Learning any semantics for dynamical systems represented by logic programs

Tony Ribeiro, Maxime Folschette, Morgan Magnin, Katsumi Inoue

▶ **To cite this version:**

Tony Ribeiro, Maxime Folschette, Morgan Magnin, Katsumi Inoue. Learning any semantics for dynamical systems represented by logic programs. 2020. hal-02925942v2

HAL Id: hal-02925942

https://hal.science/hal-02925942v2

Preprint submitted on 2 Sep 2020 (v2), last revised 13 Oct 2021 (v5)

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

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

Noname manuscript No.
(will be inserted by the editor)

Learning any semantics for dynamical systems represented by logic programs

Tony Ribeiro · Maxime Folschette · Morgan Magnin · Katsumi Inoue

Received: date / Accepted: date

Abstract Learning from interpretation transition (LFIT) automatically constructs a model of the dynamics of a system from the observation of its state transitions. So far the systems that LFIT handled were mainly restricted to synchronous deterministic dynamics. However, other dynamics exist in the field of logical modeling, in particular the asynchronous semantics which is widely used to model biological systems. In this paper, we propose a modeling of discrete memory-less multi-valued dynamic systems as logic programs in which a rule represents *what can occur rather than what will occur*. This modeling allows us to represent non-determinism and to propose an extension of LFIT to learn regardless of the update schemes. We also propose a second algorithm which is able to learn a whole system dynamics, including its semantics, in the form of a single propositional logic program with constraints. We show through theoretical results the correctness of our approaches. Practical evaluation is performed on benchmarks from biological literature.

Tony Ribeiro
Independant Researcher
Laboratoire des Sciences du Numérique de Nantes, 1 rue de la Noë, 44321 Nantes, France
National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
E-mail: tony_ribeiro@ls2n.fr,

Maxime Folschette
Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France

Morgan Magnin
Laboratoire des Sciences du Numérique de Nantes, 1 rue de la Noë, 44321 Nantes, France
National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan

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

Keywords inductive logic programming · dynamic systems · logical modeling · dynamic semantics

1 Introduction

Learning the dynamics of systems with many interactive components becomes more and more important in many applications such as physics, cellular automata, biochemical systems as well as engineering and artificial intelligence systems. In artificial intelligence systems, knowledge like action rules is employed by agents and robots for planning and scheduling. In biology, learning the dynamics of biological systems corresponds to the identification of influence of genes, signals, proteins and molecules that can help biologists to understand their interactions and biological evolution.

In modeling of dynamical systems, the notion of concurrency is crucial. When modeling a biological regulatory network, it is necessary to represent the respective evolution of each component of the system. One of the most debated issues with regard to semantics targets the choice of a proper update mode of every component, that is, synchronous [20], asynchronous [42] or more complex ones. The differences and common features of different semantics w.r.t. properties of interest (attractors, oscillators, etc.) have thus resulted in an area of research per itself [15,29,6]. But the biologists often have no idea whether a model of their system of interest should intrinsically be synchronous, asynchronous, generalized... It thus appears crucial to find ways to model systems from raw data without burdening the modelers with an a priori choice of the proper semantics.

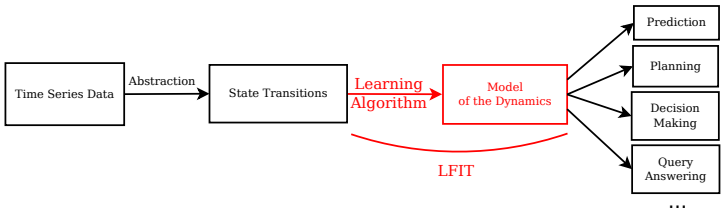

Fig. 1: Assuming a discretization of time series data of a system as state transitions, we propose a method to automatically model the system dynamics.

So far, learning from interpretation transition (LFIT) [16] has been proposed to automatically construct a model of the dynamics of a system from the observation of its state transitions. Figure 1 shows this learning process. Given some raw data, like time-series data of gene expression, a discretization of those data in the form of state transitions is assumed. From those state transitions, according to the semantics of the system dynamics, several inference algorithms modeling the system as a logic program have been proposed. The semantics of a system's dynamics can indeed differ with regard to the synchronism of its variables, the determinism of its evolution and the

influence of its history. The LFIT framework proposes several modeling and learning algorithms to tackle those different semantics. To date, the following systems have been tackled: memory-less deterministic systems [16], systems with memory [38], probabilistic systems [28] and their multi-valued extensions [39,27]. [40] proposes a method that allows to deal with continuous time series data, the abstraction itself being learned by the algorithm. As a summary, the systems that LFIT handles so far are restricted to synchronous deterministic dynamics.

In this paper, we extend this framework to learn systems dynamics independently of its update semantics. For this purpose, we propose a modeling of discrete memory-less multi-valued systems as logic programs in which each rule represents that a variable possibly takes some value at the next state, extending the formalism introduced in [16,37]. Research in multi-valued logic programming has proceeded along three different directions [21]: bilattice-based logics [13,14], quantitative rule sets [43] and annotated logics [5,4]. Our representation is based on annotated logics. Here, to each variable corresponds a domain of discrete values. In a rule, a literal is an atom annotated with one of these values. It allows us to represent annotated atoms simply as classical atoms and thus to remain at a propositional level. This modeling allows us to characterize optimal programs independently of the update semantics, allowing to model the dynamics of a wide range of discrete systems. To learn such semantic-free optimal programs, we propose **GULA**: the General Usage LFIT Algorithm. We show from theoretical results that this algorithm can learn under a wide range of update semantics including synchronous (deterministic or not), asynchronous and generalized semantics. Empirical evaluation is provided regarding both scalability and prediction accuracy over the three aforementioned semantics for Boolean network benchmarks from biological literature [23,9]. We also extend this modeling to propose the **Synchronizer** algorithm, that is able to learn a whole system dynamics, including its semantics behavior, in the form of a single propositional logic program with constraints. We show through theoretical results that this second algorithm can learn a program able to reproduce any given set of discrete state transitions and thus the behavior of any discrete memory-less dynamical semantics.

The organization of the paper is as follows. Section 2 provides a formalization of discrete memory-less dynamics system as multi-valued logic program. Section 3 formalizes dynamical semantics under logic programs. Section 4 presents the first algorithm, **GULA**, which learns optimal programs regardless of the semantics. Section 5 provides extension of the formalization and a second algorithm, the **Synchronizer**, to represent and learn the semantics behavior itself. Section 6 provides experimental evaluations. Section 7 discusses related work and Section 8 concludes the paper. All proofs of theorems and propositions are given in Appendix.

2 Logical Modeling of Dynamic Systems

In this section, the concepts necessary to understand the learning algorithms we propose are formalized. In Section 2.1, the basic notions of *multi-valued logic* ($\mathcal{M}$VL) are presented. Then, Section 2.2 presents a modeling of dynamics systems using this formalism. In the following, we denote by $\mathbb{N} := \{0, 1, 2, ...\}$ the set of natural numbers, and for all $k, n \in \mathbb{N}$, $[\![k; n]\!] := \{i \in \mathbb{N} \mid k \leq i \leq n\}$ is the set of natural numbers between k and n included. For any set S, the cardinal of S is denoted $|S|$ and the power set of S is denoted $\wp(S)$.

2.1 Multi-valued Logic Program

Let $\mathcal{V} = \{\mathrm{v}_1, \ldots, \mathrm{v}_n\}$ be a finite set of $n \in \mathbb{N}$ variables, $\mathcal{V}al$ the set in which variables take their values and $\mathsf{dom} : \mathcal{V} \to \wp(\mathcal{V}al)$ a function associating a domain to each variable. The atoms of $\mathcal{M}$VL are of the form v^{val} where $\mathrm{v} \in \mathcal{V}$ and $val \in \mathsf{dom}(\mathrm{v})$. The set of such atoms is denoted by $\mathcal{A}^{\mathcal{V}}_{\mathsf{dom}} = \{\mathrm{v}^{val} \in \mathcal{V} \times \mathcal{V}al \mid val \in \mathsf{dom}(\mathrm{v})\}$ for a given set of variables $\mathcal{V}$ and a given domain function dom. In the following, we work on specific $\mathcal{V}$ and dom that we omit to mention when the context makes no ambiguity, thus simply writing $\mathcal{A}$ for $\mathcal{A}^{\mathcal{V}}_{\mathsf{dom}}$.

Example 1 For a system of 3 variables, the typical set of variables is $\mathcal{V} = \{a, b, c\}$. In general, $\mathcal{V}al = \mathbb{N}$ so that domains are sets of natural integers, for instance: $\mathsf{dom}(a) = \{0, 1\}$, $\mathsf{dom}(b) = \{0, 1, 2\}$ and $\mathsf{dom}(c) = \{0, 1, 2, 3\}$. Thus, the set of all atoms is: $\mathcal{A} = \{a^0, a^1, b^0, b^1, b^2, c^0, c^1, c^2, c^3\}$.

A $\mathcal{M}$VL rule R is defined by:

$$R \quad = \quad \mathrm{v}_0^{val_0} \leftarrow \mathrm{v}_1^{val_1} \wedge \cdots \wedge \mathrm{v}_m^{val_m} \tag{1}$$

where $\forall i \in [\![0; m]\!], \mathrm{v}_i^{val_i} \in \mathcal{A}$ are atoms in $\mathcal{M}$VL so that every variable is mentioned at most once in the right-hand part: $\forall j, k \in [\![1; m]\!], j \neq k \Rightarrow \mathrm{v}_j \neq \mathrm{v}_k$. Intuitively, the rule R has the following meaning: the variable v_0 can take the value val_0 in the next dynamical step if for each $i \in [\![1; m]\!]$, variable v_i has value val_i in the current dynamical step.

The atom on the left-hand side of the arrow is called the *head* of R and is denoted $h(R) := \mathrm{v}_0^{val_0}$. The notation $\mathrm{var}(h(R)) := \mathrm{v}_0$ denotes the variable that occurs in $h(R)$. The conjunction on the right-hand side of the arrow is called the *body* of R, written $b(R)$ and can be assimilated to the set $\{\mathrm{v}_1^{val_1}, \ldots, \mathrm{v}_m^{val_m}\}$; we thus use set operations such as $\in$ and $\cap$ on it. The notation $\mathrm{var}(b(R)) := \{\mathrm{v}_1, \cdots, \mathrm{v}_m\}$ denotes the set of variables that occurs in $b(R)$. More generally, for all set of atoms $X \subseteq \mathcal{A}$, we denote $\mathrm{var}(X) := \{\mathrm{v} \in \mathcal{V} \mid \exists val \in \mathsf{dom}(\mathrm{v}), \mathrm{v}^{val} \in X\}$ the set of variables appearing in the atoms of X. A *multi-valued logic program* ($\mathcal{M}$VLP) is a set of $\mathcal{M}$VL rules.

Definition 1 introduces a domination relation between rules that defines a partial anti-symmetric ordering, as stated by Theorem 1. Rules with the most

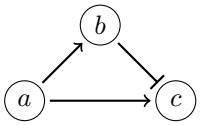

Fig. 2: Example of interaction graph of a regulation network representing an incoherent feed-forward loop [18] where a positively influences b and c, while b (and thus, indirectly, a) negatively influences c.

general bodies dominate the other rules. In practice, these are the rules we are interested in since they cover the most general cases.

Definition 1 (Rule Domination) Let R_1, R_2 be two $\mathcal{M}$VL rules. The rule R_1 *dominates* R_2, written $R_2 \leq R_1$ if $h(R_1) = h(R_2)$ and $b(R_1) \subseteq b(R_2)$.

Theorem 1 *Let R_1, R_2 be two $\mathcal{M}$VL rules. If $R_1 \leq R_2$ and $R_2 \leq R_1$ then $R_1 = R_2$.*

2.2 Dynamic Multi-valued Logic Program

We are interested in modeling non-deterministic (in a broad sense, which includes deterministic) discrete memory-less dynamical systems. In such a system, the next state is decided according to dynamics that depend on the current state of the system. From a modeling perspective, the variables of the system at time step t can be seen as *target variables* and the same variables at time step $t-1$ as *features variables*. Furthermore, additional variables that are external to the system, like *stimuli* or *checkpoints* for example, can appear only as feature or target variables. Such a system S can be represented by a $\mathcal{M}$VLP with some restrictions. First, the set of variables is divided into two disjoint subsets: $\mathcal{T}$ (for targets) encoding system variables at time step t plus optional external variables like checkpoints, and $\mathcal{F}$ (for features) encoding system variables at $t-1$ and optional external variables like stimuli. It is thus possible that $|\mathcal{F}| \neq |\mathcal{T}|$. Second, rules only have a conclusion at t and conditions at $t-1$, i.e., only an atom of a variable of $\mathcal{T}$ can be a head and only atoms of variables in $\mathcal{F}$ can appear in a body. In the following, we also re-use the same notations as for the $\mathcal{M}$VL of Section 2.1 such as $h(R)$, $b(R)$ and $\text{var}(h(R))$.

Definition 2 (Dynamic $\mathcal{M}$VLP) Let $\mathcal{T} \subset \mathcal{V}$ and $\mathcal{F} \subset \mathcal{V}$ such that $\mathcal{F} = \mathcal{V} \setminus \mathcal{T}$. A $\mathcal{DM}$VLP P is a $\mathcal{M}$VLP such that $\forall R \in P, \text{var}(h(R)) \in \mathcal{T}$ and $\forall \text{v}^{val} \in b(R), \text{v} \in \mathcal{F}$.

In the following, when there is no ambiguity, we suppose that $\mathcal{F}$, $\mathcal{T}$, $\mathcal{V}$ and $\mathcal{A}$ are already defined and we omit to defined them again.

