THÈSE DE DOCTORAT DE

L'ÉCOLE CENTRALE DE NANTES
COMUE UNIVERSITE BRETAGNE LOIRE

Ecole Doctorale N°601
Mathèmatique et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

« Xinwei CHAI »

« Reachability Analysis and Revision of Dynamics of Biological Regulatory Networks »

«Analyse d'accessibilité et révision de la dynamique dans les réseaux de régulations biologiques»

Thèse présentée et soutenue à L'ÉCOLE CENTRALE DE NANTES, le 24 mai, 2019

Rapporteurs avant soutenance :

Gilles Bernot	Professeur des universités	Université Côte d'Azur, Sophia Antipolis
Pascale Le Gall	Professeur des universités	CentraleSupélec, Gif sur Yvette

Composition du jury :

Président :	Béatrice Duval	Professeur des universités	Université d'Angers
Examinateurs :	Gilles Bernot	Professeur des universités	Université Côte d'Azur, Sophia Antipolis
	Pascale Le Gall	Professeur des universités	CentraleSupélec, Gif sur Yvette
	Morgan Magnin	Professeur des universités	École Centrale de Nantes
	Loïc Paulevé	Chargé de recherche	Université de Bordeaux
	Olivier Roux	Professeur des universités	École Centrale de Nantes
Dir de thèse :	Olivier Roux	Professeur des universités	École Centrale de Nantes
Co-dir. de thèse :	Morgan Magnin	Professeur des universités	École Centrale de Nantes

Acknowledgement

First and foremost, I would like to express my sincere gratitude to my advisors Prof. Olivier ROUX and Prof. Morgan MAGNIN for the continuous support of my Ph.D. study for their patience, ideas and great contribution of time. This kind support also came to my personal life which helped me to regain motivation and carry on the research when I encountered at the same time academy and mental difficulties. I especially appreciate their tolerance allowing me to explore on my will even the outcome was not satisfying.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Gilles BERNOT, Prof. Pascal LE GALL, Prof. Béatrice DUVAL and Dr. Loïc PAULEVÉ, for their patient reading, insightful comments and encouragement, but also for the questions which incented me to widen my research from various perspectives.

My sincere thanks also goes to my kind team members. Dr. Emna BEN ABDALLAH, Dr. Maxime FOLSCHETTE, Samuel BUCHET and Dr. Loïc PAULEVÉ provided me an opportunity to work with them even after their graduation. Without their precious but abundant support and our interesting discussion, it would not be possible for me to conduct this research. I want to especially acknowledge Tony RIBEIRO Sensei for his kindness, intensity and genius ideas which and accelerated my Ph.D. study. I would like to show my deep respect to our collaborator Prof. Katsumi INOUE, who is kind to all, enthusiast in research, and showed us remarkable inspect in various domains during our collaboration of three months. Without his help, I could not have redirected my research.

I gratefully acknowledge the funding provided by China Scholarship Council (CSC) that made my Ph.D. work possible.

My life in Nantes was enriched by my warm-hearted neighbors Geneviève ROCHE and the family of MARCHAND. Also I thank my friends in École Centrale de Nantes for all the fun we have had in the last four years.

Last but not the least, I would like to thank my family for all their love and encouragement. For my parents who raised me with curiosity in science and supported me in all my pursuits. For my uncle who talked with me intimately in my sadness and confusion. For my girlfriend who accompanied me spiritually everyday across the oceans.

Thank you.

Xinwei Chai
École Centrale de Nantes
May 2019

Contents

List of Figures

List of Tables

Abstract

Concurrent systems have been of interest for decades. With their simple but expressive semantics, concurrent systems become a good choice to fit the data and analyze the underlying mechanics. However, *learning* and *analyzing* such concurrent systems are computationally difficult. When dealing with big data sets, the state-of-the-art techniques appear to be insufficient, either in term of efficiency or in term of precision.

In this thesis, we propose a refined modeling framework ABAN (Asynchronous Binary Automata Network) and develop reachability analysis techniques based on ABAN: PermReach (Reachability *via* Permutation search) and ASPReach (Reachability *via* Answer Set Programming). Then we propose two model learning/constructing methods: CRAC (Completion *via* Reachability And Correlations) and M2RIT (Model Revision *via* Reachability and Interpretation Transitions) using respectively continuous and discrete data to fit the model and using reachability properties to constrain the output models.

Chapter 1 states briefly the background and the contribution of our research.

Chapter 2 introduces the state of the art on modeling frameworks, model checkers, different update schemes of modelings and model learning techniques. Some of them are referenced in the following chapters.

Chapter 3 presents our modeling framework and its related reachability analyzers based on static analysis. We focus on the inconclusive cases of pure static analysis and extract the key components preventing from a direct solution. We then apply heuristics on these components, solving them with a limited search to reach a more conclusive result of the reachability problem.

Chapter 4 presents the methodology of model learning. Our model learners CRAC and M2RIT perform in fact model selection. They choose a model from the candidates satisfying the provided reachability constraints. However the number of candidate models can be exponential, our model learners can shrink the search space with constraints when generating the models.

Chapter 5 shows some comparative and exploratory tests and their results on the methods presented in Chapter 3 and Chapter 4. PermReach and ASPReach are more efficient than traditional model checkers on the reachability analysis and they perform a more conclusive analysis while holding the running time in the same scale as pure static analyzers have.

Chapter 6 concludes the thesis and proposes some possible future work.

Résumé

Les systèmes concurrents présentent un intérêt depuis des décennies. Avec leur sémantique simple mais expressive, les systèmes concurrents deviennent un bon choix pour ajuster les données et analyser les mécanismes sous-jacents. Cependant, *l'apprentissage et l'analyse* de tels systèmes concurrents sont difficiles pour ce qui concerne les calculs. Lorsqu'il s'agit de grands ensembles de données, les techniques les plus récentes semblent insuffisantes, que ce soit en termes d'efficacité ou de précision.

Dans cette thèse, nous proposons un cadre de modélisation raffiné ABAN (Asynchronous Binary Automata Network) et développons des techniques d'analyse d'atteignabilité basées sur ABAN: PermReach (Reachability *via* Permutation search) et ASPReach (Reachability *via* Answer Set Programming). Nous proposons ensuite deux méthodes de construction et d'apprentissage des modèles: CRAC (Completion *via* Reachability And Correlations) et M2RIT (Model Revision *via* Reachability and Interpretation Transitions) en utilisant respectivement des données continues et discrètes pour s'ajuster au modèle et des propriétés d'accessibilité afin de contraindre les modèles résultants.

Le chapitre 1 décrit brièvement le contexte et la contribution de nos recherches. Le chapitre 2 présente l'état de l'art des modélisations, des model checkers, des différentes dynamiques associé aux modès et les techniques d'apprentissage des modèles. Certains d'entre eux sont référencés dans les chapitres suivants.

Le chapitre 3 présente notre cadre de modélisation et ses analyseurs d'accessibilité associés, qui sont basés sur l'analyse statique. Nous nous concentrons sur les cas non concluants d'analyse statique pure et extrayons les composants clés empêchant une solution directe. Nous appliquons ensuite des heuristiques sur ces composants, en les résolvant avec une recherche limitée pour obtenir un résultat plus concluant du problème d'accessibilité.

Le chapitre 4 présente la méthodologie de l'apprentissage par modèle. Nos systèmes de construction de modèles par apprentissage CRAC et M2RIT effectuent en fait une sélection des modèles. Ils choisissent un modèle parmi les candidats qui satisfont à toutes les contraintes d'accessibilité données. Cependant, le nombre de modèles candidats pouvant être de très grande taille, nos réviseurs de modèles peuvent réduire l'espace de recherche avec des contraintes lors de la génération des modèles.

Le chapitre 5 présente quelques tests comparatifs et exploratoires et leurs résultats sur les méthodes présentées aux chapitres 3 et 4. PermReach et ASPReach sont plus efficaces que les vérificateurs de modèle traditionnels pour l'analyse de l'accessibilité. Ils effectuent une analyse plus concluante tout en maintenant la durée de fonctionnement à la même échelle que les analyseurs statiques purs.

Le chapitre 6 conclut la thèse et propose des travaux futurs possibles.

List of symbols

$\wedge$	logic AND
$\vee$	logic OR
$\oplus$	logic XOR
$\|A\|$	the cardinality of set A
$[i,j]$	the interval comprising real numbers x satisfying $i \leq x \leq j$
$[i;j]$	the interval comprising integers x satisfying $i \leq x \leq j$
C_n^k	the k-combination in the set of n elements, $C_n^k = \frac{n!}{k!(n-k)!}$
a_i	automaton a is taking value i
$x :: y$	event x happens just before y in the sequence
$a.next$	the successor of a
$a.pred$	the predecessor of a
$A \rightarrow b_j$	condition A allows automaton b to reach qualitative level j
$\mathbb{N}$	the set of all natural numbers
$\bar{x}$	the mean of variable x

Chapter 1

Introduction

In the domain of systems biology, more big data are becoming available with the development of biotechnology. However, extracting information from such big data could be difficult due to its high computational complexity and the potential fuzziness of biological systems. Modelings and their related analytic techniques are drawing increasing attention. The dilemma of efficiency and precision always persists.

This thesis is dedicated to attacking this dilemma by proposing our modeling framework and its related dynamic properties analyzers as well as model learning methods.

1.1 Context and Motivations

In the studies of concurrent systems, modeling is an inevitable topic. The modeling frameworks discussed in this thesis are all designed for biological use and some features are drawn from biology but they can be potentially useful in other domains, *e.g.* robotics, human engineering.

Models are supposed to represent the operations of a real system and help one to access, analyze and control the real system. It is a tool to help people to understand the interaction of the components in real systems and the integral behavior of the systems.

A good model is a model which:

- is consistent with the corresponding real system

  The model reproduces certain important behaviors. In the ideal situation, the model bisimulates the real system.

- is observable

  To allow one to verify the behaviors, the state (historical, current and future) and the mechanics of the model have to be observable.

- allows one to access the I/O of the model

  With full control of the I/O, we can carry some unfeasible tests in real system as the mechanical of the model is known.

- has related analyzers of various properties

  Some properties are not verifiable via finite enumeration.

- can be translated from/to other models

Normally, the above metrics are self-constrained: The finer the model is, the bigger the computational complexity is (simulation, verification, *etc.*). In this thesis, we focus on modelings, their related analyzers of system properties and model revision based on these properties.

1.1.1 Models in Computational Biology

Systems biologists are interested in highly abstracted models because they need abstract representation and/or flexibility to make model compatible with unknown biological knowledge.

Tractability with big data is also important. “Big” refers to two meanings: one is that biological systems can be huge, with enormous number of components and interactions in between; another is that the number and the size of data sets can be huge.

To model the real system, we need components to represent genes, RNA messengers, proteins, metabolites, *etc.* At this stage, we can carry out a first-step abstraction. The synthesis of proteins is under the instruction of RNA messengers which are synthesized according to genes. This linear process allows to compress the three entities into one. Their inner behaviors (*e.g.* protein phosphorylation, activation/inhibition of genes) are characterized by the values of the associated entities.

1.1.2 Classification of Models

The values in models can be continuous or discrete which differentiate the modeling frameworks.

Continuous values correspond directly to the measurement and can be used in the models based on the family of Ordinary Differential Equation

(ODE), for example Stochastic Differential Equation (SDE), Delay Differential Equation (DDE).

Discrete values come from an approximation from sigmoid function to step functions. Sigmoid function is a monotonic function and its change rate is high around a certain point (Proof in Appendix C). Many biological behaviors are similar with sigmoid functions: certain entity starts to influence the system if its value goes beyond a certain threshold. If the value is far from the threshold, it is either insufficient (low level) or saturated (high level) [42, 81]. This fact inspires scientists to study discrete models. One can encode low level as 0 and high level as 1 or even add additional discrete levels to represent more behaviors.

In the term of concurrency, synchronous models make components to evolve at the same time while asynchronous ones allow at most one component to evolve at one time. Due to the fuzziness of biological system, asynchronous models are compatible with more configurations of system parameters [6].

However, the compatibility of asynchronous models is also a shortcoming as they explore more state space at each system transition compared with synchronous ones. The exploration of state space leads to so-called state space explosion problem. To deal with such problem, Paulevé *et al.* proposed the Process Hitting framework and its related static model checker [61] and Folschette *et al.* proposed Asynchronous Automata Network enriching model semantics [29]. In this thesis, we mainly work on **asynchronous discrete models** based on the models above.

1.1.3 Model Checking

If one has an existing model, he might want to know what kind of properties this model satisfies, such as fixed points, safety, reachability.

As stated in the beginning of this chapter, verifying such properties in a concurrent system is costly in computation (PSPACE-complete) [35]. We have to make a compromise on either efficiency or precision: exact analyzers are precise but need to traverse big state space; abstract analyzers solve a simplified version of the original problem so that the solution is not equivalent to the one of the original problem.

1.1.4 Model Learning

Models are built by the biological knowledge, obtained either by certain experiments, either by generalized conclusions from biologists.

Model learning turns the data coming from biological experiment into model parameters, if the modeling framework is given. We will focus on the LFIT-based method (Learning From Interpretation Transition) proposed by Ribeiro *et al.* [66, 65, 67]. LFIT is a precise learning technique which takes all the inputs into account.

However, model learning is not enough, as we are not sure the resulted models are consistent with empirical conclusions.

1.2 Problem Statement

With the background of model checking and model learning, we can now formulate the two main problems of this thesis:

- How to analyze efficiently (less runtime) and precisely (less false positive/negative rate) the reachability properties within an asynchronous discrete model?

- How to build a model from time-series data such that the model satisfies desired reachability properties?

1.3 Contributions

The main contributions corresponding to the problems are the followings:

- Development of efficient and precise heuristic reachability analyzers based on static analysis [12]

- Design of model revisers: using *a priori* knowledge and reachability properties to revise existing models

This thesis aims at solving the problems in the last sections by refining existing modeling frameworks and learning approaches:

Reachability analyzers

To solve both the state space explosion problem and the unsatisfying precision of pure static analysis, we developed two approaches based on static analysis with different weights on *efficiency* and *precision.*

Model Revisers

As far as we know, model revision based on reachability properties has never been considered in the literature. According to the problems of model inference above, we designed two algorithms:

- an algorithm based on learning from **raw** time-series data

  This algorithm uses correlation coefficients to infer the correlations between the change rate of each variable and other variables in order to suggest hypothetical regulations of the system. With the hypothetical regulations and *a priori* biological knowledge, we can revise incomplete models by adding transitions consistent with the real system.

- an algorithm based on learning from **discretized** time-series data

  The learning approach we applied is the one using Inductive Logic Programming [65]. This approach has more strict rule constraints, the model is either consistent with the original time-series data or not. We try to revise the model in order to make it consistent with *a priori* biological knowledge and the rule constraints at the same time.

1.4 Organization of the Manuscript

Chapter 2 introduces the state of the art on several modeling frameworks, model checkers, different update schemes of modelings and model learning/revision techniques. We are especially interested in Asynchronous Automata Network, reachability analysis and the learning from state transitions.

Chapter 3 presents our modeling framework adapted from Automata Network and its relating reachability analyzers based on static analysis. We will focus on the inconclusive cases of pure static analysis and analyze the key components preventing from a direct solution. We then apply some different heuristics on the key components to solve them dynamically in order to reach a conclusive result on the reachability problem.

Chapter 4 presents the methodology of model revision in this thesis and our model revisers mentioned above. These model revisers perform a model selection. They choose a model from the candidates satisfying all the provided constraints. However the number of candidate models can be huge, our model revisers can shrink the search space drastically to obtain a result.

Chapter 5 shows some comparative and exploratory tests and their results on the approaches presented in Chapter 3 and Chapter 4.

Chapter 6 concludes the whole thesis.

Chapter 2

State of the Art

This chapter is dedicated to the introduction of the basic notions of this thesis. Concretely speaking, we are going to present the state of the art constituting of mainly three basic contents:

- Several modeling frameworks, some of which are the ancestors of the new modelings to be used in this thesis and some of which will be used in comparison
- The notion of model-checking and some existing model checkers
- Different update schemes of modeling frameworks
- Several model learning/inference techniques

These notions will be helpful for readers to understand our new model-checkers and new model inference approaches.

In computational biology, there are plenty of modeling frameworks suiting different needs. Models are a useful tool to represent the abstraction of the real system, as the real system is usually mechanically complicated and not completely known.

From the point of view of data type, models are classified into continuous ones, discrete ones and their combination, hybrid models, where the last one is not the focus of this thesis.

Continuous models are usually derived from real world data, as the data (no matter time-series data or static data) are obtained by measurement and can be used as input without preconditioning. Differential equations models [33, 76, 80] are based on the hypothesis that in biological systems, the change rate of variables is numerically related to the values of other variables. Differential equations-based models deal with mainly the system dynamics, *e.g.* biological regulations. Models based on Bayesian probability and Markov chain [39, 45] study static topics, *e.g.* phylogenetics.

However, the following difficulties usually arise in the analysis of continuous models:

- System mechanics is unclear
- System parameters need to be precise which is difficult for biological system
- Solving differential equations numerically is expensive

Unlike straightforward continuous modelings, discrete modelings perform an abstraction of continuous dynamics and an over-approximation of continuous constraints [6]. Discrete models can be also inferred from discrete data by discretization [24].

On the aspect of dynamics, discrete models have the following update pattern: $discrete\_state1 \xrightarrow{conditions} discrete\_state2$, where the states correspond to the intervals in the continuous models. States here can be qualitative levels or *combination*s of qualitative levels, compensating the imprecision or the incompleteness of the original system. Conditions here could be either a necessary time delay or a state of the current system, *etc.*

As to the computation of various properties, discrete transition models generally cost less than continuous dynamical models. Among discrete models, Boolean Networks [42], Thomas Models [79], Petri nets [63], Process Hitting Framework [59] could more or less avoid these disadvantages when facing the problems related to system evolution.

Hybrid (in the sense of discrete/continuous) models [83, 48, 75] usually behave like continuous models, because in each discrete block, the sub-model also need continuous parameters as continuous models need. Their computational complexities are of the same scale. We are not going to detail hybrid models during this thesis.

Continuity characterizes the evolution pattern of each variable. However, models contain multiple variables. From the point of view on concurrency, models can be classified according to update schemes into mainly three genres: synchronous models, asynchronous models and generalized models.

Synchronous update scheme designates that every component will transit simultaneously to one of its possible future states regardless the time needed for transition or other components. However in real world, constraints always exist. Biologically it is not probable that multiple components in one system change their state simultaneously. This deficiency demands us to consider more update schemes. Detailed discussion is in Section 2.2.

In this chapter, we will first discuss several discrete modeling frameworks and updating schemes, comparing their differences and the possibility of translating from each other. Some of these models will be used in the following chapters. Models need related analytic tools, we will then study some model checkers especially those focusing on reachability properties as many other complex properties can be formulated with the help of reachability. At last we will be interested in the model learning and revising techniques which are the key of constructing a model.

2.1 Discrete Modeling Frameworks

Original biological problems are usually difficult to be studied directly due to the uncertainty and the big scale of biological systems. Modeling is a process of abstracting the real system into a more concise and more easily automatized system. To solve a certain problem, an appropriate modeling framework is crucial because different models have different bias from reality and have also different advantages in computation, *e.g.* fixed point, reachability. Here we are going to introduce several most frequenly applied modeling frameworks and analyze their advantages and disadvantages.

2.1.1 Regulatory Network

Regulatory Networks (RN) have characteristics of a static network, representing the interactions between components [6]. With the analysis of topological features, it can be applied in for example gene expression analysis [74].

Definition 2.1 (Regulatory network (RN))**.** A regulatory network is a labeled digraph $G = (V, E)$ where

- each vertex v of V, called variable, is provided with a boundary $b_v \in \mathbb{N}$ less or equal to the out-degree of v in G.
- each arc $u \in v$ of E is labelled with a couple (t_{uv}, α_{uv}) where t_{uv} is an integer between 1 and b_v, called qualitative threshold and where $\alpha_{uv} \in \{+, -\}$ is the sign of the regulation.

An example is shown in Figure 2.1 A, consisting of three entities X, Y, Z on page 11. Sharp arrows $\rightarrow$ stand for promotion while blunt arrows $\dashv$ stand for inhibition, *e.g.* X inhibits Z, Z promotes Y.

RN is a special instance among discrete modelings. It does not require a threshold setting for each variable. As a result, RN is usually applied to study static problems, *e.g.* model completion, finding fixed points [85]. Without quantitative representation, one can barely analyze the system dynamics because RN does not possess an update pattern which describes state transitions.

2.1.2 Bayesian Network

Bayesian networks are a type of probabilistic graphical models. They represent joint probability distribution of a set of variables. More concretely, if one obtains the value of certain variable, a Bayesian network can help him analyze the values of its linked variables.

Bayesian networks are usually used for inferring causal dependencies between genes in gene regulatory networks with the goal of estimating the posterior probability of chosen features being inherent in the network, given the data [30].

A Bayesian network is defined as (G, θ) where G is a *directed acyclic* graph whose vertices connect the random variables of the network. θ is a probability distribution associated to the vertices. These variables can be continuous or discrete. Directed edges correspond to dependencies between variables. θ describes a conditional distribution for each variable of the network, given its "parents" as defined by the relations in G.

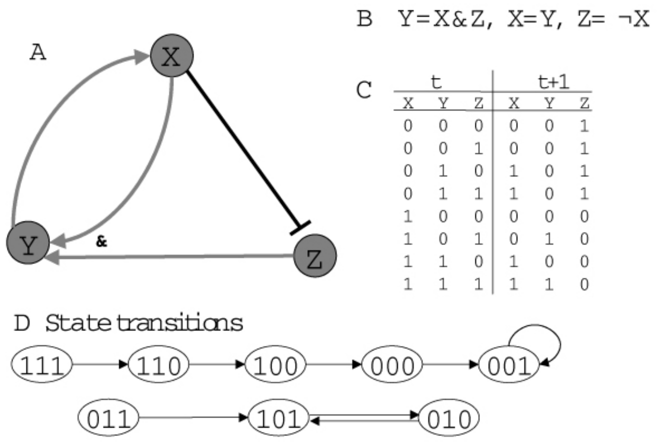
One disadvantage of Bayesian networks is that they can only represent acyclic topology as they must be acyclic in order to guarantee that their underlying probability distribution is normalized to 1. However feedback loops appear very frequently in biology which narrow the application of Bayesian networks.

2.1.3 Boolean Network

Boolean Networks (BN) are a traditional framework studied for decades [42]. They discretize every variable of the system into Boolean variables taking values 0 or 1, presenting active/inactive, high/low concentration, *etc.* Transitions in BNs are defined by Boolean functions. Here we introduce the basic definition of BN.

Definition 2.2 (Boolean Network (BN))**.** A Boolean Network $G(V, F)$ consists of a set of nodes $V = \{v_1, \cdots, v_n\}$ and a set of Boolean functions $F = \{f_1, \cdots, f_n\}$ where function f_i decides the value of node v_i of the next time point: $v_i(t+1) = f_i(v_1(t), \cdots, v_n(t))$.

In some applications, the nodes are classified into incoming nodes, outgoing nodes and inner nodes to represent an input-output system [3].



t			t+1		
X	Y	Z	X	Y	Z
0	0	0	0	0	1
0	0	1	0	0	1
0	1	0	1	0	1
0	1	1	1	0	1
1	0	0	0	0	0
1	0	1	0	1	0
1	1	0	1	0	0
1	1	1	1	1	0

Figure 2.1: Four ways of representing biological topology: A. regulatory network, B. Boolean functions, C. transition interpretation table, D. State transition graph

Figure 2.1 shows different representations of biological topology. RN (A) shows only a qualitative inference graph. BN (B) is concise but does not indicate directly the state change between moments t and $t+1$. Its update scheme needs to be precised (in Section 2.2). Transition interpretation table (C) and state transition graph (D) are straightforward but there are two drawbacks:

- the state space increases exponentially with the number of variables leading to the lack of memory
- they are not equivalent to Boolean functions

One set of transition interpretations or one state transition graph could correspond to multiple set of Boolean functions.

In term of expressiveness, BNs can be translated to Normal Logic Programs (NLP) [41]. NLP provides a more dynamical representation and also for applying SAT techniques in the computation of point attractors of both synchronous and asynchronous semantics [25, 36].

2.1.4 Normal Logic Program (NLP)

Logic programming is a type of programming paradigm which is largely based on formal logic. Any program written in a logic programming language is a set of sentences in logical form, expressing facts and rules about some problem domain. Major logic programming language families include Prolog, Answer set programming (ASP, which we will detail in Section 3.4.2 of Chapter 3) and Datalog.

In all of these languages, the basic element is called an *atom*, representing a variable with certain value. The set of atom is denoted $\mathcal{B}$. Rules are written in the form of clauses consisting of atoms (formula representation on the left while code representation on the right):

$$H \leftarrow B_1 \wedge \ldots \wedge B_n. \quad \text{or} \quad \texttt{H :- B1, ... , Bn.}$$

$H, B_1, \ldots, B_n$ are atoms. The rule is read declaratively as logical implications:

$$H \text{ if } B_1 \text{ and } \ldots \text{ and } B_n.$$

H is called the head of rule and $B_1, \wedge \ldots \wedge B_n$ is called the body of rule. Facts are rules having no body, and are written in the simplified form:

$$H.$$

Using this notation, one can describe the transition of variable:

$$var_0^{val_0}(t+1) \leftarrow var_1^{val_1}(t) \wedge \ldots \wedge var_n^{val_n}(t).$$

which reads variable var_0 will take value val_0 at the next time point if variable var_1 takes value val_1 and ... and variable var_n is taking value val_n. In this thesis we simplify the notation as

$$var_0^{val_0} \leftarrow var_1^{val_1} \wedge \ldots \wedge var_n^{val_n}.$$

Remark 2.1. NLP appears to have almost the same formula as BN according to Inoue *et al.* [41], the only difference is $\leftarrow$ of NLP and $=$ of BN. However, logic OR is not allowed in NLP. The way to represent logic OR is to use additional rules. For example, Boolean function $Z = X \vee Y$ can be translated to the following rules: $Z^1 \leftarrow X^1$, $Z^1 \leftarrow Y^1$ and $Z^0 \leftarrow X^0 \wedge Y^0$. This translation may create non-equivalence and non-determinism.

Detailed applications are in Section 4.4.1 of Chapter 4.

2.1.5 Process Hitting (PH)

If one wants to describes the dynamics more finely with reasonable memory use, Process Hitting (PH) framework is a good choice, which was introduced by Paulevé *et al.* [59].

PH is inspired by π-calculus, which expresses the communication between canals. In PH, the corresponding meaning becomes the interaction between different components. Process Hitting is an asynchronous automata network, *i.e.* allowing at most one transition fired simultaneously. PH is more expressive than Asynchronous Thomas' model [79] or Asynchronous BN [32]. Actions in Process Hitting are more capable of describing various transitions than Boolean functions or attractors as it specifies the regulating and regulated components and their quantitative levels. Also it expresses explicitly the cooperation between several components and stochastic features using π-calculus which are not detailed in this thesis [58].

Moreover, in order to define efficient analysis techniques that avoid to build the whole state space of the model causing state space explosion (in Thomas' model and Boolean network), various abstract structures have been introduced, and one of them is graph of causality, which allows a reasoning of the reachability of local states instead of traverse of global states.

It gathers a finite number of concurrent processes grouped into a finite set of sorts. A process belongs to one and only one sort and is denoted as a_i where a is the sort and i the identifier of the process within the sort a. At any time, only one process of each sort is present, forming a global state of the PH.

Definition 2.3 (Process Hitting (PH))**.** A PH consists of a tuple (Σ, L, H):

- $\Sigma = \{a, b, ...\}$ is the finite set of sorts
- $L = \prod_{a\in\Sigma} L_a$ is the set of states with $L_a = \{a_0, ..., a_{l_a}\}$ the finite and countable set of states of sort $a \in \Sigma$ and l_a a positive integer with: $a \neq b \rightarrow \forall(a_i, b_j) \in L_a \times L_b, a_i \neq b_j$
- $H = \{h = a_i \rightarrow b_j \Rsh b_k \mid (a, b) \in \Sigma^2, (a_i, b_j, b_k) \in L_a \times L_b \times L_b, b_j \neq b_k, a = b \rightarrow a_i = b_j\}$ is the finite set of actions, which defines the regulations and dynamics of the PH: a_i, b_j, b_k are denoted $hitter(h)$, $target(h)$ and $bounce(h)$ respectively of the action $h = a_i \rightarrow b_j \Rsh b_k$.

