \cup v^{val'}, h(R_i) = v^{val'}), b(R) \subseteq s_{i-1}$. From α, the reached state s_n by t is denoted $\alpha \cdot t$.

Definition 6 (Reachability)**.** Given a logic program P, ω is said reachable in P from α iff there exists a trajectory t s.t. $\alpha \cdot t = \omega$ and is denoted $reachable(P, \alpha, \omega)$, otherwise $unreachable(P, \alpha, \omega)$.

Definition 7 (Consistent program)**.** Let P be a logic program, Re (resp. Un) be a set of reachability (resp. unreachability) properties. P is said to be *consistent* with Re (resp. Un) iff $\forall(\alpha, \omega) \in Re, \exists$ a trajectory t in P s.t. $\alpha.t = \omega$ and $\forall(\alpha, \omega) \in Un, \nexists$ a trajectory t in P s.t. $\alpha.t = \omega$.

Specializing a rule is to add elements in the body of a rule, thus to make the condition of a rule more difficult to be satisfied (in a more specialized situation) as the condition of firing becomes more strict.

Definition 8 (Least Specialization of a rule)**.** Let R be a rule, a least specialization of R is a rule $R' \in ls(R) := \{h(R) \leftarrow b(R) \cup \{var^{val}\}, \nexists var^{val'} \in b(R)\}$. If R contains already all the variables in its body, the only way to specialize R is to remove R.

Similarly, generalization of a rule is to remove certain elements in the body of a rule, thus to make the condition of a rule easier to be satisfied.

Definition 9 (Least Generalization of a rule)**.** Let R be a rule, a least generalization of R is a rule $R' \in lg(R) := \{h(R) \leftarrow b(R) \setminus \{x\}, x \in b(R)\}$.

Definition 10 (Revisable)**.** A logic program P is said revisable w.r.t. a reachability (resp. unreachability) property if: $\exists P' \in \{(P \setminus R_P) \cup \{R' \mid R \in R_P, R' \in ls(R) \cup lg(R)\}\} \mid R_P \subseteq P\}$. P is revisable w.r.t. to a set of property S: if their exists an ordering S' of the elements of S such that each ith revision, $0 \leq i \leq |S'|$, (P being the 0th revision) is revisable w.r.t. the $i+1$th property.

From definition 10, it follows that the revision of logic program P w.r.t. a set of reachability/unreachability properties S can be found (or proved to be non-existent) by brute force enumeration of all possible ordering of S and trying all possible iterative revisions of P. In the next section we propose an algorithm exploiting the LCG structure to restrict the search to valid ordering of the properties.

3 Algorithm

In this section we propose an algorithm that exploits the previous formalization to fit a logic program to reachability properties. Given a set of transition E of an asynchronous system S, a logic program P is learned via the adaptation of **LFIT** of section 2.1. When E is partial, the learned program P does not have the exact dynamics of S. Given a set of reachability properties Re and a set of unreachability properties Un of S, we propose an algorithm to revise P so

that the dynamics of P satisfy S. As discussed previously, this can be done by complete brute force but here we propose a first attempt to reduce the search space. Furthermore, our aim is to find what could be considered a metric of minimal revision of P: a revision P' s.t. $\nexists P'', (P'' \setminus P \cap P'') \subseteq (P' \setminus P \cap P')$

Specialization/generalization operations aim to revise the rule nearest to the target state in the LCG. If it is not possible, they try to revise the successor, if there is no possible solution, return $\varnothing$ to show the input logic program is not revisable. Specialization operation is limited by the observation. If P after specialization can not explain all the transitions, the specialization is not admissible. Generalization is similar but without the constraint of the observation, as the observation is partial, P may describe some state transitions never observed.

Specialization:

- Input: a logic program P, an unsatisfied element (α, ω), a reachable set Re, an unreachable set Un
- Output: modified logic program P or $\varnothing$ if not revisable

1. $Rev \leftarrow \{\omega\}$
2. For each R s.t. $h(R) = Rev$, for each $R' \in \{R'' | R'' \in ls(R) \wedge \nexists (I, J) \in E,$ s.t. $\nexists R''' \in P \cup \{R''\} \setminus \{R\}, h(R''') \in J, b(R''') \in I\}$
   - If $P' \leftarrow P \setminus \{R\} \cup \{R'\}$, $unreachable(P', \alpha, \omega)$ and P' satisfies all previous properties, return P'
3. $Rev \leftarrow b(R)$ with $h(R) = Rev$ and back to step 2
4. There is no revision for (α, ω), return $\varnothing$