Example 2 Figure 2 gives an example of regulation network with three elements a, b and c. The information of this network is not complete; notably, the relative "force" of the components a and b on the component c is not explicit. Multiple dynamics are then possible on this network, among which four possibilities are given below by Program 1 to 4, defined on $\mathcal{T} := \{a_t, b_t, c_t\}$, $\mathcal{F} := \{a_{t-1}, b_{t-1}, c_{t-1}\}$ and $\forall \mathrm{v} \in \mathcal{T} \cup \mathcal{F}, \mathsf{dom}(\mathrm{v}) := \{0, 1\}$. Program 1 is a direct translation of the relations of the regulation network. It only contains rule producing atoms with value 1 which is equivalent to a set of Boolean functions. In Program 2, a always takes value 1 while in Program 3 it always takes value 0, a having no incoming influence in the regulation network this can represent some kind of default behavior. In Program 3, the two red rules introduce potential non-determinism in the dynamics since both conditions can holds at the same time. In Program 4, the rule apply the conditions of the regulation network but it also allows each variable to keep the value 1 at t if it has it at $t-1$ and no inhibition occurs. We insist on the fact that the index notation t or $t-1$ is part of the variable name, not its value. This allows to distinguish variables from $\mathcal{T}$ (t) or $\mathcal{F}$ $(t-1)$.

Program 1	**Program 2**	**Program 3**	**Program 4**
$b_t^1 \leftarrow a_{t-1}^1$	$a_t^1 \leftarrow \emptyset$	$a_t^0 \leftarrow \emptyset$	$a_t^1 \leftarrow a_{t-1}^1$
$c_t^1 \leftarrow a_{t-1}^1 \wedge b_{t-1}^0$	$b_t^0 \leftarrow a_{t-1}^0$	$b_t^0 \leftarrow a_{t-1}^0$	$b_t^1 \leftarrow b_{t_1}^1$
	$b_t^1 \leftarrow a_{t-1}^1$	$b_t^1 \leftarrow a_{t-1}^1$	$b_t^1 \leftarrow a_{t-1}^1$
	$c_t^0 \leftarrow a_{t-1}^0$	$c_t^0 \leftarrow a_{t-1}^0$	$c_t^1 \leftarrow c_{t-1}^1 \wedge b_{t-1}^0$
	$c_t^0 \leftarrow b_{t-1}^1$	$c_t^0 \leftarrow b_{t-1}^1$	$c_t^1 \leftarrow a_{t-1}^1 \wedge b_{t-1}^0$
	$c_t^1 \leftarrow a_{t-1}^1 \wedge b_{t-1}^0$	$c_t^1 \leftarrow a_{t-1}^1$	

The dynamical system we want to learn the rules of is represented by a succession of *states* as formally given by Definition 3. We also define the "compatibility" of a rule with a state in Definition 4 and with a transition in Definition 5.

Definition 3 (Discrete state) A *discrete state* s on $\mathcal{T}$ (resp. $\mathcal{F}$) of a $\mathcal{DMVLP}$ is a function from $\mathcal{T}$ (resp. $\mathcal{F}$) to $\mathbb{N}$, i.e., it associates an integer value to each variable in $\mathcal{T}$ (resp. $\mathcal{F}$). It can be equivalently represented by the set of atoms $\{\mathrm{v}^{s(\mathrm{v})} \mid \mathrm{v} \in \mathcal{T} \text{ (resp. } \mathcal{F})\}$ and thus we can use classical set operations on it. We write $\mathcal{S}^{\mathcal{T}}$ (resp. $\mathcal{S}^{\mathcal{F}}$) to denote the set of all discrete states of $\mathcal{T}$ (resp. $\mathcal{F}$), and a couple of states $(s, s') \in \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$ is called a *transition*.

Example 3 The two sets of possible states of a program defined on the two sets of variables $\mathcal{T} = \{a_t, b_t, ch\}$ and $\mathcal{F} = \{a_{t-1}, b_{t-1}, st\}$, and the set of atoms $\mathcal{A} = \{a_t^0, a_t^1, b_t^0, b_t^1, ch^0, ch^1, a_{t-1}^0, a_{t-1}^1, b_{t-1}^0, b_{t-1}^1, c^0, c^1\}$ are :

$\mathcal{S}^{\mathcal{F}} = \{$
$\{a_{t-1}^0, b_{t-1}^0, st^0\}, \{a_{t-1}^0, b_{t-1}^0, st^1\},$
$\{a_{t-1}^0, b_{t-1}^1, st^0\}, \{a_{t-1}^0, b_{t-1}^1, st^1\},$
$\{a_{t-1}^1, b_{t-1}^0, st^0\}, \{a_{t-1}^1, b_{t-1}^0, st^1\},$
$\{a_{t-1}^1, b_{t-1}^1, st^0\}, \{a_{t-1}^1, b_{t-1}^1, st^1\}\}$

$\mathcal{S}^{\mathcal{T}} = \{$
$\{a_t^0, b_t^0, ch^0\}, \{a_t^0, b_t^0, ch^1\},$
$\{a_t^0, b_t^1, ch^0\}, \{a_t^0, b_t^1, ch^1\},$
$\{a_t^1, b_t^0, ch^0\}, \{a_t^1, b_t^0, ch^1\},$
$\{a_t^1, b_t^1, ch^0\}, \{a_t^1, b_t^1, ch^1\}\}$

Here, a and b are regular variables of the system and thus appear in both $\mathcal{F}$ and $\mathcal{T}$ encoded with time as label to make them different variables from a

$\mathcal{M}$VL perspective, they appear in both kind of state $\mathcal{S}^{\mathcal{F}}, \mathcal{S}^{\mathcal{T}}$. Variables *st* and *ch* are respectively a stimuli and a checkpoint and thus only appear in $\mathcal{F}, \mathcal{S}^{\mathcal{F}}$ or $\mathcal{T}, \mathcal{S}^{\mathcal{T}}$. Depending on the number of stimuli and checkpoint, states of $\mathcal{S}^{\mathcal{F}}$ can have a different size to state in $\mathcal{S}^{\mathcal{T}}$ (see Figure 3).

Definition 4 (Rule-state matching) Let $s \in \mathcal{S}^{\mathcal{F}}$. The $\mathcal{M}$VL rule R *matches* s, written $R \sqcap s$, if $b(R) \subseteq s$.

We note that this definition of matching only concerns feature variables. Target variables are never meant to be matched. The final program we want to learn should both:

- match the observations in a complete (all transitions are learned) and correct (no spurious transition) way;
- represent only minimal necessary interactions (according to Occam's razor: no overly-complex bodies of rules)

The following definitions formalize these desired properties. In Definition 5 we characterize the fact that a rule of a program is useful to describe the dynamics of one variable in a transition; this notion is then extended to a program and a set of transitions, under the condition that there exists such a rule for each variable and each transition. A conflict (Definition 6) arises when a rule describes a change that is not featured in the considered set of transitions. Finally, Definition 8 and Definition 7 give the characteristics of a complete (the whole dynamics is covered) and consistent (without conflict) program.

Definition 5 (Rule and program realization) Let R be a $\mathcal{M}$VL rule and $(s, s') \in \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$. The rule R *realizes* the transition (s, s'), written $s \xrightarrow{R} s'$, if $R \sqcap s \wedge h(R) \in s'$.

A $\mathcal{DM}$VLP P *realizes* $(s, s') \in \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$, written $s \xrightarrow{P} s'$, if $\forall \mathrm{v} \in \mathcal{T}, \exists R \in P, \mathrm{var}(h(R)) = \mathrm{v} \wedge s \xrightarrow{R} s'$. It *realizes* a set of transitions $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$, written $\overset{P}{\hookrightarrow} T$, if $\forall (s, s') \in T, s \xrightarrow{P} s'$.

In the following, for all set of transitions $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$, we denote: $\mathrm{first}(T) := \{s \in \mathcal{S}^{\mathcal{F}} \mid \exists (s_1, s_2) \in T, s_1 = s\}$ the set of all initial states of these transitions. We note that $\mathrm{first}(T) = \emptyset \iff T = \emptyset$.

Definition 6 (Conflict and Consistency) A $\mathcal{M}$VL rule R *conflicts* with a set of transitions $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$ when $\exists s \in \mathrm{first}(T), \big(R \sqcap s \wedge \forall (s, s') \in T, h(R) \notin s'\big)$. R is said to be *consistent* with T when R does not conflict with T.

A rule is consistent if for all initial states of the transitions of T (first(T)) matched by the rule, there exists a transitions of T for which it verifies the conclusion.

Definition 7 (Consistent program) A $\mathcal{DM}$VLP P is *consistent* with a set of transitions T if P does not contain any rule R conflicting with T.

Definition 8 (Complete program) A $\mathcal{DM}$VLP P is *complete* if $\forall s \in \mathcal{S}^{\mathcal{F}}, \forall \mathrm{v} \in \mathcal{T}, \exists R \in P, R \sqcap s \wedge \mathrm{var}(h(R)) = \mathrm{v}$.

A complete $\mathcal{DM}$VLP realizes at least one transition for each possible initial state. Definition 9 groups all the properties that we want the learned program to have: suitability and optimality, and Proposition 1 states that the optimal program of a set of transitions is unique.

Definition 9 (Suitable and optimal program) Let $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$. A $\mathcal{DM}$VLP P *is suitable for* T when:

- P is consistent with T,
- P realizes T,
- P is complete
- for all $\mathcal{M}$VL rules R not conflicting with T, there exists $R' \in P$ such that $R \leq R'$.

If in addition, for all $R \in P$, all the $\mathcal{M}$VL rules R' belonging to $\mathcal{DM}$VLP suitable for T are such that $R \leq R'$ implies $R' \leq R$ then P is called *optimal*.

Proposition 1 *Let $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$. The $\mathcal{DM}$VLP optimal for T is unique and denoted $P_{\mathcal{O}}(T)$.*

3 Dynamical semantics

The aim of this section is to formalize the general notion of *dynamical semantics* as an update policy based on a program, and to give characterizations of several widespread existing semantics used on discrete models.

In the previous section, we supposed the existence of two distinct sets of variables $\mathcal{F}$ and $\mathcal{T}$ that represent conditions (features) and conclusions (targets) of rules. Conclusion atoms allow to create one or several new state(s) made of target variables, from conditions on the current state which is made of feature atoms.

In Definition 10, we formalize the notion of dynamical semantics which is a function that, to a program, associates a set of transitions where each state has at least one outgoing transition. Such a set of transitions can also be seen as a function that maps any state to a non-empty set of states, regarded as possible dynamical branchings. We give examples of semantics afterwards.

Definition 10 (Dynamical Semantics) A *dynamical semantics* (on $\mathcal{A}$) is a function that associates, to each $\mathcal{DM}$VLP P, a set of transitions $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$ so that: $\mathrm{first}(T) = \mathcal{S}^{\mathcal{F}}$. Equivalently, a dynamical semantics can be seen as a function of $\big(\mathcal{DM}\text{VLP} \to (\mathcal{S}^{\mathcal{F}} \to \wp(\mathcal{S}^{\mathcal{T}}) \setminus \{\emptyset\})\big)$ where $\mathcal{DM}$VLP is the set of $\mathcal{DM}$VLPs.

We now aim at characterizing a set of semantics of interest for the current work, as given in Theorem 2. Beforehand, Definition 11 allows to denote as $\mathsf{Ccl}(s, P)$ the set of heads of rules, in a program P, matching a state s, and

Definition 12 introduces a notation B_X to consider only atoms in a set $B \subseteq \mathcal{A}$ that have their variable in a set $X \subseteq \mathcal{V}$. These two notations will be used in the next theorem and afterwards. In the following, we especially use the notation of Definition 12 with $\mathcal{A}$ (denoted $\mathcal{A}_X$) and on Ccl (denoted $\mathsf{Ccl}_X(s, P)$).

Definition 11 (Program Conclusions) Let s in $\mathcal{S}^{\mathcal{F}}$ and P a $\mathcal{M}$VLP. We denote: $\mathsf{Ccl}(s, P) := \{h(R) \in \mathcal{A} \mid R \in P, R \sqcap s\}$ the set of conclusion atoms in state s for the program P.

Definition 12 (Restriction of a Set of Atoms) Let $B \subseteq \mathcal{A}$ be a set of atoms, and $X \subseteq \mathcal{V}$ be a set of variables. We denote: $B_X = \{\mathrm{v}^{val} \in B \mid \mathrm{v} \in X\}$ the set of atoms of B that have their variables in X. If B is instead a function that outputs a set of atoms, we note $B_X(params)$ instead of $\big(B(params)\big)_X$, where $params$ is the sequence of parameters of B.

With Theorem 2, we characterize semantics which for any $\mathcal{DM}$VLP produce the same behavior using the corresponding optimal program, that is, any semantics DS such that for any $\mathcal{DM}$VLP $P, DS(P) = DS(P_{\mathcal{O}}(DS(P)))$. Such a semantics produces new states based only on the initial state s and the heads of matching rules of the given program $\mathsf{Ccl}(s, P)$, as stated by point (2). Moreover, $P_{\mathcal{O}}(DS(P))$ being consistent with $DS(P)$, each of those heads appears in a state of $DS(P)(s)$, thus the semantics should produce the same states being given the atoms of all those next states as possibilities, as stated by point (1). Those two conditions are sufficient to ensure that $DS(P_{\mathcal{O}}(DS(P))) = DS(P)$ and thus can be used to assert if the dynamics of a given semantics, for any given original program P, can be reproduced using the corresponding optimal program $P_{\mathcal{O}}(DS(P))$ with the same semantics.

Theorem 2 (Pseudo-idempotent Semantics and Optimal $\mathcal{DM}$VLP) *Let DS be a dynamical semantics. For all P a $\mathcal{DM}$VLP, if:*

- $\exists \mathsf{pick} \in (\mathcal{S}^{\mathcal{F}} \times \wp(\mathcal{A}_{\mathcal{T}}) \to \wp(\mathcal{S}^{\mathcal{T}}) \setminus \{\emptyset\})$ *so that*
  (1) $\forall D \subseteq \mathcal{A}_{\mathcal{T}}, \mathsf{pick}(s, \bigcup_{s' \in \mathsf{pick}(s,D)} s') = \mathsf{pick}(s, D) \wedge$
  (2) $\forall s \in \mathcal{S}^{\mathcal{F}}, \big(DS(P)\big)(s) = \mathsf{pick}(s, \mathsf{Ccl}(s, P))$,

then: for all P a $\mathcal{DM}$VLP, $DS(P_{\mathcal{O}}(DS(P)))) = DS(P)$.