Example 2.1. Figure 2.2 shows a PH $PH = (\Sigma, L, H)$ constituting of four sorts $\Sigma = \{a, b, c, d\}$, and the possible states of each sort are $L_a = \{a_0, a_1\}$, $L_b = \{b_0, b_1, b_2\}$, $L_c = \{c_0, c_1\}$, $L_d = \{d_0, d_1, d_2\}$. The initial state of PH is $\alpha = \langle a_1, b_0, c_0, d_1\rangle$. It has actions $H = \{a_0 \rightarrow c_0 \Rsh c_1,\ a_1 \rightarrow b_1 \Rsh b_0,\ c_1 \rightarrow b_0 \Rsh b_1,\ b_1 \rightarrow a_0 \Rsh a_1,\ b_0 \rightarrow d_0 \Rsh d_1,\ d_1 \rightarrow b_0 \Rsh b_2,\ c_1 \rightarrow d_1 \Rsh d_0\}$. For example $a_0 \rightarrow c_0 \Rsh c_1$ means if sort a is at level 0, it allows sort c to jump from level 0 to level 1.

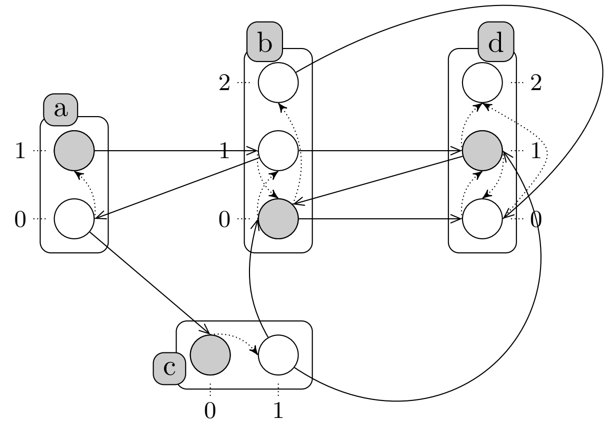


Figure 2.2: Gray circles represent the initial state of sorts. Full arrows are regulations while the following dashed arrows are the actions under the condition of the corresponding regulations.

Nevertheless, PH cannot encode equivalently the conjunctions in Boolean functions like $f(a) = b \wedge c$.

To overcome this drawback, it is needed to introduce a cooperative sort bc to represent the conjunction of b and c, with 8 actions $b_0 \rightarrow bc_{10} \Rsh bc_{00}$, $b_0 \rightarrow bc_{11} \Rsh bc_{01}$, $b_1 \rightarrow bc_{00} \Rsh bc_{10}$, $b_1 \rightarrow bc_{01} \Rsh bc_{11}$, $c_0 \rightarrow bc_{01} \Rsh bc_{00}$, $c_0 \rightarrow bc_{11} \Rsh bc_{10}$, $c_1 \rightarrow bc_{00} \Rsh bc_{01}$, $c_0 \rightarrow bc_{10} \Rsh bc_{11}$.

However, the size of this representation grows exponentially with the size of the conjunction and the behavior of cooperative sorts are not equivalent to that of BN. Also, this encoding introduces extra reactions, producing a temporal shift between the presence of the reactants and the playability of the reaction.

2.1.6 Asynchronous Automata Network (AAN)

Facing the drawback of PH, *i.e.* only cooperative sorts can encode reactions with conjunctions in the *hitters*, Asynchronous Automata Network (AAN) is introduced by Folschette *et al.* [29]. AAN allows one to naturally model cooperations by defining several requisites for a transition. Moreover, such automata networks are still compatible with the notion of priority, that can also be used to model different reaction rates in the model. AAN (and, a fortiori, their restriction, the PH framework) can be considered as a subset of Communicating Finite State Machines or safe Petri Nets [60].

Basically the main difference between PH and AAN is state below. Sorts in PH are called automata in AAN. Also, the definition of H becomes

- $H = \{A \to b_j \Rsh b_k \mid b \in \Sigma \wedge (b_j, b_k) \in L_b \times L_b \wedge b_j \neq b_k \wedge \forall a \in \Sigma,\ |A \cap L_a| \leq 1 \wedge A \cap L_b = \varnothing\}$ is the finite set of actions, which defines the regulations and dynamics of the AAN: A, b_j, b_k are denoted $\mathit{hitter}(h)$, $\mathit{target}(h)$ and $\mathit{bounce}(h)$ respectively of the action $h = A \to b_j \Rsh b_k$.

where the action conditions are no longer limited to the state of only one automaton. With the new definition of H, we can now define actions needing multiple local states as condition.

In Chapter 3, we will make use of the definition of AAN by limiting the variables from multi-value ones to Boolean ones in order to obtain some interesting properties in the forthcoming reachability analysis.

2.2 Semantics of Modelings

For a given modeling framework, even if the components and transitions are defined, the dynamics of the system is not unique. Different update schemes lead to different dynamics. The main difference lies on the relations of the number of transitions that *can* be fired and the number of transitions that *will* be fired at given time point t [65, 14].

2.2.1 Synchronicity

Literally, synchronous update scheme implies that all fireable transitions are fired simultaneously.

Example 2.2. Taking the transition interpretation table in Table 2.1 as the given system dynamics, Figure 2.3 (a) shows the synchronous case.

t		$t+1$	
u	v	u	v
0	0	2	0
1	0	2	1
2	0	2	1
0	1	0	0
1	1	0	1
2	1	2	1

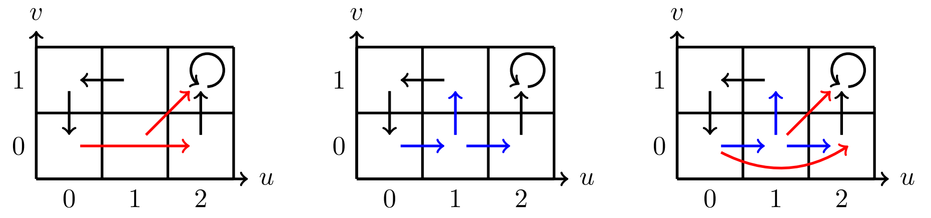
Table 2.1: Exemplary transition interpretation table indicating the tendency of system evolution from one state to another.

Synchronous update scheme seems to be deterministic. However, when there are multiple fireable transitions available for one variable, there are multiple possible future states which cannot be fired simultaneously.

Example 2.3. Given an NLP with rules $var_3^1 \leftarrow var_1^1$ and $var_3^2 \leftarrow var_2^1$ and initial state $\langle var_1^1, var_2^1, var_3^0 \rangle$, these two rules are in conflict. Even though the semantics is synchronous, a choice is need to be made between these transitions to decide the next state of var_3 is 1 or 2.

To avoid the conflicts in Example 2.3, one possible solution in Boolean NLP is to clarify the state transition metrics: for one variable, if it *can* change its value at the next time point, it cannot keep its current value the NLP is Boolean, there is only one choice for it.

On the computational aspect, one of the benefits of the synchronous model is tractability, while classical state space exploration algorithms fail if there are multiple possible future states at each state transition.



(a) Synchronous semantics (b) Asynchronous semantics (c) Generalized semantics

Figure 2.3: State transition graphs of different updating schemes

2.2.2 Asynchronicity

For biological applications, asynchronous semantics is said to capture more realistic behaviors: at a given time, a single gene can change its expression level. However, these rich behaviors result in a potential combinatorial explosion of the number of reachable states.

Example 2.4. Let us take the same transition interpretation table as Example 2.3, due to the limit of number of variables which can change their values, the asynchronous case is shown in Figure 2.3 (b).

Given BN with n variables, from a certain state, for deterministic synchronous semantic, there is only one path to be exploited. However, for asynchronous semantic, there are at most n future states, after t step of evolution, there are $O(n^t)$ possible branches in the arborescent searching graph which leads to state space explosion problem.

Another problem of asynchronous update scheme is the compatibility with time series data. One cannot guarantee when timeline is discretized evenly whether data of adjacent time points have at most one state transition. Our solution is to discretize timeline according to the moments when variables change their qualitative levels.

Example 2.5. In Figure 2.4, let us consider a dense enough time-series data (quasi-continuous) with two variables $x = \cos(t)$ and $y = \sin(t)$. The thresholds for both variables are set to 0.5. The left diagram shows equi-temporal discretization of step size 0.5π, while the right diagram discrete time at the moment when x and y reach the threshold, *i.e.* $\pi/6, \pi/3, 5\pi/6, 5\pi/3$. The latter discretization has no conflict with asynchronous update scheme, as there are at most one variable changing its value at each time point.

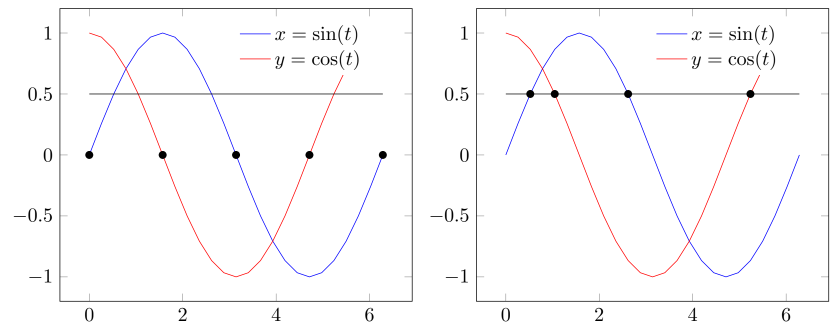


Figure 2.4: Adapted version of discretization to asynchronous systems

This PhD thesis focuses on the study of reachability of asynchronous modeling frameworks. We will still introduce in the last part of this section, a more global semantics.

2.2.3 Generalized Semantics

Generalized semantics is even more complex than asynchronous semantics. At any given time, the system can change the values of *any* number of variables. Given a BN and a current state, if there are m variables which may change their value at the next time point, there will be 2^m possibilities of the next state. Generalized semantics contains synchronous and asynchronous semantics.

Example 2.6. Let us take the same transition interpretation table as Example 2.3, we can update any number of variables at each time point. The generalized case is shown in Figure 2.3 (c).

To sum up, Table 2.2 shows the difference of those three update schemes.

For synchronous update scheme, m is the number of different variables in all the heads of rule. The formal definition m is as follows: let R be the set of rules and current state be $S = \langle var_0^{val_0}, \cdots, var_n^{val_n}\rangle$. $V = \{head(r) \mid \exists r \in R, body(r) \subseteq S\}$, then $m = |V|$.

For synchronous cases we can consider the possible future states is of $O(1)$ as it is little probable that many rules are in conflict even though one can create an extreme counterexample making the number of possible future states reach its theoretical limit $O(3^m)$ if the number of total states of all the variables is fixed (the proof is given in Appendix C). The possible future states is polynomial for asynchronous cases. However generalized update scheme has to consider the combinatorial results of the fireable transitions.

As to asynchronous case, there can be no fireable transitions where $n = 0$. Hence, the number of transitions will be fired is $\min(1, n)$.

The benchmark part of [65] shows the complexity of generalized semantics, where the model inference fails with 12 components in the model.

2.3 Model Checking

Model Checking is an automatic verification technique for large state transition systems and was independently developed by Clarke and Emerson [19] and by Queille and Sifakis [64] in the early 1980s. It was originally developed for reasoning about finite-state concurrent systems. Typically, a

	Synchronous	Asynchronous	Generalized
nb of fireable transitions		n	
nb of transitions *will* be fired	m	$\min(1, n)$	$[0; m]$
nb of possible future states	at most $O(3^m)$	m	$[2^m; 2^n]$

Table 2.2: Numbers of fireable transitions in different updating scheme, where m stands for the number of different variables in all the heads of fireable transitions.

model checker has three basic components: a modeling formalism adopted to encode a state machine representing the system to be verified, a specification language based on Temporal Logic, and a verification algorithm [20] which employs an exhaustive searching of the entire state space to determine whether the specification holds or not.

In this thesis we focus on reachability (**EF** in Temporal Logic) as most temporal properties can be reduced to reachability problems due to the expressiveness of hybrid modeling frameworks.

2.3.1 Exact Model Checkers

At first, Model Checking was done by the search in the state transition graphs, which are encoded in adjacent lists [19]. This representation however requires a memory growing exponentially with the number of components. To avoid such explicit representation, state transition graphs were replaced by Boolean formulas. OBDD-based (Ordinary Binary Decision Diagram) Model Checkers were developed, having reached 10^{120} states, *e.g.* SMV [51], NuSMV [16], and VIS [9]. However, the performance is still not enough to analyze problems in systems biology due to their complexity (PSPACE-complete) [35]. The tests are illustrated in Chapter 5.

2.3.2 Static Analyzers

Model Checkers are widely applied to hardware and software. Especially when applied to software, algorithmic verification techniques have to deal with infinite state space of software, requiring abstraction techniques to make problems tractable. SPIN [38] and Goanna [27] are designed as source code analyzers, by verifying a set of over-approximate conditions, they managed to check the safety/liveness properties of a program or whether certain program behaves as expected. However, the static code analyzers only determine run-time properties of programs by examining the code structure, which may produce false-positive and false-negative results [82].

Inspired by these ideas, Pint [55] takes the initiative to apply pure static analysis, combine over-approximation and under-approximation [61] to squeeze the state space in order to try to solve the original reachability problem of a PH or an AAN (See Figure 2.5). Similarly, due to the approximations, the result of Pint is not necessarily conclusive [29].

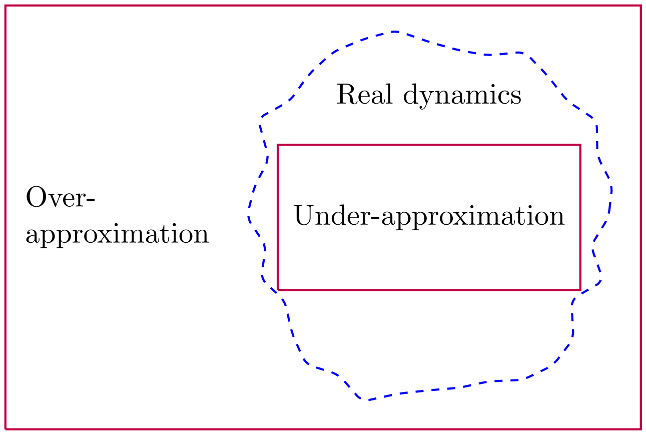


Figure 2.5: Schema of real dynamics and over-approximation and under-approximation in Pint

2.3.3 Reachability Problem

In the domain of model checking, reachability has been of great interest for over 30 years [18, 20]. Various modeling frameworks and semantics in bioinformatics have been studied: Boolean network [3], Petri nets [50, 26], timed-automata [21, 84]. These approaches rely on global search and thus face state explosion problem as the state space grows exponentially with the number of variables. In [62], it has been shown that the reachability problem of Petri net is exponential time-hard and exponential space-hard, and this conclusion does not change even under some specific conditions [26]. For 1-safe Petri nets, the complexity of reachability analysis is generally PSPACE-complete [15]. Li *et al.* [46, 47] investigated theoretically the stability, the controllability and the reachability of Switched Boolean Networks, but their method remains computationally expensive; Saadatpour *et al.* [70] researched only the reachability of fixed points.

To tackle the complexity issue, symbolic model checking [10] based on OBDDs and SAT-solvers (satisfiability) [2] have been studied over years, but still fail to analyze big biological systems with more than 1000 variables.

Bounded Model Checking (BMC) [17] is an efficient approach but generally not complete as its searching depth is limited to a given integer k.

Model checking is not only related to the verification of models, it can be of help to the learning of model and modification according to the unsatisfied properties.

2.4 Model Learning and Model Revision

All the modeling frameworks and model-checkers mentioned above are not effective unless they are fed with trustworthy system topology. Model learning is used to classify the original data into generalized topological knowledge of the system where the original data come from raw data after being discretized, normalized, denoised *etc.* In this thesis we do not assess the quality of raw data as it is related to the results of biological experiments.

Among the important contributions, Khalis *et al.* [43] have studied parameter learning in a given model topology. Rodrigues *et al.* [69] have studied active learning of relational action model whose dynamics is however not compatible with BRNs. Bonneau *et al.* [7] have developed a learning algorithm based on regression but with the limit on the size of clusters. Opgen-Rhein *et al.* [54] have studied the learning using correlation coefficients, but their resulting regulatory networks are undirected. Ribeiro *et al.* have designed LFIT-based (Learning From Interpretation Transitions) learning methods [66, 65, 67] but these approaches cannot deal with noisy or imprecise inputs as all the errors are taken into account during the learning phase.

However, the sensitivity of LFIT can be of benefit in model revision. The choices of revised models are often combinatorial w.r.t the satisfiability of certain dynamic properties. If we consider another constraint that the revised model has to reproduce exactly the input time-series data, the number of consistent revised models will decrease. Here we introduce some basic ideas of LFIT which will be of help in Chapter 4.

As for model revision, to our knowledge, model revision based on reachability properties has never been considered in the literature. One possible related work is cut set [57] to be introduced after LFIT. Cut set is used to detect the atoms cutting critical paths from the initial state and the desired states. By inhibiting these elements, we can ensure the unreachability of the desired states. In practice, cut sets are useful for proposing potential therapeutic targets that have been formally identified from the model for preventing the activation of a particular molecule. However, cut sets are not of direct help in model revision, as they inhibit certain atoms to be reached rather than modifying the system topology.

2.4.1 Learning From Interpretation Transitions (LFIT)

Ribeiro *et al.* [66] have designed LFIT aiming at learning a logic program P (definition in Section 2.1.4 on page 12) from a set of state transitions E in the form $S_1 \rightarrow S_2$ where S_1, S_2 are the states of the system. E can be extracted from discretized time-series data. Here LFIT considers only synchronous updating scheme, hence we name it as "synchronous LFIT".

Basically, the mechanics of synchronous LFIT is as follows: it starts from the most general logic program. It verifies whether every element in E is consistent with P. Obtained inconsistencies are classified as conflicts. Then LFIT tries to specialize the conflicting rules by adding atoms in their body to make them harder to be matched. When there is no conflict, P can reproduce perfectly all the state transitions in E.

Algorithm 2 in Appendix B describes the details of synchronous LFIT. To prevent potential ambiguities, we denote assignment operation as $:=$ instead of $\leftarrow$.

2.4.2 Cut set

Paulevé *et al.* [57] have designed an algorithm for identifying sets of atoms whose activity is necessary for the reachability of a given local state. If all the atoms from such a set are disabled in the model, the concerned reachability is impossible. Those sets are referred to as cut sets and are computed from a particular abstract causality structure, so-called Local Causality Graph (detailed in Section 3.2.2). *Via* such manipulation on atoms, one may control certain dynamical properties of a BRN.

However, we are going to try to control the systems dynamics by revising the transition rules of a model instead of inhibiting/imposing certain atoms. This need urges us to make modifications on the elements to be inhibited. The detail of cut sets is in Section 4.2.2.

2.5 Résumé

In this chapter, we presented mainly four main basic contents of the state of the art:

- Modeling frameworks

- Update schemes of models
- Model checking and model checkers
- Model learning/inference techniques

Modeling frameworks allow one to encode real biological regulatory systems into computable models according to his need (completion, static analysis, dynamic analysis, simulation *etc.*). In the next chapter, we will focus on a finer reachability analysis. To achieve this goal, we will propose a new modeling framework ABAN based on existing ones presented in this chapter. Also, to better position our new modeling framework and define its dynamics, we introduced different update schemes of models. To validate and evaluate our work (in Chapter 5), we introduced here several representative model checkers to be used for comparison. With the help of model learning technique LFIT and our reachability analysis methods, we managed to develop a model revision method based on desired reachability properties which has never been consider before.

Chapter 3

Refined Reachability Analysis *via* Heuristics

Several modeling frameworks and model checking techniques were introduced in Chapter 2. We noticed that even though there exist already exact model checkers and static analyzers for reachability problems, they are not sufficient. Exact model checkers always face the state space explosion problem when analyzing large models (with more than 50 variables); Static analyzer PINT, designed for Process Hitting/Automata Network is however, theoretically inconclusive, *i.e.* not able to provide a global solution to arbitrary input. This chapter is going to deepen into the reachability problem for Asynchronous Binary Automata Network *via* the following steps:

- Why the inconclusiveness problem arises in static analysis methods
- What are the problematic topological structures in static analysis
- How to deal with such topological structures

As a result, we try to recover the consequence of the information lost due to non-exhaustive search of static analysis and construct a more close approximation of the real dynamics in order to gain a better conclusiveness.

The contribution of this chapter was published and presented at SASB 2018 in Freiburg, Germany [12].

In this chapter, we are going to formally define the main modeling framework studied in this thesis, Asynchronous Binary Automata Network (ABAN) and the related static analyzer we have specifically designed, Simplified Local Causality Graph (SLCG). These two new definitions based on the one of Automata Network in order to adapt to our new reachability analyzers.

Also, to deal with the inconclusiveness problem persisting in previous work [29], we propose at first doing some preprocessings by simplifying the topology of the models in order to try to remove the parts leading to inconclusiveness. Then we will introduce two new analyzers (PermReach and ASPReach) based on over-approximation. They perform different heuristics, trying to avoid most of the inconclusiveness due to pure static analysis.

3.1 Background

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

To tackle the complexity issue, symbolic model checking [10] based on ordered binary decision diagrams (OBDDs) [34] and that based on SAT-solvers (satisfiability) [2] have been studied over years, but still fail to analyze big biological systems with more than 1000 variables. Bounded Model Checking (BMC) [17] is a state-of-the-art approach, it is efficient but generally not complete as its searching depth is limited to a given integer k. One has no idea whether there exists a solution beyond step k or not.

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

However, like BMC, abstract approaches do not solve all the instances. In fact, they solve a simplified version instead of the original reachability problem. If the result of the simplified version is not sufficient to imply the one of the original problem, abstract approaches fail (inconclusive). In the following, we are going to formally define the reachability problem and discuss what are the causes of the inconclusiveness and how to solve them.

3.2 Asynchronous Binary Automata Network

In [29], Paulevé *et al.* have worked on the modeling of concurrent systems by Asynchronous Automata Network (AAN) and they invented Local causality graph (LCG) [56, 29, 59] to analyze the reachability of AAN. This interpretation drastically reduces the searching state-space thus avoids costly global search [61]. However, this pure static analysis is not complete as there are inconclusive cases which can not be decided reachable or not. LCG can only conclude with the following two constraints:

- With no cycles (Section 3.3.1)
- With no **AND gates** (Section 3.2.3)

We are thus going to refine the reachability analysis to deal with more instances. To attack the inconclusiveness problem, we have designed a new discrete modeling framework for concurrent systems [11]: Asynchronous Binary Automata Network (ABAN). In ABAN, we adapted LCG to SLCG (Simplified LCG) to address reachability problem. This approach refers to a static abstraction of the reachability (with an over-approximation of the real dynamics). In binary situation, the approximation of reachability has some interesting properties simplifying the whole reachability analysis. These properties do not hold in multi-valued models (See Section 3.2.2).

3.2.1 Definitions

Definition 3.1 (ABAN)**.** An ABAN is a tuple $\mathbb{A} = (\Sigma, T)$, where:

- $\Sigma = \{a, b, \ldots\}$ is the finite set of automata with every automaton having a Boolean state;
- The states of $\mathbb{A}$ can then be defined: $LS = \bigcup\limits_{a \in \Sigma} \{a_0, a_1\}$ is the set of all *local states*, $L = \bigtimes\limits_{a \in \Sigma'} \{a_0, a_1\}$ is the set of *joint states* where $\Sigma' \subseteq \Sigma$. Particularly, if $\Sigma' = \Sigma$, L is the set of *global states.*
- $T = \{A \to b_i \mid b \in \Sigma \wedge A \in L\}$ is the set of transitions. For transition $tr = A \to b_i$, A (called head, noted $head(tr)$) is the set of required state(s), which allows to flip b_{1-i} to b_i (called body, noted $body(tr)$). In other words, transition tr is said fireable iff $A \subseteq s$, where s is the current global state.

Remark: In AAN and PH, their transitions (or called actions) are noted $A \to b_i \Rsh b_j$ to express "automaton b changes its value from i to j under condition A". However, the states in ABAN are all binary, the transition can only be realized from 0 to 1 or conversely. Thus we omit the state before transition while avoiding ambiguity. It might also be noted the notation $A \to b_i$ resembles the equivalent notation in Normal Logic Program (NLP): $b_i \leftarrow A$.

Also, the notions of different states are *crucial* in this thesis. A *local state* represents the state of one automaton, *e.g.* a_1 means automaton a is at level 1. A *joint state* represents the state of a set of automata, *e.g.* $\langle a_1, b_0 \rangle$ means automaton a is at level 1 and automaton b is at level 0. In fact, when we take all the automata in the system as the set of automata, the corresponding *joint state* becomes the state of the whole system, which is the global state, *e.g.* given $\Sigma = \{a, b, c\}$, $\langle a_0, b_1, c_0 \rangle$ shows the state of *all* the automata. To conclude, joint state is the most general case, when $|\Sigma'| = 1$, it becomes a local state; when $\Sigma' = \Sigma$, it becomes a global state.

Definition 3.2 (Dynamics)**.** From current global state s, the global state after firing transition $tr = A \to b_j$ is denoted $s \cdot tr = s \backslash \{b_i\} \cup \{b_j\}, b_i \in s$. If there does not exist fireable transition, s remains unchanged. The state of a certain automaton a is noted $(s \cdot tr)[a]$.

The definition of dynamics allows one to describe how the system state interacts with the transitions. Moreover, to describe the evolution in an ABAN, we use the notion of trajectory.

Definition 3.3 (Trajectory)**.** Given an ABAN $\mathbb{A} = (\Sigma, T)$ and a global initial state $\alpha \in L$, a trajectory t from α is a sequence of transitions $t = tr_1 ::$

$\cdots :: tr_i :: \cdots :: tr_n$ with $tr_i \in T$ and each tr_i is fireable in $(\alpha \cdot tr_1 \cdot \ldots \cdot tr_{i-1})$. From α, the global state after firing all transitions of t is $(\alpha \cdot tr_1 \cdot \ldots \cdot tr_n)$, denoted $\alpha \cdot t$.

A trajectory describe the historical evolution of the system or one possible future evolution by recording the fired transitions. An alternative is to record the state changes using state sequence:

Definition 3.4 (State sequence)**.** Given an ABAN $\mathbb{A} = (\Sigma, T)$ and a global initial state $\alpha \in L$ and trajectory t, the state sequence $seq = s_1 :: \cdots :: s_i :: \cdots :: s_n$ with $s_i \in LS$ is formed by the updated local states during the trajectory t.

Thanks to asynchronicity, at each time step ABAN changes the value of at most one automaton. That is why we can distinguish the order of state changes, thus form a state sequence.

Example 3.1 illustrates all the definitions above.

Example 3.1. Figure 3.1 shows an ABAN of 5 automata a, b, c, d, e, with the set of transitions $T = \{\{b_1, c_1\} \rightarrow a_1, \{e_1\} \rightarrow a_1, \{d_0\} \rightarrow b_1, \{d_1\} \rightarrow c_1, \{b_1\} \rightarrow d_1\}$ and the initial state $\alpha = \langle a_0, b_0, c_0, d_0, e_0 \rangle$. A possible trajectory from α is $t = \{d_0\} \rightarrow b_1 :: \{b_1\} \rightarrow d_1 :: \{d_1\} \rightarrow c_1 :: \{b_1, c_1\} \rightarrow a_1$. After firing the transitions in trajectory t, the global state becomes $\Omega = s \cdot t = \langle a_1, b_1, c_1, d_1, e_0 \rangle$, and the local state of a is $(\alpha \cdot t)[a] = a_1$. The corresponding state sequence is $seq = b_1 :: d_1 :: c_1 :: a_1$.

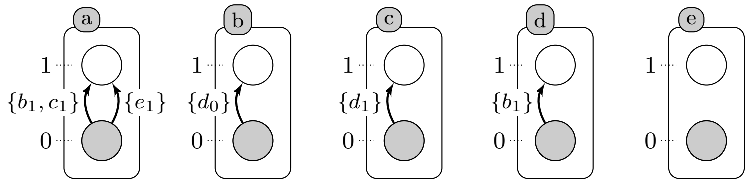


Figure 3.1: An example of ABAN

With the definition of trajectory and that of state sequence, we can address reachability problem.

Definition 3.5 (Reachability problem)**.** Given an ABAN, the *joint reachability* $REACH(\alpha, \Omega)$ can be formalized as: joint state Ω is reachable iff there exists a trajectory t s.t. $\alpha \cdot t = \Omega$. *Partial reachability* $reach(\alpha, \omega)$ is defined analogously: local state $\omega = a_i$ is reachable iff there exists a trajectory t s.t. $(\alpha \cdot t)[a] = a_i$. $REACH(\alpha, \Omega)$ and $reach(\alpha, \omega)$ take Boolean values **True**, **False** or **Inconclusive** if it cannot be decided.

Example 3.2. Taking the same ABAN as in Example 3.1, target global state $\Omega = \langle a_1, b_1, c_1, d_1, e_0 \rangle$ or target local state $\omega = a_1$ are reachable from the initial state α *via* trajectory t or state sequence s, *i.e.* $reach(\alpha, a_1) =$ **True** and $REACH(\alpha, \Omega) =$ **True**.

One can define various dynamical properties by using reachability, *e.g.* safety (there exists no trajectory from any initial state to an unwanted state), robustness (there exist trajectories from any initial state to a wanted state). Moreover, Proposition 3.1 explains the reachability of a joint state even a global state can be transformed to that of a local state.