Generalization:

- Input: a logic program P, an unsatisfied element (α, ω), a reachable set Re, an unreachable set Un
- Output: modified logic program P or $\varnothing$ if not revisable

1. $Rev \leftarrow \{\omega\}$
2. For each R s.t. $h(R) = Rev$, for each $R' \in lg(R)$
   - If $P' \leftarrow P \setminus \{R\} \cup \{R'\}$, $reachable(P', \alpha, \omega)$ and P' satisfies all previous properties, return P'
3. $Rev \leftarrow b(R)$ with $h(R) = Rev$ and back to step 2
4. There is no revision for (α, ω), return $\varnothing$

Complete revision:

- Input: a logic program P, a reachable set Re and an unreachable set Un
- Output: revised logic program P or $\varnothing$ if not revisable

1. Construct the cycle-free LCGs for the elements in Re and Un and compute unsatisfied sets $Re' \subseteq Re$ and $Un' \subseteq Un$ which are to be revised
2. If $Re' = \varnothing$ and $Un' = \varnothing$, return P
3. Let $L = \{l_i, \ldots\}$ with $i \in Re' \cup Un'$, $l_i = \{j, \ldots\}$, with $j = (\alpha, \omega)$, $\omega \in LCG(i)$ and $j \in Re \cup Un$
4. Pick one of $l_i \in L$ of the smallest cardinality: $\nexists l'_i$, $|l'_i| < |l_i|$
5. If $l_i \cap (Re' \cup Un') \neq \varnothing$,
   (a) Reconstruct the LCG for i
   (b) If l_i becomes consistent because of former revision, $L \leftarrow L \setminus \{l_i\}$ and back to step 4
6. If $i \in Un'$, specialize P to make i unreachable, if not revisable, return $\varnothing$
7. Otherwise generalize P to make i reachable, if it is not revisable, return $\varnothing$
8. $L \leftarrow L \setminus \{l_i\}$
9. If $L \neq \varnothing$, back to step 1
10. Return P

The main algorithm starts with constructing the LCGs to verify Re and Un in order to ensure the reachability/unreachability properties to be satisfied. Then, for the unsatisfied properties, the program P has to be revised. LCG can share the elements s.t. revising one can modify the others. By starting with the LCGs with least dependencies with others, i.e. the ones with the smallest cardinality of l_i, it increases the chance of partially satisfying other unsatisfied properties (step 3 and 4). Then all possible revision of P are generated using least specialization or generalization according to $l_i \in Re$ or $l_i \in Un$ (step 6 and 7). Each revision of P is checked against Re and Un to verify that all properties satisfied by P are still satisfied. If new ones are satisfied, L is updated accordingly (step 5). We update P until there is no unsatisfied properties (step 8 and 9). Finally, if a revision of P consistent with all given properties is found the algorithm terminates and output it.

4 Conclusion

In this paper, we strengthen the capability of LFIT framework to the learning of Boolean asynchronous systems in the form of logic programs. Asynchronicity implies non-determinism which is meaningful to the modeling of uncertain parts in biology. We propose a method revising the logic program learned by LFIT w.r.t. the knowledge on reachability properties. If the logic program is revisable, the revision is consistent with both state transitions and reachability information. However, our algorithm does not guarantee the minimal revision of the logic program. As future works, considering the metric for minimal revision and designing a related algorithm will be interesting. Adapting more dynamical properties other than reachability also remains as our future work.

References

[1] G. Bernot and F. Tahi. Behaviour preservation of a biological regulatory network when embedded into a larger network. *Fundamenta Informaticae*, 91(3-4):463–485, 2009.

[2] X. Chai, M. Magnin, and O. Roux. A Heuristic for Reachability Problem in Asynchronous Binary Automata Networks, 2018. arXiv:1804.07543v1.

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

[4] K. Inoue. Logic programming for boolean networks. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Two*, pages 924–930. AAAI Press, 2011.

[5] D. Martínez Martínez, T. Ribeiro, K. Inoue, G. Alenyà Ribas, and C. Torras. Learning probabilistic action models from interpretation transitions. In *Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015)*, pages 1–14, 2015.

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

[7] T. Ribeiro and K. Inoue. Learning prime implicant conditions from interpretation transition. In *Inductive Logic Programming*, pages 108–125. Springer, 2015.