Up to this point, no link has been made between corresponding feature (in $\mathcal{F}$) and target (in $\mathcal{T}$) variables or atoms. In other words, the formal link between the two atoms v_t^{val} and v_{t-1}^{val} has not been made yet. This link, called *projection*, is established in Definition 13, under the only assumption that $\mathsf{dom}(\mathrm{v}_t) = \mathsf{dom}(\mathrm{v}_{t-1})$. It has two purposes:

– When provided with a set of transitions, for instance by using a dynamical semantics, one can describe dynamical paths, that is, successions of next states, by using each next state to generate the equivalent initial state for the next transition;

– Some dynamical semantics (such as the asynchronous one, see Definition 15) make use of the current state to build the next state, and as such need a way to convert target variables into feature variables.

However, such a projection cannot be defined on the whole sets of target ($\mathcal{T}$) and feature ($\mathcal{F}$) variables, but only on two subsets $\overline{\mathcal{F}} \subseteq \mathcal{F}$ and $\overline{\mathcal{T}} \subseteq \mathcal{T}$. Note that we require the projection to be a bijection, thus: $|\overline{\mathcal{F}}| = |\overline{\mathcal{T}}|$. These subsets $\overline{\mathcal{T}}$ and $\overline{\mathcal{F}}$ contain variables that we call afterwards *regular variables*: they correspond to variables that have an equivalent in both the initial states (at $t-1$) and the next states (at t). Variables in $\mathcal{F} \setminus \overline{\mathcal{F}}$ can be considered as *stimuli* variables: they can only be observed in the previous state but we do not try to explain their next value in the current state; this is typically the case of external stimuli (sun, stress, nutriment...) that are unpredictable when observing only the studied system. Variables in $\mathcal{T} \setminus \overline{\mathcal{T}}$ can be considered as *checkpoint* variables: they are only observed in the present state as the result of the combination of other (regular and stimuli) variables; they can be of use to assess the occurrence of a specific configuration in the previous state but cannot be used to generate the next step. For the rest of this section, we suppose that $\overline{\mathcal{F}}$ and $\overline{\mathcal{T}}$ are given and that there exists such projection functions, as given by Definition 13. Figure 3 gives a representation of these sets of variables.

It is noteworthy that projections on states are not bijective, because of stimuli variables that have no equivalent in target variables, and checkpoint variables that have no equivalent in feature variables (see Figure 3). Therefore, the focus is often made on regular variables (in $\overline{\mathcal{F}}$ and $\overline{\mathcal{T}}$). Especially, for any pair of states $(s, s') \in \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$, having $\mathsf{sp}_{\overline{\mathcal{T}} \to \overline{\mathcal{F}}}(s') \subseteq s$, which is equivalent to $\mathsf{sp}_{\overline{\mathcal{F}} \to \overline{\mathcal{T}}}(s) \subseteq s'$, means that the regular variables in s and their projection in s' (or conversely) hold the same value, modulo the projection.

Definition 13 (Projections) A *projection on variables* is a bijective function $\mathsf{vp}_{\overline{\mathcal{T}} \to \overline{\mathcal{F}}} : \overline{\mathcal{T}} \to \overline{\mathcal{F}}$ so that $\overline{\mathcal{T}} \subseteq \mathcal{T}$, $\overline{\mathcal{F}} \subseteq \mathcal{F}$, and: $\forall \mathrm{v} \in \overline{\mathcal{T}}, \mathsf{dom}(\mathsf{vp}_{\overline{\mathcal{T}} \to \overline{\mathcal{F}}}(\mathrm{v})) = \mathsf{dom}(\mathrm{v})$. The *projection on atoms* (based on $\mathsf{vp}_{\overline{\mathcal{T}} \to \overline{\mathcal{F}}}$) is the bijective function:

$$\begin{aligned} \mathsf{ap}_{\overline{\mathcal{T}} \to \overline{\mathcal{F}}} : \mathcal{A}_{\overline{\mathcal{T}}} &\to \mathcal{A}_{\overline{\mathcal{F}}} \\ \mathrm{v}^{val} &\mapsto \left(\mathsf{vp}_{\overline{\mathcal{T}} \to \overline{\mathcal{F}}}(\mathrm{v})\right)^{val} \quad . \end{aligned}$$

The inverse function of $\mathsf{vp}_{\overline{\mathcal{T}} \to \overline{\mathcal{F}}}$ is denoted $\mathsf{vp}_{\overline{\mathcal{F}} \to \overline{\mathcal{T}}}$ and the inverse function of $\mathsf{ap}_{\overline{\mathcal{T}} \to \overline{\mathcal{F}}}$ is denoted $\mathsf{ap}_{\overline{\mathcal{F}} \to \overline{\mathcal{T}}}$.

The *projections on states* (based on $\mathsf{ap}_{\overline{\mathcal{T}} \to \overline{\mathcal{F}}}$ and $\mathsf{ap}_{\overline{\mathcal{F}} \to \overline{\mathcal{T}}}$) are the functions:

$$\begin{aligned} \mathsf{sp}_{\overline{\mathcal{T}} \to \overline{\mathcal{F}}} : \mathcal{S}^{\mathcal{T}} &\to \mathcal{S}^{\overline{\mathcal{F}}} \\ s' &\mapsto \{\mathsf{ap}_{\overline{\mathcal{T}} \to \overline{\mathcal{F}}}(\mathrm{v}^{val}) \in \mathcal{A} \mid \mathrm{v}^{val} \in s' \wedge \mathrm{v} \in \overline{\mathcal{T}}\} \\ \mathsf{sp}_{\overline{\mathcal{F}} \to \overline{\mathcal{T}}} : \mathcal{S}^{\mathcal{F}} &\to \mathcal{S}^{\overline{\mathcal{T}}} \\ s &\mapsto \{\mathsf{ap}_{\overline{\mathcal{F}} \to \overline{\mathcal{T}}}(\mathrm{v}^{val}) \in \mathcal{A} \mid \mathrm{v}^{val} \in s \wedge \mathrm{v} \in \overline{\mathcal{F}}\} \quad . \end{aligned}$$

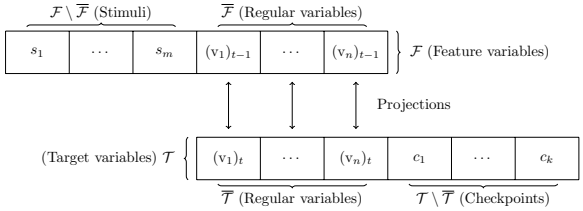

Fig. 3: Representation of a state transition of a dynamic system over n variables, m stimuli and k checkpoints, i.e., $|\mathcal{F}| = n + m, |\mathcal{T}| = n + k$.

3.1 Synchronous, Asynchronous and General Semantics

In the following, we present a formal definition and a characterization of three particular semantics that are widespread in the field of complex dynamical systems: synchronous, asynchronous and general, and we also treat the particular case of the deterministic synchronous semantics. Note that some points in these definitions are arbitrary and could be discussed depending on the modeling paradigm. For instance, the policy about rules R so that $\exists s \in \mathcal{S}^{\mathcal{F}}, R \sqcap s \wedge \mathsf{ap}_{\overline{\mathcal{T}} \to \overline{\mathcal{F}}}(h(R)) \in s$, which model stability in the dynamics, could be to include them (such as in the synchronous and general semantics) or exclude them (such as in the asynchronous semantics) from the possible dynamics. The modeling method presented so far in this paper is independent to the considered semantics as long as it respects Definition 10 and the capacity of the optimal program to reproduce the observed behavior is ensured as long as the semantics respects Theorem 2.

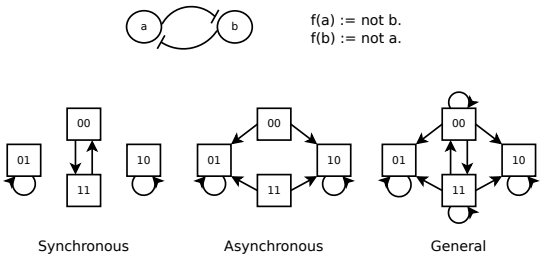

Fig. 4: A Boolean network with two variables inhibiting each other (top) and its synchronous, asynchronous and general state transitions diagrams (bottom).

Definition 14 introduces the synchronous semantics, consisting in updating all variables at once in each step in order to compute the next state. The value of each variable in the next state is taken amongst a "pool" of atoms containing all conclusions of rules that match the current state (using Ccl) and atoms produced by a "default function" d that is explained below. However, this is taken in a loose sense: as stated above, atoms that make a variable

change its value are not prioritized over atoms that don't. Furthermore, if several atoms on the same variable are provided in the pool (as conclusions of different rules or provided by the default function), then several transitions are possible, depending on which one is chosen. Thus, for a self-transition $(s, s') \in \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$ with $\mathsf{sp}_{\overline{\mathcal{T}} \to \overline{\mathcal{F}}}(s') \subseteq s$ to occur, there needs to be, for each atom $\mathrm{v}^{val} \in s'$ so that $\mathrm{v} \in \overline{\mathcal{T}}$, either a rule that matches s and whose head is v^{val} or that the default function gives the value v^{val}. Note however that such a loop is not necessarily a point attractor; it is only the case if all atoms in the pool are also in $\mathsf{sp}_{\overline{\mathcal{T}} \to \overline{\mathcal{F}}}(s)$.

As explained above, for a given state s and a given set of variables W, the function d provides a set of "default atoms" added to the pool of atoms used to build the next state, along with rules conclusions. This function d, however, is not explicitly given: the only constraints are that:

- d produces atoms at least for a provided set of variables W, specifically, the set of variables having no conclusion in a given state, which is necessary in the case of an incomplete program,
- $d(s, \emptyset)$ is a subset of $d(s, W)$ for all W, as it intuitively represents a set of default atoms that are always available.

Note that $d(s, \emptyset) = \emptyset$ is always valid. In the case of a complete program, that is, a program providing conclusions for every variables in every state, d is always called with $W = \emptyset$ and the other cases can thus be ignored. Another typical use for d is the case of a system with Boolean variables (i.e., such that $\forall \mathrm{v} \in \mathcal{V}, \mathsf{dom}(\mathrm{v}) = \{0, 1\}$) where a program P is built by importing only the positive rules of the system, that is, only rules with atoms v_t^1 as heads. This may happen when importing a model from another formalism featuring only Boolean formulas, such as Boolean networks. In this case, d can be used to provide a default atom w_t^0 for all variables w that do not appear in $\mathsf{Ccl}(s, P)$, thus reproducing the dynamics of the original system.

Definition 14 (Synchronous semantics) Let $d \in (\mathcal{S}^{\mathcal{F}} \times \wp(\mathcal{T}) \to \wp(\mathcal{A}_{\mathcal{T}}))$, so that $\forall s \in \mathcal{S}^{\mathcal{F}}, \forall W \subseteq \mathcal{T}, W \subseteq \mathrm{var}(d(s, W)) \wedge d(s, \emptyset) \subseteq d(s, W)$. The *synchronous semantics* $\mathcal{T}_{syn}$ is defined by:

$$\mathcal{T}_{syn} : P \mapsto \{(s, s') \in \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}} \mid s' \subseteq \mathsf{Ccl}(s, P) \cup d(s, \mathcal{T} \setminus \mathrm{var}(\mathsf{Ccl}(s, P)))\} \ .$$

In Definition 15, we formalize the asynchronous semantics that imposes that no more than one regular variable can change its value in each transition. The checkpoint variables are not counted since they have no equivalent in feature variables to be compared to. As for the previous synchronous semantics, we use here a "pool" of atoms, made of rules conclusions and default atoms, that may be used to build the next states. The default function d used here is inspired from the previous synchronous semantics, with an additional constraint: its result always contains the atoms of the initial state. Constrains are also added on the next state to limit to at most one regular variable change. Moreover, contrary to the synchronous semantics, the asynchronous semantics prioritizes the changes. Thus, for a self-transition $(s, s') \in \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$ with

$\mathsf{sp}_{\overline{\mathcal{T}}\to\overline{\mathcal{F}}}(s') \subseteq s$ to occur, it is required that all atoms of regular variables in the pool are in $\mathsf{sp}_{\overline{\mathcal{F}}\to\overline{\mathcal{T}}}(s)$, i.e., this only happens when (s, s') is a point attractor, in the sense that all regular variables cannot change their value.

Definition 15 (Asynchronous semantics) Let $d \in (\mathcal{S}^{\mathcal{F}} \times \wp(\mathcal{T}) \to \wp(\mathcal{A}_{\mathcal{T}}))$, so that $\forall s \in \mathcal{S}^{\mathcal{F}}, \forall W \subseteq \mathcal{T}, W \subseteq \mathrm{var}(d(s, W)) \wedge \mathsf{sp}_{\overline{\mathcal{F}}\to\overline{\mathcal{T}}}(s) \subseteq d(s, \emptyset) \subseteq d(s, W)$. The *asynchronous semantics* $\mathcal{T}_{asyn}$ is defined by:

$$\mathcal{T}_{asyn} : P \mapsto \{(s, s') \in \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}} \mid s' \subseteq \mathsf{Ccl}(s, P) \cup d(s, \mathcal{T} \setminus \mathrm{var}(\mathsf{Ccl}(s, P))) \wedge \big(|\mathsf{sp}_{\overline{\mathcal{F}}\to\overline{\mathcal{T}}}(s) \setminus s'| = 1 \vee \mathsf{Ccl}_{\overline{\mathcal{T}}}(s, P) \cup d_{\overline{\mathcal{T}}}(s, \overline{\mathcal{T}} \setminus \mathrm{var}(\mathsf{Ccl}(s, P))) = \mathsf{sp}_{\overline{\mathcal{F}}\to\overline{\mathcal{T}}}(s)\big)\}$$

where the notations $\mathcal{A}_{\mathcal{T}}$, $\mathsf{Ccl}_{\overline{\mathcal{T}}}$ and $d_{\overline{\mathcal{T}}}$ come from Definition 12.

A typical mapping for d is: $d : (s, W) \mapsto \mathsf{sp}_{\overline{\mathcal{F}}\to\overline{\mathcal{T}}}(s)$, thus conserving the previous values and ignoring the second argument.