Proposition 3.1 (Transformation of reachability)**.** Given an ABAN $\mathbb{A} = (\Sigma, T)$ and a joint reachability problem $REACH(\alpha, \Omega)$, there exists an ABAN $\mathbb{A}' = (\Sigma, T')$ with $\Sigma' = \Sigma \cup \{x\}$ and $T' = T \cup \{\Omega \to x_1\}$ s.t. the local reachability problem in $\mathbb{A}'$, $reach(\alpha', x_1)$ with $\alpha' = \alpha \cup \{x_0\}$ is equivalent to $REACH(\alpha, \Omega)$ in $\mathbb{A}$.

Proof. If $REACH(\alpha, \Omega) =$ **True**, there must exists a trajectory t satisfying $\alpha \cdot t = \Omega$. t is consistent with $\mathbb{A}'$ with initial state α' as $\mathbb{A}'$ contains all the elements in $\mathbb{A}$ and $\alpha \subset \alpha'$. After firing all the transitions in t, the global state becomes $\Omega \cup \{x_0\}$, transition $\Omega \to x_1$ is fireable and x_1 is reachable from α' *via* $t' = t :: \Omega \to x_1$, thus $reach(\alpha', x_1) =$ **True**.

If $REACH(\alpha, \Omega) =$ **False**, there does not exist a trajectory t satisfying $\alpha \cdot t = \Omega$. In $\mathbb{A}'$, this conclusion remains true as the only added transition $\Omega \to x_1$ is useless in the reachability of Ω. The only pathway towards x_1 is through $\Omega \to x_1$, as Ω is not reachable, x_1 is not reachable, $reach(\alpha', x_1) =$ **False**.

Similarly, we can prove the global reachability from local one. $\square$

One advantage of ABAN Many biological regulatory networks are encoded in Boolean style, *e.g.* in [3, 42], because BN is a simple formalism but with strong applicability: discretization in BN is a way to handle the imprecision of *a priori* knowledge on the model. However BN may be not expressive enough. If one wants to model the dynamic behavior "$a \leftarrow 1$ at moment $t + 1$ if $b = 1$ at moment t", the translation is $a(t + 1) = b(t)$ in BN. It means a always follows the evolution of b but with a redundant behavior "$a \leftarrow 0$ when $b = 0$ at moment t" which is not defined in his need. ABAN models this dynamics as $\{b_1\} \to a_1$ without this redundancy. Besides, BNs are transformable to ABANs, and this property makes our approach applicable to a wider domain (Appendix A.1 Translation between Models).

3.2.2 Simplified Local Causality Graph (SLCG)

Paulevé *et al.* [59] have invented Local Causality Graph (LCG) to analyze reachability problems statically. LCG abstracts the original problem through an over-approximation (necessary condition) and an under-approximation (sufficient condition). It is a very efficient tool as there is no global search and all the operations are bounded in polynomial complexity. However LCG does not guarantee to obtain a result, *i.e.* some inconclusive instances satisfy the necessary condition but fail sufficient conditions, thus one has no clue about the reachability of these instances.

In this thesis, we make use of the LCG by removing objective nodes needed only in multi-valued networks, naming it SLCG (Simplified LCG), then we try to analyze it more deeply to solve inconclusive cases of binary valued systems. In fact, Didier *et al.* [23] have shown a technique to transform multi-valued networks to Boolean networks, which broadened the applicability to multi-valued networks.

SLCG is aimed at studying the reachability of a target state while given an ABAN and a global initial state. SLCG is a goal-oriented method. It starts with the target state ω, looks for transitions reaching the target state, then replaces ω with the bodies of these transitions. If the current target state is included in the initial state, we find the causal path from the initial state towards ω, otherwise we continue the process, until we reach the initial state or local states we have already traversed. This process terminates because the local states are finite, of size $O(n)$, where n is the number of automata. In the worst case, SLCG contains all the local states of the ABAN.

Definition 3.6 (Over-approximate SLCG)**.** Given an ABAN $\mathbb{A} = (\Sigma, T)$, a global initial state α and a target local state ω, SLCG $l = (V_{\text{state}}, V_{\text{sol}}, E)$ is the smallest recursive structure with $E \subseteq (V_{\text{state}} \times V_{\text{sol}}) \cup (V_{\text{sol}} \times V_{\text{state}})$ which satisfies:

$$\begin{array}{rcl} \omega & \in & V_{\text{state}} \\ a_i \in V_{\text{state}} & \Leftrightarrow & \{(a_i, A \to a_i)\} \subseteq E \\ A \to a_i \in V_{\text{sol}} & \Leftrightarrow & \{(A \to a_i, X) \mid X = \varnothing \text{ if } a_i \in \alpha, \\ & & \text{else } \forall b_j \in A,\ X = b_j\} \subseteq E \end{array}$$

where $V_{\text{state}} \subseteq LS$ is a set of local states, $V_{\text{sol}} \subseteq T$ is the set of solutions and X is one of the required local states of $A \to a_i$.

It is worth noticing that every state node in V_{state} forms an **OR gate** as a local state a is reachable if there exists one fireable transition with body

a. Similarly, every solution node in V_{sol} forms an **AND gate** as a transition is fireable only if all the local states in its head are reachable.

Remark 3.1. Original LCGs consist of three kinds of nodes: **state nodes** corresponding to the local states of automata (V_{state}), **objective nodes** corresponding to the state transition paths within one automata ($V_{\text{objective}}$), **solution nodes** corresponding to the transitions to be used for each state transition path (V_{sol}). However, under the circumstance of ABAN, objective nodes are no longer needed. Because for one state a_i to be reached, the only possible path is $a_{1-i} \rightarrow a_i$ ($0 \rightarrow 1$ or $1 \rightarrow 0$). Unlike multi-valued case, for example, if one wants to reach a_1 from a_0 (suppose possible states for a are 0,1,2), there are in fact infinite possible paths: $0 \rightarrow 1$, $0 \rightarrow 2 \rightarrow 1$, $0 \rightarrow 2 \rightarrow 0 \rightarrow 1$, ... This simplification in fact reduces the searching space and reinforces the conclusiveness.

Example 3.3. Figure 3.2 shows the SLCG for analyzing $reach(a_1)$ in Example 3.1. There are two pathways from a_1 to α: $a_1 \rightarrow \circ \rightarrow b_1 \rightarrow \circ \rightarrow d_0$ and $a_1 \rightarrow \circ \rightarrow c_1 \rightarrow \circ \rightarrow d_1 \rightarrow \circ \rightarrow b_1 \rightarrow \circ \rightarrow d_0$.

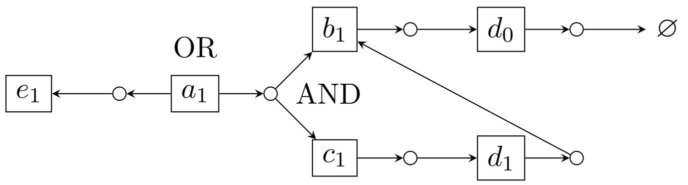


Figure 3.2: Visualization of SLCG, with the squares representing local states and small circles representing solution nodes. $\varnothing$ signifies that there is no need to link any transitions, *i.e.* the former state d_0 is in the initial state.

Algorithm 4 in Appendix describes how to construct an SLCG from an ABAN $\mathbb{A} = (\Sigma, T)$. Starting from a given target local state ω, one can find all the transitions $T_s \subseteq T$ reaching ω and add edges $\omega \rightarrow T_s$. Then we find all the heads A of T_s and add edges $T_s \rightarrow A$ and replace Ls with A (recursion). Finally, we update the structure until $Ls \subseteq \alpha$ or there is no transition with body in Ls.

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

With SLCG, it is easy to verify whether their are potential pathways from the target state ω to the initial state α as the causal relations with form *state* $\to$ *transition* and *transion* $\to$ *state* are indicated on the graph. If there does not exist such a pathway, one can ensure that ω is not reachable from α.

Pseudo-reachability is a procedure computing the existence of such pathway by confirming whether the transitions are enough (in causal sense) for ω to be reachable without considering the system evolution. However, pseudo-reachability is named "pseudo" because it is only an over-approximation of the reachability, *i.e.* it verifies a necessary condition of the reachability.

Definition 3.7 (Pseudo-reachability)**.** Given an SLCG $l = (V_{\text{state}}, V_{\text{sol}}, E)$ with global initial state α, the pseudo-reachability of node $v \in V_{\text{state}}$ is defined as

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

where $\text{fireable}(sol) = \bigwedge_{(sol,s)\in E} reach'(\alpha, s)$.

3.2.3 Conclusiveness

When pseudo-reachability suggests **True**, due to its non-equivalence with reachability, we cannot assure the pathway from ω to α is *dynamically realizable.* We are going to show several counter-examples as follows:

Example 3.4. In Figure 3.3, $reach'(\alpha, c_1) = reach'(\alpha, a_1) \wedge reach'(\alpha, b_1) = reach'(\alpha, a_0) \wedge reach'(\alpha, b_0) =$ **True**. Both a_1 and b_1 are reachable, but they can not be reached simultaneously. In such SLCG, there are two branches, $a_1 \to b_0$ and $b_1 \to a_0$, the automata a and b involve themselves in different branches, the reachability of a_1 impedes the reachability of b_1 and *vice versa.*

Also, the recursive reasoning does not terminate if there exists cycles in SLCG. While computing the pseudo-reachability, self-dependent structure $reach'(\alpha, a_i) = \ldots = reach'(\alpha, a_i)$ might appear and cannot be computed by using Definition 3.7. In Figure 3.4, dealing with cycles becomes inevitable.

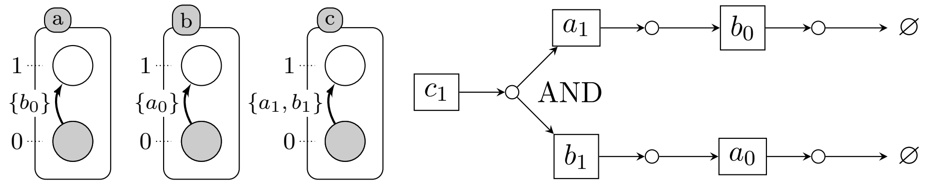


Figure 3.3: $\Sigma = \{a, b, c\}$, $T = \{\{b_0\} \to a_1,\ \{a_0\} \to b_1,\ \{a_1, b_1\} \to c_1\}, \omega = c_1$

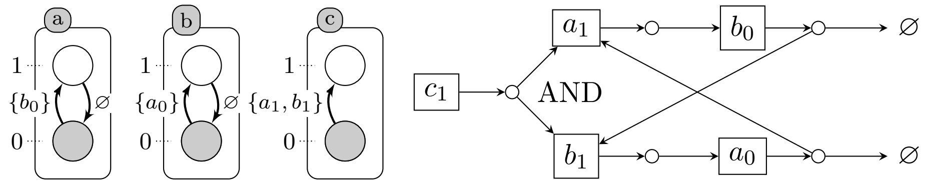


Figure 3.4: SLCG with cycles, $\Sigma = \{a, b, c\}$, $T = \{\{b_0\} \to a_1,\ \{a_0\} \to b_1,\ \{a_1, b_1\} \to c_1, \varnothing \to a_0, \varnothing \to b_0\}, \omega = c_1$

3.3 Topological Preprocessing

We have shown some example of inconclusiveness due to cycles and **AND gates**. In this section, we are going to analyze these two special structures and offer two solutions to the inconclusiveness.

3.3.1 Detection and Removal of Cycles

Definition 3.8 (Cycle)**.** In an SLCG, a cycle is formed by a sequence of nodes linked as follows: $a_i \to \circ \to \cdots \to \circ \to a_i$, where circles stand for solution nodes.

To identify cycles, we search Strongly Connected Components (SCC) of size greater than one instead of cycles, because there may be common nodes and edges shared by multiple cycles. When removing cycles, modifying such nodes or edges effects multiple cycles causes excessive task. In other words, a SCC contains as many as possible nested cycles which strongly connect to each other. Removing all the SCCs guarantees the nonexistence of cycles. [77] shows that the detection of SCCs can be done in $O(|V|+|E|)$ time, with $|V|$ the number of the vertices and $|E|$ the number of the edges. SLCG is usually a sparse graph, as in biological systems, the automata mostly interact with only a part of the system, hence the out-degree can be considered of

$O(1)$ and the detection of SCCs[1] can be done in $O(|V|)$, *i.e.* linear time.

Theorem 3.1. Given a cycle $x \rightarrow \circ \rightarrow \cdots \rightarrow \circ \rightarrow x$ in an SLCG, if there is at most one incoming edge to the cycle, the cycle can be removed.

Proof. If there is no incoming edge, the target state y must be in the cycle. The edge $y.pred \rightarrow \circ \rightarrow y$ can be removed, because the reachability of $y.pred$ requires y, but y is the target state, which is never reached before the other local states in the SLCG are reached. Thus the transition corresponding to this edge is never fired and the edge can be removed. Similarly, if there is an outside incoming edge $a \rightarrow \circ \rightarrow x$, a must be the successor of target state y or the target itself, $x.pred \rightarrow \circ \rightarrow x$ can hence be removed. $\square$

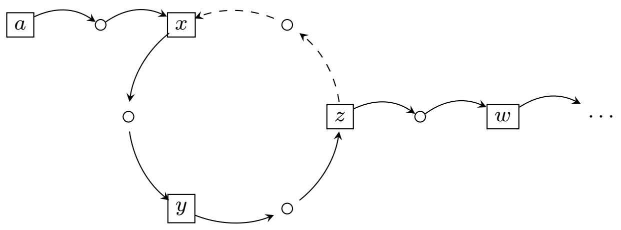


Figure 3.5: SLCG l containing cycle $x \rightarrow \circ \rightarrow y \rightarrow \circ \rightarrow z \rightarrow \circ \rightarrow x$

Example 3.5. In Figure 3.5, the pseudo-reachability of a in SLCG l is

$$\begin{aligned} reach'(\alpha, a) &= reach'(\alpha, x) = reach'(\alpha, y) = reach'(\alpha, z) \\ &= reach'(\alpha, x) \vee reach'(\alpha, w) \end{aligned}$$

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

Unfortunately, not all cycles are removable *via* Theorem 3.1.

When there are cycles that cannot be deleted according to Theorem 3.1, we can apply Theorem 3.2. It is associated with the decomposition of SLCG in the next section. The decomposition of SLCG replaces all the **OR gates** with one of their branches, then the cycles are either broken, either has no outgoing edge which leads to unreachability. Example 3.6 explains the issue.

Theorem 3.2. Given a cycle, if it contains no edge towards outside of the cycle, all the local states in the cycle are unreachable.

[1]Implementation in Python3 by Mario Alviano at `https://github.com/alviano/python/blob/master/rewrite_aggregates/scc.py`

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

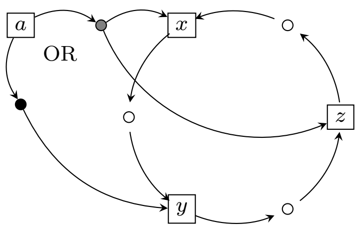


Figure 3.6: x, y, z all have external links, thus none of the links can be discarded.

Example 3.6. In Figure 3.6, cycle $C = x \to \circ \to y \to \circ \to z \to \circ \to x$ possesses 3 incoming edges, which is unbreakable according to Theorem 3.1. But with Theorem 3.2, if **OR gates** are removed, the cycle can be dealt with. At node a, there is an **OR gate** with two branches (filled circles). No matter which is branch chosen in the decomposition phase, there is no edge towards cycle C. x, y, z are all unreachable, hence a is unreachable.

3.3.2 Decomposition of SLCG

For every **OR gate**, it has multiple successor transitions (solution nodes) for reaching its corresponding local state. Fixing the transition choice of all the **OR gates** is called an *assignment*. If one wants to discover all the solutions, he needs to traverse all the assignments which are exponential. To avoid combinatorial explosion, we use a simple heuristic: choose randomly one assignment for each trial. Then, we can construct a new SLCG without **OR gate**, every state node has exactly one successor solution node, see Figure 3.7.

3.4 Reachability Analysis

After the preprocessings introduced in the previous section, we can get rid of cycles and **OR gates**. The next step is to analyze an SLCG with only **AND**

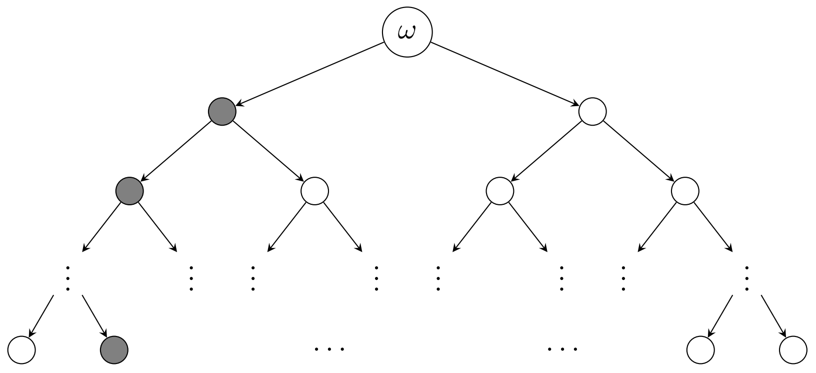


Figure 3.7: Random choice on **OR gates**. Descending from the target state, when we encounter an **OR gate**, we choose randomly one of its branches. Circles filled gray stand for one possible assignment.

gates. We need to find a trajectory reaching all the components of the **AND gates** simultaneously. Usually one cannot achieve good conclusiveness and low complexity at the same time, that is why we propose two solutions for different needs of conclusiveness-complexity balance: Reachability *via* search in permutations (PermReach) which is a partial search and Reachability *via* Answer Set Programming (ASPReach) which is a exhaustive search.

3.4.1 Reachability *via* Permutations (PermReach)

Before running into the definitions, we compare Example 3.4 and Example 3.7 with minor difference.

Example 3.7. Figure 3.8 shows the SLCG for the reachability of c_1 in ABAN with transitions $\mathbf{T} = \{\{a_1, b_1\} \to c_1, \{b_0\} \to a_1, \{c_0\} \to b_1\}$. The only difference with Example 3.4 is transition $\{\mathbf{c_0}\} \to b_1$. a_1 and b_1 are reachable respectively but is not necessarily for the joint state $s = \{a_1, b_1\}$. If we begin with the branch with a_1, s is reachable with trajectory $\{b_0\} \to a_1 :: \{c_0\} \to b_1 :: \{a_1, b_1\} \to c_1$. However, if we begin with the branch b_1, after firing $\{c_0\} \to b_1$, b_0 is no longer reachable, resulting the unreachability of a_1.

The head of a transition forms a joint state. If such joint state is reachable, the corresponding transition can be fired. However in the Example 3.7,

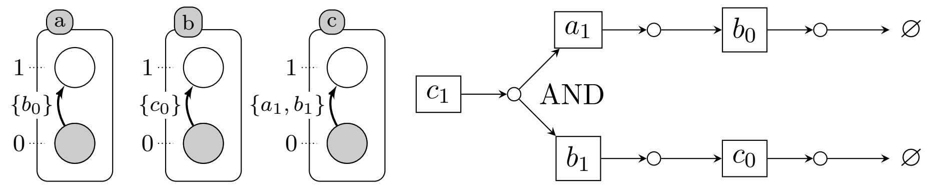


Figure 3.8: The ABAN and the SLCG of *Example 3.7*, $\alpha = \langle a_0, b_0, c_0 \rangle$. The only difference with *Example 3.4* on page 33 is the transition $\{\mathbf{c_0}\} \to b_1$.

c_1 can be reached only in *certain orders*. These orders cannot be retrieved by SLCG, as SLCG works statically. The following contents are contributed to retrieve an admissible order to reach a joint state.

The reachability of a joint state can be formulated as sequential reachability:

Definition 3.9 (Sequential reachability)**.** Let $s = \{ls_1, \dots, ls_n\}$ be a joint state, $p_1, \dots, p_n$ be a permutation of $1, \dots, n$ and $seq = ls_{p_1} :: \dots :: ls_{p_n}$ be a sequence. The sequential reachability of *seq* is defined as $REACH(\alpha, seq) = reach(s_1, ls_{p_1}) :: \dots :: reach(s_n, ls_{p_n})$, where $s_1 = \alpha$ and for $i > 1$, $s_i = s_{i-1} \backslash \{\neg ls_{p_{i-1}}\} \cup \{ls_{p_{i-1}}\}$. For a Boolean local state a_j, we abuse the notation by $\neg a_j = a_{1-j}$. $REACH(\alpha, seq) =$ **True** if from an initial state α, s is reached by the ordered state changes in *seq*, otherwise $REACH(\alpha, seq) =$ **False**.

In Example 3.7, $REACH(\alpha, a_1 :: b_1 :: c_1) =$ **True** and $REACH(\alpha, b_1 :: a_1 :: c_1) =$ **False**.

As the firing order matters, we come to verify all the possible sequential reachabilities of certain joint state to verify its reachability.

Proposition 3.2. Given joint state $s = \{ls_1, \dots, ls_n\}$, with all the local states in s are reachable: $reach(\alpha, ls_i) =$ **True**, $\forall i \in [1; n]$. The set of all the permutations of s is denoted $Perm(s) = \{(ls_1 :: ls_2, :: \dots :: ls_n), \cdots, (ls_n :: ls_{n-1} :: \dots, :: ls_1)\}$. $\bigvee_{j \in Perm(s)} REACH(\alpha, j) =$ **True** is a sufficient condition of $REACH(\alpha, s) =$ **True**.

Proof. If $\exists perm_i \in Perm(s)$ s.t. $REACH(\alpha, perm_i) =$ **True**, s can be reached according to the order in $perm_i$. To reach s, every local state in the SLCG of s is mandatory to be reached. Because the definition of SLCG suggests it is the smallest structure which contains all the needed local states and transitions for the target state. As long as there is no **OR gates**, all the transitions in the SLCG must be fired to reach the target state. As

Perm(s) covers all the possible orders, if there are any admissible ones, they are verified. □

In case where the successors of certain **AND gate** contain other **AND gates**, we cannot directly obtain its reachability because the reachability of the successor **AND gates** are unknown. We analyze first the simple **AND gates** *simp*, *i.e.* the successors of *simp* do not contain any **AND gates**. If all local states within *simp* are reachable *via* the search of permutations, we can update the initial state by firing all the transitions and also update the SLCG by deleting the successors of *simp*. Then, we restart this process from new simple **AND gates** until we reach finally the target state. Detailed algorithm is in Appendix Algorithm 7.

However the method of PermReach is not complete. If there are constraints in different branches, traversing all the permutations may be not sufficient to find admissible trajectories towards the target state as in the Example 3.8.

Example 3.8. In Figure 3.9, among the simple **AND gates**, if sol_{c_1} is solved first, automaton d will be at the state d_1, which disables the reachability of b_1. The trajectory towards a_1 may not be retrievable by PermReach even if a_1 is reachable.

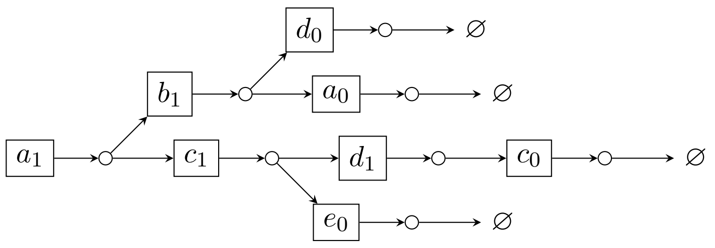


Figure 3.9: A counterexample of PermReach with $\alpha = \langle a_0, b_0, c_0, d_0, e_0\rangle$. $reach(\alpha, a_1)$=**True** but **Inconclusive** is given by PermReach.

We state here several algorithmic properties of PermReach.

Theorem 3.3 (Termination and correctness of PermReach)**.** Let $l = (V_{\text{state}}, V_{\text{sol}}, E)$ be an SLCG with initial state α and target local state ω and $k > 0$ be an integer.

- The call $PermReach(l, k)$ terminates.
- $PermReach(l, k) = (\textbf{False}, \varnothing)$ if $\nexists t$ a trajectory in l from α to ω.

The proof is given in Theorem C.3 in Appendix.

Theorem 3.4 (Complexity of PermReach)**.** Let $l = (V_{\text{state}}, V_{\text{sol}}, E)$ be an SLCG with initial state α and $k > 0$ be an integer. Let $s = |V_{\text{sol}}|$ be the number of target state of l. Let $v = |V_{\text{state}}|$ be the number of vertices of l. Let $e = |E|$ be the number of edges of l. The complexity of $PermReach(l, k)$ is $O(v + s + e + (v + s)/2 \times v \times e \times s + v^2 \times e + v \times e + k \times (v \times e^2 + \frac{v}{2}!))$ which is bounded by $O(k \times \frac{v}{2}!)$.

The proof is given in Theorem C.4 in Appendix.

3.4.2 Reachability *via* ASP (ASPReach)

As we have seen in the last section, PermReach is more conclusive than pure static analysis, but still fails in the cases where there are nested **AND gates** and the order of these **AND gates** influence the reachability.

To deal with the inconclusiveness left by PermReach, we use an ASP-based method (Answer Set Programming) [5] instead of the search in permutation to analyze the preprocessed SLCG with only **AND gates**.

Also, Ben Abdallah *et al.* [1] have shown that pure ASP approach is costly (See Section D.1 in Appendix) which suggests a hybrid solution.

ASP is a Prolog-like declarative programming paradigm. It uses the description and the constraints of the problem (called rule) instead of imperative orders. ASP solvers tackle problems by generating all the possibilities respecting the constraints. This search in all the possibilities allows us to filter automatically all the admissible orders no matter the order-sensitive cases exist in **AND gates** or inside **AND gates**.

Introduction to ASP

We use Clingo [31] which is a combination of grounder Gringo and solver Clasp. Given an input program with first-order variables, a grounder computes an equivalent ground (variable-free) program for an ASP program, while a solver selects admissible solutions (answer sets) in the ground.

A rule is in the following form:

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

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

ASP Programs can yield no answer set, one answer set, or multiple answer sets. For example, the following program produces two answer sets: $\{b\}$ and $\{c\}$.

```
b:- not c.
c:- not b.
```

Indeed, the absence of c makes b true, and conversely absence of b makes c true. Cardinality constraints are another way to obtain multiple answer sets. The most usual way of using a cardinality is in place of *head*:

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

Or corresponding ASP code with $k = 2$, $m = 3$ and $n = 2$:

```
l{q1, q2}u :- a1, a2, a3,..., not a4, not a5.
```

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

Encoding of SLCG

Suppose all the **OR gates** are deleted *via* preprocessing, we begin encoding the reachability problem in ASP. As SLCGs already contain all the local states and the transitions to be used, there is no need to describe the elements in ABAN. Here is how we encode the SLCGs: they are regarded as *fact* they were created (fixed) already without any variables.

Predicate `init(a,i)` shows the automaton a is at initial state i. Predicate `node(a,i,n)` shows the node a_i in the SLCG is numbered n, while `parent(n1,n2)` expresses the node numbered n_1 is the predecessor of the node numbered n_2. The SLCG in Figure 3.3 on page 34 is encoded as follows:

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

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

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

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

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

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

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

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

Example 3.9. Let us simulate the analysis of the SLCG in Figure 3.3 using ASP. Rule 1 gives $b_0 \rhd a_1$, $a_1 \rhd c_1$, $a_0 \rhd b_1$, $b_1 \rhd c_1$; Rule 2 gives $a_0 \rhd c_1$ and $b_0 \rhd c_1$; Rule 3 gives $a_1 \rhd b_1$ and $b_1 \rhd a_1$ which is impossible, therefore there does not exist a state sequence to reach c_1 from initial state. c_1 is unreachable.

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

Reachability Analyzer ASPReach

In this section, we integrate the ASP code into our analyzer **ASPReach**: an algorithm for checking the reachability of a target local state ω from a global initial state α (which can also be partial) in a given ABAN. However, exhaustive search leads to heavy computation and huge need of memory (tests in Chapter 5 shows pure ASP method is time and memory-consuming).

The algorithm proposed below tries to overcome those shortcomings by combining static analysis and stochastic search into the following hybrid approach. First, we try to use only SLCG to solve the reachability problem, SLCG illustrates the causality between necessary transitions to be fired to reach the target state. If sole SLCG is not sufficient, we simplify the SLCG using the preprocessings introduced in Section 3.3.1 and Section 3.3.2. The tentative of removing cycles simplifies the SLCG and keep the reachability unchanged. pseudo-reachability allows one to filter some unreachable cases based on the topology of SLCG. After that, the heuristic part is the core of our algorithm. Stochastic choices avoid combinatorial explosion on different **OR gates**. The ASP part searches thoroughly the result but does not traverse the whole state space (ASP solver starts from constraints, finds one consistent order and terminate the search).

ASPReach:

- Input: An ABAN $\mathbb{A}$, an initial state α, a target state ω and a max number of iterations k

- Output: $reach(\omega) \in \{\mathbf{False}, \mathbf{True}, \mathbf{Inconclusive}\}$

1. Construct the SLCG $l = SLCG(\mathbb{A}, \alpha, \omega)$
2. Try to remove all cycles and prune useless edges from l
3. Try to prove unreachability of ω in l using $reach'(l, \omega)$
   Return **False** if $reach'(l, \omega) = \mathbf{False}$
4. Try at most k times
   - $l' \leftarrow l$
   - Simplify each **OR gate** such that l' is an SLCG with only **AND gates**
   - If there remain cycles:
     - Back to step (iv)
   - Generate all trajectory that starts with α in l' using ASP
     - If a trajectory t ending with ω is found, return **True**
5. return **Inconclusive**

To be more precise, Algorithm 9 in Appendix provides the detailed pseudocode of the algorithm taking an SLCG l as input whose detailed construction is given in Algorithm 4. Lines 4-8 shows how to delete all the cycles with at most one incoming edge. After removing cycles, the SLCG may contain nodes without successor. Such nodes can be pruned since they do not lead to initial state (Line 9-18). This preprocessing reduces the search space of the stochastic search performed in step 4. Now l is pruned and might be cycle-free. Static analysis of l can then be used as heuristics to check pseudo-reachability (Definition 3.7) in order to detect some unreachability cases (Lines 19-21) which may conclude before searching. SLCG shows the dependencies between local states and transitions. A pathway in SLCG suggests a possible trajectory of reaching the target state. If $reach'(l, \omega) = \mathbf{False}$, we can ensure that ω is unreachable, as pseudo-reachability checks a necessary condition of reachability. If $reach'(l, \omega) = \mathbf{True}$, static analysis is not sufficient for reachability analysis.

When static analysis fails, a stochastic search is performed at most k times (line 22-30) to find a state sequence from the initial state α to target state ω. If there remain cycles with multiple incoming edges, according to Theorem 3.2, ω is unreachable.

The value of k will be discussed later in Chapter 5. Keep in mind that every state node is an **OR gate**, we have to choose one of its successor solution nodes to access the state. Random choices are made to fix a value for each **OR gate** of the SLCG allowing to perform a reachability check by generating all possible variable assignment order using ASP.

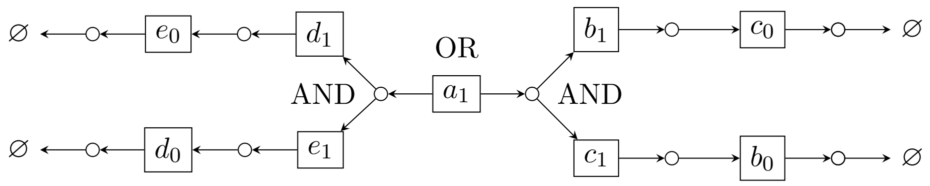


Figure 3.10: If an SLCG contains such structure, the result could be inconclusive. However the inconclusiveness requires a_1 does not possess other reachable branches.

Even though ASPReach is able to solve the inconclusive cases left by PermReach, still, ASPReach is not complete.

Example 3.10. A counter-example is shown in Figure 3.10, ABAN with transitions $T = \{\{b_1, c_1\} \to a_1, \{d_1, e_1\} \to a_1, \{e_0\} \to d_1, \{d_0\} \to e_1, \{c_0\} \to b_1, \{b_0\} \to c_1\}$ and initial state $\langle a_0, b_0, c_0, d_0, e_0 \rangle$. When there are multiple branches of one **OR gate** leading to unreachability, the result can be inconclusive. Because in the assignment phase, no matter which branch we choose at a_1 (b_1, c_1 or d_1, e_1), we cannot find an admissible order as both side are exactly the case of Example 3.4. At the end, the program will return **Inconclusive** due to the limit of iteration k is reached.

There is a tricky way to deal with this issue when |**OR gates**| is not big: we set a limit n, if |**OR gates**| $< n$, we shift the heuristics on the assignment of **OR gates** to the enumeration of all possible assignments. This "hack" can deal with the inconclusive cases with small size, including the one in Example 3.10. In the benchmarks in Chapter 5, inconclusive instances appear neither in biological examples nor in random generated tests.

At last, we are going to show some algorithmic properties of ASPReach which are important for a model checker. A model checker must terminate for any input like the standard for any algorithm; also, its complexity is crucial for its wider applicability and scalability.

Theorem 3.5 (Termination and correctness of ASPReach). Let $l = (V_{\text{state}}, V_{\text{sol}}, E)$ be an SLCG with initial state α and target local state ω and $k > 0$ be an integer.

- The call *ASPReach*(l, k) terminates.
- *ASPReach*$(l, k) = (\textbf{False}, \varnothing)$ if $\nexists t$ a trajectory in l from α to ω.
- *ASPReach*$(l, k) = (\textbf{True}, t)$ only if $\exists t$ a trajectory in l from α to ω.

The proof is given in Theorem C.5 in Appendix.

Theorem 3.6 (Complexity of ASPReach)**.** Let $l = (V_{\text{state}}, V_{\text{sol}}, E)$ be an SLCG with initial state α and $k > 0$ be an integer. Let $s = |V_{\text{sol}}|$ be the number of target state of l. Let $v = |V_{\text{state}}|$ be the number of vertices of l. Let $e = |E|$ be the number of edges of l. Assume that ASP solver is equivalent to a pure enumerator. The complexity of *ASPReach*(l, k) is $O(v+s+e+(v+s)/2 \times v \times e \times s + v^2 \times e + v \times e + k \times (v \times e^2 + \frac{v}{2}!))$ which is bounded by $O(k \times \frac{v}{2}!)$.

The proof is given in Theorem C.6 in Appendix.

ASPReach has an factorial complexity due the exhaustive search of admissible order by ASP solver. However ASP solver is a black-box system, random tests in Chapter 5 shows even when the number of automata increases to 1000, the runtime is still acceptable (several seconds).

3.5 Extension to Multi-valued Models

The use of either pseudo-reachability, PermReach or ASPReach can be extended to multi-valued models for wider applications.

To avoid the issues of inconclusiveness due to the notion of objective in PH or AAN (Remark 3.1 on page 32), we restrict the semantics of AAN (Section 2.1.6 on page 15). We allow only state changes of one step size ($|j - k| = 1$ in the Definition of restricted AAN below) and presume that there is no need to leave and return to certain local state in order to achieve certain reachability. These modifications assure that there is only one path going from the target local state and its corresponding initial state, *e.g.* from state 1 to 3, the only path is $1 \rightarrow 2 \rightarrow 3$.

Definition 3.10 (Restricted AAN)**.** A restricted Asynchronous Automata Network is a tuple $\mathbb{A} = (\Sigma, T)$, where:

- $\Sigma = \{a, b, \ldots\}$ is the finite set of automata with every automaton having a discrete state. The max discrete level of a is denoted l_a and this level is 1 by default (Boolean) and omitted in the notation;

- The states of $\mathbb{A}$ can then be defined: $LS = \bigcup\limits_{a \in \Sigma} \{a_0, \ldots, a_{l_a}\}$ is the set of all *local states*, $L = \bigtimes\limits_{a \in \Sigma'} \{a_0, \ldots, a_{l_a}\}$ is the set of *joint states* where $\Sigma' \subseteq \Sigma$. Particularly, if $\Sigma' = \Sigma$, L is the set of *global states*.

- $T = \{A \rightarrow b_k \mid A \in L,\ b \in \Sigma,\ k \in [1; l_b] \wedge \text{if } l_b > 1,\ \exists b_j \in A,\ j \in [1; l_b],\ |j - k| = 1,\ \text{else } \nexists b_j \in A\}$ is the finite set of transitions, which defines the regulations and dynamics of the restricted AAN: A, b_k are denoted $head(h)$, $body(h)$ respectively of the transition $tr = A \rightarrow b_k$.

By putting the affected state into the condition of transition, we can distinguish on which state the transition functions for multi-valued automata. Affected states are regarded as equivalent conditions as other states in the SLCG for restricted AAN. The only difference is that when encountered a multi-valued automaton, SLCG seeks to find only the transitions in the direction from the wanted state towards the initial state. This preference avoids the appearance of cycles due to returning to a given local state.

Definition 3.11 (Extended over-approximate SLCG)**.** Given an ABAN $\mathbb{A} = (\Sigma, T)$, a global initial state α and a target local state ω, SLCG $l = (V_{\text{state}}, V_{\text{sol}}, E)$ is the smallest recursive structure with $E \subseteq (V_{\text{state}} \times V_{\text{sol}}) \cup (V_{\text{sol}} \times V_{\text{state}})$ which satisfies:

$$\begin{aligned}
\omega \in V_{\text{state}} \quad & \& \quad \exists a_j \in \alpha \\
a_i \in V_{\text{state}} \quad & \Leftrightarrow \quad \{(a_i, A \rightarrow a_i) \mid \text{if } l_a > 1,\ a_k \in A, \\
& \qquad \text{where } k = i + 1 \times sgn(i - j)\} \subseteq E \\
A \rightarrow a_i \in V_{\text{sol}} \quad & \Leftrightarrow \quad \{(A \rightarrow a_i, X) \mid X = \varnothing \text{ if } i = j, \\
& \qquad \text{else } \forall b_m \in A,\ X = b_m\} \subseteq E
\end{aligned}$$

where $V_{\text{state}} \subseteq LS$ is a set of local states, $V_{\text{sol}} \subseteq T$ is the set of solutions and X is one of the required local states of $A \rightarrow a_i$.

Example 3.11 shows that SLCG succeeds to find a trajectory jumping multiple qualitative levels (a_0 to a_2).

Example 3.11. The left of Figure 3.11 shows a restricted AAN $\mathbb{A} = (\Sigma, T)$ where $\Sigma = \{a, b, c\}$ with $l_a = 2, l_c = 2$ and $T = \{\{a_1, b_1\} \rightarrow a_2, \{a_0, c_1\} \rightarrow a_1, \{c_2\} \rightarrow b_1, \{c_1, b_0\} \rightarrow c_2\}$. The initial state is $\langle a_0, b_0, c_1 \rangle$. An extended SLCG for computing the reachability of a_2 is on the right.

PermReach or ASPReach can be applied to analyze the real reachability of a_2 (reachable *via* $a_0 \rightarrow a_1 \rightarrow a_2$).

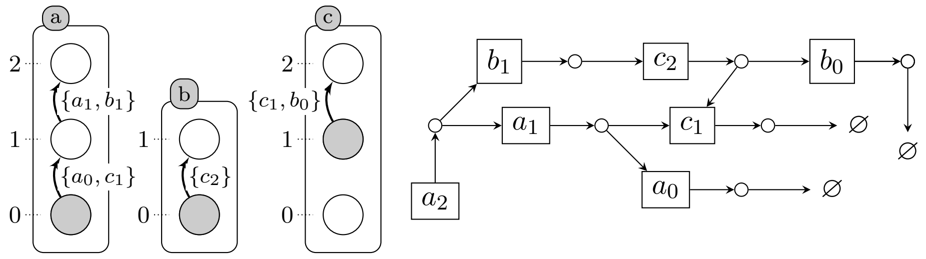


Figure 3.11: Restricted AAN and extended SLCG

However, this extension is limited enough because we do not consider the possibility to leave and return to certain local state. In multi-valued situation, certain local states with different direction to the target state need to be reached to "unlock" the condition of certain transitions. Example 3.12 shows a counterexample suggesting under certain conditions, it is necessary to leave and return to some local state (a_2 is reached to "unlock" b_1) and SLCG does not cover all the needed transitions.

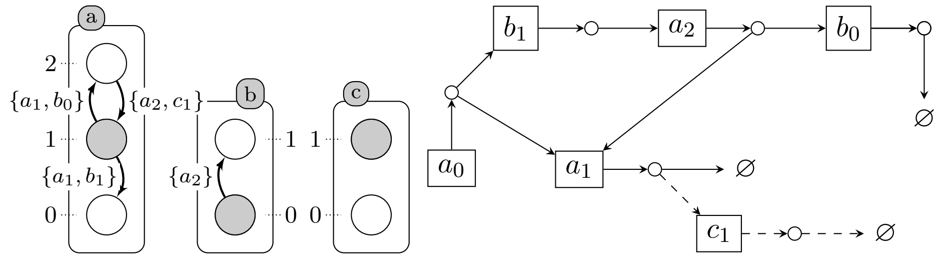


Figure 3.12: Restricted AAN and extended SLCG, dashed arrows represent the transition actually used but it does not appear in the extended SLCG.

Example 3.12. The left of Figure 3.12 shows a restricted AAN $\mathbb{A} = (\Sigma, T)$ where $\Sigma = \{a, b, c\}$ with $l_a = 2$ and $T = \{\{a_1, b_1\} \to a_0, \{a_1, b_0\} \to a_2, \{a_2, c_1\} \to a_1, \{a_2\} \to b_1\}$. The initial state is $\langle a_1, b_0, c_1 \rangle$. An extended SLCG for computing the reachability of a_0 is on the right. As a_2 appears in the extended SLCG, it needs to be reached. However a_1 is no longer reachable with the only help of the transitions in the extended SLCG. a_0 is in fact reachable *via* $\{a_1, b_0\} \to a_2 :: \{a_2\} \to b_1 :: \{a_2, c_1\} \to a_1 :: \{a_1, b_1\} \to a_0$ and the state change of a is $a_1 \to a_2 \to a_1 \to a_0$.

Even though this extension is not a complete method, it remains to be a heuristics to discover certain trajectories if there exist such.

3.6 Résumé

In this chapter, we formally (re)defined the modeling framework used for static analysis: Asynchronous Binary Automata Network and some terminology. Then we dug into the details of static analysis, figured out why they are not conclusive under certain conditions. To get rid of these constraints, we carry first preprocessing (Section 3.3) to detect and try to delete cycles. With the preprocessed ABAN, we introduced two reachability analyzers based on SLCG: PermReach and ASPReach.

PermReach relies on a complete search on the permutations of **AND gates**. However, permutations do not cover all the possible trajectories but PermReach is very efficient.

ASPReach does a finer work than PermReach, searching all the possible trajectories of a preprocessed ABAN (without cycles and **OR gates**).

An extension of the above reachability analysis to multi-valued models (restricted AAN and extended SLCG) allows one to partly discover the solutions but it appears to be also an interesting heuristics.

The experimental results are shown in Chapter 5 Tests and Benchmarks. From the results of multiple tests, these two approaches can deal with more problems than pure static analysis.

Like static analysis, those two analyzers are not fully conclusive. However, if one wants total conclusiveness, he will probably need a complete search over state space like exact model checkers do, resulting in state space explosion.

In the next chapter, we will introduce three model inference approaches based on these reachability analyzers.

Chapter 4

Model Inference and Revision

In the previous chapter, we introduced several approaches of refined reachability analysis, which are more suitable for practical use (more efficient and more conclusive). Nevertheless, these approaches can never take effect no matter how powerful they are, if the original model does not reflect the reality.

To model a biological computational system, one may consider two possible aspects: a first-step model built by biologists (variables, some confirmed transitions, some *a priori* properties *etc.*) and the observations from experiments (time-series data, steady states, oscillations, *etc.*).

This chapter is dedicated to the introduction of three different approaches of model inference/revision:

- *via* reachability properties and candidate regulations
- *via* partial correlation (statistics)
- *via* reachability properties and time-series data

There lies very little nuance between model inference and model revision: these two operations begin with some *a priori* information and/or observations, aiming at constructing a new model/modifying an existed model. As a result, they take the information apart from the existing transitions into account, using different tools to transform this information into possible transitions in the model (also can modify or deny existing transitions).

Following a 3-month research stay at Inoue Lab. at National Institute of Informatics, preliminary ideas behind Section 4.4 were presented in the "work in progress" track at ILP 2018 in Ferrara, Italy [13].

4.1 Background

Model inference/learning can be useful not only in biology [66] but also in other domains with the need of abstraction, *e.g.* robotics [53], multi-agent systems [28]. In the community of bioinformatics, DREAM challenge[1] is a well-known organization calling for the participation of the whole world on the newest bio-medical problems (called challenges). Among these challenges, some of them are in the domain of inference and prediction: DREAM2[2], DREAM4[3], DREAM8[4], DREAM11[5].

Like in biology, the modeling structures in robotics are usually complex and with big scale, which is difficult in control, prediction and simulation. Constructing an abstracted (often approximated) model is a compromising way of dealing with the mechanical and computational complexity. However, the abstracted models are theoretically non-equivalent to the original ones because they contain different amount of information. Even though there always remains a non-equivalence, we wish that the abstracted models are in bisimulation with the original systems with respect to certain variables and keep the same important properties as original systems.

As is mentioned in Section 1.3 of Chapter 1, as far as we know, there has been no work combining model inference and model revision. Some related works: Opgen-Rhein *et al.* [54] have studied the undirectional inference *via* correlation, Rodrigues *et al.* [69] have studied the learning of action models and Bonneau *et al.* [7] have studied the learning from continuous time-series data.

Figure 4.1 shows the methodology of this chapter. There are two parallel pathways:

- One starts with biological *a priori* knowledge. The knowledge comes

[1]`http://dreamchallenges.org/about-dream`
[2]`https://www.synapse.org/#!Synapse:syn2825374/wiki/71143`
[3]`https://www.synapse.org/#!Synapse:syn2825304/wiki/71129`
[4]`https://www.synapse.org/#!Synapse:syn1720047/wiki/55342`
[5]`https://www.synapse.org/#!Synapse:syn6131484/wiki/402026`

from biological literature, certain empirical conclusion can be translated to temporal properties (especially reachability).

- The other one starts with partial observation. It is said partial because the real system is not completely observable, only a part of parameters can be taken as I/O. Learning approaches will build a model consistent with partial observation but not necessarily consistent with the real system. Model checkers can verify the temporal properties and we can come up with some modifications to make the model consistent with *a priori* knowledge.

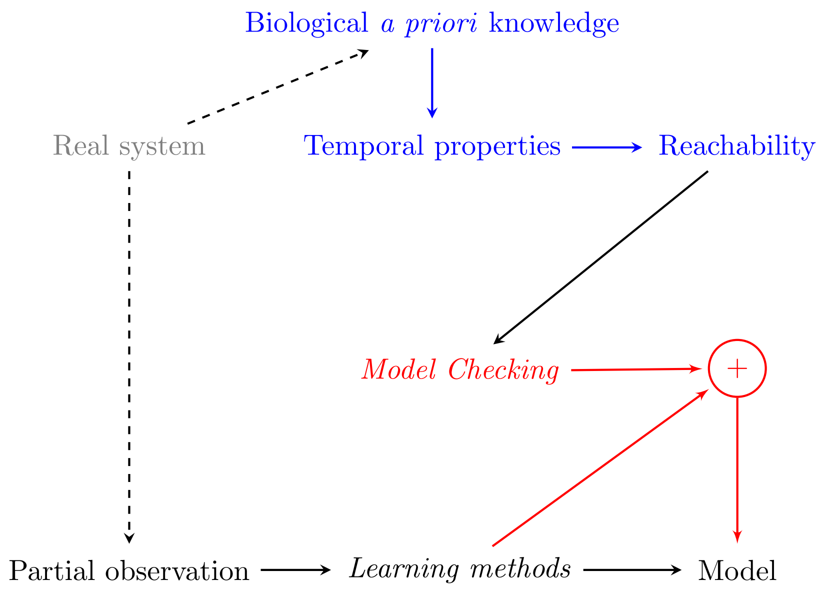


Figure 4.1: Big picture of model inference

We developed two learning approaches following the Big Picture in Figure 4.1.

- CRAC (Completion *via* Reachability And Correlations)

  CRAC is decomposed into two steps:

  1. Infer candidate regulations from continuous time-series data
  2. Select transitions consistent with candidate regulations and add them into/delete them from an existing incomplete model (can be empty) to make it satisfy *a priori* reachability information

- M2RIT (Model Revision *via* Reachability and Interpretation Transitions)

  M2RIT is also run with two steps:

  1. Learn a Normal Logic Program from *discretized* time-series data by asynchronous version of LFIT (synchronous version in Section 2.4.1)
  2. Modify the obtained NLP as little as possible to make it consistent with *a priori* reachability information while keeping the consistency with the constraints of time-series data, *i.e.* the NLP has to reproduce the time-series data

It is worth noticing that step 2 of CRAC could be an individual model inference algorithm if the set of candidate regulations/transitions is given. This step has the least complexity. Because the most naive method, *i.e.* brute force search is to verify every combination of the candidates if it satisfies the constraints. The models to be verified are of $O(2^n)$, where n is the number of candidate transitions.

One may ask if we can run CRAC or M2RIT without being provided with time-series data, *i.e.* run directly step 2 of either method. Deleting can be done without additional data, as the number of transitions already *assuring* certain reachability is very limited, which is shown in Section 2.4.2 or [57].

Nevertheless, adding transitions can be a more complex task. Suppose we begin with no constraints on the model, to obtain a transition $A \to a_i$ with fixed a_i, there are at most $m-1$ possibilities for A having one local state, C^2_{m-1} possibilities for A having 2 local states ... where m is the number of model variables. This factorial number of possibilities for **one** transition is not acceptable.

4.2 Model Completion *via* Candidate Regulations

We begin with the part which seems to have the smallest complexity. Brute force search that we just presented is still meaningless in real application. Roughly speaking, we prefer finding the transitions among the candidates which meet the unsatisfied constraints and keep already satisfied constraints unchanged.

4.2.1 Problem Description

Given an incomplete model, at first it does not satisfy wanted reachability properties. With given set of candidade regulations, we can consider the transitions inferred by these candidates are more likely to be the good ones than the randomly generated ones. Then we select the transitions which are prone to make certain unsatisfied reachability property satisfied and add them into the set of model transitions.

Briefly speaking, completion problem is:

incomplete model + candidate regulations + constraints $\rightarrow$ new model

The resulted model is expected to contain all the elements in the incomplete model and be consistent with all the *a priori* information.

Definition 4.1 (Model completion in ABAN semantics)**.** Given an incomplete ABAN $\mathbb{A} = (\Sigma, T)$ (where T can be empty), a complete model is an ABAN $\mathbb{A}' = (\Sigma, T')$ which satisfies $T' \supseteq T$, $\mathbb{A}'$ consistent with the set of reachability constraints C and the completion set $CS = T' \setminus T$ is consistent with the set of candidate regulations R, where

- $R = \{(body, head, sgn) \mid body \in \Sigma,\ head \in \Sigma,\ sgn \in \{-, +\}\}$
- $C = \{(\alpha, \omega, r) \mid \alpha \in L,\ \omega \in LS,\ r \in \{\mathbf{True}, \mathbf{False}\}\}$

R is the set of possible relations between variables, *body* may inhibit ($sgn = -$) or promote ($sgn = +$) *head*. C is the set of reachability properties to be satisfied, ω is (un)reachable from α. The definition of consistency is straightforward, but its meaning changes for reachability constraints and candidate regulations.

Definition 4.2 (Consistency)**.**

- An ABAN $\mathbb{A}$ is said consistent with the set of reachability constraints C iff $\forall(\alpha, \omega, r) \in C$ s.t. $reach(\alpha, \omega) = r$
- A completion set CS is said consistent with the set of candidate regulations R if $\forall\{a_i\} \rightarrow b_j \in CS, \exists(body, head, sgn) \in R$ s.t. $body = a,\ head = b$ and $sgn = +$ if $i = j$, $sgn = -$ if $i \neq j$.

However, completion operation is not able to remove or modify existing trajectories towards the states to be reached as the transitions used in the trajectories are still present. To obtain an ABAN meeting all the unreachable constraints in C, we have to apply cut sets.

Before our research began, Paulevé *et al.* [57] had worked on cut sets for reachability in large scale automata networks, which can be a reverse operation of completion sets. Cut set is used to inhibit certain local states to make certain local states unreachable.

By combining completion and cut set, one can construct/revise a model according to *a priori* information (candidate regulations) and constraints (reachability). We introduce first the notion and the application of cut sets.

4.2.2 Cut set

Cut set operation works on the local states mandatory for reachability. It is used to inhibit certain local states to make wanted local states unreachable. However we cannot eliminates the states in a given network, what we can manipulate here are the transitions. We present how cut set functions and then adapt it to our requirements.

We first obtain the rank of a digraph SLCG (detailed in Appendix C.1) which refers to the parental relations. Child nodes cannot have higher rank than their parent nodes. Nodes in the same SCC (Strongly Connected Components) have the same rank.

The computation of cut sets is recursive. We begin with the nodes of low rank, computing the valuation of each node, called update (defined below). The process stops when the valuation becomes saturated, *i.e.* the valuation of every node does not change after update. Valuation shows the global dependencies of reachability and saturated valuation is the cut set we want.

For different elements n in the SLCG,

- if n is a solution $A \to a_i \in V_{\text{sol}}$, it is sufficient to prevent the reachability of any local state in a_i to cut n; therefore, the cut sets results from the union of the cut sets of the successor of n (all local states).
- if n is a local state $a_i \in V_{\text{state}}$, it is sufficient to cut all its successors (all solutions) to prevent the reachability of a_i from α. In addition, if $a_i = \omega$, $\{a_i\}$ is added to the set of its cut sets.

Definition 4.3 (Valuation)**.** Given an SLCG $l = (V_{\text{state}}, V_{\text{sol}}, E)$, its valuation $\mathbb{V}$ is a mapping from each node of $V_{\text{state}} \cup V_{\text{sol}}$ to a set of sets of local states. $\mathbb{V}_0$ refers to the initial valuation s.t $\forall n \in V_{\text{state}} \cap V_{\text{sol}}, \mathbb{V}_0(n) = \varnothing$.

Definition 4.4 (update)**.**

$$\mathsf{update}(\mathbb{V}, n) = \begin{cases} \zeta^N(\bigcup_{m \in n.next} \mathbb{V}(m)) & \text{if } n \in \mathbf{Sol} \\ \zeta^N(\prod_{m \in n.next} \mathbb{V}(m)) & \text{if } n \in \mathbf{LS} \backslash \{\omega\} \\ \zeta^N(\{\{\omega\}\} \cup \prod_{m \in n.next} \mathbb{V}(m)) & \text{if } n = \omega \end{cases} \qquad (4.1)$$

where $\zeta^N(\{e_1, \ldots, e_n\}) = \{e_i \mid i \in [1;n] \wedge |e_i| \leq N \wedge \nexists j \in [1;n], j \neq i, e_j \subset e_i\}$, e_i being sets.

Algorithm 1 describes the whole process, where $rk(n)$ refers to the rank of n in the SLCG (See Definition C.1 in Appendix), characterizing the parent-child relations between nodes where a node with lower rank cannot be the parent of a node with higher rank.

We associate a first valuation with the nodes of low rank, check if it changes the valuation of its parent node. If yes, we have to add the unsaturated node back to the set to be updated ($\mathcal{M}$). The algorithm terminates when valuation $\mathbb{V}$ becomes saturated, *i.e.* the valuation becomes the cut set.

Algorithm 1 Cut set

```
Input: an SLCG $l = (V_{\text{state}}, V_{\text{sol}}, E)$
Output: cut set $\mathbb{V}$
Initialization: $\mathcal{M} \leftarrow V_{\text{state}} \cup V_{\text{sol}}$, $\mathbb{V} \leftarrow \mathbb{V}_0 = \varnothing$
while $\mathcal{M} \neq \varnothing$ do
    $n \leftarrow \text{argmin}_{m \in \mathcal{M}}\{rk(m)\}$
    $\mathcal{M} \leftarrow \mathcal{M} \backslash \{n\}$
    $\mathbb{V}' \leftarrow \mathsf{update}(\mathbb{V}, n)$
    if $\mathbb{V}'(n) \neq \mathbb{V}(n)$ then
        $\mathcal{M} \leftarrow \mathcal{M} \cup n.pred$
    $\mathbb{V} \leftarrow \mathbb{V}'$
return $\mathbb{V}$
```

We are going to run a tiny example on the mechanics of cut set.

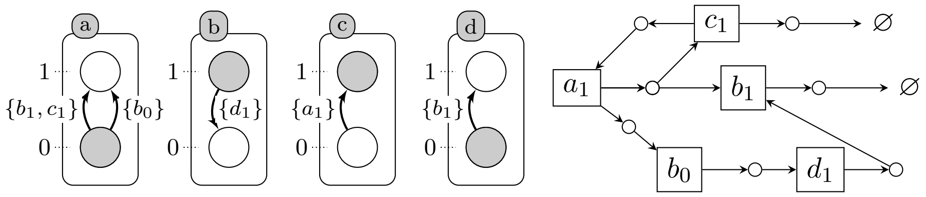


Figure 4.2: ABAN and SLCG for the reachability of a_1

Example 4.1. Figure 4.2 shows an ABAN and its under approximate SLCG of a_1. The ABAN has transitions $T = \{\{b_1, c_1\} \rightarrow a_1, \{a_1\} \rightarrow c_1, \{b_0\} \rightarrow a_1, \{d_1\} \rightarrow b_0, \{b_1\} \rightarrow d_1\}$, and initial state $\alpha = \langle a_0, b_1, c_1, d_0 \rangle$, the target state is $\omega = a_1$.