In Definition 16, we formalize the general semantics as a more permissive version of the synchronous one: any subset of the variables can change their value in a transition. This semantics uses the same "pool" of atoms than the synchronous semantics containing rules conclusions of P and default atoms provided by d, and no constraint. However, as for the asynchronous semantics, the atoms of the initial state must always be featured as default atoms. Thus, a self-transition $(s, s') \in \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$ with $\mathsf{sp}_{\overline{\mathcal{F}}\to\overline{\mathcal{T}}}(s) \subseteq s'$ occurs for each state s because, intuitively, the empty set of variables can always be selected for update. However, as for the synchronous semantics, such a self-transition is a point attractor only if all atoms of regular variables in the "pool" are in $\mathsf{sp}_{\overline{\mathcal{F}}\to\overline{\mathcal{T}}}(s)$. Finally, we note that the general semantics contains the dynamics of both the synchronous and the asynchronous semantics, but also other dynamics not featured in these two other semantics.

Definition 16 (General semantics) Let $d \in (\mathcal{S}^{\mathcal{F}} \times \wp(\mathcal{T}) \to \wp(\mathcal{A}_{\mathcal{T}}))$, so that $\forall s \in \mathcal{S}^{\mathcal{F}}, \forall W \subseteq \mathcal{T}, W \subseteq \mathrm{var}(d(s, W)) \wedge \mathsf{sp}_{\overline{\mathcal{F}}\to\overline{\mathcal{T}}}(s) \subseteq d(s, \emptyset) \subseteq d(s, W)$. The *general semantics* $\mathcal{T}_{gen}$ is defined by:

$$\mathcal{T}_{gen} : P \mapsto \{(s, s') \in \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}} \mid s' \subseteq \mathsf{Ccl}(s, P) \cup d(s, T \setminus \mathrm{var}(\mathsf{Ccl}(s, P)))\}.$$

Finally, with Theorem 3, we state that the definitions and method developed in the previous section are independent of the chosen semantics as long as it respect Theorem 2.

Theorem 3 (Semantics-free correctness) *Let P be a $\mathcal{DM}$*VLP*.*

- $\mathcal{T}_{syn}(P) = \mathcal{T}_{syn}(P_{\mathcal{O}}(\mathcal{T}_{syn}(P)))$,
- $\mathcal{T}_{asyn}(P) = \mathcal{T}_{asyn}(P_{\mathcal{O}}(\mathcal{T}_{asyn}(P)))$,
- $\mathcal{T}_{gen}(P) = \mathcal{T}_{gen}(P_{\mathcal{O}}(\mathcal{T}_{gen}(P)))$.

4 GULA

Until now, the LF1T algorithm [16,37,39] only tackled the learning of synchronous deterministic programs. Using the formalism introduced in the previous sections, it can now be revised to learn systems from transitions produced from any semantics respecting Theorem 2 like the three semantics defined above. Furthermore, both deterministic and non-deterministic systems can now be learned.

4.1 Learning operations

This section focuses on the manipulation of programs for the learning process. Definition 17 and Definition 18 formalize the main atomic operations performed on a rule or a program by the learning algorithm, whose objective is to make minimal modifications to a given $\mathcal{DM}$VLP in order to be consistent with a new set of transitions.

Definition 17 (Rule least specialization) Let R be a $\mathcal{M}$VL rule and $s \in \mathcal{S}^{\mathcal{F}}$ such that $R \sqcap s$. The *least specialization* of R by s according to $\mathcal{F}$ and $\mathcal{A}$ is:

$$L_{\text{spe}}(R, s, \mathcal{A}, \mathcal{F}) := \{h(R) \leftarrow b(R) \cup \{\mathrm{v}^{val}\} \mid \mathrm{v} \in \mathcal{F} \wedge \mathrm{v}^{val} \in \mathcal{A} \wedge \mathrm{v}^{val} \notin s \wedge \forall val' \in \mathbb{N}, \mathrm{v}^{val'} \notin b(R)\}.$$

The least specialization $L_{\text{spe}}(R, s, \mathcal{A}, \mathcal{F})$ produces a set of rule which matches all states that R matches except s. Thus $L_{\text{spe}}(R, s, \mathcal{A}, \mathcal{F})$ realizes all transitions that R realizes except the ones starting from s. Note that $\forall R \in P, R \sqcap s \wedge |b(R)| = |\mathcal{F}| \implies L_{\text{spe}}(R, s, \mathcal{A}, \mathcal{F}) = \emptyset$, i.e., a rule R matching s cannot be modified to make it not match s if its body already contains all feature variables, because nothing can be added in its body.

Definition 18 (Program least revision) Let P be a $\mathcal{DM}$VLP, $s \in \mathcal{S}^{\mathcal{F}}$ and $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$ such that $\text{first}(T) = \{s\}$. Let $R_P := \{R \in P \mid R \text{ conflicts with } T\}$. The *least revision* of P by T according to $\mathcal{A}$ and $\mathcal{F}$ is $L_{\text{rev}}(P, T, \mathcal{A}, \mathcal{F}) := (P \setminus R_P) \cup \bigcup_{R \in R_P} L_{\text{spe}}(R, s, \mathcal{A}, \mathcal{F})$.

Note that according to Definition 18, $\text{first}(T) = \{s\}$ implies that all transitions for T have s as initial state. Theorem 4 states properties on the least revision, in order to prove it suitable to be used in the learning algorithm.

Theorem 4 (Properties of least revision) *Let R be a $\mathcal{M}$VL rule and $s \in \mathcal{S}^{\mathcal{F}}$ such that $R \sqcap s$. Let $S_R := \{s' \in \mathcal{S}^{\mathcal{F}} \mid R \sqcap s'\}$ and $S_{\text{spe}} := \{s' \in \mathcal{S}^{\mathcal{F}} \mid \exists R' \in L_{\text{spe}}(R, s, \mathcal{A}, \mathcal{F}), R' \sqcap s'\}$.*

Let P be a $\mathcal{DM}$VLP and $T, T' \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$ such that $|\text{first}(T)| = 1 \wedge \text{first}(T) \cap \text{first}(T') = \emptyset$. The following results hold:

1. $S_{\text{spe}} = S_R \setminus \{s\}$,

2. $L_{\text{rev}}(P,T,\mathcal{A},\mathcal{F})$ *is consistent with* T,
3. $\overset{P}{\hookrightarrow} T' \implies \overset{L_{\text{rev}}(P,T,\mathcal{A},\mathcal{F})}{\hookrightarrow} T'$,
4. $\overset{P}{\hookrightarrow} T \implies \overset{L_{\text{rev}}(P,T,\mathcal{A},\mathcal{F})}{\hookrightarrow} T$,
5. P *is complete* $\implies L_{\text{rev}}(P,T,\mathcal{A},\mathcal{F})$ *is complete.*

The next properties are directly used in the learning algorithm. Proposition 2 gives an explicit definition of the optimal program for an empty set of transitions, which is the starting point of the algorithm. Proposition 3 gives a method to obtain the optimal program from any suitable program by simply removing the dominated rules; this means that the $\mathcal{DM}$VLP optimal for a set of transitions can be obtained from any $\mathcal{DM}$VLP suitable for the same set of transitions by removing all the dominated rules. Finally, in association with these two results, Theorem 5 gives a method to iteratively compute $P_{\mathcal{O}}(T)$ for any $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$, starting from $P_{\mathcal{O}}(\emptyset)$.

Proposition 2 $P_{\mathcal{O}}(\emptyset) = \{\mathrm{v}^{val} \leftarrow \emptyset \mid \mathrm{v} \in \mathcal{T} \wedge \mathrm{v}^{val} \in \mathcal{A}_{\mathcal{T}}\}$.

Proposition 3 *Let* $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$. *If* P *is a* $\mathcal{DM}$VLP *suitable for* T, *then* $P_{\mathcal{O}}(T) = \{R \in P \mid \forall R' \in P, R \leq R' \implies R' \leq R\}$

Theorem 5 (Least revision and suitability) *Let* $s \in \mathcal{S}^{\mathcal{F}}$ *and* $T, T' \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$ *such that* $|\text{first}(T')| = 1 \wedge \text{first}(T) \cap \text{first}(T') = \emptyset$. $L_{\text{rev}}(P_{\mathcal{O}}(T), T', \mathcal{A}, \mathcal{F})$ *is a* $\mathcal{DM}$VLP *suitable for* $T \cup T'$.

4.2 Algorithm

In this section we present **GULA**: the General Usage LFIT Algorithm, a revision of the **LF1T** algorithm [16,37] to capture a set of multi-valued dynamics that especially encompass the classical synchronous, asynchronous and general semantics dynamics. For this learning algorithm to operate, there is no restriction on the semantics. **GULA** learns the optimal program that, under the same semantics, is able to exactly reproduce a complete set of observations, if the semantics respect Theorem 2. Section 5 will be devoted to also learning the behaviors of the semantics itself, if it is unknown.

GULA learns a logic program from the observations of its state transitions. Given as input a set of transitions $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$, **GULA** iteratively constructs a $\mathcal{DM}$VLP that models the dynamics of the observed system by applying the method formalized in the previous section as shown in Algorithm 1. From the set of transitions T, **GULA** learns the conditions under which each $\mathrm{v}^{val} \in \mathcal{A}' \subseteq \mathcal{A}, \mathrm{v} \in \mathcal{T}' \subseteq \mathcal{T}$ may appear in the next state. The algorithm starts by computing the set of all negative examples of the appearance of v^{val} in next state: all states such that v never takes the value *val* in the next state of a transition of T. Those negative examples are then used during the following learning phase to iteratively learn the set of rules $P_{\mathcal{O}}(T)$. The learning phase starts by initializing a set of rules $P_{\mathrm{v}^{val}}$ to $\{R \in P_{\mathcal{O}}(\emptyset) \mid h(R) = \mathrm{v}^{val}\} = \{\mathrm{v}^{val} \leftarrow \emptyset\}$. $P_{\mathrm{v}^{val}}$ is iteratively revised against each negative example *neg* in

Algorithm 1 GULA

- **INPUT:** a set of atoms $\mathcal{A}'$, a set of transitions $T \subseteq \mathcal{S}^{\mathcal{F}'} \times \mathcal{S}^{\mathcal{T}'}$, two sets of variables $\mathcal{F}'$ and $\mathcal{T}'$.
- For each atom $\mathrm{v}^{val} \in \mathcal{A}'$ of each variable $\mathrm{v} \in \mathcal{T}'$:
  - Extract all states from which no transition to v^{val} exist: $Neg_{\mathrm{v}^{val}} := \{s \mid \nexists (s, s') \in T, \mathrm{v}^{val} \in s'\}$.
  - Initialize $P_{\mathrm{v}^{val}} := \{\mathrm{v}^{val} \leftarrow \emptyset\}$.
  - For each state $s \in Neg_{\mathrm{v}^{val}}$:
    - Extract and remove the rules of $P_{\mathrm{v}^{val}}$ that match s: $M_{\mathrm{v}^{val}} := \{R \in P \mid b(R) \subseteq s\}$ and $P_{\mathrm{v}^{val}} := P_{\mathrm{v}^{val}} \setminus M_{\mathrm{v}^{val}}$.
    - $LS := \emptyset$
    - For each rule $R \in M_{\mathrm{v}^{val}}$:
      - Compute its least specialization $P' = L_{\mathrm{spe}}(R, s, \mathcal{A}', \mathcal{F}')$.
      - Remove all the rules in P' dominated by a rule in $P_{\mathrm{v}^{val}}$.
      - Remove all the rules in P' dominated by a rule in LS.
      - Remove all the rules in LS dominated by a rule in P'.
      - $LS := LS \cup P'$.
    - Add all remaining rules of LS to $P_{\mathrm{v}^{val}}$: $P_{\mathrm{v}^{val}} := P_{\mathrm{v}^{val}} \cup LS$.
  - $P := P \cup P_{\mathrm{v}^{val}}$
- **OUTPUT:** $P_{\mathcal{O}}(T) := P$.

$Neg_{\mathrm{v}^{val}}$. All rules R_m of $P_{\mathrm{v}^{val}}$ that match neg have to be revised. In order for $P_{\mathrm{v}^{val}}$ to remain optimal, the revision of each R_m must not match neg but still matches every other state that R_m matches. To ensure that, the least specialization (see Definition 17) is used to revise each conflicting rule R_m. For each variable of $\mathcal{F}'$ so that $b(R_m)$ has no condition over it, a condition over another value than the one observed in state neg can be added. None of those revision match neg and all states matched by R_m are still matched by at least one of its revisions. Each revised rule can be dominated by a rule in $P_{\mathrm{v}^{val}}$ or another revised rules and thus dominance must be checked from both. The non-dominated revised rules are then added to $P_{\mathrm{v}^{val}}$. Once $P_{\mathrm{v}^{val}}$ has been revised against all negatives example of $Neg_{\mathrm{v}^{val}}$, $P_{\mathrm{v}^{val}} = \{R \in P_{\mathcal{O}}(T) \mid h(R) = \mathrm{v}^{val}\}$. Finally, $P_{\mathrm{v}^{val}}$ is added to P and the loop restarts with another atom. Once all values of each variable have been treated, the algorithm outputs P which is then equal to $P_{\mathcal{O}}(T)$. More discussion of the implementation and detailed pseudocode are given in appendix. The source code of the algorithm is available at `https://github.com/Tony-sama/pylfit` under GPL-3.0 License.

Theorem 6 gives the properties of the algorithm: **GULA** terminates and **GULA** is sounds, complete and optimal w.r.t. its input, i.e., all and only non-dominated consistent rules appear in its output program which is the optimal program of its input. Finally, Theorem 7 characterizes the algorithm time and memory complexities.

Theorem 6 (GULA Termination, soundness, completeness, optimality) *Let $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$.*

(1) *Any call to* **GULA** *on finite sets terminates,*

(2) $\mathbf{GULA}(\mathcal{A}, T, \mathcal{F}, \mathcal{T}) = P_{\mathcal{O}}(T)$,

(3) $\forall \mathcal{A}' \subseteq \mathcal{A}_{\mathcal{T}}, \mathbf{GULA}(\mathcal{A}_{\mathcal{F}} \cup \mathcal{A}', T, \mathcal{F}, \mathcal{T}) = \{R \in P_{\mathcal{O}}(T) \mid h(R) \in \mathcal{A}'\}$.