Table 4.1 shows the steps to obtain a saturated valuation. The cut set is $\{\{a_1\},\{b_1\},\{b_0,c_1\},\{c_1,d_1\}\}$, meaning that by inhibiting any of $\{a_1\}$ or $\{b_1\}$ or $\{b_0,c_1\}$ or $\{c_1,d_1\}$ leads to the unreachability of a_1.

Node	Rank	$\mathbb{V}$
$\varnothing$ (of b_1)	1	$\varnothing$
b_1	2	$\{\{b_1\}\}$
$\{b_1\} \to d_1$	3	$\{\{b_1\}\}$
d_1	4	$\{\{b_1\},\{d_1\}\}$
$\{d_1\} \to b_0$	5	$\{\{b_1\},\{d_1\}\}$
b_0	6	$\{\{b_0\},\{b_1\},\{d_1\}\}$
$\{b_0\} \to a_1$	7	$\{\{b_0\},\{b_1\},\{d_1\}\}$
$\varnothing$ (of c_1)	8	$\varnothing$
c_1	9	$\{\{c_1\}\}$
$\{b_1,c_1\} \to a_1$	9	$\{\{b_1\},\{c_1\}\}$
a_1	9	$\{\{a_1\},\{b_1\},\{b_0,c_1\},\{c_1,d_1\}\}$
$\{a_1\} \to c_1$	9	$\{\{a_1\},\{b_1\},\{b_0,c_1\},\{c_1,d_1\}\}$

Table 4.1: Result of Algorithm 1 on the SLCG in Figure 4.2. In column of $\mathbb{V}$, it is sufficient to prevent the local states $\{\{a_1\},\{b_1\},\{b_0,c_1\},\{c_1,d_1\}\}$ in order to inhibit the reachability of a_1.

Cut set inhibits certain *local states* to prevent the reachability. However, it mismatches the operations of revision in this thesis as revision aims at adding/deleting transitions to change the dynamics properties.

Here, we introduce the cut set w.r.t transitions. We change the update definition to $\mathsf{update}_{\mathsf{trans}}$ in order to capture the transitions needed for each local state.

Definition 4.5 ($\mathsf{update}_{\mathsf{trans}}$ for transitions)**.**

$$\mathsf{update}_{\mathsf{trans}}(\mathbb{V}, n) = \begin{cases} \zeta^N(\{\{n\}\} \cup \bigcup_{m \in n.next} \mathbb{V}(m)) & \text{if } n \in \mathbf{Sol} \\ \zeta^N(\prod_{m \in n.next} \mathbb{V}(m)) & \text{if } n \in \mathbf{LS} \end{cases} \tag{4.2}$$

If we apply this $\mathsf{update}_{\mathsf{trans}}$ to the same example, Example 4.1 and its cut set, resulted cut set w.r.t transitions is $\mathbb{V}_t = \{\{\{b_1,c_1\} \to a_1, \{b_0\} \to a_1\}, \{\{b_1,c_1\} \to a_1, \{d_1\} \to b_0\}, \{\{b_1,c_1\} \to a_1, \{b_1\} \to d_1\}\}$. By deleting any set of transitions in the cut set w.r.t transitions, we can ensure the unreachability of a_1 if a_1 is not at initial state.

4.2.3 Completion Set

Likewise, for an SLCG with suggesting certain local state is unreachable, we can use an analogous algorithm to compute completion sets satisfying that if the elements one completion set are reachable, the target local state becomes reachable.

Definition 4.6 (update′ for completion set)**.**

$$\mathsf{update}'(\mathbb{V}, n) = \begin{cases} \zeta^N(\prod_{m \in n.next} \mathbb{V}(m)) & \text{if } n \in \mathbf{Sol} \\ \{\varnothing\} & \text{if } \bigcup_{m \in n.next} \mathbb{V}(m) = \{\varnothing\} \\ \zeta^N(\{\{n\}\} \cup \bigcup_{m \in n.next} \mathbb{V}(m)) & \text{else} \end{cases}$$

Using the same definition of valuation, update′ for completion set assigns every node with a valuation $\mathbb{V}$. When the valuations are saturated, they stand for the completion sets of ω.

Example 4.2. Figure 4.3 shows an ABAN and its over approximate SLCG of a_1. The ABAN has transitions $T = \{\{b_1, c_0\} \to a_1, \{c_1, e_1\} \to a_1, \{d_1\} \to c_1\}$ and initial state $\alpha = \langle a_0, b_0, c_0, d_0 \rangle$, $\omega = a_1$.

Table 4.2 shows the steps to complete the graph like the procedures in cut set.

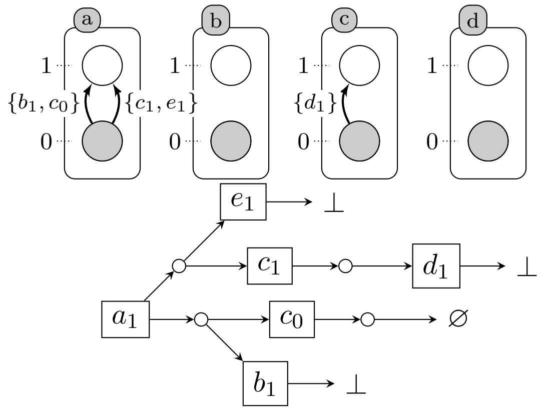


Figure 4.3: ABAN and SLCG for reachability of a_1

4.2.4 Completion by Over-Approximation

As stated in the previous chapter, verifying exact reachability is a hard task. Here we use two metrics, over-approximation and under-approximation to

Node	Rank	$\mathbb{V}$
$\perp$ (of b_1)	1	$\varnothing$
b_1	2	$\{\{b_1\}\}$
$\varnothing$ (of c_0)	3	$\{\varnothing\}$
c_1	4	$\{\varnothing\}$
$\{b_1, c_0\} \to a_1$	5	$\{\{b_1\}\}$
$\perp$ (of d_1)	6	$\varnothing$
d_1	7	$\{\{d_1\}\}$
$\{d_1\} \to c_1$	8	$\{\{d_1\}\}$
c_1	9	$\{\{c_1, d_1\}\}$
$\perp$ (of e_1)	10	$\varnothing$
e_1	11	$\{\{e_1\}\}$
$\{c_1, e_1\} \to a_1$	12	$\{\{c_1, e_1\}, \{d_1, e_1\}\}$
a_1	13	$\{\{a_1\}, \{b_1\}, \{c_1, e_1\}, \{d_1, e_1\}\}$

Table 4.2: Result of Algorithm 1 by replacing update with update′ on the SLCG in Figure 4.3. In column $\mathbb{V}$, it is sufficient to make one of the sets of the local states $\{\{a_1\}, \{b_1\}, \{c_1, e_1\}, \{d_1, e_1\}\}$ reachable in order to reach a_1.

replace and approach the notion of reachability (as shown in Figure 2.5 on page 20).

Over-approximation is the reasoning of pseudo-reachability in Definition 3.7 on page 33. It associates local states and transitions with the initial state according to the causalities: if there exists a reverse pathway from the target local state to the initial state, this target state can be reachable. Over-approximation does not take into consideration the order of transitions to be fired, which is the cause of inconclusiveness (literally it over-approximates the reachability). Thus over-approximation is a necessary condition of the reachability.

Figure 4.4 gives a first impression of this method. The visualization of the set of candidate regulations $R = \{(a, b, +), (c, a, -)\}$ is on the left. On the right, the initial ABAN has only three Boolean variables $\Sigma = \{a, b, c\}$ and initial state $\langle a_0, b_0, c_0 \rangle$, with transitions $T = \{\{a_1\} \to b_1\}$. b_1 is not reachable because of the unreachability of a_1. As $(c, a, -) \in R$, consistent transition $\{c_0\} \to a_1$ is added then a_1 becomes reachable which makes b_1 also reachable. $T' = \{\{a_1\} \to b_1, \{c_0\} \to a_1\}$, $CS = \{\{c_0\} \to a_1\}$.

Algorithm 10 in Appendix shows the detailed algorithm of the completion by over-approximation.

Likewise, we can define the completion by under-approximation.

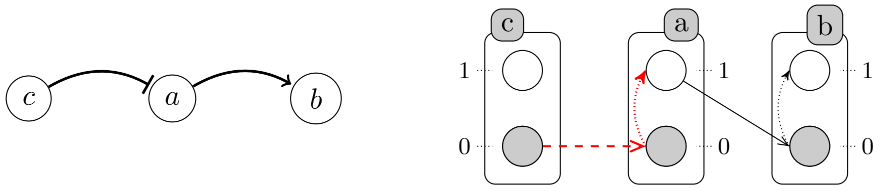


Figure 4.4: Completion by over-approximation. Dashed arrows represent added action.

4.2.5 Completion by Under-Approximation

Over-approximation is only a necessary condition of reachability. If one wants the guarantee of the reachability even at the price of redundant transitions, he may consider the completion by under-approximation.

Under-approximation is a variation of SLCG reasoning. It associates the local states and the transitions with the initial state and *all the states which may appear*. This association covers more orders of occurrence than over-approximation as it takes into account the mutual reachability between local Boolean states if both appear in the SLCG. The proof was done by Paulevé *et al.* [61]. Similar to over-approximation, in the SLCG of under-approximation, if there is a pathway between target local state and the initial state, this target state might be reachable.

However, all the orders of occurrence do not necessarily appear during the simulation, which suggest that this approach "under-approximates" the reachability.

Additionally, the computation of under-approximation is more complicated than that of over-approximation, as the set of associated local states grows during the computation. Former added local states need to be regarded as new "target states". This process is called *update*. Update does not stop until the set of associated local states becomes stable.

Definition 4.7 (Under-approximate SLCG)**.** Given ABAN $\mathbb{A} = (\Sigma, T)$, a global initial state α and a target local state ω, under-approximate SLCG $l = (V_{\text{state}}, V_{\text{sol}}, E)$ is the smallest recursive structure with $E \subseteq (V_{\text{state}} \times V_{\text{sol}}) \cup (V_{\text{sol}} \times V_{\text{state}})$ which satisfies:

$$\begin{aligned}
\omega &\in V_{\text{state}} \\
a_i \in V_{\text{state}} &\Leftrightarrow \{(a_i, A \to a_i)\} \subseteq E \\
A \to a_i \in V_{\text{sol}} &\Leftrightarrow \{(A \to a_i, X) \mid X = \varnothing \text{ if } a_i \in \alpha \wedge a_{1-i} \notin V_{\text{state}}, \\
&\qquad \text{else } \forall b_j \in A,\ X = b_j\} \subseteq E
\end{aligned}$$

where $V_{\text{state}} \subseteq LS$ is a set of local states, $V_{\text{sol}} \subseteq T$ is the set of solutions and X is one of the required local states of $A \rightarrow a_i$.

If we compare the definition of under-approximation with Definition 3.6 of over-approximation on page 31, the difference lies in the condition of X. The reasoning of over-approximation stops when certain automaton a reaches the initial state. However, to guarantee all the possible orders, we exige a_i and a_{1-i} are reachable from each other if they are both present in the under-approximate SLCG. The corresponding formula is $X = \varnothing$ if $a_i \in \alpha \wedge a_{1-i} \notin V_{\text{state}}$.

The visualization of under-approximate SLCGs has a slight difference with the one of over-approximate SLCGs. $a_0 \Rsh a_1$ means a_1 is to be reached *via* a_0. This notation seems redundant but it is useful when we need to continue the reasoning even if we reach the initial state until the SLCG is saturated. Local states in the initial state may have other successors besides $\bigcirc$. For example in Figure 4.7, d_0 is in the initial state, but the reasoning continues because d_1 appears in another branch.

Example 4.3. Given ABAN $\mathbb{A} = (\Sigma, T)$ in Figure 4.5 with $\Sigma = \{a, b, c, d\}$, $T = \{\{b_1, c_1\} \rightarrow a_1, \{b_1\} \rightarrow d_1, \{d_0\} \rightarrow b_1\}$, the initial state $\langle a_0, b_0, c_0, d_0\rangle$. With candidate regulations $R = \{(d, c, -), (c, d, -)\}$, Figure 4.6, Figure 4.7 and Figure 4.8 show the procedures of completion by under-approximation: after two additions of actions and one update, the SLCG becomes stable and a_1 becomes reachable. $T' = \{\{b_1, c_1\} \rightarrow a_1, \{b_1\} \rightarrow d_1, \{d_0\} \rightarrow b_1, \{d_1\} \rightarrow c_1, \{c_1\} \rightarrow d_0\}$, $CS = \{\{d_1\} \rightarrow c_1, \{c_1\} \rightarrow d_0\}$.

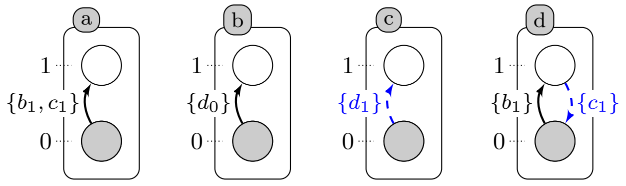


Figure 4.5: Example of completion by under-approximation. Dashed arrows represent added transitions.

Figure 4.6 shows the under-approximate SLCG of the reachability a_1. $\boxed{c_1}$ is unreachable then the joint state $\langle b_1, c_1\rangle$ is not reachable, a_1 is not reachable.

In Figure 4.7, according to candidate regulation $(d, c, -) \in R$, transition $\{d_1\} \rightarrow c_1$ is added. Thanks to the existence of $\{b_1\} \rightarrow d_1$, d_1 has successor b_1. However, the reachability of under-approximation requires all the possible occurrences are realizable, *i.e.* we need a transition with body d_0 for d_0

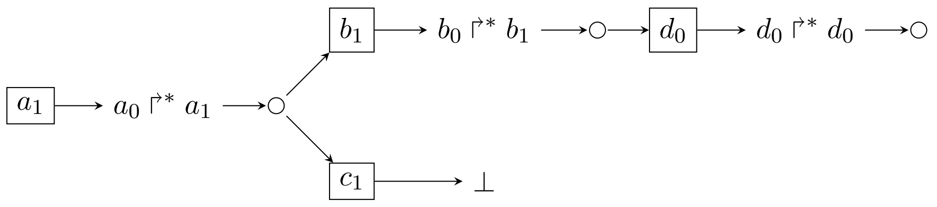


Figure 4.6: Step 1, under-approximation SLCG of Figure 4.5 studying the reachability of a_1 (small circles stand for solutions and squares stand for required local states).

and d_1 can be reached from each other, because both d_0 and d_1 are present in the SLCG.

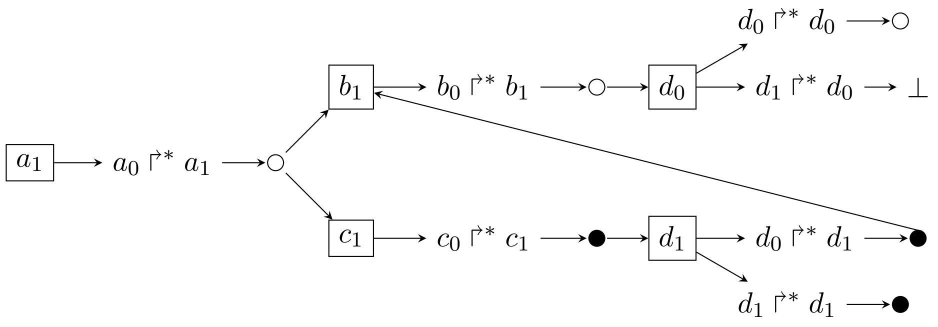


Figure 4.7: Step 2 of completion by under-approximation (filled small circles stand for new possible solutions after completion).

In Figure 4.8, the completed under-approximate SLCG shows a_1 is now reachable after adding $\{c_1\} \rightarrow d_0$ according to $(c, d, -) \in R$ as there is no more $\perp$ in the SLCG.

To sum up the whole process, we begin with an incomplete model and a set of candidate regulations.

Algorithm 11 in Appendix shows the detailed procedures of completion by under-approximation.

Remark 4.1. Neither completion by under-approximation nor completion by over-approximation can output complex transitions (transitions with multiple variables in the head). If one wants to assess if certain complex transition can be added to the incomplete model, he can replace regulations in the input with transitions, skip the step of checking the consistency, then use directly the transition to complete the incomplete model.

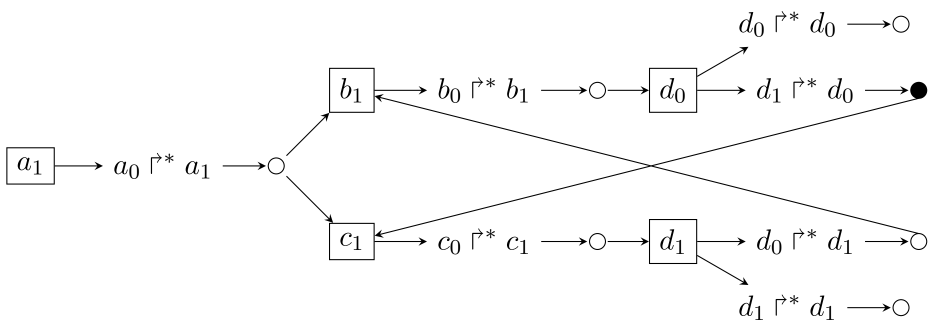

a_1
$a_0 \upharpoonright^* a_1$
b_1
$b_0 \upharpoonright^* b_1$
c_1
$c_0 \upharpoonright^* c_1$
d_0
d_1
$d_0 \upharpoonright^* d_0$
$d_1 \upharpoonright^* d_0$
$d_0 \upharpoonright^* d_1$
$d_1 \upharpoonright^* d_1$

Figure 4.8: Step 3 of completion by under-approximation (filled small circles stand for new possible solutions after completion).

4.3 Model Inference via Statistics

As we already have the completion methods, to make CRAC applicable, we need to obtain candidate regulations R. Here we introduce a method to generate candidate regulations to be verified from time series data.

In the previous contents, only the systems consisting of transitions with delay of 1 time unit are discussed (BN, ABAN, *etc.*), *i.e.* the influence done by variables will take place at the next time point (immediately). However, in biological context, some reactions have a long duration, and some even need a whole observation period to take place. For this reason, we require approaches to reveal transitions of different delays: for an observation of period T, possible delays lie in $[1; T-1]$.

4.3.1 Preliminaries

In Definition 2.1 on page 9, the definition of regulatory network (RN) is originally related to a set of approximated ordinary differential equations (ODEs) [43] (continuous modeling):

$$\frac{\mathrm{d}x_v}{\mathrm{d}t} = k_v - \lambda_v x_v + \sum_{u \in v.pred} x_u k_{uv} \tag{4.3}$$

where x_v is the concentration of variable v, $k_v, \lambda_v \in \mathbb{R}$ are self-regulation kinetic parameters (containing protein degradation) and $k_{uv} \in \mathbb{R}$ are kinetic parameters of external regulations.

Here, considering the following reasons, we make several modifications to the ODE.

1. The change rate done by external regulations is not always proportional to the concentration of corresponding external variables. Consider an elementary biochemical reaction $m\mathrm{A} + n\mathrm{B} \rightarrow \mathrm{C}$, according to the rate law, the synthesis rate of C is $r_{\mathrm{C}} = k[\mathrm{A}]^m[\mathrm{B}]^n$ where k is a fixed positive real number and m, n are the parameters characterizing reaction order.

2. In the community of model inference, self-regulations are often considered difficult to be detected because nearly every transition could be explained as the result of a self-regulation. For example, in the problem description of DREAM8[6], self-regulations are *a priori* ignored.

3. According to the differential equation, the influence takes effect immediately without delay. However, considering the discretization on time, we have to replace the differential mark d in 4.3 into difference mark Δ and set a delay δ which takes only natural numbers to simulate state change.

To integrate these hypotheses, the equations are modified to:

$$\frac{\Delta x_v(t)}{\Delta t} = \sum_{u \in v.pred} x_u^n(t-\delta)k_{uv} \tag{4.4}$$

where $\delta \in \mathbb{N}^*$ is the delay of the regulation, $n \in \mathbb{R}^+$ is the order of regulation (corresponding to the reaction order). Also, self-regulations are not taken into account, *i.e.* k_v and λ_v are set to 0.

The definition of RN (Definition 2.1) requires every regulation take effect immediately. To cooperate with the new equations, it is adapted to the version with delay.

Definition 4.8 (Regulatory network with delay)**.** A regulatory network with delay is a labeled directed graph $G = (V, E)$ where

- each vertex v of V, called variable, is provided with a boundary $b_v \in \mathbb{N}$ less or equal to the out-degree of v in G.
- each arc $u \in v$ of E is labelled with a triplet $(t_{uv}, \alpha_{uv}, \delta)$ where t_{uv} is an integer between 1 and b_v, called qualitative threshold, where $\alpha_{uv} \in \{+, -\}$ is the sign of the regulation and $\delta \in \mathbb{N}^+$ is the delay of the regulation.

With the new formalization, we can now address the inference problem.

[6]`https://www.synapse.org/#!Synapse:syn1720047/wiki/55342`

4.3.2 Partial Correlation

In huge biological networks, it is not possible to compute candidate regulations by hand nor to traverse all the possible addable transitions, as stated formerly, the verification of $O(3^{|GlobalStates|})$ states is a huge task: if the influenced variable is fixed, every other variable has 3 possible influence, promotion, inhibition and no regulation.

To deal with the high complexity of the verification of $O(3^{|GlobalStates|})$ states, *a priori* knowledge is needed. Some unsatisfied states can be eliminated without global verification. In order to take all the information into account, statistical approaches come into sight. We use correlation coefficients to try to match the evolution of variables with Equation 4.4 on page 65. Correlation coefficients characterizes how linear or monotonous the relations between variables are all along the sampling period. The possible values of all the mentioned correlation coefficients lie in the interval $[-1, 1]$. The closer to 1 the absolute value of the coefficient is, the more correlated the variables are. Particularly, 1 suggests total positive correlation and -1 suggests total negative correlation.

In Equation 4.4, for different n,

1. $n = 1$, the change rate of certain variable is the linear sum of other variables. The *linear* correlation between the change rate of one variable and the value of other variables is detectable via an approach using Pearson correlation coefficients (PCC).

2. $n \neq 1$, the correlations between variables are no longer linear but still monotonous. Analogously the *monotonous* correlation is detectable by Spearman correlation coefficients (SCC), which is the application of Pearson correlation coefficients on the rank of variables.

One additional profit of statistical approaches is that original time series data are discretized before being used, which cause a loss of information. This loss could lead to an imprecise model. However, some statistical approaches could use directly the continuous time series data as input which avoid this drawback.

Definition 4.9 (Pearson correlation coefficient)**.** Pearson correlation coefficient (r) is the covariance of the two variables divided by the product of their standard deviations. When applied to a sample (time series data), the formula is converted to:

$$r_{x,y} = \frac{\operatorname{cov}(x,y)}{\sigma_x \sigma_y} = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{N}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{N}(y_i - \bar{y})^2}}$$

Where x, y are variables, $\mathrm{cov}(x, y)$ is the covariance of x and y, σ_x is the standard deviation of x, N is the sample size and x_i, y_i are the i-th sample points of x, y.

The definition of Spearman correlation coefficient is related to that of Pearson correlation coefficient.

Definition 4.10 (Spearman correlation coefficient)**.** Spearman correlation coefficient (rs) is the Pearson correlation coefficient between the ranked variables.

$$rs_{x,y} = r_{\mathrm{rg}_x,\mathrm{rg}_y} = \frac{\mathrm{cov}(\mathrm{rg}_x, \mathrm{rg}_y)}{\sigma_{\mathrm{rg}_x}\sigma_{\mathrm{rg}_y}}$$

Where rg_x is the rank variable of x.

Example 4.4. Variable $x = \{3, 10, 1, 7, 6\}$, the rank variable of x is $rg_x = \{4, 1, 5, 2, 3\}$.

If two variables are highly correlated and form an independent system, their correlation can be perfectly detected by PCC or SCC. However, variables are usually simultaneously influenced by more than one variables, which is the biological reality. For example, for three variables a, b, c, if a is activated by b and inhibited by c, the PCC or SCC of $r_{\frac{\mathrm{d}a}{\mathrm{d}t},b}$ is biased by c. To get rid of certain variables, one can apply partial Pearson correlation coefficients [4] or partial Spearman correlation coefficients [8].

Partial correlation coefficients are denoted pr.

Definition 4.11 (Partial Pearson correlation coefficient (PPCC))**.** $pr_{xy\cdot\mathbf{z}}$ is the partial Pearson correlation coefficient of x and y ignoring the influence by $\mathbf{z}$.

$$pr_{xy\cdot\mathbf{z}} = \frac{pr_{xy\cdot\mathbf{z}\setminus\{z_0\}} - pr_{xz_0\cdot\mathbf{z}\setminus\{z_0\}}pr_{z_0y\cdot\mathbf{z}\setminus\{z_0\}}}{\sqrt{1 - pr^2_{xz_0\cdot\mathbf{z}\setminus\{z_0\}}}\sqrt{1 - pr^2_{z_0y\cdot\mathbf{z}\setminus\{z_0\}}}}$$

Where x, y are variables, $\mathbf{z}$ is a set of variables, z_0 is an arbitrary element in z. Particularly, if $\mathbf{z}$ has only one element z_0, the formula becomes

$$pr_{x,y\cdot\{z_0\}} = \frac{r_{x,y} - r_{x,z_0}r_{z_0,y}}{\sqrt{1 - r^2_{x,z_0}}\sqrt{1 - r^2_{z_0,y}}}$$

The partial Spearman correlation coefficient is defined likewise.

Definition 4.12 (Partial Spearman correlation coefficient (PSCC))**.** Partial Spearman correlation coefficient is denoted prs

$$prs_{x,y\cdot\mathbf{z}} = pr_{rg_x,rg_y\cdot rg_\mathbf{z}}$$

The possible value of all the mentioned correlation coefficients are in the interval $[-1, 1]$. The closer to 1 the absolute value of the coefficient is, the more correlated the variables are. Particularly, 1 suggests total positive linear/monotonous correlation and -1 suggests total negative linear/monotonous correlation. *e.g.* if we find the PSCC of $\frac{\mathrm{d}a}{\mathrm{d}t}$ and b is -0.9 (high enough), we can add $b \xrightarrow{-} a$ to the set of candidate regulations.

Overall Process

With the former definitions, we propose a method applying partial correlation to detect the relevance of each pair of variables.

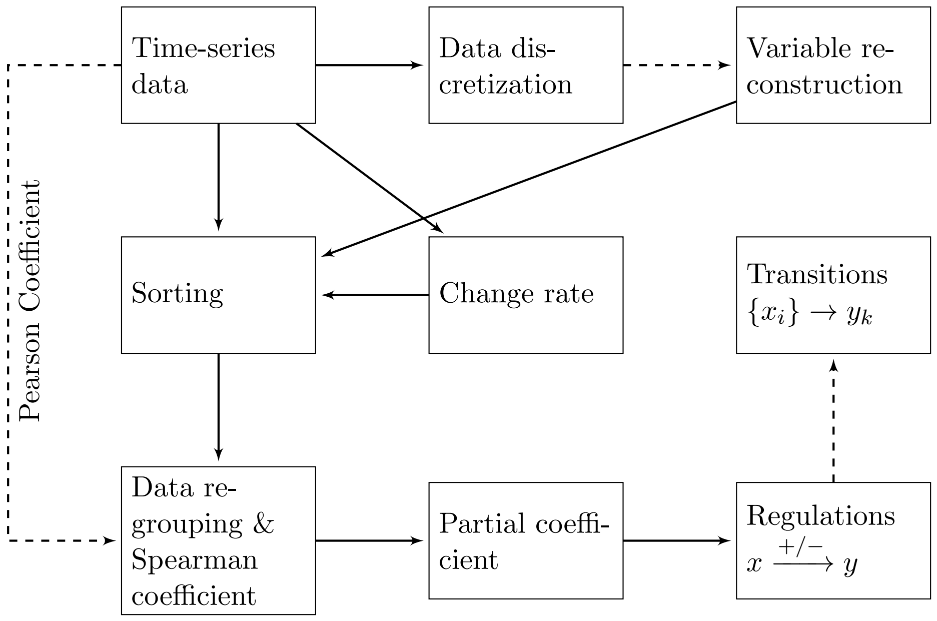


Figure 4.9: Workflow of the whole procedure of candidate regulations generation (dashed arrows stand for optional processes)

Figure 4.9 shows the procedure of regulation generation: before discretizing original data, Pearson or Spearman correlation coefficients [72, 37] are computed for identifying the relevance between original data and change rates of variables (in linear or monotonic way respectively). If cooperation between variables exists, former coefficients need replacing by partial ones [22] for more precise result. Resulting coefficients above the threshold (One can set a threshold of correlation, *e.g.* 0.7) suggest there probably exist regulations between variables.

Furthermore, to complete Biological Regulatory Networks in the form of ABAN, more accurate regulations are inferred through variable reconstruction, which splits a variable into several new variable according to its qualitative levels. For example, variable a has two discrete levels a_0 a_1, then the correlation coefficients as well as the partial coefficients are computed in the domain of a_0 and of a_1 separately. As a result, the correlation between a_0 and other variables and that of a_1 are computed, with which candidate transitions are deduced.

As all the subroutines depicted in Figure 4.9 are defined, starting from original data, regulations in form of René Thomas' model [79] or ABAN are resulted step by step. In the next section, the data processing in Figure 4.9 will be introduced with examples.

Data Regrouping

To gain a better understanding of correlations between observation data and change rate, certain observed data of one variable are replaced by corresponding change rate. By calculating the correlation coefficients of such matrix, regulation on this variable is characterized.

Definition 4.13 (Regrouping)**.** Let A be a $n \times T$ matrix, representing time-series data, where n is the number of variables and T is the period of the time-series data. A_{ij} is the i-th variable at time j. Let δ be the delay we want to explore, there are in total n matrices of size $n \times (T-\delta)$ after regrouping. The k-th matrix A'^k is defined:

$$A'^k_{ij} = \begin{cases} \dfrac{A_{i,j+\delta} - A_{ij}}{\delta} & \text{if } i = k \\ A_{ij} & \text{else} \end{cases}$$

We regroup the change rate of one variable and the values of the other variables in order to infer the correlations between them.

Example 4.5. Let us take the time-series data of 4 variables a, b, c, d as an example. Matrix A below records the state of the system of discrete time point $t = 0, 1, 2, 3$, A'^1 is the first regrouped matrix ($k = 1$) as the time-series of a is placed on the first row. A'^1 is used to compute the correlation between the change rate of a and other variables of delay $\delta = 1$.

$$a'(t) = \frac{\Delta a(t)}{\Delta t} = \frac{a(t+1) - a(t)}{(t+1) - t} = a(t+1) - a(t) \tag{4.5}$$

$$A = \begin{array}{c} t \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \left[\begin{array}{cccc} a(0) & a(1) & a(2) & a(3) \\ b(0) & b(1) & b(2) & b(3) \\ c(0) & c(1) & c(2) & c(3) \\ d(0) & d(1) & d(2) & d(3) \end{array}\right] \end{array} \rightarrow A'^1 = \begin{array}{c} t \\ a' \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \left[\begin{array}{ccc} a'(0) & a'(1) & a'(2) \\ b(0) & b(1) & b(2) \\ c(0) & c(1) & c(2) \\ d(0) & d(1) & d(2) \end{array}\right] \end{array}$$

In this way, regulations of b, c, d on a are then evaluated by PPCC or PSCC (see Definition 4.11 and 4.12).

With the matrices A'^k, we can compute the matrix containing the correlation information of all the variable pairs.

Definition 4.14 (Correlation matrix)**.** Let A'^k be the regrouped matrices of time-series data matrix A with $k \in [1; n]$ and Σ be the set of all the variables, the $n \times n$ correlation matrix is defined:

$$\mathcal{R}_{ij} = pr_{ij,\Sigma\setminus\{i,j\}}$$

where the original data for computing $pr_{ij,\Sigma\setminus\{i,j\}}$ come from A'^i. As we do not study self-regulation in this thesis, we set $pr_{ij,\Sigma\setminus\{i,j\}} = \text{N}/A$. Also, pr (PPCC) can be replaced by prs (PSCC). In the following, we abuse the simplification of the notation $pr_{ij,\Sigma\setminus\{i,j\}}$ by r_{ij}.

To expand the applicability, matrices A'^k and $\mathcal{R}$ representing different delays can be formed analogously by only changing the value of δ.

4.3.3 Variable Reconstruction

By following previous steps, candidate regulations in form $(a, b, +/-)$ are deduced, but the result is not in the precise form of ABAN. As is stated on page 30, ABAN has a finer description. We need to study the regulations in different qualitative levels of each variable. To obtain a result in ABAN form, variable reconstruction is necessary.

Definition 4.15 (Variable Reconstruction)**.** Let $x(t)$ be a variable in time series data with l levels from 0 to $l-1$, the corresponding intervals are $[0, x_0], [x_0, x_1], \cdots, [x_{l-2}, x_{l-1}]$, with x_i the thresholds. The reconstructed variable of x are: $x'_i(t) = x(t)$ with their domain in $\{t | x(t) \in [x_{i-2}, x_{i-1}]\}$ and $i \in [0; l-1]$.

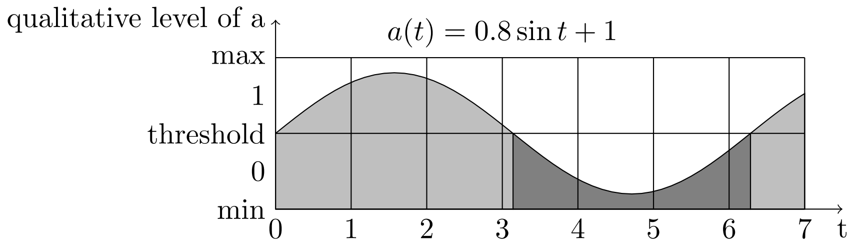


Figure 4.10: Example of variable reconstruction

Example 4.6. In Figure 4.10, with the Boolean discretization on the threshold, variable $a(t) = 0.8\sin t + 1$ $(t \in [0;7])$ is split into 2 variables $a'_0(t) = 0.8\sin t{+}1$ $(t \in (\pi, 2\pi))$ (dark gray) and $a'_1(t) = 0.8\sin t{+}1(t \in [0,\pi]\cup[2\pi, 7])$ (light gray) with different domains according to the qualitative threshold.

For the system with two Boolean variables a, b, the PCCC matrix is created according to Definition 4.14:

$$\mathcal{R} = \begin{bmatrix} r_{a'a} & r_{a'b} \\ r_{b'a} & r_{b'b} \end{bmatrix} = \begin{bmatrix} \text{N/A} & r_{a'b} \\ r_{b'a} & \text{N/A} \end{bmatrix}$$

After variable reconstruction, a and b are split into four, a_0, a_1, b_0, b_1. The corresponding PCCC matrix $\mathcal{R}'$ is:

$$\mathcal{R}' = \begin{bmatrix} r_{a'_0a_0} & r_{a'_0a_1} & r_{a'_0b_0} & r_{a'_0b_1} \\ r_{a'_1a_0} & r_{a'_1a_1} & r_{a'_1b_0} & r_{a'_1b_1} \\ r_{b'_0a_0} & r_{b'_0a_1} & r_{b'_0b_0} & r_{b'_0b_1} \\ r_{b'_1a_0} & r_{b'_1a_1} & r_{b'_1b_0} & r_{b'_1b_1} \end{bmatrix} = \begin{bmatrix} \text{N/A} & \text{N/A} & r_{a'_0b_0} & r_{a'_0b_1} \\ \text{N/A} & \text{N/A} & r_{a'_1b_0} & r_{a'_1b_1} \\ r_{b'_0a_0} & r_{b'_0a_1} & \text{N/A} & \text{N/A} \\ r_{b'_1a_0} & r_{b'_1a_1} & \text{N/A} & \text{N/A} \end{bmatrix}$$

Here, even though the size of matrix has doubled (it can also be l times large, depending on the number of discrete levels), as the domains of a_0, a_1, b_0, b_1 are not continuous and the variables with same origins (like a_0 and a_1) have no common domain, hence $r_{a_0a_1}$, $r_{a_1a_0}$, $r_{b_0b_1}$, $r_{b_1b_0}$ are meaningless. Also, even some variables of from different origins may have no common domain, that will lead to $r_{a_ib_j} = 0$, making resulted matrix more sparse, which gives possibilities of optimization.

4.3.4 Toy Example

In order to better illustrate the whole process of generating candidate regulations, a toy example of 4 variables a, b, c and d is given below. To prepare the inputs for the model inference in Section 4.2.5 on page 61, we want to output candidate regulations.

Input: time series data in Table 4.3, equi-temporal measurement of 8 time units.

Output: candidate regulations for model completion by over-/under-approximation.

t	0	1	2	3	4	5	6	7	8
a	2.01	2.51	1.97	1.17	0.94	0.70	0.31	0.06	0.06
b	0.74	0.87	0.78	0.33	0.51	0.82	0.86	1.81	1.08
c	0.43	0.18	0.42	0.23	0.17	0.23	0.32	0.53	0.80
d	1.62	1.22	1.07	0.57	0.27	0.28	0.24	0.27	0.31

Table 4.3: Original time-series data generated by Gene Net Weaver [73]

Change rates are obtained in Table 4.4 by Equation 4.5.

t	0	1	2	3	4	5	6	7
a	0.5	−0.54	−0.8	−0.23	−0.24	−0.39	−0.25	0.0
b	0.13	−0.09	−0.45	0.18	0.31	0.04	0.95	−0.73
c	-0.25	0.24	−0.19	−0.06	0.06	0.09	0.21	0.27
d	-0.4	−0.15	−0.5	−0.3	0.01	−0.04	0.03	0.04

Table 4.4: Change rates derived from original data by $x'[t] = x[t+1] - x[t]$

After data regrouping, we obtain 4 matrices consisting of the change rate of each variable respectively, four correlation matrices are then computed:

$$\begin{array}{c|cccc} & a' & b & c & d \\ \hline a' & 1.0 & 0.09 & -0.30 & -0.09 \\ b & 0.09 & 1.0 & -0.74 & 0.42 \\ c & -0.30 & -0.74 & 1.0 & -0.38 \\ d & -0.09 & 0.42 & -0.38 & 1.0 \end{array} \quad \begin{array}{c|cccc} & a & b' & c & d \\ \hline a & 1.0 & 0.65 & 0.90 & -0.98 \\ b' & 0.65 & 1.0 & 0.75 & -0.62 \\ c & 0.90 & 0.75 & 1.0 & -0.89 \\ d & -0.98 & -0.62 & -0.88 & 1.0 \end{array}$$

$$\begin{array}{c|cccc} & a & b & c' & d \\ \hline a & 1.0 & 0.72 & -0.56 & -0.93 \\ b & 0.72 & 1.0 & -0.71 & -0.71 \\ c' & -0.56 & -0.71 & 1.0 & 0.70 \\ d & -0.93 & -0.71 & 0.70 & 1.0 \end{array} \quad \begin{array}{c|cccc} & a & b & c & d' \\ \hline a & 1.0 & -0.68 & 0.78 & 0.89 \\ b & -0.68 & 1.0 & -0.93 & -0.88 \\ c & 0.78 & -0.93 & 1.0 & 0.91 \\ d' & 0.88 & -0.88 & 0.91 & 1.0 \end{array}$$

r' is formed by taking the i-th line from the i-th matrix, which suggests the relevance between change rate of one variables and the others.

$$r' = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a' & b' & c' & d' \end{array} \\ \left[\begin{array}{cccc} 1.0 & 0.09 & -0.30 & -0.09 \\ 0.65 & 1.0 & 0.75 & -0.62 \\ -0.56 & -0.71 & 1.0 & 0.70 \\ 0.88 & -0.87 & 0.91 & 1.0 \end{array}\right] \end{array}$$

We can set an arbitrary threshold *e.g.* 0.6. All the coefficients with their absolute value greater than 0.6 are listed below:

$$(b, a, 0.65),\ (b, c, 0.75),\ (b, d, -0.62),\ (c, b, -0.71)$$

$$(c, d, 0.70),\ (d, a, 0.89),\ (d, b, -0.87),\ (d, c, 0.91)$$

For example $(d, b, 1, -0.87)$ tells that $d \xrightarrow{-} b$ is probably a good candidate regulation as its absolute value of correlation coefficient is close enough to 1. Like this, a BRN model is formed, see Figure 4.11. According to the configuration of ABAN (whether absence of regulation is regarded as counter regulation), an ABAN is then deduced.

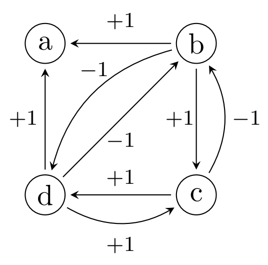


Figure 4.11: Resulted candidate regulations of the toy example

In fact, the choice of threshold of correlation coefficients has little influence if it is above 0.5: we can even lower the threshold if resulting regulations do not satisfy desired properties. Because when coefficient $r = 0.5$, then the 95% prediction interval of $y|x$ will be about 13% smaller than the 95% prediction interval of y, *i.e.* y behaves more relevantly than individually [40].

It is worth noticing that this method does not take into account the interactions between regulations, *i.e.* we do not distinguish between the conjunctions and disjunctions. For candidate regulations $(a, c, +)$ and $(b, c, +)$, the following set of transitions are both consistent $\{\{a_1, b_1\} \rightarrow c_1\}$ and

$\{\{a_1\} \rightarrow c_1, \{b_1\} \rightarrow c_1\}$. In CRAC, we do not consider conjunctions for candidate regulations.

Last but not least, the high-correlated variable pairs are not necessarily the reality but can also be a coincidence, *i.e.* the inferred candidate regulations/transitions can probably reproduce the system dynamics but cannot guarantee the identity. In fact, there is no method that can guarantee it reveals the reality, as what model inference does is to infer *via* the correlations instead of causality. Causality, or the reason behind the observation is very hard to retrieve.

4.4 Model Revision *via* Reachability and Interpretation Transitions (M2RIT)

When modeling a real system, instead of causality, one usually requires to assess the *consistency* between a given modeling network and the concrete system by checking whether 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.

Inoue [41] has shown that Boolean networks can be represented by logic programs. In this section, we provide an approach 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 [66], 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, asynchronous semantics helps one to capture more realistic behaviors [6]. 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.

Considering the ignorance of conjunctions by CRAC, we use a more precise learning technique (LFIT) to obtain the model to be revised in M2RIT [13]. The trade-off is that M2RIT cannot deal with noisy data sets and there are more constraints in the revision phase because we have to keep the consistency of the resulted network with the original time-series data.

Here we follow the same methodology as shown in Figure 4.1 on page 53. First we obtain the model to be revised via learning method LFIT, then we modify the learned transitions according to the SLCG but in a way that maintaining the revised model can always reproduce the original time-series data.

4.4.1 Learning From Interpretation Transitions (LFIT)

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 thesis, we adapt the algorithms of [66, 49] 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 the learned rules consistent.

4.4.2 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 [66] is not suitable. Martínez *et al.* [49] have proposed a modeling of multi-valued synchronous systems as annotated logic program. This modeling can be applied to represent 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. Let us consider a *multi-valued logic program* as a set of *rules* of the form

$$var^{val} \leftarrow var_1^{val_1} \wedge \cdots \wedge var_n^{val_n} \tag{4.6}$$

where var^{val} and $var_i^{val_i}$ are atoms ($n \geq 1$). For any rule R of the form (4.6), 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 (4.6) as $b(R) = \{var_1^{val_1}, \ldots, var_n^{val_n}\}$. A rule R of the form (4.6) 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 states is denoted **LS**. The subset of a 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.

4.4.3 Modeling and Learning of Asynchronous Dynamics

Due to the non-deterministic nature of asynchronous systems and its restriction to at most one variable change per transition, the notions of consistency, realization and successor have to be redefined as follows in order to be adapted to further revision process.

Definition 4.16 (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.

Former definition of consistency is Definition 4.2 on page 55. Here we enlarge the domain to the consistency between logic programs and state transitions (time-series data), *i.e.* all rules are applied in the state transitions.

Definition 4.17 (Program realization)**.** Let P be a logic program and E be a set of state transitions. P realizes E if $\forall (I, J) \in E, \exists R, b(R) \subseteq I, (I \setminus J) = \{h(R)\}$.

Similarly, program realization describes that all the state transitions can be explained by the existing rules.

Definition 4.18 (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

$$T_P^{as}(I, S) = \{I \backslash \{v^{val'}\} \cup \{v^{val}\} | v^{val'} \in I, v^{val} \in T_P(I, S)\} \cup \{I | T_P(I, S) = \varnothing\}$$

Like the definition of ABAN dynamics (Definition 3.2 on page 28), here asynchronous successors defines formally system dynamics of logic programs.

We now adapt the **LFIT** algorithm of [66] to the learning of asynchronous systems. The idea of learning asynchronous and generalized semantics using **LFIT** was published on [65]. Here we incorporate the learning of asynchronous dynamics with SLCG and its related model checkers PermReach and ASPReach.

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. This mechanics allows one to model 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 $var^{val''} \in b(R), var^{val'''} \in b(R') \implies val'' = val'''$.

In [49], multi-valued least specialization was used to learn multi-valued *synchronous* systems dynamics. Like in previous versions of LFIT, asynchronous LFIT takes a set of state transitions E as input and outputs a logic program P that realizes E.

Starting from the most general rules, least specialization allows one to learn the minimal rules of such system iteratively from its state transitions $(I, J) \in E$. 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 is specialized using least specialization. The resulting program P' realizes all previously treated state transitions plus (I, J). By doing so iteratively for each transition, the algorithm outputs a program P which models the dynamics of the system observed in the state transitions E.

Asynchronous LFIT algorithm

- INPUT: $\mathcal{B}$ a set of annotated atoms and E a set of state 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: $P$
```

Like in Definition 4.2, here we introduce the consistency between reachability properties and logic programs.

Definition 4.19 (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 4.20 (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 $b(R)$ contains already all the variables in the logic program P, the only way to specialize R is to remove R from P as there is no more available variables to be added in $b(R)$.

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 4.21 (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 4.22 (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. a set of property S: if their exists an ordering S' of the elements of S such that each i-th revision, $0 \leq i \leq |S'|$, (P being the 0-th revision) is revisable w.r.t. the $(i+1)$-th property.

From definition 4.22, 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 SLCG structure to restrict the search to valid ordering of the properties.

4.4.4 Revision

In this section, we are going to present our algorithm **M2RIT** (**M**odel **R**evision *via* **R**eachability and **I**nterpretation **T**ransitions) exploiting the previous formalization to fit a logic program to reachability properties. Given a set of state transitions E of an asynchronous system S, a logic program P is learned via the adaptation of **asynchronous LFIT** of section 4.4.3. 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 P is consistent with S in the meaning of reachability. 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 goal is to find what could be considered a metric of minimal revision of P: a revision P' s.t. $\nexists P'', (P'' \setminus P \cap P'') \subseteq (P' \setminus P \cap P')$

To make use of SLCG to study reachability properties, we need the model noted in the form of ABAN. As the notation of transitions in ABAN is $\{a_i, b_j\} \rightarrow c_k$, which can be perfectly translated to NLP $c_k \leftarrow a_i, b_j$. This property is valid only in binary situation.

Specialization and generalization algorithms aim at revising the rules nearest to the target state in the SLCG. 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. If the direct revision on the unsatisfied element fails, we replace the element with its successsors in the SLCG.

Well-formed detailed algorithms of specialization and generalization are Algorithm 12 and Algorithm 13 in Appendix B.

Specialization algorithm:

- **Input**: a logic program P, an unsatisfied element $reach(\alpha, \omega) =$ **False**, a satisfied reachable set Re
- **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 $R' \in ls(R)$ and $P \cup \{R'\} \backslash \{R\}$ is consistent with E

   (This consistency can be verified by the condition $\nexists (I, J) \in E$ s.t. $\nexists R' \in P \cup \{R''\} \backslash \{R\}$, $h(R'') \in J, b(R'') \subseteq I\}$)

   - **If** $P' \leftarrow P \backslash \{R\} \cup \{R'\}$, $reach(\alpha, \omega) =$ **False** for P' and P' satisfies Re, **return** P'
3. $Rev \leftarrow b(R)$ with $h(R) = Rev$ and back to step 2
4. There is no revision for $reach(\alpha, \omega) =$ **False**, **return** $\varnothing$

Generalization is similar to specialization but without the constraint of the observation, as the observation is partial, P may describe some state transitions never observed.

Generalization algorithm:

- **Input**: a logic program P, an unsatisfied element $reach(\alpha, \omega) =$ **True**, a satisfied 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** $P' \leftarrow P \backslash \{R\} \cup \{R'\}$, $reach(\alpha, \omega) =$ **True** for $P'$and P' satisfies Un, **return** P'
3. $Rev \leftarrow b(R)$ with $h(R) = Rev$ and back to step 2
4. There is no revision for $reach(\alpha, \omega) =$ **True**, **return** $\varnothing$

The main revision algorithm (see below) starts with constructing the SLCGs to verify Re and Un so as to obtain the reachability/unreachability properties to be satisfied. Then, for the unsatisfied properties, the program P has to be revised. SLCG for one target state may contain the topology of SLCGs for other target states, hence revision based on one SLCG can influence other SLCGs. By starting with the SLCGs with least dependencies on others, *i.e.* the ones containing the least other SLCGs (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 revisions 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 outputs it.

Main revision algorithm:

- **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 SLCGs 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 $j = (\alpha, \omega)$, $\omega \in SLCG(i)$ 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 SLCG for $i$
    (b) **If** $l_i$ becomes consistent because of former revision, $L \leftarrow L \backslash \{l_i\}$ 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. $L \leftarrow L \backslash \{l_i\}$
9. **If** $L \neq \varnothing$ , back to step 1
10. **Return** $P$
```

4.4.5 Toy Example

After introducing the algorithms, we are going to show how they work in a minimum toy example.

Example 4.7. Let us consider a logic program P with rules: $a_1 \leftarrow b_1$, $a_1 \leftarrow d_1 \wedge c_0$, $b_1 \leftarrow c_0$, $c_1 \leftarrow b_0$ and initial state: $\alpha = \langle a_0, b_0, c_0, d_0 \rangle$. Reachability properties to be verified: $Un = \{(\alpha, b_1), (\alpha, d_1)\}$ and $Re = \{(\alpha, a_1)\}$. Figure 4.12 shows the SLCG of a_1.

We compute first L revealing the inclusion relations between the SLCGs: $L = \{\{(\alpha, a_1), (\alpha, b_1), (\alpha, d_1)\}, \{(\alpha, b_1)\}, \{(\alpha, d_1)\}\}$. The SLCG of b_1 and that of d_1 are contained in the SLCG of a_1. We begin with the SLCG containing least others SLCG of b_1 or d_1. $b_1 \leftarrow c_0$ can be specialized to $b_1 \leftarrow c_0 \wedge a_1$ to make b_1 unreachable. Here the a_1 becomes unreachable due to the unreachability of b_1. Generalizing $a_1 \leftarrow d_1 \wedge c_0$ can solve this problem, it can only be generalized to $a_1 \leftarrow c_0$ as $d_1 \in U_K$.

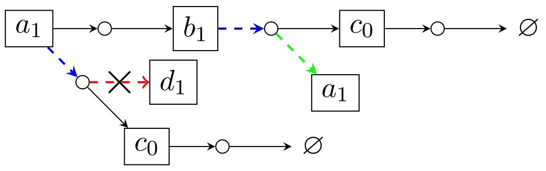


Figure 4.12: Toy example of M2RIT, where dashed arrows are the revisions.

4.5 Résumé

In this chapter we presented two methods to infer/revise models based on different *a priori* knowledge.

The first method **CRAC** (Completion *via* Reachability And Correlations) consists of two parts. The first part allows one to construct a model among a large number of candidate regulations while aiming at (un)reaching certain states, but it is unable to obtain the candidates. To cover this disadvantage, the second part, statistic approach *via* correlation coefficients provides us with candidate regulations to be verified by the first part. Correlation coefficient uses all the continuous time-series data to suggest what a complete model might be, thus can feed the first part with candidate regulations.

Considering the disadvantages of the combination of CRAC, the second method **M2RIT** (Model Revision *via* Reachability and Interpretation

Transitions) does not need candidate transitions and can adjust the result by reachability constraints. It revises the logic program learned by LFIT w.r.t. the knowledge on reachability properties. When talking about reachability, we should fix at first update scheme of the dynamic system. We use asynchronicity as the update scheme as it implies non-determinism which is meaningful to the modeling of nuanced uncertain parts in biology. From the point of view of revisability, asynchronicity gives the possibilities of modifying the existing transitions. If the logic program is revisable, the revision is consistent with both state transitions and reachability information. Intuitively speaking, a given set of time-series data is usually consistent with less synchronous systems than asynchronous systems, thus it is more likely to revise an asynchronous system to satisfy certain reachabilty constraints.

The drawback of M2RIT is that there is a loss of information due to discretization (input time-series data must be discretized), while the CRAC makes full use of the original continuous data. Moreover, M2RIT does not guarantee the minimal revision of the logic program.

From the contents of this chapter, we propose several topics as possible future work:

- Developing heuristics to improve the performance of the existing algorithms
- Considering the metric for minimal revision and designing a related algorithm
- Reachability in the meaning of continuous models
- Adapting more dynamical properties other than reachability

Chapter 5

Tests and Benchmarks

After the presentation of the main theoretical contents, we are going to show in this chapter some results of the implementation:

- Comparison of PermReach, ASPReach which were introduced in Chapter 3 with several state-of-the-art model checkers: exhaustive reachability analyzers (Mole, NuSMV and ASP solver without optimization), Pint on their scalability (biggest tractable model size), efficiency (runtime) and precision (global conclusiveness)
- As there is no model revision technique based on reachability in the literature, we are going to show the performance of CRAC and M2RIT in Chapter 4 on random examples of different sizes.

In Chapter 3, we have illustrated the concepts of the new modeling framework ABAN and several related definitions which describe the reachability problem under this framework. Afterwards, we have shown the theoretical effectiveness and correctness of our reachability analyzers PermReach and ASPReach. To ensure their capacity, we use small models to verify the correctness, *i.e.* PermReach and ASPReach obtain the same result as other model checkers. Then we apply all the analyzers on a series of models with increasing sizes to obtain the limit of the capacity of each analyzer. These tests show our analyzers can be applied to models with more than 1000 automata while traditional ones fail at models with 50 automata.

As for CRAC and M2RIT, CRAC uses reachability properties and *continuous* time-series data while M2RIT uses reachability properties and *discretized* time-series data.

5.1 Comparison of Reachability Analyzers

In this section, we evaluate the reachability analyzers through tests on computing power, conclusiveness and results on random examples. The competitors are

- traditional model checkers Mole[1] and NuSMV[2], pure ASP solver [1]
- pure static analyzer Pint [56]
- our hybrid analyzers PermReach and ASPReach

All tests were run on an Intel Core i7-3770 CPU, 3.4GHz with 8GB RAM computer.

5.1.1 Performance on Computing Power

To evaluate the scalability in *in silico* networks, we take T-cell Receptor model (TCR) [71] and epidermal growth factor receptor model (EGFR) [72] as examples, with the former one containing 95 automata and 206 transitions and the latter one containing 104 automata and 389 transitions respectively.

These models are originally Boolean networks. According to the approach in Appendix A.1, BNs are transformed into ABANs. Here, we ran the same test as in [29]. In the TCR model, we take 3 automata as input (`cd4 cd28 tcrlig`), vary exhaustively their initial states combinations (2^3) and finally take the reachability of states of 5 automata (`sre ap1 nfkb nfat sigmab`) as output. Similarly we carried a bigger test on EGFR model with 13 automata as input and 12 automata as output. We first tested the performance of traditional model checkers and pure ASP approach which have the biggest theoretical complexities. For traditional model checkers, Mole turns out to be memory-out for 6 in 12 outputs, and all memory-out for NuSMV in model EGFR.

Ben-Abdallah *et al.* [1] have implemented reachability analyzer using pure ASP solver, showing it has a runtime of the same scale (See Appendix D). Thanks to the efficiency of Clingo[3], pure ASP solver begins to fail at 80 automata rather than 50 which is the limit of traditional model checkers. But this computational capacity is still not enough as the number of automata in systems biology is usually in the scale of 10^3 or bigger.

[1]`http://www.lsv.fr/~schwoon/tools/mole`
[2]`http://nusmv.fbk.eu`
[3]`http://potassco.sourceforge.net/`

Due to the big state space, traditional model checkers and pure ASP method are not applicable but they can be used to validate non-exhaustive approaches on small examples. The runtime results of Pint, PermReach and ASPReach are listed in Table 5.1 on page 89. Pint runs faster than ASPReach in TCR test but slower in EGFR test because there are inconclusive instances which cost more the runtime.

5.1.2 Performance on Conclusiveness

To validate our approaches, we carried tests on a small example, λ-phage model [78] to compare with an alternative reachability analyzer Pint [61] implementing solely an analysis using LCG [56, 29, 59]. λ-phage is originally a multivalued model. We compressed its multivalued variables to transform it into a Boolean model. In this model with 4 automata and 12 transitions (without taking consideration of the self-regulations), PermReach and ASPReach show complete conclusiveness while Pint cannot (Figure 5.1). Pint cannot decide whether cro_1 is reachable or not, because it does not consider the order in the state sequence even though there exists a solution of length 3: $cII_0 :: cI_0 :: cro_0$ corresponding to the trajectory $\{cI_1\} \rightarrow cII_0 :: \{cII_0\} \rightarrow cI_0 :: \{cII_0, cI_0\} \rightarrow cro_0$.