From Theorem 6: for any dynamical semantics DS and any $\mathcal{DM}$VLP P, $\mathbf{GULA}(\mathcal{A}, DS(P), \mathcal{F}, \mathcal{T}) = P_{\mathcal{O}}(DS(P))$. If DS is as in Theorem 2, then $DS(\mathbf{GULA}(\mathcal{A}, DS(P), \mathcal{F}, \mathcal{T})) = DS(P_{\mathcal{O}}(DS(P))) = DS(P)$. From Theorem 3, the following holds:

- $\mathcal{T}_{syn}(\mathbf{GULA}(\mathcal{A}, \mathcal{T}_{syn}(P), \mathcal{F}, \mathcal{T})) = \mathcal{T}_{syn}(P_{\mathcal{O}}(\mathcal{T}_{syn}(P))) = \mathcal{T}_{syn}(P)$
- $\mathcal{T}_{asyn}(\mathbf{GULA}(\mathcal{A}, \mathcal{T}_{asyn}(P), \mathcal{F}, \mathcal{T})) = \mathcal{T}_{asyn}(P_{\mathcal{O}}(\mathcal{T}_{asyn}(P))) = \mathcal{T}_{asyn}(P)$
- $\mathcal{T}_{gen}(\mathbf{GULA}(\mathcal{A}, \mathcal{T}_{gen}(P), \mathcal{F}, \mathcal{T})) = \mathcal{T}_{gen}(P_{\mathcal{O}}(\mathcal{T}_{gen}(P))) = \mathcal{T}_{gen}(P)$

Thus the algorithm can be used to learn from transitions produced from both synchronous, asynchronous and general semantics.

Theorem 7 (GULA Complexity) *Let $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$ be a set of transitions, Let $n := max(|\mathcal{F}|, |\mathcal{T}|)$ and $d := \max(\{|\mathsf{dom}(\mathrm{v})|) \in \mathbb{N} \mid \mathrm{v} \in \mathcal{F} \cup \mathcal{T}\}$. The worst-case time complexity of* **GULA** *when learning from T belongs to $\mathcal{O}(|T|^2 + |T| \times (2n^4 d^{2n+2} + 2n^3 d^{n+1}))$ and its worst-case memory use belongs to $\mathcal{O}(d^{2n} + 2nd^{n+1} + nd^{n+2})$.*

5 Learning From Any Dynamical Semantics

Any non-deterministic (and thus deterministic) discrete memory-less dynamical system S can be represented by a $\mathcal{M}$VLP with some restrictions and a dedicated dynamical semantics. For this, programs must contain two types of rules: *possibility rules* which have conditions on variables at $t-1$ and conclusion on one variable at t, same as for $\mathcal{DM}$VLP; and *constraint rules* which have conditions on both t and $t-1$ but no conclusion. In the following, we also re-use the same notations as for the $\mathcal{M}$VL of Section 2.1 such as $h(R)$, $b(R)$ and $\mathrm{var}(h(R))$.

5.1 Constraints $\mathcal{DM}$VLP

Definition 19 (Constrainted $\mathcal{DM}$VLP) Let P' be a $\mathcal{DM}$VLP on $\mathcal{A}^{\mathcal{F}\cup\mathcal{T}}_{\mathsf{dom}}$, $\mathcal{F}$ and $\mathcal{T}$ two sets of variables, and ε a special variable with $\mathsf{dom}(\varepsilon) = \{0, 1\}$ so that $\varepsilon \notin \mathcal{T} \cup \mathcal{F}$. A $\mathcal{CDM}$VLP P is a $\mathcal{M}$VLP such that $P = P' \cup \{R \in \mathcal{M}\mathrm{VL} \mid h(R) = \varepsilon^1 \wedge \forall \mathrm{v}^{val} \in b(R), \mathrm{v} \in \mathcal{F} \cup \mathcal{T}\}$. A $\mathcal{M}$VL rule R such that $h(R) = \varepsilon^1$ and $\forall \mathrm{v}^{val} \in b(R), \mathrm{v} \in \mathcal{F} \cup \mathcal{T}$ is called a $\mathcal{M}$VL *constraint*.

Moreover, in the following we denote $\mathcal{V} = \mathcal{F} \cup \mathcal{T} \cup \{\varepsilon\}$. This $\mathcal{V}$ is different than the one of P' (which is $\mathcal{F} \cup \mathcal{T}$, without the special variable ε). From now, a constraint C is denoted: $\leftarrow b(C)$.

Example 4 $\leftarrow a^0_t \wedge a^0_{t-1}$ is a constraint that can prevent a to take the value 0 in two successive states. $\leftarrow b^1_t \wedge d^2_t \wedge c^2_{t-1}$ is a constraint that can prevent to have both b^1 and d^2 in the next state if c^2 appears in the initial state. $\leftarrow a^0_t \wedge b^0_t$ is a constraint with only conditions in $\mathcal{T}$, it prevents a and b to take value 0 at

same time. $\leftarrow a^0_{t-1} \wedge b^0_{t-1}$ is a constraint with only conditions in $\mathcal{F}$, it prevents any transitions from a state where a and b have value 0, thus creating final states.

Definition 20 (Constraint-transition matching) Let $(s, s') \in \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$. The constraint C *matches* (s, s'), written $C \sqcap (s, s')$, iff $b(C) \subseteq s \cup s'$.

Using the notion of rule and constraint matching we can use a $\mathcal{CDM}$VLP to compute the next possible states. Definition 21 provides such a method based on synchronous semantic and constraints. Given a state, the set of possible next states is the Cartesian product of the conclusion of all matching rules and default atoms. Constraints rules are then used to discard states that would generate non-valid transitions.

Definition 21 (Synchronous constrained Semantics) The *synchronous constrained semantics* $\mathcal{T}_{syn-c}$ is defined by:

$$\mathcal{T}_{syn-c} : P \mapsto \{(s, s') \in \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}} \mid s' \subseteq \mathsf{Ccl}(s, P) \wedge \nexists C \in P, h(C) = \varepsilon^1 \wedge C \sqcap (s, s')\}$$

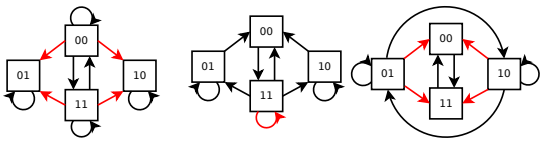

Fig. 5: States transitions diagrams corresponding to three semantics that do not respect Theorem 2 (in black) applied on the Boolean network of Figure 4. Using the synchronous semantics on the optimal program of the black transitions will produce in addition the red ones.

Figure 5 shows the dynamics of the Boolean network of Figure 4 under three semantics which dynamics cannot be reproduced using synchronous, asynchronous or general semantics on a program learned using **GULA**. In the first example (left), either all Boolean functions are applied simultaneously or nothing occurs (self-transition using projection). In the second example (center), the Boolean functions are applied synchronously but their is also always a possibility for any variable to take value 0 in the next state. In the third example (right), either the Boolean functions are applied synchronously, or each variable value is reversed (0 into 1 and 1 into 0). The original transitions of each dynamics are in black and the additional non-valid transitions in red. Using the original black transitions as input, **GULA** learns programs which, under the synchronous semantics (Definition 14), would realize the original black transitions plus the non-valid red ones. The idea is to learn constraints that would prevent those non-valid transitions to occur so that the observed dynamics is exactly reproduced using the synchronous constrained semantics of Definition 21.

Definition 22 (Conflict and Consistency of constraints) The constraint C *conflicts* with a set of transitions $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$ when $\exists (s, s') \in T, C \sqcap (s, s')$. C is said to be *consistent* with T when C does not conflict with T.

Therefore, a constraint is consistent if it does not match any transitions of T.

Definition 23 (Complete set of constraints) A set of constraints SC is *complete* with a set of transitions T if $\forall (s, s') \in (\mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}), (s, s') \notin T \implies \exists C \in SC, C \sqcap (s, s')$.

Definition 24 groups all the properties that we want the learned set of constraints to have: suitability and optimality, and Proposition 4 states that the optimal set of constraints of a set of transitions is unique.

Definition 24 (Suitable and optimal constraints) Let $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$. A set of $\mathcal{M}$VL constraints SC is *suitable* for T when:

- SC is consistent with T,
- SC is complete with T,
- for all constraints C not conflicting with T, there exists $C' \in P$ such that $C \leq C'$.

If in addition, for all $C \in SC$, all the constraint rules C' belonging to a set of constraints suitable for T are such that $C \leq C'$ implies $C' \leq C$, then SC is called *optimal*.

Proposition 4 *Let $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$. The optimal set of constraints for T is unique and denoted $C_{\mathcal{O}}(T)$.*

The subset of constraints of $C_{\mathcal{O}}(T)$ that prevent transitions permitted by $P_{\mathcal{O}}(T)$ but not observed in T from happening, or, in other terms, constraints that match transitions in $\mathcal{T}_{syn-c}(P_{\mathcal{O}}(T))) \setminus T$, is denoted $C'_{\mathcal{O}}(T)$ and given in Definition 25. All constraints of $C_{\mathcal{O}}(T)$ that are not in this set can never match a transition produced by $P_{\mathcal{O}}(T)$ with $\mathcal{T}_{syn-c}$ and can thus be considered useless. Finally, Theorem 8 shows that any set of transitions T can be reproduced, using synchronous constrained semantics of Definition 21 on the $\mathcal{CDM}$VLP $P_{\mathcal{O}}(T) \cup C'_{\mathcal{O}}(T)$.

Definition 25 Let $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$.

$C'_{\mathcal{O}}(T) := \{C \in C_{\mathcal{O}}(T) \mid \forall \mathrm{v}^{val} \in b(C), \mathrm{v} \in \mathcal{T}, \exists R \in P_{\mathcal{O}}(T), h(R) = \mathrm{v}^{val} \wedge \big(\forall \mathrm{w} \in \mathcal{F}, \forall val', val'' \in \mathsf{dom}(\mathrm{w}), \mathrm{w}^{val'} \in b(R) \wedge \mathrm{w}^{val''} \in b(C) \implies val' = val''\big)\}$.

Theorem 8 (Optimal $\mathcal{DM}$VLP and constraints correctness under synchronous constrainted semantics) *Let $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$, it holds that $T = \mathcal{T}_{syn-c}(P_{\mathcal{O}}(T) \cup C'_{\mathcal{O}}(T))$.*

5.2 Algorithm

In previous sections we presented a modified version of **GULA**: the General Usage LFIT Algorithm from [35], which takes as arguments a different set of variables for conditions and conclusions of rules. This modification allows to use this modified algorithm to learn constraints and thus $\mathcal{CDM}$VLP.

Algorithm 2 show the **Synchronizer** algorithm, which given a set of transitions $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$ will output $P_{\mathcal{O}}(T) \cup C'_{\mathcal{O}}(T)$ using **GULA** and the properties introduced in the previous section. With the new version of GULA it is possible to encode meaning in the transitions we give as input to the algorithm. The constraints we want to learn are technically rules whose head is ϵ^1 with conditions on both $\mathcal{F}$ and $\mathcal{T}$. It is sufficient to make the union of the two states of a transition and feed it to GULA to make it learn such rules. Constraints should match when an impossible transition is generated by the rules of the optimal program of T. GULA learns from negative examples and negative examples of impossible transitions are just the possible transitions, thus the transitions observed in T. Using the set of transitions $T' := \{(s \cup s', \{\varepsilon^0\}) \mid (s, s') \in T\}$ we can use GULA to learn such constraints with $GULA(\mathcal{A} \cup \{\varepsilon^1\}, T', \mathcal{F} \cup \mathcal{T}, \{\varepsilon\})$. Note that ε, from the algorithmic viewpoint, is just a dummy variable used to make every transition of T' a negative example of ε^1 which will be the only head of the rule we will learn here. The program produced will contain a set of rules that match none of the initial states of T' and thus none of the transitions of T but matches all other possible transitions according to GULA properties. Their head being ε^1, those rules are actually constraints over T. Since all and only such minimal rules are output by this second call to **GULA**, it correspond to $C_{\mathcal{O}}(T)$, which prevent every transitions that are not in T to be produced using the constraint synchronous semantics. Finally, the non-essential constraints can be discarded following Definition 25 and finally $P_{\mathcal{O}}(T) \cup C'_{\mathcal{O}}(T)$ is output. The source code of the algorithm is available at `https://github.com/Tony-sama/pylfit` under GPL-3.0 License.

Algorithm 2 **Synchronizer**

- **INPUT:** a set of atoms $\mathcal{A}$, a set of transitions $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$, two sets of variables $\mathcal{F}$ and $\mathcal{T}$.
- // 1) Learn what is possible locally in a transition using GULA
- $P := GULA(\mathcal{A}, T, \mathcal{F}, \mathcal{T})$
- // 2) Encode negative examples of constraints, i.e., observed transitions
- Let ε be a special variable not in the system: $\varepsilon \notin \mathcal{F} \cup \mathcal{T}$
- $T' := \{(s \cup s', \{\varepsilon^0\}) \mid (s, s') \in T\}$
- // 3) Learn what is impossible in form of constraint using GULA
- $P' := GULA(\mathcal{A}_{\mathcal{F} \cup \mathcal{T} \cup \{\varepsilon^1\}}, T', \mathcal{F} \cup \mathcal{T}, \{\varepsilon\})$
- $P'' := \{C \in P' \mid \forall \mathrm{v}^{val} \in b(C), \mathrm{v} \in \mathcal{T}, \exists R \in P, h(R) = \mathrm{v}^{val} \wedge \big(\forall \mathrm{w} \in \mathcal{F}, \forall val', val'' \in \mathsf{dom}(\mathrm{w}), \mathrm{w}^{val'} \in b(R) \wedge \mathrm{w}^{val''} \in b(C) \implies val' = val''\big)\}$
- **OUTPUT:** $P_{\mathcal{O}}(T) \cup C'_{\mathcal{O}}(T) := P \cup P''$.