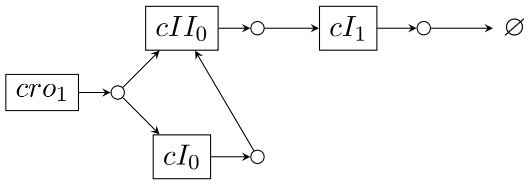


Figure 5.1: An SLCG of λ-phage model, automaton cI appears in both branches of the **AND gate**. Pint cannot decide the reachability of cro_0.

When dealing with more complex topology of SLCGs in term of branches, PermReach is not able to handle some of the special cases where multiple states of one automaton appear in different branches (Figure 5.2). Theses cases are solvable by ASPReach.

PermReach is not able to deal with the cases where multiple states of one automaton appear in the branches of one **AND gate** (d_0 and d_1 in this example). There exists a consistent state sequence: $d_1 :: b_1 :: c_1 :: a_1$ corresponding to the trajectory $\{c_0\} \rightarrow d_1 :: \{d_0, a_0\} \rightarrow b_1 :: \{d_1, e_0\} \rightarrow c_1 :: \{b_1, c_1\} \rightarrow a_1$ which could be found by ASPReach.

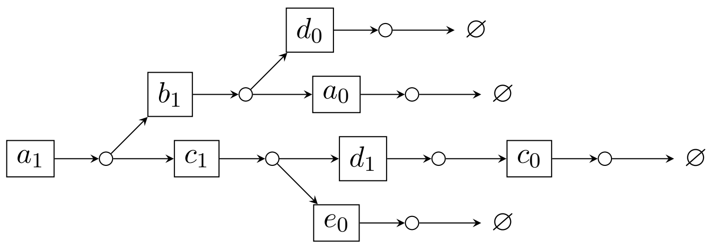


Figure 5.2: Counterexample of SLCG that cannot be solved by PermReach. The former counterexample shows PermReach is not fully inconclusive.

The whole approach is implemented in Python3[4]. The call of ASP in Python is done by package *pyasp*[5].

In the TCR tests, our approach gives exactly the same result as Pint did. As for EGFR tests, ASPReach returned no inconclusive output.

As seen in Table 5.1 on page 89, our approach can be more conclusive than Pint for ABANs. Model-checkers using global search are perfectly conclusive for all tests (including λ-phage model) and memory-out on latter two tests so they are not listed in the table. In the configuration of heuristics, we set a threshold for **OR gates**. If there are less than 10 **OR gates** after preprocessing, the computation will be shifted from heuristic to the enumeration of all combinations of **OR gates**. This is the case for these three benchmarks. The experiments show the ability of ASPReach is already more conclusive than Pint in “simple” cases.

5.1.3 Performance on Random Examples

Besides the tests on the examples coming from the literature, we have also carried tests on some randomly generated ABANs to check the generality and the runtime performance of PermReach. The ABANs are generated as follows:

Given the number of transitions, for every transition $tr = A \to a_h$ to be generated, its head a_h is randomly chosen from $\mathbf{LS}$, the first element of the body A_1 is randomly chosen from $\mathbf{LS}_1 = \mathbf{LS} \backslash \{a_h, a_{1-h}\}$. For $i > 1$, if A_{i-1} exists (suppose $A_{i-1} = b_x$), we generate A_i with an 80% probability, choosing randomly from $\mathbf{LS}_i = \mathbf{LS}_{i-1} \backslash \{b_x, b_{1-x}\}$.

One test is on the different numbers of automata with the same density

[4]Code and testing data available at `https://github.com/XinweiChai/reach_and_revision`

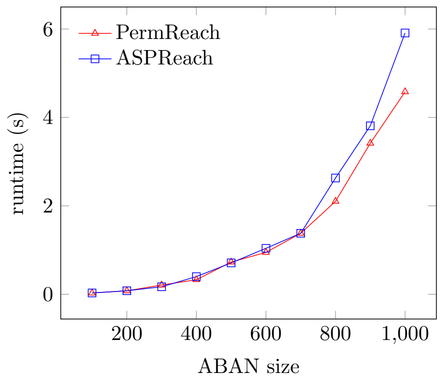
[5]`https://pypi.python.org/pypi/pyasp`

Model	λ-phage		
Inputs	4	Outputs	4
Total tests	$2^4 \times 4 = 64$		
Analyzer	Pint	**PermReach**	**ASPReach**
Reachable	36(56%)	38(59%)	
Unreachable	26(41%)		
Inconclusive	**2(3%)**	**0(0%)**	
Total time	< 1s		
Model	TCR		
Inputs	3	Outputs	5
Total tests	$2^3 \times 5 = 40$		
Analyzer	Pint	**PermReach**	**ASPReach**
Reachable	16(40%)		
Unreachable	24(60%)		
Inconclusive	**0(0%)**		
Total time	7s	0.85s	40s
Model	EGFR		
Inputs	13	Outputs	12
Total tests	$2^{13} \times 12 = 98,304$		
Analyzer	Pint	**PermReach**	**ASPReach**
Reachable	64,282(65.4%)	74,268(75.5%)	
Unreachable	24,036(24.5%)		
Inconclusive	**9,986(10.1%)**	**0(0%)**	
Total time	**9h50min**	**15min31s**	**3h46min**

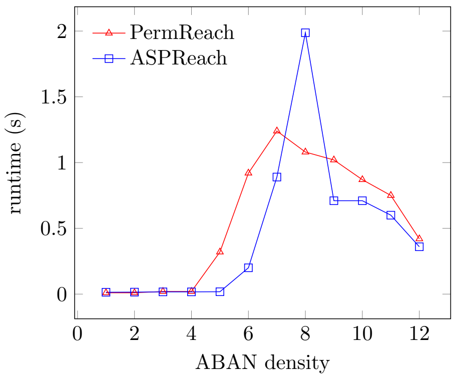
Table 5.1: Results of the tests on small (λ-phage) and large (TCR, EGFR) examples from literature. "Reachable", "Inconclusive" and "Unreachable" give respectively the number of different results of reachability, while "Total time" depict the maximum time of the individual computations.

(average number of transitions per automaton) of ABAN. Fixing the density to 3, we vary the number of automata from $10, 20, \dots, 100, 200, \dots, 1000$. In the instances with less than 300 automata, the runtime of each reachability check is less than 0.1s. Figure 5.3(a) and Figure 5.3(b) show the average runtime is less than 5 seconds even if there are 1000 automata. Moreover, the longest runtime among the test sets is less than 20s. Because we stop the computation if one reachability check takes more than 20s and we note it as timeout. We find no timeout case.

Another test is on different densities with the same number of automata. In Figure 5.3(b), we fixed $|\Sigma| = 20$ and vary the number of the transitions per automaton (density) from 1 to 12. The runtime peak is at density 8. A possible explanation is that even if the topology of the network is more complex with the growth of density, more available transitions lead to more pathways from the initial state to the target state, thus the heuristics may end with less trials.



(a) Runtime with fixing the density to 3



(b) Runtime with fixing the number of automata to 20

Figure 5.3: Average Runtime tests of PermReach and ASPReach on random generated ABANs

5.2 Implementation of CRAC and M2RIT

CRAC and M2RIT are algorithms for revising existing models using reachability information and sets of revisable candidate transitions, with the former adding and removing transitions and the latter revising existing transitions.[6]. We suppose that the sets come from time-series data.

5.2.1 CRAC

CRAC is based on the hypothesis that time-series data is consistent with differential equations. We test it by the following steps:

1. Generate random differential equations E
2. Generate time-series data tsd from E
3. Construct an ABAN $A = (\Sigma, T)$ consistent with E, compute its reachability information Re, Un using ASPReach

[6]Code and testing data available at `https://github.com/XinweiChai/reach_and_revision`

4. Obtain a partial ABAN $A' = (\Sigma, T')$ with $T' \subset T$, compute its reachability information Re' and Un' using ASPReach

5. Infer candidate regulations R from E

6. Construct an ABAN $A'' = (\Sigma, T')$ based on A', Re, Un, Re', Un' and R s.t. A'' satisfies Re, Un using CRAC

In step 1, given the set of variables Σ, every differential equation associated to $x_v \in \Sigma$ is $\Delta x_v = \sum_{u \in \Sigma'} x_u k_{uv}$. To generate such equation, we generate first $\Sigma' \subseteq \Sigma \backslash \{v\}$ randomly. The first element of Σ' is randomly chosen from $\Sigma \backslash \{v\}$. For $i > 1$, if $(i-1)$-th element of Σ' exists, we generate the i-th element with an 80% probability, choosing from $\Sigma \setminus (\Sigma' \cup \{v\})$. For parameter k_{uv}, we choose a random number in $[-1, -0.5] \cup [0.5, 1]$.

In step 2, we generate a possible time-series. To make it consistent with differential equations (simultaneous change), we choose the next state from synchronous successors. Every state change can be regarded as a composition of several asynchronous state changes without considering the orders. One of the drawbacks is that time is not taken into account.

We then hide randomly a part of transitions (20% of all the transitions) to obtain T'.

From the running result, we discovered that CRAC is not able to retrieve all the transitions in T but can satisfy the reachability properties provided by A. We noticed that the added transitions can be similar to the hidden transitions but with different logic operators, *e.g.* $\{b_1, c_1\} \to a_1$ could be replaced by $\{b_1\} \to a_1$ and $\{c_1\} \to a_1$.

The performance of CRAC also depends on the discretization which can be a possible future topic of studies.

5.2.2 M2RIT

The test of M2RIT is analogous:

1. Generate random time-series data *tsd*

2. Obtain partial time-series data $tsd' \subset tsd$

3. Infer ABAN $A = (\Sigma, T)$ from *tsd* using asynchronous LFIT, compute its reachability information Re and Un using ASPReach

4. Infer ABAN $A' = (\Sigma, T')$ from tsd' using asynchronous LFIT, compute its reachability information Re' and Un' using ASPReach

5. Construct an ABAN $A'' = (\Sigma, T')$ based on A', Re, Un, Re', Un' s.t. A'' satisfies Re, Un using M2RIT

We generate a random ABAN like in Section 5.1. We then choose the next state from asynchronous successors are generated by an equi-probable next state. This operation allows us to obtain a time-series data matching perfectly asynchronous update scheme as there is at most one state change for all automata at each time point. Hence, such time-series data fit perfectly asynchronous LFIT algorithm. However, they do not exist in real world as we cannot limit the number of state changes at each observation.

$$A = \begin{array}{c} t \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} \mathbf{0} & \mathbf{1} & 1 & 1 \\ \mathbf{0} & \mathbf{1} & 0 & 1 \\ \mathbf{1} & \mathbf{0} & 1 & 1 \\ \mathbf{0} & \mathbf{0} & 0 & 1 \end{bmatrix} \end{array} \qquad A' = \begin{array}{c} t \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} \mathbf{0} & \mathbf{1} & 1 & 1 \\ 0 & \mathbf{0} & \mathbf{1} & 1 \\ 1 & 1 & \mathbf{1} & \mathbf{0} \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

The left matrix A is an example of generalized time series data, where at each time point multiple variables can change their values. The right matrix A' is an example of "asynchronous" time-series data, where at each time point at most one variable changes its value. From $t = 0$ to $t = 1$, A shows a direct transition $(0, 0, 1, 0) \rightarrow (1, 1, 0, 0)$ where A' interprets this transition in details (from $t = 0$ to $t = 3$) under the hypothesis that two variables cannot change their values exactly at the same time.

Both can be used as input for CRAC but only the latter one is proper for M2RIT to fit asynchronous LFIT algorithm.

Like in the last section, we hide 20% of the transitions and offer a set of reachability information.

From the running result, we discover that M2RIT is not able to retrieve all the transitions in T but can construct a new ABAN A'' which simulates A in the meaning of reachability. This behavior resembles that of LFIT algorithm, as it aims at obtaining a model that reproduces the time series data without considering if the model is identical to the real one.

Limited by the computing capacity of ASPReach, CRAC and M2RIT can deal with models with up to 1000 variables.

5.3 Résumé

In this chapter, we have shown the performance of the practical implementation of the algorithm introduced in this thesis. Different results approved the properties of different methodologies shown in the previous chapters.

PermReach and ASPReach show that they are more conclusive than Pint, the former work of our laboratory according to conclusiveness tests. They are also more efficient than traditional reachability analyzers like Mole and NuSMV according to computing power tests. Between PermReach and of ASPReach, there is a trade-off between conclusiveness and efficiency: PermReach is more efficient than ASPReach but with less conclusiveness.

The tests of CRAC and M2RIT show that they can be applied to revise models with up to 1000 variables if proper and sufficient reachability information is given. Unfortunately, there is no existing work that uses reachability properties to revise models, we cannot run a competition between CRAC and M2RIT and the state-of-the-art methods.

Chapter 6

Conclusion and Outlooks

We will recap in this chapter what have been discussed during this thesis and propose some possible future work. We answered the two questions in Section 1.2, how to analyze the reachability efficiently and precisely (Chapter 3) and how to build a model by given time series data and reachability information (Chapter 4).

With the increasing amount of biological data, the needs of analyzing, extracting knowledge and predicting system behaviors based on these data is becoming crucial. Modeling is one of the ways to answer all these needs by collecting, classifying, analyzing the common features of the data and make reasonable prediction.

The application of modeling in biological engineering can be diverse. By analyzing the mechanics of a cell or a bacterium, one may locate the gene to be knocked-off or knocked-in in order to let the system perform his desired system behaviors; by analyzing the interaction between human body and medicine in molecular scale, one may ameliorate existing targeted therapies or design new ones; in pharmaceutics, a thorough understanding of bio-chemical reactions allows one to design new medicine or new synthesis processes.

As also mentioned in the introduction chapter, robotic models are also of importance. For a non black-box system, model checking is helpful to controlling the system *via* setting specific initial states, ensuring system safety and robustness.

In robotics, one prefers synchronous models as every move of a robot is programmed, the system parameters are usually accessible. However modelings in systems biology are different. Since the inner mechanics of biological systems are usually unknown, or partially known and biological systems have intrinsic non-determinism (*e.g.* cell differentiation), these two facts suggest us to consider a non-deterministic model. As discussed in Section 2.2, asynchronicity can nearly impose non-determinism. Moreover, to avoid the solution of differential equations and tolerate some noise, discretized models are preferred. In all, *asynchronous modelings* are used in the most of this thesis.

6.1 Contributions

Based on the background of systems biology, our goal is to *enrich and correct* the existing models with additional knowledge. To do so, we have to first develop efficient model checkers to verify whether the model is consistent with qualitative properties such as reachability (Chapter 3).

In Section 2.3, we stated that exact model checkers provide conclusive results of dynamic properties but the computational cost is unaffordable while the computational cost of abstract model checkers are acceptable but the results are not necessarily conclusive.

With the previous work on the Asynchronous Automata Network (AAN) by Paulevé *et al.* [29], we squeeze the application domain of AAN and simplified its semantics, called ABAN (Asynchronous Binary Automata Network). This semantic change is to give a possibility of making related reachability analyzer more conclusive.

Our research focuses on a static analysis called *over-approximation* of the system dynamics as this abstract method relies on only topological information of the model instead of simulation. We also proposed Simplified Local Causality Graph (SLCG) to visualize the static analysis on ABANs. The computation is efficient but still inconclusive.

However, the result can be used in the refined analysis if it is not conclusive. During the theoretical analysis of the reason of inconclusiveness of the over-approximation, we developed two model checkers to deal with the key components impeding conclusiveness. They are based on heuristic methods and pure static analysis:

- PermReach (Section 3.4.1)

  which performs a *limited search of permutations* on conjunctive nodes in the SLCG as the existence of conjunctive node is one of the factor of inconclusiveness. Since PermReach does not take the nodes appearance orders into account, the analysis remains theoretically inconclusive.

- ASPReach (Section 3.4.2)

  which performs a *global search of possible nodes* order in the SLCG with the help of Answer Set Programming (ASP). ASPReach covers the weak-point of PermReach but is still inconclusive due to the heuristic preprocessing which simplifies the problem but creates inequivalence.

PermReach and ASPReach perform normally on models with 1000 components while traditional model checkers fail to compute and static analyzer Pint also fails to give conclusive results on certain instances (Section 5.1). Moreover, ASPReach is more conclusive than PermReach but need extra runtime.

As a tentative, we tried to extend the application of PermReach and ASPReach to multi-valued models (restricted AAN). However, this extension requires a presumption which is not realistic enough in biological systems but remains to be an interesting heuristics when the requirement is to find only one reachable trajectory instead of a global solution (Section 3.5).

Next, with the unsatisfied properties detected by our model checkers, we need a way to correct the model in order to make the system consistent with all the wanted properties (Chapter 4).

Finally, we provide with two model learning/revision techniques based on reachability analysis, covering the incapability of taking background knowledge into account.

- CRAC (Completion *via* Reachability And Correlations)

  first applies ordinal differential equation model to generate regulation candidates. Then it tries to satisfy all the reachability requirements by adding/deleting transitions in the model according to the regulation candidates.

- M2RIT (Model Revision *via* Reachability and Interpretation Transitions)

  is a strengthening to the capability of LFIT (Learning From Interpretation Transitions) framework to the learning of Boolean asynchronous systems in the form of logic programs. Unlike CRAC, M2RIT uses the reproducibility as a constraint. When changing the transitions in the model, the model has to always be able to reproduce the original time-series data.

As far as we know, the revision of dynamic model based on reachability properties has never been considered in the literature. To our knowledge, this field has been explored only by Yamamoto *et al.* [85], who have studied the completion of static models limited by *static* data.

CRAC and M2RIT are able to deal with models of 1000 variables thanks to the high performance of ASPReach. However they are not theoretical conclusive as they are built on heuristics, possessing risks of failure.

Moreover, even though M2RIT can precisely reproduce all the provided time-series data, it is sensitive to noise, inheriting the fragility of LFIT.

After all, to enhance learning methods, we need a lot of studies to prove the causality instead of correlation/consistency by fitting the data.

To sum up the contents in this thesis, we have:

1. studied different modeling frameworks and semantics
2. investigated the weakness of existing model checkers and model learning methods
3. developed reachability analyzers PermReach and ASPReach
4. developed model revisors CRAC and M2RIT

6.2 Future Work

- From the analysis of the drawbacks of PermReach and ASPReach, we assert that the way towards high efficiency and precision is probably hybrid analysis where the needs of precision and effectiveness meet.

  We propose to use more accurate heuristics (for example at the choice on **OR gates** of an SLCG) instead of random choice so as to access branches with higher possibility of reachability.

- We studied only Boolean-related models in order to use a stronger conclusiveness of SLCG. It is however possible to generalize the study to multi-valued systems.

  Normal Logic Program (NLP) might be a proper modeling because it ignores the state of departure while different pathways of a variable might be a cause of inconclusiveness. For example, transition $a_1 \to b_0 \Rsh b_2$ has multiple pathways for automaton b: $b_0 \to b_2, b_0 \to b_3 \to b_2$, *etc.* When this transition is translated to $b_2 \leftarrow a_1$, it does not require b_0 as a state of departure which does not cause such problems.

- We are now considering the incorporation of parametrization space abstraction, like the work of [44], to improve the performance of our model checkers/revisors regarding versatility.

  The study of parametrization space allows us to obtain different properties of a set of models rather than only squeezing consistent transitions only by reachability properties. Also, parametrization space methods is still restrictive: they can explore the full dynamics up to dozens of variables. New model checkers with versatility and less time or memory cost may be interesting.

- M2RIT does not guarantee the minimal revision of the logic program.

  Considering the metric for minimal revision and designing a related algorithm will be interesting. The definition of "minimal" can be of different means: allowing changing minimal number of transitions, allowing modifying minimal number of elements of each transition or allowing least inconsistency, *etc.* Also, when revising the models, there might be some common parts in the SLCGs of inconsistent reachability properties. Making use of these common parts might be useful to reaching the goal of minimal revision.

Bibliography

[1] Emna Ben Abdallah, Maxime Folschette, Olivier Roux, and Morgan Magnin. Exhaustive analysis of dynamical properties of biological regulatory networks with answer set programming. In 2015 IEEE International Conference on Bioinformatics and Biomedicine (BIBM), pages 281–285. IEEE, 2015.

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

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

[4] Kunihiro Baba, Ritei Shibata, and Masaaki Sibuya. Partial correlation and conditional correlation as measures of conditional independence. Australian & New Zealand Journal of Statistics, 46(4):657–664, 2004.

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

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

[7] Richard Bonneau, David J Reiss, Paul Shannon, Marc Facciotti, Leroy Hood, Nitin S Baliga, and Vesteinn Thorsson. The Inferelator: an algorithm for learning parsimonious regulatory networks from systems-biology data sets de novo. Genome biology, 7(5):R36, 2006.

[8] Connie M Borror. Practical nonparametric statistics. Journal of Quality Technology, 33(2):260, 2001.

[9] Robert K Brayton, Gary D Hachtel, Alberto Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen Edwards, Sunil Khatri, Yuji Kukimoto, Abelardo Pardo, et al. VIS: A system for verification and synthesis. In International conference on computer aided verification, pages 428–432. Springer, 1996.

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

[11] Xinwei Chai, Morgan Magnin, and Olivier Roux. A heuristic for reachability problem in asynchronous binary automata networks, 2018. arXiv:1804.07543v1.

[12] Xinwei Chai, Tony Ribeiro, Morgan Magnin, Olivier Roux, and Katsumi Inoue. Static analysis and stochastic search for reachability problem. In 9th Static Analysis in Systems Biology, affiliated with Static Analysis Symposium, 2018. In press.

[13] Xinwei Chai, Tony Ribeiro, Morgan Magnin, Olivier Roux, and Katsumi Inoue. Using reachability properties of logic program for revising biological models. Work in Progress of International Conference on Inductive Logic Programming, 2018.

[14] Thomas Chatain, Stefan Haar, and Loïc Paulevé. Boolean networks: Beyond generalized asynchronicity. In International Workshop on Cellular Automata and Discrete Complex Systems, pages 29–42. Springer, 2018.

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

[16] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: a new symbolic model checker. International Journal on Software Tools for Technology Transfer, 2(4):410–425, 2000.

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

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

[19] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Workshop on Logic of Programs, pages 52–71. Springer, 1981.

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

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

[22] Alberto De La Fuente, Nan Bing, Ina Hoeschele, and Pedro Mendes. Discovery of meaningful associations in genomic data using partial correlation coefficients. Bioinformatics, 20(18):3565–3574, 2004.

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

[24] Elena Stanimirova Dimitrova. Polynomial models for systems biology: Data discretization and term order effect on dynamics. PhD thesis, Virginia Tech, 2006.

[25] Elena Dubrova and Maxim Teslenko. A SAT-based algorithm for finding attractors in synchronous Boolean networks. IEEE/ACM transactions on computational biology and bioinformatics, 8(5):1393–1399, 2011.

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

[27] Ansgar Fehnker, Ralf Huuck, Patrick Jayet, Michel Lussenburg, and Felix Rauch. Goannaa static model checker. In International Workshop on Formal Methods for Industrial Critical Systems, pages 297–300. Springer, 2006.

[28] Jakob Foerster, Ioannis Alexandros Assael, Nando de Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. In Advances in Neural Information Processing Systems, pages 2137–2145, 2016.

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

[30] Nir Friedman, Michal Linial, Iftach Nachman, and Dana Pe'er. Using Bayesian networks to analyze expression data. Journal of computational biology, 7(3-4):601–620, 2000.

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

[32] Leon Glass. Classification of biological networks by their qualitative dynamics. Journal of Theoretical Biology, 54(1):85–107, 1975.

[33] Leon Glass and Stuart A Kauffman. The logical analysis of continuous, non-linear biochemical control networks. Journal of theoretical Biology, 39(1):103–129, 1973.

[34] RH Hardin, RP Kurshan, SK Shukla, and MY Vardi. A new heuristic for bad cycle detection using BDDs. In International Conference on Computer Aided Verification, pages 268–278. Springer, 1997.

[35] David Harel, Orna Kupferman, and Moshe Y Vardi. On the complexity of verifying concurrent transition systems. Information and Computation, 173(2):143–161, 2002.

[36] Inman Harvey and Terry Bossomaier. Time out of joint: Attractors in asynchronous random Boolean networks. In Proceedings of the Fourth European Conference on Artificial Life, pages 67–75. MIT Press, Cambridge, 1997.

[37] Jan Hauke and Tomasz Kossowski. Comparison of values of Pearson's and Spearman's correlation coefficients on the same sets of data. Quaestiones geographicae, 30(2):87–93, 2011.

[38] Gerard J. Holzmann. The model checker SPIN. IEEE Transactions on software engineering, 23(5):279–295, 1997.

[39] John P Huelsenbeck and Fredrik Ronquist. MRBAYES: Bayesian inference of phylogenetic trees. Bioinformatics, 17(8):754–755, 2001.

[40] Clark L Hull. The correlation coefficient and its prognostic significance. The Journal of Educational Research, 15(5):327–338, 1927.

[41] Katsumi Inoue. Logic programming for boolean networks. In IJCAI proceedings-international joint conference on artificial intelligence, volume 22, page 924, 2011.

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

[43] Zohra Khalis, Jean-Paul Comet, Adrien Richard, and Gilles Bernot. The SMBioNet method for discovering models of gene regulatory networks. Genes, genomes and genomics, 3(1):15–22, 2009.

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

[45] Bret Larget and Donald L Simon. Markov chain monte carlo algorithms for the bayesian analysis of phylogenetic trees. Molecular biology and evolution, 16(6):750–759, 1999.

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

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

[48] Patrick Lincoln and Ashish Tiwari. Symbolic systems biology: Hybrid modeling and analysis of biological networks. In International Workshop on Hybrid Systems: Computation and Control, pages 660–672. Springer, 2004.

[49] David Martínez Martínez, Tony Ribeiro, Katsumi Inoue, Guillem Alenyà Ribas, and Carme 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.

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

[51] Kenneth L McMillan. Symbolic model checking. In Symbolic Model Checking, pages 25–60. Springer, 1993.

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

[53] Duy Nguyen-Tuong and Jan Peters. Model learning for robot control: a survey. Cognitive processing, 12(4):319–340, 2011.

[54] Rainer Opgen-Rhein and Korbinian Strimmer. From correlation to causation networks: a simple approximate learning algorithm and its application to high-dimensional plant gene expression data. BMC systems biology, 1(1):37, 2007.

[55] Loïc Paulevé. Pint: a static analyzer for transient dynamics of qualitative networks with IPython interface. In CMSB 2017 - 15th conference on Computational Methods for Systems Biology, volume 10545 of Lecture Notes in Computer Science, pages 309–316. Springer International Publishing, 2017.

[56] Loïc Paulevé. Reduction of qualitative models of biological networks for transient dynamics analysis. IEEE/ACM Transactions on Computational Biology and Bioinformatics, 15(4):1167–1179, 2018.

[57] Loïc Paulevé, Geoffroy Andrieux, and Heinz Koeppl. Under-approximating cut sets for reachability in large scale automata networks. In International Conference on Computer Aided Verification, pages 69–84. Springer, 2013.

[58] Loïc Paulevé, Courtney Chancellor, Maxime Folschette, Morgan Magnin, and Olivier Roux. Logical Modeling of Biological Systems, chapter Analyzing Large Network Dynamics with Process Hitting, pages 125 – 166. Wiley, 2014.

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

[60] Loïc Paulevé, Morgan Magnin, and Olivier Roux. From the Process Hitting to Petri Nets and Back. Technical Report hal-00744807, ETH Zürich, October 2012.

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

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

[63] John W Pinney, David R Westhead, and Glenn A McConkey. Petri net representations in systems biology. Biochemical Society Transactions, 31(6):1513–1515, 2003.

[64] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In International Symposium on programming, pages 337–351. Springer, 1982.

[65] Tony Ribeiro, Maxime Folschette, Morgan Magnin, Olivier Roux, and Katsumi Inoue. Learning dynamics with synchronous, asynchronous and general semantics. In the 28th International Conference on Inductive Logic Programming, 2018.

[66] Tony Ribeiro and Katsumi Inoue. Learning prime implicant conditions from interpretation transition. In Inductive Logic Programming, pages 108–125. Springer, 2015.

[67] Tony Ribeiro, Sophie Tourret, Maxime Folschette, Morgan Magnin, Domenico Borzacchiello, Francisco Chinesta, Olivier Roux, and Katsumi Inoue. Inductive learning from state transitions over continuous domains. In International Conference on Inductive Logic Programming, pages 124–139. Springer, 2017.

[68] Alexandre Rocca, Nicolas Mobilia, Éric Fanchon, Tony Ribeiro, Laurent Trilling, and Katsumi Inoue. ASP for construction and validation of regulatory biological networks. Logical Modeling of Biological Systems, pages 167–206, 2014.

[69] Christophe Rodrigues, Pierre Gérard, Céline Rouveirol, and Henry Soldano. Active learning of relational action models. In International Conference on Inductive Logic Programming, pages 302–316. Springer, 2011.