Theorem 9 (Synchronizer correctness) *Given any set of transitions T,* **Synchronizer***$(\mathcal{A}, T, \mathcal{F}, \mathcal{T})$ outputs $P_{\mathcal{O}}(T) \cup C'_{\mathcal{O}}(T)$.*

From Theorem 8 and Theorem 9, given a set of transitions $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$, it holds that $\mathcal{T}_{syn-c}(Synchronizer(\mathcal{A}, T, \mathcal{F}, \mathcal{T})) = T$, i.e., the algorithm can be used to learn a $\mathcal{CDM}$VLP that reproduce exactly the input transitions whatever the semantics that produced them.

Theorem 10 (Synchronizer Complexity) *Let $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$ be a set of transitions, Let $n := max(|\mathcal{F}|, |\mathcal{T}|)$ and $d := \max(\{|\mathsf{dom}(\mathrm{v})| \in \mathbb{N} \mid \mathrm{v} \in \mathcal{F} \cup \mathcal{T}\})$ and $m := |\mathcal{F}| + |\mathcal{T}|$. The worst-case time complexity of* **Synchronizer** *when learning from T belongs to $\mathcal{O}((|T|^2 + |T| \times (2n^4 d^{2n+2} + 2n^3 d^{n+1})) + (|T|^2 + |T| \times (2m^4 d^{2m+2} + 2m^3 d^{m+1})))$ and its worst-case memory use belongs to $\mathcal{O}((d^{2n} + 2nd^{n+1} + nd^{n+2}) + (d^{2m} + 2md^{m+1} + md^{m+2}))$.*

6 Evaluation

In this section, both the scalability and the accuracy of **GULA** are evaluated using Boolean network benchmarks from the biological literature. The scalability of **Synchronizer** is also evaluated (details are given in appendix). All experiments[1] were conducted on one core of an AMD Ryzen 7 (2700X, 3.7 GHz) with 64 Gb of RAM.

6.1 **GULA** Scalability

Benchmark	size	synchronous	asynchronous	general
arellano_rootstem	9	2s/1.8s/0.9s/0.3s/512	2.4s/1.4s/1.1s/0.2s/1,940	1.1s/0.5s/0.3s/0.3s/11K
davidich_yeast	10	16s/10s/4s/0.6s/1,024	12s/6s/4s/0.5s/4,364	3s/1.5s/1s/0.9s/39K
faure_cellcycle	10	15s/10s/4s/0.8s/1,024	12s/5.6s/4.7s/0.6s/4,273	4s/1.2s/0.9s/0.9s/31K
fission_yeast	10	16s/10s/4.8s/0.8s/1,024	12s/5.8s/4.6s/0.4s/4,157	3.6s/1.2s/1s/0.8s/34K
mammalian	10	14.8s/11s/4.8s/0.8s/1,024	12s/5.7s/3.4s/0.6s/4,273	3.4s/1.4s/1s/0.9s/31K
budding_yeast	12	564s/194s/61s/3.7s/4,096	216s/107s/85s/2.6s/20K	51s/14s/5.9s/4.1s/260K
n12c5	12	468s/200s/64s/2.8s/4,096	213s/103s/144s/1.3s/30K	4.7s/6s/8.6s/11s/1,122K
tournier_apoptosis	12	369s/164s/54s/2.7s/4,096	199s/98s/94s/2s/22K	26s/6.7s/4.6s/4.6s/358K
dinwoodie_stomatal	13	-/748s/221s/6.1s/8,192	-/548s/628s/4s/53K	70s/18s/15s/18s/1.5M
multivalued	13	-/-/406s/6s/8,192	-/565s/765s/4.9s/49K	61s/18s/13s/13s/1M
saadatpour_guardcell	13	-/757s/219s/6s/8,192	-/575s/638s/4.2s/53K	68s/17s/15s/18s/1.5M
arabidopsis	15	-/-/-/53s/32K	-/-/-/50s/213K	-/352s/123s/103s/7M
dinwoodie_life	15	-/-/-/37s/32K	-/-/-/30s/245K	-/352s/240s/256s/20M
randomnet_n15k3	15	-/-/-/51s/32K	-/-/-/31s/262K	731s/219s/226s/280s/22M
irons_yeast	18	-/-/-/653s/262K	-/-/-/324s/2M	memory out

Table 1: Run time of **GULA** for 9 to 18 nodes Boolean networks of [9,23] for the three semantics: run time in seconds for 25%/50%/75%/100% of the transitions as input, and total number of transitions (K for thousands and M for millions).

Table 1 shows the run time of **GULA** when learning from the transitions of Boolean networks from Boolenet [9] and PyBoolnet [23]. Boolean networks

[1] Available at: `https://github.com/Tony-sama/pylfit`. Using command "`python3 src/evaluations/mlj2020/mlj2020_all.py`" from the repository's root folder, results will be in the `/tmp` folder.

are converted to $\mathcal{DM}$VLP where $\forall v \in \mathcal{V}, \mathsf{dom}(v) = \{0, 1\}$. For each variable, Boolean functions are given in disjonctive normal form (DNF), a disjonction of conjonction clauses that can be considered as a set of Boolean atoms of the form v or $\neg$v. Each clause c of the DNF of a variable v is directly converted into a rule R such that, $h(R) = \mathrm{v}^1$ and $\mathrm{v}'^1 \in b(R) \iff \mathrm{v}' \in c$ and $\mathrm{v}'^0 \in b(R) \iff \neg\mathrm{v}' \in c$. For each such $\mathcal{DM}$VLP the set T of all transitions are generated for the three considered semantics (see Section 3). For each generation, to simulate the cases where Boolean functions are false, each semantics uses a default function that gives $\mathrm{v}^0, \forall \mathrm{v} \in \mathcal{T}$ when no rule $R, \mathrm{v}(h(R)) = \mathrm{v}$ matches a state. Learning is performed on several random subsets of 25%/50%/75%/100% of the whole set of transitions. The run time needed by the algorithm to learn $P_{\mathcal{O}}(T)$ is reported for each case.

We observe that for each benchmark we get a better run time if we are given more input transitions. More transitions possibly implies more specialization of non-optimal rules, increasing the chance for them to be dominated by another rule, thus reducing the number of rules to compare. The same reasoning applies between the semantics. It is important to note that those systems are deterministic with the synchronous semantics and thus the number of transitions in the synchronous case is much lower than for the two other semantics. The rules are simpler for the two other semantics since rules of the form $\mathrm{v}_t^{val} \leftarrow \mathrm{v}_{t-1}^{val}$ are always consistent and quickly obtained. Such simple rules have great dominance power, reducing the quantity of rules stored and thus checked for domination at each step.

GULA succeeds in learning the benchmarks with less than 12 variables for all semantics before the time-out ("-" in Table 1) of 1,000 seconds for all subsets of transitions. Run times of smaller benchmarks from the same sources (3 to 7 variables) are omitted in the table since they are lower than one second in all cases. Benchmarks from 13 variables need a substantial amount of input transitions to prevent the explosion of consistent rules and thus reaching the time out. The 18 variables benchmark could be learned for both the synchronous and asynchronous semantics. For the general semantics, however, the number of transitions generated (about 80 millions) is too large for our naive usage of the memory. The current implementation of the algorithm is rather naive and better performances are expected from future optimizations. In particular, the algorithm can be parallelized into as many threads as the number of different rule heads (one thread per target variable value). We are also developing[2] an approximated version of **GULA** that outputs a subset of $P_{\mathcal{O}}(T)$ sufficient to realize T [36]. The complexity of this new algorithm is polynomial, greatly improving the scalability of our approach. Because of space limitations we could not incorporate this algorithm and its evaluation in this paper.

Learning constraints is obviously more costly than learning regular rules since both feature and target variables can appear in the body, i.e., the num-

[2] The polynomial approximation of **GULA**, currently named **PRIDE** is also available at: `https://github.com/Tony-sama/pylfit`

ber of features becomes $|\mathcal{F}| + |\mathcal{T}|$. Under the same experimental settings, the **Synchronizer** reached the time-out of 1,000 seconds on the benchmarks of 9 nodes. The contribution regarding $\mathcal{CDM}$VLP being focused on theoretical results, we provided the detailed evaluation of the **Synchronizer** in appendix to save space.

6.2 **GULA** Predictive power

In this experiment, we evaluate the quality of the models learned by **GULA** in their ability to correctly predict possible values for each variable from unseen initial states. Regarding our modeling, it consists in learning an approximation of $P_{\mathcal{O}}(T)$ and check both consistency and realization of T. For each Boolean network benchmark, the set T of all possible transitions are generated as in the previous experiment. First the transitions are grouped by initial state and 10% to 90% are chosen randomly to form a training set and the rest for a test set so that $\text{first}(training) \cap \text{first}(test) = \emptyset$ and $training \cup test = T$ (except for for 100% where $training = T = test$). The training set is given as input to **GULA** and its output program is used to predict the possible values appearing in the next states from the initial states of the test set. Figure 6 (left) shows the accuracy of the predicted possible values w.r.t. the ratio of training data going from 10% to 100% with the synchronous semantics.

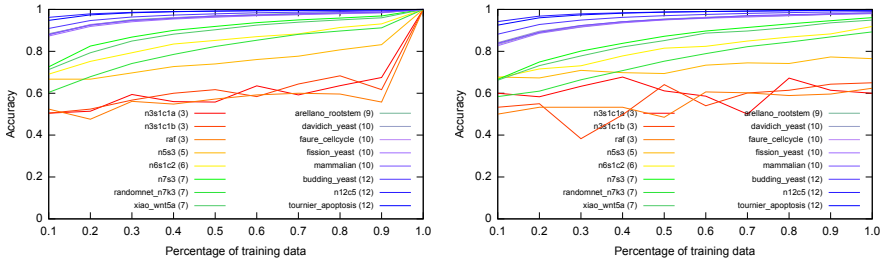

Fig. 6: Accuracy of the models learned by **GULA** when predicting possible target variable values from unseen states with different amounts of training data in the synchronous semantics in two different settings: (left) experiment 1, with a complete set of input transitions from a partial number of initial states; and (right) experiment 2, with a potentially incomplete set of input transitions from an incomplete set of initial states.

In this first experiment, the input of **GULA** is complete for the initial states observed, i.e., every transitions from all $s \in \text{first}(training)$ are given. For the second experiment, the algorithm is provided an incomplete input, i.e., for some $s \in \text{first}(training)$, $\exists (s, s') \in T, (s, s') \notin training$. Thus in the second case, having missing transitions from some initial state is a form of noise. The test set must remain complete to ensure a correct evaluation. Thus, as in the

first experiment, transitions are grouped by initial states. Here, 20% of the initial states are randomly chosen to form a complete test set with all their outgoing transitions. From the remaining 80% of initial states of T, 10% to 100% of the transitions with such initial states are randomly chosen as training set and used as input by **GULA**. Figure 6 (right) shows the accuracy of the predicted possible values w.r.t. the ratio of input data with the synchronous semantics.

In order to predict from unseen states, for each target atom we need rules that assert possibility and rules that assert impossibility. The first kind of rules can be learned from a regular call to **GULA** on the training transitions and form a first $\mathcal{DM}$VLP P. To obtain impossibility rules we just have to reverse the computation of negative example in the algorithm so that: $\forall \mathrm{v}^{val} \in \mathcal{A}_{\mathcal{T}}, Neg_{\mathrm{v}^{val}} := \{s \mid \exists (s, s') \in T, \mathrm{v}^{val} \in s'\}$. Rules learned using least revision from this $Neg_{\mathrm{v}^{val}}$ will match when v^{val} is not possible in the next state and form a second $\mathcal{DM}$VLP $\overline{P}$. Both types of rules are then weighted by counting the number of initial state they match in the training set: let R be a rule, then $weight(R, T) := |\{s \mid \exists (s, s') \in T, R \sqcap s, h(R) \in s'\}|$. To forecast how likely each atom is to appear in a transition from an unseen state s, we take the matching rules of P and $\overline{P}$ with the maximal weight, as follows. Let $s \in \mathcal{S}^{\mathcal{F}}$, then $forecast(\mathrm{v}^{val}, s, P, \overline{P}, T) = 0.5 + \frac{p - np}{2(p + np)}$ be the forecast probabilities of v^{val} being in some next state of s according to $P, \overline{P}$ and T with $p := max(\{0\} \cup \{weight(R, T) \mid R \in P, h(R) = \mathrm{v}^{val}, b(R) \subseteq s\})$ and let $np := max(\{0\} \cup \{weight(R, T) \mid R \in \overline{P}, h(R) = \mathrm{v}^{val}, b(R) \subseteq s\})$. The forecast probabilities are compared to the observed values of the test set. Let $actual(\mathrm{v}^{val}, s, T) = \begin{cases} 1, & \text{if } \exists (s, s') \in T, \mathrm{v}^{val} \in s' \\ 0, & \text{otherwise} \end{cases}$. Given $T, T' \in \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$, $P, \overline{P} \in \mathcal{DM}$VLP, the accuracy of prediction from $P, \overline{P}$ according to T over T' is:

$$accuracy(P, \overline{P}, T, T') = 1 - \frac{1}{|\text{first}(T')|} \sum_{s \in \text{first}(T')} error(P, \overline{P}, T, T', s) \text{ , with}$$

$$error(P, \overline{P}, T, T', s) = \frac{1}{|\mathcal{T}|} \sum_{\mathrm{v}^{val} \in \mathcal{T}} \frac{abs(actual(\mathrm{v}^{val}, s, T') - forecast(\mathrm{v}^{val}, s, P, \overline{P}, T))}{abs(actual(\mathrm{v}^{val}, s, T'))}$$

The evolution of $accuracy(P, \overline{P}, training, test)$ is shown in Figure 6 for experiments 1 (left) and 2 (right). For both experiments, we observe poor results for the smallest benchmarks of 3 to 5 variables unless most transitions are given. In those cases, the number of training samples are too low for the weighting heuristic to help choices between possibility and impossibility rules. Starting from 7 variables, we observe much better results. From 10 variables, 80% accuracy can be achieved with no more than 10% of the possible transitions as training examples. Performance is quite similar in both experiments, showing that our method can handle some noise caused by missing observations. We observed similar results with asynchronous and general semantics and thus did not presented it here because of the lack of space.