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

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

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

[73] Thomas Schaffter, Daniel Marbach, and Dario Floreano. GeneNetWeaver: in silico benchmark generation and performance profiling of network inference methods. Bioinformatics, 27(16):2263–2270, 2011.

[74] Kazuo Shinozaki, Kazuko Yamaguchi-Shinozaki, and Motoaki Seki. Regulatory network of gene expression in the drought and cold stress responses. Current opinion in plant biology, 6(5):410–417, 2003.

[75] Rajat Singhania, R Michael Sramkoski, James W Jacobberger, and John J Tyson. A hybrid model of mammalian cell cycle regulation. PLoS computational biology, 7(2):e1001077, 2011.

[76] El Houssine Snoussi. Qualitative dynamics of piecewise-linear differential equations: a discrete mapping approach. Dynamics and stability of Systems, 4(3-4):565–583, 1989.

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

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

[79] René Thomas. Logical analysis of systems comprising feedback loops. Journal of Theoretical Biology, 73(4):631–656, 1978.

[80] René Thomas and Richard d'Ari. Biological feedback. CRC press, 1990.

[81] George Von Dassow, Eli Meir, Edwin M Munro, and Garrett M Odell. The segment polarity network is a robust developmental module. Nature, 406(6792):188, 2000.

[82] Kostyantyn Vorobyov and Padmanabhan Krishnan. Comparing model checking and static program analysis: A case study in error detection approaches. proceedings of 5th International Conference on Systems Software Verification, Vancouver, Canada, pages 1–7, 2010.

[83] Qinsi Wang, Paolo Zuliani, Soonho Kong, Sicun Gao, and Edmund M Clarke. SReach: A probabilistic bounded delta-reachability analyzer for stochastic hybrid systems. In International Conference on Computational Methods in Systems Biology, pages 15–27. Springer, 2015.

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

[85] Yoshitaka Yamamoto, Adrien Rougny, Hidetomo Nabeshima, Katsumi Inoue, Hisao Moriya, Christine Froidevaux, and Koji Iwanuma. Completing SBGN-AF networks by logic-based hypothesis finding. In International Conference on Formal Methods in Macro-Biology, pages 165–179. Springer, 2014.

Appendix A

Representation of Different Models

A.1 Transformation from BNs to ABANs

Given Boolean functions $v_i(t+1) = f_i(\mathbf{V}_i)$, with $\mathbf{V}_i$ the set of participating variables among $v_1(t), \cdots, v_n(t)$. Boolean functions could be transformed to equivalent CNF (conjunctive normal form) and DNF (disjunctive normal form) if the length of Boolean functions is limited to $O(1)$ [52] which is often the case.

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

Example A.1. Let $G_B = (V, F)$ a BN with $V = \{a, b, c, d, e\}$, and has only one Boolean function, $F = \{f(a) = (b \vee c) \wedge (d \vee e)\}$, we have $f(a) = (b \wedge d) \vee (b \wedge e) \vee (c \wedge d) \vee (c \wedge e)$, and $\neg f(a) = (\neg b \wedge \neg c) \vee (\neg d \wedge \neg e)$. The equivalent ABAN is then constructed: 5 automata $\mathbf{\Sigma} = \{a, b, c, d, e\}$, with transitions: $\mathbf{T} = \{\{b_1, d_1\} \rightarrow a_1,\ \{b_1, e_1\} \rightarrow a_1,\ \{c_1, d_1\} \rightarrow a_1,\ \{c_1, e_1\} \rightarrow a_1,\ \{b_0, c_0\} \rightarrow a_0,\ \{d_0, e_0\} \rightarrow a_0\}$.

Appendix B

Algorithms

Algorithm 2 Synchronous LFIT

```
  **Input**: a set of annotated atoms $\mathcal{B}$ and a set of state transitions $E$
  **Output**: an NLP $P$
  Initialization: $P := \{var^{val} \leftarrow \varnothing \mid var^{val} \in \mathcal{B}\}$
  **while** $E \neq \varnothing$ **do**
      Pick $(I, J) \in E, E := E \backslash \{(I, J)\}$
      **for** $A \in J$ **do**
          // Create all the local state transitions
          $R_A^I := (A \leftarrow \bigwedge_{B_i \in I} B_i \wedge \bigwedge_{C_j \in (\mathcal{B} \backslash I)} \neg C_j)$
          // Delete the rules in conflict
          $P :=$ **Specialize**$(P, R_A^I)$
  **return** $P$
```

Algorithm 3 Specialize in synchronous LFIT algorithm

```
Input: an NLP $P$ and a rule $R$
Output: the maximal specialization of $P$ that does not subsume $R$
Initialization: $conflicts := \varnothing$
// Search rules that need to be specialized
for $R_P \in P$ do
    if $b(R_P) \subseteq b(R)$ then
        $conflicts := conflicts \cup \{R_P\}$
        $P := P \setminus \{R_P\}$
// Revise the rules by least specialization
for $R_c \in conflicts$ do
    for $l \in b(R)$ do
        if $l \notin b(R_c)$ and $\neg l \notin b(R_c)$ then
            $R'_c := (h(R_c) \leftarrow (b(R_c) \cup \{\neg l\}))$
            if $P$ does not subsume $R_c$ then
                $P := P \setminus$ all the rules subsumed by $R'_c$
                $P := P \cup \{R'_c\}$
return $P$
```

Algorithm 4 Construction of SLCG (over-approximation)

```
Input: an ABAN $\mathbb{A} = (\Sigma, T)$, an initial state $\alpha$, a target state $\omega$
Output: SLCG $l = (V_{\text{state}}, V_{\text{sol}}, E)$
Initialization: $Ls \leftarrow \{\omega\}$, $V_{\text{state}} \leftarrow \varnothing$, $V_{\text{sol}} \leftarrow \varnothing$, $E \leftarrow \varnothing$
while $Ls \neq \varnothing$ do
    $Ls = Ls \setminus V_{\text{state}}$
    for $a_i \in Ls$ do
        $Ls \leftarrow Ls \setminus \{a_i\}$
        if $a_i \in \alpha$ then
            $E \leftarrow E \cup \{(a_i, \varnothing)\}$
        else
            // Choose the transitions reaching $a_i$
            for $sol = A \rightarrow a_i \in \mathbf{T}$ do
                $V_{\text{sol}} \leftarrow V_{\text{sol}} \cup \{sol\}$
                $E \leftarrow E \cup \{(a_i, sol)\}$
                $V_{\text{state}} \leftarrow V_{\text{state}} \cup A$
                for $b_j \in A$ do
                    $E \leftarrow E \cup \{(sol, b_j)\}$
                $Ls \leftarrow Ls \cup A$
                $V_{\text{state}} \leftarrow V_{\text{state}} \cup Ls$
            $V_{\text{sol}} \leftarrow V_{\text{sol}} \cup \{a_i.next\}$
return $(V_{\text{state}}, V_{\text{sol}}, E)$
```

Algorithm 5 Construction of SLCG (under-approximation)

```
Input: an ABAN $\mathbb{A} = (\Sigma, T)$, an initial state $\alpha$, a target state $\omega$
Output: SLCG $l = (V_{\text{state}}, V_{\text{sol}}, E)$
Initialization: $Ls \leftarrow \{\omega\}$, $V_{\text{state}} \leftarrow \varnothing$, $V_{\text{sol}} \leftarrow \varnothing$, $E \leftarrow \varnothing$, $rev \leftarrow \varnothing$
while $Ls \neq \varnothing$ do
    $Ls = Ls \setminus V_{\text{state}}$
    for $a_i \in Ls$ do
        $Ls \leftarrow Ls \backslash \{a_i\}$
        if $a_i \in \alpha$ then
            $E \leftarrow E \cup \{(a_i, \varnothing)\}$
            // Revise the reachability of initial state
            $rev \leftarrow rev \cup \{a_i\}$
        else
            // Check if local initial state $a_{1-i}$ needs to be revised
            if $a_{1-i} \in rev$ then
                $Ls \leftarrow Ls \cup \{a_{1-i}\}$
                $\alpha \leftarrow \alpha \backslash \{a_{1-i}\}$
            // Choose the transitions reaching $a_i$
            for $sol = A \rightarrow a_i \in \mathbf{T}$ do
                $V_{\text{sol}} \leftarrow V_{\text{sol}} \cup \{sol\}$
                $E \leftarrow E \cup \{(a_i, sol)\}$
                $V_{\text{state}} \leftarrow V_{\text{state}} \cup A$
                for $b_j \in A$ do
                    $E \leftarrow E \cup \{(sol, b_j)\}$
                $Ls \leftarrow Ls \cup A$
                $V_{\text{state}} \leftarrow V_{\text{state}} \cup Ls$
            $V_{\text{sol}} \leftarrow V_{\text{sol}} \cup \{a_i.next\}$
return $(V_{\text{state}}, V_{\text{sol}}, E)$
```

Algorithm 6 Pseudo-reachability $reach'$

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

Algorithm 7 PermReach

```
Input: an SLCG $l = (V_{\text{state}}, V_{\text{sol}}, E)$, an initial state $\alpha$, a target state $\omega$,
an integer $k$
Output: reachability $r(\alpha, \omega)$
// 1) Try to break cycles
for each $scc = (V'_{\text{state}}, V'_{\text{sol}}) \in \text{SCC}(l)$ with $V'_{\text{state}} \subseteq V_{\text{state}}, V'_{\text{sol}} \subseteq V_{\text{sol}}$ do
    if $scc$ has less than one incoming edge then
        for each $v \in V'_{\text{state}}$ do
            if $\exists (v, v') \in E, v' \in (V_{\text{sol}} \setminus V'_{\text{sol}})$ then
                $E \leftarrow E \backslash \{(v, v'') | v'' \in V'_{\text{sol}}, (v, v'') \in E\}$
// 2) Remove useless nodes/edges
pruned = true
while pruned do
    pruned = false
    for $v \in V_{\text{state}}$ do
        if $\nexists (v, v') \in E$ then
            $V_{\text{state}} \leftarrow V_{\text{state}} \backslash \{v\}$; $E \leftarrow E \backslash \{(v'', v) \in E\}$
            $E \leftarrow E \backslash \{(v'', sol) \in E | sol \in \{sol = (A \to a) \in V_{\text{sol}} | v \in A\}\}$
            $V_{\text{sol}} \leftarrow V_{\text{sol}} \backslash \{sol = (A \to a) \in V_{\text{sol}} | v \in A\}$
            pruned = true
// 3) Check pseudo-reachability
if PSEUDOREACH($l$) = False then
    return False
```

Algorithm 8 PermReach (continued)

```
// 4) main search loop
for each $i$ in $1 \ldots k$ do
    $l' = (V'_{\text{state}}, V'_{\text{sol}}, E') \leftarrow (V_{\text{state}}, V_{\text{sol}}, E)$
    for $v \in V'_{state}$ do // Treat each OR gates
        pick a random element $(v, v') \in E'$
        $E' \leftarrow E' \backslash \{(v, v'') \in E' | v'' \neq v'\}$
    if $l'$ contains cycles then
        continue
    Obtain simple AND gates $simp$ from $l'$
    while $simp \neq \varnothing$ do
        $re \leftarrow$ False
        // check the reachability of $simp$, if true, update initial state
        for $i \in perm(simp)$ do
            if $REACH(i) =$ True then
                $re \leftarrow$ True
                break
        if $re =$ True then
            update $\alpha$ by firing transitions in $simp$
            $V'_{\text{sol}} \leftarrow V'_{\text{sol}} \setminus simp$
        else
            break
        Obtain simple AND gates $simp$ from $l'$
    if $re =$ True then
        return True
return Inconclusive
```

Algorithm 9 ASPReach

```
Input: SLCG $l = (V_{\text{state}}, V_{\text{sol}}, E)$, an integer $k$
Output: reachability $r$ and a trajectory $t$
Compute SCCs, classify them into SCC1($l$) with at most 1 incoming edge
and SCC2($l$) otherwise
// 1) Break all cycles and prune useless branches
for each $(V'_{\text{state}} \subseteq V_{\text{state}}, V'_{\text{sol}} \subseteq V_{\text{sol}}) \in \text{SCC1}(l)$ do
    for each $v \in V'_{\text{state}}$ do
        if $\exists (v, v') \in E, v' \in (V_{\text{sol}} \setminus V'_{\text{sol}})$ then
            $E \leftarrow E \backslash \{(v, v'') | v'' \in V'_{\text{sol}}, (v, v'') \in E\}$
// 2) remove useless nodes/edges
pruned = True
while pruned do
    pruned = False
    for $v \in V_{\text{state}}$ do
        if $\nexists (v, v') \in E$ then
            $V_{\text{state}} \leftarrow V_{\text{state}} \backslash \{v\}$; $E \leftarrow E \backslash \{(v'', v) \in E\}$
            $E \leftarrow E \backslash \{(v'', sol) \in E | sol \in \{sol = (A \to a) \in V_{\text{sol}} | v \in A\}\}$
            $V_{\text{sol}} \leftarrow V_{\text{sol}} \backslash \{sol = (A \to a) \in V_{\text{sol}} | v \in A\}$
            pruned = True
// 3) Check pseudo-reachability
if $pseudoReach(l)$ = False then
    return (False, $\varnothing$)
// 4) main search loop
for each $i$ in $1 \dots k$ do
    $l' = (V'_{\text{state}}, V'_{\text{sol}}, E') \leftarrow (V_{\text{state}}, V_{\text{sol}}, E)$
    for $v \in V'_{state}$ do // Treat each OR gates
        pick a random element $(v, v') \in E'$
        $E' \leftarrow E' \backslash \{(v, v'') \in E' | v'' \neq v'\}$ with $\nexists i \in \text{SCC2}(l)$ and $i \in E'$
    $(r, t) \leftarrow$ ASPsolve($l'$)
    if $r$ = True then
        return (True, $t$)
return (Inconclusive, $\varnothing$)
```

Algorithm 10 Completion by over-approximation

```
Input: an incomplete ABAN $\mathbb{A} = (\Sigma, T)$, a set of candidate regulations $R$, targeted reachability $(\alpha, \omega)$
Output: Completed set $CS$
Initialization: $rev \leftarrow \{(\{\omega\}, \{\omega\}, \varnothing)\}$, $CS \leftarrow \varnothing$
Construct SLCG $l = SLCG(\alpha, \omega) = (V_{\text{state}}, V_{\text{sol}}, E)$ by Algorithm 4
if $reach(\alpha, \omega)$ then
    return $(\textbf{True}, \varnothing)$
while $rev \neq \varnothing$ do
    // $ls$: traversed local states, $fr$: local states to be revised in the next step, $tr$: transitions to be added
    for $i = (ls, fr, tr) \in rev$ do
        $rev \leftarrow rev \backslash \{i\}$
        if $fr = \varnothing$ then
            $CS \leftarrow CS \cup \{ls\}$
            continue
        for $j \in fr$ do
            for $k \in j.next$ do
                if $head(k) \not\subseteq ls$ then
                    $rev \leftarrow rev \cup \{(ls \cup head(k), fr \cup head(k) \backslash \{j\}, tr)\}$
        $cand \leftarrow \varnothing$
        for $j = (b, h, sgn) \in R$ do
            if $\exists h_x \in fr \wedge \{b_{x \oplus sgn}\} \rightarrow h_x \notin T$ then
                $cand \leftarrow cand \cup \{\{b_{x \oplus sgn}\} \rightarrow h_x\}$
                $rev \leftarrow rev \cup \{(ls \cup \{b_{x \oplus sgn}\}, fr \cup \{b_{x \oplus sgn}\} \backslash \{h_x\},$
                        $tr \cup \{\{b_{x \oplus sgn}\} \rightarrow h_x\})\}$
if $CS = \varnothing$ then
    return $(\textbf{False}, \varnothing)$
return $(\textbf{True}, CS)$
```

Algorithm 11 Completion by under-approximation

```
Input: an incomplete ABAN $\mathbb{A} = (\Sigma, T)$, a set of candidate regulations $R$, a targeted reachability $(\alpha, \omega)$
Output: Completed set $CS$
Initialization: $rev \leftarrow \{(\{\omega\}, \{\omega\}, \varnothing)\}$, $CS \leftarrow \varnothing$, $CS_{iter} \leftarrow \varnothing$
do
    SLCG $l = SLCG(\alpha, \omega) = (V_{\text{state}}, V_{\text{sol}}, E)$ by Algorithm 5
    while $rev \neq \varnothing$ do
        for $i = (ls, fr, tr) \in rev$ do
            $rev \leftarrow rev \backslash \{i\}$
            if $fr = \varnothing$ then
                $CS_{iter} \leftarrow CS_{iter} \cup \{ls\}$
                continue
            for $j \in fr$ do
                for $k \in j.next$ do
                    if $head(k) \nsubseteq ls$ then
                        $rev \leftarrow rev \cup \{(ls \cup head(k), fr \cup head(k) \backslash \{j\}, tr)\}$
            $cand \leftarrow \varnothing$
            for $j = (b, h, sgn) \in R$ do
                if $\exists h_x \in fr \wedge \{b_{x \oplus sgn}\} \rightarrow h_x \notin T$ then
                    $cand \leftarrow cand \cup \{\{b_{x \oplus sgn}\} \rightarrow h_x\}$
                    $rev \leftarrow rev \cup \{(ls \cup \{b_{x \oplus sgn}\}, fr \cup \{b_{x \oplus sgn}\} \backslash \{h_x\},$
                                $tr \cup \{\{b_{x \oplus sgn}\} \rightarrow h_x\})\}$
    $CS \leftarrow CS \cup CS_{iter}$
while $CS_{iter} \neq \varnothing$
return $CS$
```

Algorithm 12 Specialization

```
**Input**: a logic program $P$, a set of state transitions $E$, an unsatisfied element $(\alpha, \omega)$, a reachable set $Re$
**Output**: modified logic program $P'$ or $\varnothing$ if not revisable
Initialization: $Rev \leftarrow \{\omega\}$
do
    $RS \leftarrow \varnothing$
    for $R \in P$ do
        if $h(R) \in Rev$ then
            $RS \leftarrow RS \cup \{R\}$
    for $R \in RS$ do
        for $R' \in ls(R)$ do
            $P' \leftarrow P \cup \{R'\} \setminus \{R\}$
            if $P'$ is consistent with $E$ and $P'$ satisfies $Re$ then
                if $reach(\alpha, \omega) =$ False then
                    return $P'$
    for $R \in RS$ do
        $Rev \leftarrow b(R)$
while $RS \neq \varnothing$
```

Algorithm 13 Generalization

```
**Input**: a logic program $P$, a set of state transitions $E$, an unsatisfied element $(\alpha, \omega)$, an unreachable set $Un$
**Output**: modified logic program $P'$ or $\varnothing$ if not revisable
Initialization: $Rev \leftarrow \{\omega\}$
do
    $RS \leftarrow \varnothing$
    for $R \in P$ do
        if $h(R) \in Rev$ then
            $RS \leftarrow RS \cup \{R\}$
    for $R \in RS$ do
        for $R' \in lg(R)$ do
            $P' \leftarrow P \cup \{R'\} \setminus \{R\}$
            if $P'$ satisfies $Un$ then
                if $reach(\alpha, \omega) =$ True then
                    return $P'$
    for $R \in RS$ do
        $Rev \leftarrow b(R)$
while $RS \neq \varnothing$
```

Appendix C

Theorems and Proofs

Theorem C.1 (Change rate of sigmoid function)**.** Sigmoid function $S(x) = \dfrac{1}{1+e^{-x}}$ on $\mathbb{R}$ is monotonically increasing and its change rate is high around $x = 0$.

Proof. The derivative of $S(x)$ is $S'(x) = \dfrac{1}{e^x + e^{-x} + 2} > 0$, thus $S(x)$ is monotonically increasing.

The second derivative of $S(x)$ is $S''(x) = \dfrac{e^{-x} - e^x}{(e^x + e^{-x} + 2)^2}$, $S''(x) = 0$ iff $x = 0$, thus the change rate of $S(x)$ reaches its maximum at $x = 0$ and is high around $x = 0$. □

Theorem C.2 (Future states of synchronous semantics)**.** There are at most $O(3^m)$ possible future states for a system of synchronous update scheme if the number of total states of all the variables is fixed, where m is the number of different variables in all the heads of fireable transitions and x is the max number of qualitative levels in the system.

Proof. For every variable, there are at most x future states and the sum number of states of all variables is constant $xm = C$.

The amount of future states is $f(x) = x^{\frac{C}{x}}$, $f'(x) = \frac{C}{x}(1 - \ln x)$, for $x > 0$, $f(x)$ takes its maximum at $x = e$. As x is integer, we take the nearest value $x = 3$. Considering that C is also integer, the maximum of $f(x)$ is $\begin{cases} 3^{\lfloor \frac{C}{3} \rfloor} & \text{if } C \equiv 0 \pmod 3 \\ 3^{\lfloor \frac{C}{3} \rfloor - 1} \times 4 & \text{if } C \equiv 1 \pmod 3 \\ 3^{\lfloor \frac{C}{3} \rfloor} \times 2 & \text{if } C \equiv 2 \pmod 3 \end{cases}$ □

Theorem C.3 (Termination and correctness of PermReach)**.** Let $l = (V_{\text{state}}, V_{\text{sol}}, E)$ be an SLCG with initial state α and target local state ω and $k > 0$ be an integer.

- The call $PermReach(l, k)$ terminates.
- $PermReach(l, k) = (\textbf{False}, \varnothing)$ if $\nexists t$ a trajectory in l from α to ω.

Proof.

1. The algorithm starts by breaking the cycles in the SLCG and according to Theorem 3.1 it terminates and does not affect the reachability of α in l.
2. Then all the nodes of V_{state} (resp. V_{sol}) with no (resp. missing) outgoing edges are removed. Such nodes cannot be part of a trajectory leading to initial state α and thus this operation does not affect the reachability of α in l. The internal for loop of this operation iterates over V_{state} which is finite. To continue looping, it requires one state deletion thus this operation will terminate at least when V_{state} becomes $\varnothing$.
3. $PermReach(l, k) = \textbf{False}$ if $\nexists t$ a trajectory in l from α to $v \in V_{\text{sol}}$.
4. The call $PermReach(l, k)$ terminates.
5. After this preprocessing, pseudo reachability is checked and according to [61], it terminates and is correct. It is the only possibility for *PermReach* to output **False**.
6. Stochastic search follows by randomly reducing each **OR gate** of l to one of its edges to form l'. This operation is run a finite time k and iterates over V_{state} which is finite and thus it terminates. This operation does not create new edges, *i.e.* $E' \subseteq E$. $PermReach(l')$ traverses the permutations of **AND gates** and generates trajectories of l' leading to α. The number of permutations is finite and thus $PermReach(l')$ terminates.

□

Theorem C.4 (Complexity of PermReach)**.** Let $l = (V_{\text{state}}, V_{\text{sol}}, E)$ be an SLCG with initial state α and $k > 0$ be an integer. Let $s = |V_{\text{sol}}|$ be the number of target state of l. Let $v = |V_{\text{state}}|$ be the number of vertices of l. Let $e = |E|$ be the number of edges of l. The complexity of $PermReach(l, k)$ is $O(v + s + e + (v + s)/2 \times v \times e \times s + v^2 \times e + v \times e + k \times (v \times e^2 + \frac{v}{2}!))$ which is bounded by $O(k \times \frac{v}{2}!)$.

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

2. To remove useless nodes, *PermReach* iterates over all local states and checks if one local state has no successor in l which requires to iterates over all edges. In the worst case, all the local states will be removed one by one and thus the complexity of this operation is $op2 = O(v \times (v+s) \times e)$.

3. Computing pseudo reachability over l which has no loop corresponds to performing a depth-first search on all the branches of a tree and thus is bounded by $op3 = O((v+s) \times e)$.

4. The stochastic search iterates at most k times. Treating each **OR gate** to form l' has a cost of $O(v \times e \times e)$. $Permsolve(l')$ generates the permutations of each **AND gate** to assemble a trajectory can prove reachability of ω in l'. In the worst case, l has only one **AND gates** containing all the components of $V_{\text{state}} \setminus \alpha$. The number of total order is $O(\frac{v}{2}!)$. Thus $Permsolve(l')$ is bounded by $O(\frac{v}{2}!)$ and the whole stochastic search by $op4 = O(k \times (v \times e^2 + \frac{v}{2}!))$.

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

Conclusion: The complexity of $ASPReach(l,k)$ is bounded by $O(k \times \frac{v}{2}!)$. □

Theorem C.5 (Termination and correctness of ASPReach)**.** Let $l = (V_{\text{state}}, V_{\text{sol}}, E)$ be an SLCG with initial state α and target local state ω and $k > 0$ be an integer.

- The call $ASPReach(l,k)$ terminates.
- $ASPReach(l,k) = (\textbf{False}, \varnothing)$ if $\nexists t$ a trajectory in l from α to ω.
- $ASPReach(l,k) = (\textbf{True}, t)$ only if $\exists t$ a trajectory in l from α to ω.

Proof. The correctness of preprocessing is analogous to Theorem C.3.

After the preprocessing, we obtain a sub SLCG $l' = (V'_{\text{state}}, V'_{\text{sol}}, E')$. $ASPsolve(l')$ generates all the possible trajectories of l' leading to α. The number of possible trajectory is finite and thus $ASPsolve(l')$ terminates.

When $ASPsolve(l') = (\textbf{True}, t)$, t is a trajectory of l proving reachability of α in l and it is the only possibility for ASPReach to output

True. $ASPReach(l, k) = (\textbf{True}, t)$ only if $\exists t$ a trajectory in l from α to $v \in V_{\text{sol}}$. □

Theorem C.6 (Complexity of ASPReach)**.** Let $l = (V_{\text{state}}, V_{\text{sol}}, E)$ be an SLCG with initial state α and $k > 0$ be an integer. Let $s = |V_{\text{sol}}|$ be the number of target state of l. Let $v = |V_{\text{state}}|$ be the number of vertices of l. Let $e = |E|$ be the number of edges of l. The complexity of $ASPReach(l, k)$ is $O(v + s + e + (v + s)/2 \times v \times e \times s + v^2 \times e + v \times e + k \times (v \times e^2 + \frac{v}{2}!))$ which is bounded by $O(k \times \frac{v}{2}!)$.

Proof. 1. The complexity of preprocessing is analogous to the one in Theorem C.4,

$op1 = O(v + e + s + (v + s)/2 \times v \times e \times s)$

$op2 = O(v \times (v + s) \times e)$

$op3 = O((v + s) \times e)$

2. The stochastic search iterates at most k times. Treating each **OR gate** to form l' has a cost of $O(v \times e \times e)$. $ASPsolve(l')$ generates the trajectories within the SLCG that can prove reachability of ω in l'. Each trajectory is a sequence where each element of V_{state} appears exactly once. As ASP solver is a black-box system, we assume it solves the problem by pure brute force search.

3. In the worst case, $l' = l$ and all the components of $v_s \in V_{\text{state}} \wedge v_s \notin \alpha$ are placed in one **AND gate**. The number of total order is $O(\frac{v}{2}!)$. Thus $ASPsolve(l')$ is bounded by $O(\frac{v}{2}!)$ and the whole stochastic search by $op4 = O(k \times (v \times e^2 + \frac{v}{2}!))$.

4. Conclusion: The complexity of $ASPReach(l, k)$ is $O(op1 + op2 + op3 + op4) = O(v + e + s + (v + s)/2 \times v \times e \times s + v \times (v + s) \times e + v \times e + k \times (v \times e^2 + \frac{v}{2}!))$ and $ASPReach(l, k)$ is bounded by $O(k \times \frac{v}{2}!)$.

□

Definition C.1 (Rank of digraphs)**.** Given a digraph $G = (V, E)$, $\forall v \in V$ is associated with a rank $rk(v) \in \mathbb{N}$ s.t. $\forall v_1, v_2 \in V$, if $rk(v_1) < rk(v_2)$, $\nexists(v_1, v_2) \in E$, and if $rk(v_1) = rk(v_2)$, there exist paths $\gamma(v_1, v_2)$ and $\gamma(v_2, v_1)$.

Remark C.1. There does not necessarily exist a path starting from a node with higher rank to another node with lower rank. The rank of digraphs is a numbering that only offers possibilities of traversing the whole graph without violating a unified direction.

Appendix D

Pure ASP reachability analyzer

D.1 Pure ASP Implementation

Ben Abdallah *et al.* [1] have implemented reachability analysis using pure ASP approach (brute force search) and have done the comparison between the state-of-the-art methods of 2015.

Model-target	#automata	ASP-Th	Pint	libddd	GINsim	ASPi-PH
ERBB-whole	20	2.4s	out	1m55s	2m32s	12s
ERBB-sub	20	2.6s	0.03s	1m55s	-	5s
TCR-whole	40	-	Inconc	out	out	4m28s
TCR-sub	40	-	0.02s	out	-	1m35s

Table D.1: Compared performances of Rocca et al. method [68] denoted by ASP-Th, Pint, libddd, GINsim and our new iterative method ASP-PH

Table D.1 shows pure ASP style methods (ASP-Th and ASPi-PH) are still costly compared to PermReach and ASPReach.