If one is only interested by the prediction, it is certainly easier to achieve better results using other methods like neural networks or random forest since prediction here is basically a binary classification for each target variables

values. In the case where explainability is of interest, the rules used for the predictions and their weights may be quite simple human readable candidates for explanations. Furthermore, when good prediction model can be built from training data, it can replace our learned model to forecast next state but it can also be used to improve the output of **GULA**. Indeed, one can use such models to produce artificial transitions from both observed and unseen states that can be given as input to **GULA** in place of the raw observations. It can help to deal with noisy data and improve the diversity of initial state that can speed up and improve the quality of the rules of **GULA** and thus also its approximated version [36]. Actually, as long as feature and target variables are discrete (or can be properly discretized), **GULA** (or its approximated version for big systems) could be used to generate rules that could explain in a more human readable fashion the behavior of other less explainable models. Such a combination study is out of the scope of this paper but will be an interesting application part of our future works.

7 Related Work

7.1 Modeling Dynamics

In modeling of dynamical systems, the notion of concurrency is crucial. Historically, two main dynamical semantics have been used in the field of systems biology: synchronous (Boolean networks of Stuart Kauffman [20]) and asynchronous (René Thomas' networks [42]), although other semantics are sometimes proposed or used [12].

The choice of a given semantics has a major impact on the dynamical features of a model: attractors, trap domains, bifurcations, oscillators, etc. The links between modeling frameworks and their update semantics constitute the scope of an increasing number of papers. In [15], the author exhibited the translation from Boolean networks into logic programs and discussed the point attractors in both synchronous and asynchronous semantics. In [30], the authors studied the synchronism-sensitivity of Boolean automata networks with regard to their dynamical behavior (more specifically their asymptotic dynamics). They demonstrate how synchronism impacts the asymptotic behavior by either modifying transient behaviors, making attractors grow or destroying complex attractors. Meanwhile, the respective merits of existing synchronous, asynchronous and generalized semantics for the study of dynamic behaviors has been discussed by Chatain and Paulevé in a series of recent papers. In [7], they introduced a new semantics for Petri nets with read arcs, called the interval semantics. Then they adapted this semantics in the context of Boolean networks [8], and showed in [6] how the interval semantics can capture additional behaviors with regard to the already existing semantics. Their most recent work demonstrates how the most common synchronous and asynchronous semantics in Boolean networks have three major drawbacks that are to be costly for any analysis, to miss some behaviors and to predict spurious behaviors. To

overcome these limits, they introduce a new paradigm, called *Most Permissive Boolean Network* which offers the guarantee that no realizable behavior by a qualitative model will be missed [33].

The choice of a relevant semantics appears clearly not only in the recent theoretical works bridging the different frameworks, but also in the features of the software provided to the persons involved in Systems Biology modeling (e.g., the GinSIM tool offers two updating modes, that are fully synchronous and fully asynchronous [29]). Analysis tools offer the modelers the choice of the most appropriate semantics with regard to their own problem.

7.2 Learning Dynamics

In this paper, we proposed new algorithms to learn the dynamics of a system independently of its update semantics, and apply it to learn Boolean networks from the observation of their states transitions. Learning the dynamics of Boolean networks has been considered in bioinformatics in several works [26,1,31,24,12]. In [12], Fages discussed the differential semantics, stochastic semantics, Boolean semantics, hybrid (discrete and continuous) semantics, Petri net semantics, logic programming semantics and some learning techniques. Rather than focusing on particular semantics, our learning methods are complete algorithms that learn transition rules for any memory-less discrete dynamical systems independently of the update semantics. As in [31], we can also deal with partial transitions, but will not need to identify or enumerate all possible complete transitions. [32] learns a model as a probability distribution for the next state given the previous state and an action. Here, exactly one dynamic rule fires every time-step, which corresponds to the asynchronous semantics of Definition 15. In [41], action rules are learned using inductive logic programming but require as input background knowledge. In [3], the authors use logic program as a meta-interpreter to explain the behaviour of a system as stepwise transitions in Petri nets. They produce new possible traces of execution, while our output is an interaction model of the system that aims to explain the observed behavior. In [22], Klarner *et al.* provide an optimization-based method for computing model reduction by exploiting the prime implicant graph of the Boolean network. This graph is similar to the rules of $P_{\mathcal{O}}(T)$ that can be learned by **GULA**. But while [22] requires an existing model to work, we are able to learn this model from observations. In [24], Lähdesmäki *et al.* propose algorithms to infer the truth table of Boolean functions of gene regulatory network from gene expression data. Each positive (resp. negative) example represents a variable configuration that makes a Boolean function true (resp. false). The logic programs learned by **GULA** are a generalization of those truth tables. [10,11] propose the Apperception Engine, a system able to learn programs from a sequence of state transitions. The first difference is that our approach is limited to propositional atoms while first order logic is considered in this approach. But our input can represent transitions from multiple trajectories, while they consider

a single trajectory and thus our setting can be considered as a generalized apperception task. Another major difference is that they only consider deterministic inputs while we also capture non-deterministic behaviors. Given the same kind of single trajectory and a $\mathcal{DM}$VLP (or $\mathcal{CDM}$VLP), it should be possible to produce candidates past states or to try to fill in missing values. But in practice that would suppose to have many other transitions to build such $\mathcal{DM}$VLP using **GULA** while the Aperception Engine can perform the task with only the given single trajectory. This system can also produce a set of constraints as well as rules. The constraints of $\mathcal{CDM}$VLP can prevent some combinations of atoms to appear, but only in next states, while in [10,11], constraints can prevent some states to exist anywhere in the sequence, and ensure the conservation of atoms. From Theorem 8, the conservation can also be reproduced by $\mathcal{CDM}$VLP by the right combination of optimal rules and constraints. In [25] the authors propose a general framework named ILASP for learning answer set programs. ILASP is able to learn choice rules, constraints and preferences over answer sets. Our problem settings is related to what is called "context-dependant" tasks in ILASP. Our input can be straightforwardly represented using ILASP when variables are Boolean, but the learned program does not respect our notion of optimality, and thus our learning goals differ, i.e., we guarantee to miss no potential dynamical influence. [19] proposes an incremental method to learn and revise event-based knowledge in the form of Event Calculus programs using XHAIL [34], a system that jointly abduce ground atoms and induce first-order normal logic programs. XHAIL needs to be provided with a set of mode declarations to limit the search space of possible induced rules, while our method do not require background knowledge. Still it is possible to exploit background knowledge with **GULA**: for example one could add heuristic inside the algorithm to discard rules with "too many" conditions; influences among variables, if known, could also be exploited to reduce possible bodies. Finally, XHAIL does not model constraints, thus is not able to prevent some combinations of atoms to appear in transitions, which can be achieve using our **Synchronizer**.

8 Conclusions

While modeling a dynamical system, the choice of a proper semantics is critical for the relevance of the subsequent analysis of the dynamics. The works presented in this paper aim to widen the possibilities offered to a system designer in the learning phase. Until now, the systems that the LFIT framework handles were restricted to synchronous deterministic dynamics. However, many other dynamics exist in the field of logical modeling, in particular the asynchronous and generalized semantics which are of deep interest to model biological systems. In this paper, we proposed a modeling of memory-less multi-valued dynamic systems in the form of annotated logic programs and a first algorithm, **GULA**, that learns optimal programs for a wide range of semantics (see Theorem 2) including notably the asynchronous and generalized semantics. But

the semantics need to be assumed to use the learned model, in order to produce predictions for example. Our second proposition is a new approach that makes a decisive step in the full automation of logical learning of models directly from time series, e.g., gene expression measurements along time (whose intrinsic semantics is unknown or even changeable). The **Synchronizer** algorithm that we proposed is able to learn a whole system dynamics, including its semantics, in the form of a single propositional logic program. This logic program explains the behavior of the system in the form of human readable propositional logic rules, as well as, be able to reproduce the behavior of the observed system without the need of knowing its semantics. Furthermore, the semantics can be explained, without any previous assumption, in the form of human readable rules inside the logic program.

After having exhibited the benefits of our approach on several benchmarks, further work will consist in a practical use of our method on open problems coming from systems biology. An approximate version of the method is a necessity to tackle large systems and is under development [36]. In addition, lack of observations and noise handling is also an issue when working with biological data. Data science methodologies and deep learning techniques can then be good candidate to tackle this challenge. The combination of such techniques to improve our method may be of prime interest to tackle real data.

References

1. Akutsu, T., Kuhara, S., Maruyama, O., Miyano, S.: Identification of genetic networks by strategic gene disruptions and gene overexpressions under a boolean model. Theoretical Computer Science **298**(1), 235–251 (2003)
2. Apt, K.R., Blair, H.A., Walker, A.: Towards a theory of declarative knowledge. Foundations of deductive databases and logic programming p. 89 (1988)
3. Bain, M., Srinivasan, A.: Identification of biological transition systems using meta-interpreted logic programs. Machine Learning **107**(7), 1171–1206 (2018)
4. Blair, H.A., Subrahmanian, V.: Paraconsistent foundations for logic programming. Journal of non-classical logic **5**(2), 45–73 (1988)
5. Blair, H.A., Subrahmanian, V.: Paraconsistent logic programming. Theoretical Computer Science **68**(2), 135 – 154 (1989). DOI http://dx.doi.org/10.1016/0304-3975(89)90126-6. URL `http://www.sciencedirect.com/science/article/pii/0304397589901266`
6. Chatain, T., Haar, S., Kolčák, J., Paulevé, L., Thakkar, A.: Concurrency in boolean networks. Natural Computing **19**(1), 91–109 (2020)
7. Chatain, T., Haar, S., Koutny, M., Schwoon, S.: Non-atomic transition firing in contextual nets. In: International Conference on Applications and Theory of Petri Nets and Concurrency, pp. 117–136. Springer (2015)
8. Chatain, T., Haar, S., Paulevé, L.: Boolean networks: Beyond generalized asynchronicity. In: AUTOMATA 2018. Springer (2018)
9. Dubrova, E., Teslenko, M.: A SAT-based algorithm for finding attractors in synchronous boolean networks. IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB) **8**(5), 1393–1399 (2011)
10. Evans, R., Hernandez-Orallo, J., Welbl, J., Kohli, P., Sergot, M.: Making sense of sensory input. arXiv preprint 1910.02227 (2019)
11. Evans, R., Hernandez-Orallo, J., Welbl, J., Kohli, P., Sergot, M.: Evaluating the apperception engine. arXiv preprint 2007.05367 (2020)

12. Fages, F.: Artificial intelligence in biological modelling. In: A Guided Tour of Artificial Intelligence Research, pp. 265–302. Springer (2020)
13. Fitting, M.: Bilattices and the semantics of logic programming. The Journal of Logic Programming **11**(2), 91 – 116 (1991). DOI http://dx.doi.org/10.1016/0743-1066(91)90014-G. URL http://www.sciencedirect.com/science/article/pii/074310669190014G
14. Ginsberg, M.L.: Multivalued logics: A uniform approach to reasoning in artificial intelligence. Computational intelligence **4**(3), 265–316 (1988)
15. Inoue, K.: Logic programming for boolean networks. In: Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence, *IJCAI'11*, vol. 2, p. 924–930. AAAI Press (2011)
16. Inoue, K., Ribeiro, T., Sakama, C.: Learning from interpretation transition. Machine Learning **94**(1), 51–79 (2014)
17. Inoue, K., Sakama, C.: Oscillating behavior of logic programs. In: Correct Reasoning, pp. 345–362. Springer (2012)
18. Kaplan, S., Bren, A., Dekel, E., Alon, U.: The incoherent feed-forward loop can generate non-monotonic input functions for genes. Molecular systems biology **4**(1), 203 (2008)
19. Katzouris, N., Artikis, A., Paliouras, G.: Incremental learning of event definitions with inductive logic programming. Machine Learning **100**(2-3), 555–585 (2015)
20. Kauffman, S.A.: Metabolic stability and epigenesis in randomly constructed genetic nets. Journal of theoretical biology **22**(3), 437–467 (1969)
21. Kifer, M., Subrahmanian, V.: Theory of generalized annotated logic programming and its applications. Journal of Logic Programming **12**(4), 335–367 (1992)
22. Klarner, H., Bockmayr, A., Siebert, H.: Computing symbolic steady states of boolean networks. In: Cellular Automata, pp. 561–570. Springer (2014)
23. Klarner, H., Streck, A., Siebert, H.: PyBoolNet: a python package for the generation, analysis and visualization of boolean networks. Bioinformatics **33**(5), 770–772 (2016). DOI 10.1093/bioinformatics/btw682. URL https://doi.org/10.1093/bioinformatics/btw682
24. Lähdesmäki, H., Shmulevich, I., Yli-Harja, O.: On learning gene regulatory networks under the boolean network model. Machine Learning **52**(1-2), 147–167 (2003)
25. Law, M., Russo, A., Broda, K.: Iterative learning of answer set programs from context dependent examples. Theory and Practice of Logic Programming **16**(5-6), 834–848 (2016). DOI 10.1017/S1471068416000351
26. Liang, S., Fuhrman, S., Somogyi, R.: Reveal, a general reverse engineering algorithm for inference of genetic network architectures. In: Proceedings of the 3rd Pacific Symposium on Biocomputing, pp. 18–29 (1998)
27. Martınez, D., Alenya, G., Torras, C., Ribeiro, T., Inoue, K.: Learning relational dynamics of stochastic domains for planning. In: Proceedings of the 26th International Conference on Automated Planning and Scheduling (2016)
28. Martínez Martínez, D., Ribeiro, T., Inoue, K., Alenyà Ribas, G., Torras, C.: Learning probabilistic action models from interpretation transitions. In: Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015), pp. 1–14 (2015)
29. Naldi, A., Hernandez, C., Abou-Jaoudé, W., Monteiro, P.T., Chaouiya, C., Thieffry, D.: Logical modeling and analysis of cellular regulatory networks with ginsim 3.0. Frontiers in physiology **9**, 646 (2018)
30. Noual, M., Sené, S.: Synchronism versus asynchronism in monotonic boolean automata networks. Natural Computing **17**(2), 393–402 (2018)
31. Pal, R., Ivanov, I., Datta, A., Bittner, M.L., Dougherty, E.R.: Generating boolean networks with a prescribed attractor structure. Bioinformatics **21**(21), 4021–4025 (2005)
32. Pasula, H.M., Zettlemoyer, L.S., Kaelbling, L.P.: Learning symbolic models of stochastic domains. Journal of Artificial Intelligence Research **29**, 309–352 (2007)
33. Paulevé, L., Kolčák, J., Chatain, T., Haar, S.: Reconciling qualitative, abstract, and scalable modeling of biological networks. bioRxiv (2020)
34. Ray, O.: Nonmonotonic abductive inductive learning. Journal of Applied Logic **7**(3), 329–340 (2009)

35. Ribeiro, T., Folschette, M., Magnin, M., Roux, O., Inoue, K.: Learning dynamics with synchronous, asynchronous and general semantics. In: International Conference on Inductive Logic Programming, pp. 118–140. Springer (2018)
36. Ribeiro, T., Folschette, M., Trilling, L., Glade, N., Inoue, K., Magnin, M., Roux, O.: Les enjeux de l'inférence de modèles dynamiques des systèmes biologiques à partir de séries temporelles. In: C. Lhoussaine, E. Remy (eds.) Approches symboliques de la modélisation et de l'analyse des systèmes biologiques. ISTE Editions (2020). In edition.
37. Ribeiro, T., Inoue, K.: Learning prime implicant conditions from interpretation transition. In: Inductive Logic Programming, pp. 108–125. Springer (2015)
38. Ribeiro, T., Magnin, M., Inoue, K., Sakama, C.: Learning delayed influences of biological systems. Frontiers in Bioengineering and Biotechnology **2**, 81 (2015)
39. Ribeiro, T., Magnin, M., Inoue, K., Sakama, C.: Learning multi-valued biological models with delayed influence from time-series observations. In: 2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA), pp. 25–31 (2015). DOI 10.1109/ICMLA.2015.19
40. Ribeiro, T., Tourret, S., Folschette, M., Magnin, M., Borzacchiello, D., Chinesta, F., Roux, O., Inoue, K.: Inductive learning from state transitions over continuous domains. In: N. Lachiche, C. Vrain (eds.) Inductive Logic Programming, pp. 124–139. Springer International Publishing, Cham (2018)
41. Schüller, P., Benz, M.: Best-effort inductive logic programming via fine-grained cost-based hypothesis generation. Machine Learning **107**(7), 1141–1169 (2018)
42. Thomas, R.: Regulatory networks seen as asynchronous automata: a logical description. Journal of Theoretical Biology **153**(1), 1–23 (1991)
43. Van Emden, M.H.: Quantitative deduction and its fixpoint theory. The Journal of Logic Programming **3**(1), 37–53 (1986)
44. Van Emden, M.H., Kowalski, R.A.: The semantics of predicate logic as a programming language. Journal of the ACM (JACM) **23**(4), 733–742 (1976)

A Appendix: proofs of Section 2.2

Theorem 11 (Theorem 1: double domination is equality) *Let R_1, R_2 be two $\mathcal{M}$VL rules. If $R_1 \leq R_2$ and $R_2 \leq R_1$ then $R_1 = R_2$.*

Proof. Let R_1, R_2 be two $\mathcal{M}$VL rules such that $R_1 \leq R_2$ and $R_2 \leq R_1$. Then $h(R_1) = h(R_2)$ and $b(R_1) \subseteq b(R_2)$ and $b(R_2) \subseteq b(R_1)$, hence $b(R_1) \subseteq b(R_2) \subseteq b(R_1)$ thus $b(R_1) = b(R_2)$ and $R_1 = R_2$. □

Proposition 5 (Proposition 1: uniqueness of optimal program) *Let $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$. The $\mathcal{M}$VLP optimal for T is unique and denoted $P_{\mathcal{O}}(T)$.*

Proof. Let $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$. Assume the existence of two distinct $\mathcal{M}$VLPs optimal for T, denoted by $P_{\mathcal{O}_1}(T)$ and $P_{\mathcal{O}_2}(T)$ respectively. Then w.l.o.g. we consider that there exists a $\mathcal{M}$VL rule R such that $R \in P_{\mathcal{O}_1}(T)$ and $R \notin P_{\mathcal{O}_2}(T)$. By the definition of a suitable program, R is not conflicting with T and there exists a $\mathcal{M}$VL rule $R_2 \in P_{\mathcal{O}_2}(T)$, such that $R \leq R_2$. Using the same definition, there exists $R_1 \in P_{\mathcal{O}_1}(T)$ such that $R_2 \leq R_1$ since R_2 is not conflicting with T. Thus $R \leq R_1$ and by the definition of an optimal program $R_1 \leq R$. By Theorem 1, $R_1 = R$ and thus $R \leq R_2 \leq R$ hence $R_2 = R$, a contradiction. □

B Appendix: proofs of Section 3

Theorem 12 (Theorem 2: Pseudo-idempotent Semantics and Optimal $\mathcal{DM}$VLP) *Let DS be a dynamical semantics. For all P a $\mathcal{DM}$VLP, if:*

- $\exists \mathsf{pick} \in (\mathcal{S}^{\mathcal{F}} \times \wp(\mathcal{A}_{\mathcal{T}}) \to \wp(\mathcal{S}^{\mathcal{T}}) \setminus \{\emptyset\})$ *so that*
  (1) $\forall D \subseteq \mathcal{A}_{\mathcal{T}}, \mathsf{pick}(s, \bigcup_{s' \in \mathsf{pick}(s,D)} s') = \mathsf{pick}(s, D) \wedge$
  (2) $\forall s \in \mathcal{S}^{\mathcal{F}}, \big(DS(P)\big)(s) = \mathsf{pick}(s, \mathsf{Ccl}(s, P))$,

then: for all P a $\mathcal{DM}$VLP, $DS(P_{\mathcal{O}}(DS(P)))) = DS(P)$.

Proof.

Let DS be a dynamical semantics, P a $\mathcal{DM}$VLP, pick a function from $\mathcal{S}^{\mathcal{F}} \times \wp(\mathcal{A}_{\mathcal{T}})$ to $\wp(\mathcal{S}^{\mathcal{T}}) \setminus \{\emptyset\}$ with the properties described in (1) and (2).

In this proof, we use the following equivalent notations, for all $(s, s') \in \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$: $(s, s') \in DS(P) \iff s' \in \big(DS(P)\big)(s)$.

By Definition 10, $\text{first}(DS(P)) = \mathcal{S}^{\mathcal{F}}$ $(*)$.

By Definition 9, $P_{\mathcal{O}}(DS(P))$ realizes $DS(P)$. Therefore, according to Definition 5, for all (s, s') in $DS(P)$ and v^{val} in s', because $\mathrm{v} \in \mathcal{T}$, there exists R in $P_{\mathcal{O}}(DS(P))$ so that $\text{var}(h(R)) = \mathrm{v} \wedge R \sqcap s \wedge h(R) \in s'$. Because of Definition 3, a discrete state cannot contain two different atoms on the same variable: from $\text{var}(h(R)) = \mathrm{v} \wedge \mathrm{v}^{val} \in s' \wedge h(R) \in s'$, it comes: $h(R) = \mathrm{v}^{val}$. Moreover, by definition of Ccl, because $R \in P \wedge R \sqcap s$, we have: $\mathrm{v}^{val} \in \mathsf{Ccl}(s, P_{\mathcal{O}}(DS(P)))$. By generalizing on all v^{val}, it comes: $s' \subseteq \mathsf{Ccl}(s, P_{\mathcal{O}}(DS(P)))$. By generalizing on all s', it comes: $\forall s \in \mathcal{S}^{\mathcal{F}}, \bigcup_{s' \in (DS(P))(s)} s' \subseteq \mathsf{Ccl}(s, P_{\mathcal{O}}(DS(P)))$ (†).

By Definition 9, $P_{\mathcal{O}}(DS(P))$ is also consistent with $DS(P)$. Therefore, according to Definition 7: $\forall R \in P_{\mathcal{O}}(DS(P)), \forall s \in \text{first}(DS(P)), R \sqcap s \implies \exists s' \in \big(DS(P)\big)(s), h(R) \in s'$. From $(*)$, $\text{first}(DS(P)) = \mathcal{S}^{\mathcal{F}}$, thus $\forall s \in \mathcal{S}^{\mathcal{F}}, \forall \mathrm{v}^{val} \in \mathsf{Ccl}(s, P_{\mathcal{O}}(DS(P))), \exists s' \in DS(P)(s), \mathrm{v}^{val} \in s'$. Thus: $\forall s \in \mathcal{S}^{\mathcal{F}}, \mathsf{Ccl}(s, P_{\mathcal{O}}(DS(P))) \subseteq \bigcup_{s' \in (DS(P))(s)} s'$ (§).

From (†) and (§): $\forall s \in \mathcal{S}^{\mathcal{F}}, \mathsf{Ccl}(s, P_{\mathcal{O}}(DS(P))) = \bigcup_{s' \in (DS(P))(s)} s'$ $(\star)$.

From $(\star)$ and (2): $\forall s \in \mathcal{S}^{\mathcal{F}}, \mathsf{Ccl}(s, P_{\mathcal{O}}(DS(P))) = \bigcup_{s' \in \mathsf{pick}(s, \mathsf{Ccl}(s,P))} s'$ $(\Diamond)$.

Let s in $\mathcal{S}^{\mathcal{F}}$.

– From (2): $\big(DS(P_{\mathcal{O}}(DS(P)))\big)(s) = \mathsf{pick}(s, \mathsf{Ccl}(s, P_{\mathcal{O}}(DS(P))))$.

- From ($\Diamond$): $\big(DS(P_{\mathcal{O}}(DS(P)))\big)(s) = \mathsf{pick}(s, \bigcup_{s' \in \mathsf{pick}(s, \mathsf{Ccl}(s,P))} s'$
- From (1): $\big(DS(P_{\mathcal{O}}(DS(P)))\big)(s) = \mathsf{pick}(s, \mathsf{Ccl}(s, P))$
- From (2): $\big(DS(P_{\mathcal{O}}(DS(P)))\big)(s) = \big(DS(P)\big)(s)$.

Thus: $\forall s \in \mathcal{S}^{\mathcal{F}}, \big(DS(P_{\mathcal{O}}(DS(P)))\big)(s) = \big(DS(P)\big)(s)$, QED. □

Theorem 13 (Theorem 3 Semantics-free correctness) *Let P be a $\mathcal{DM}$VLP.*

- $\mathcal{T}_{syn}(P) = \mathcal{T}_{syn}(P_{\mathcal{O}}(\mathcal{T}_{syn}(P)))$,
- $\mathcal{T}_{asyn}(P) = \mathcal{T}_{asyn}(P_{\mathcal{O}}(\mathcal{T}_{asyn}(P)))$,
- $\mathcal{T}_{gen}(P) = \mathcal{T}_{gen}(P_{\mathcal{O}}(\mathcal{T}_{gen}(P)))$.

Proof. Let $d \in (\mathcal{S}^{\mathcal{F}} \times \wp(\mathcal{T}) \to \wp(\mathcal{A}_{\mathcal{T}}))$, so that $\forall s \in \mathcal{S}^{\mathcal{F}}, \forall W \subseteq \mathcal{T}, W \subseteq \text{var}(d(s, W)) \wedge d(s, \emptyset) \subseteq d(s, W)$.

Let p be a function from $\mathcal{S}^{\mathcal{F}} \times \wp(\mathcal{A}_{\mathcal{T}})$ to $\wp(\mathcal{S}^{\mathcal{T}}) \setminus \{\emptyset\}$ so that $\forall s \in \mathcal{S}^{\mathcal{F}}, \forall D \subseteq \mathcal{A}_{\mathcal{T}}, p(s, D) = \{s' \in \mathcal{S}^{\mathcal{T}} \mid s' \subseteq D \cup d(s, \mathcal{T} \setminus \text{var}(D))\}$. Since $\mathcal{T} \setminus \text{var}(D) \subseteq \text{var}(d(s, W)), \emptyset \notin p(s, D)$. Thus from Definition 14, $\forall s \in \mathcal{S}^{\mathcal{F}}, \mathcal{T}_{syn}(P)(s) = p(s, \mathsf{Ccl}(s, P))$ (property 1).

Since $\forall W \subseteq \mathcal{T}, d(s, \emptyset) \subseteq d(s, W)$, $\forall D \subseteq \mathcal{A}_{\mathcal{T}}, d(s, \emptyset) \subseteq D \cup d(s, \mathcal{T} \setminus \text{var}(D))$, thus $d(s, \emptyset) \subseteq \bigcup_{s' \in p(s,D)} s'$ (property 2).

Moreover, $\forall D \subseteq \mathcal{A}_{\mathcal{T}}$, let $D' := \bigcup_{s' \in p(s,D)} s'$. Straightforwardly: $D' = D \cup d(s, \mathcal{T} \setminus \text{var}(D))$ because we can always create a state with any atom in $D \cup d(s, \mathcal{T} \setminus \text{var}(D))$, thus all atoms of this set are in D', and conversely (property 3). $p(s, D') = \{s' \in \mathcal{S}^{\mathcal{T}} \mid s' \subseteq D' \cup d(s, \mathcal{T} \setminus \text{var}(D'))\}$ by definition of p. $p(s, D') = \{s' \in \mathcal{S}^{\mathcal{T}} \mid s' \subseteq D' \cup d(s, \emptyset)\}$ since $\text{var}(D') = \mathcal{T}$ by definition of D' and p. $p(s, D') = \{s' \in \mathcal{S}^{\mathcal{T}} \mid s' \subseteq D'\}$ from property 2. $p(s, D') = \{s' \in \mathcal{S}^{\mathcal{T}} \mid s' \in D \cup d(s, \mathcal{T} \setminus \text{var}(D))\} = p(s, D)$ from property 3. Therefore p respects (1). Since $\mathcal{T}_{syn}(P) = p(s, \mathsf{Ccl}(s, P))$, p also respects (2). Thus, $\mathcal{T}_{syn}(P) = \mathcal{T}_{syn}(P_{\mathcal{O}}(\mathcal{T}_{syn}(P)))$ according to Theorem 2.

By definition of $\mathcal{T}_{gen}$: $\forall s \in \mathcal{S}^{\mathcal{F}}, (\mathcal{T}_{gen}(P))(s) = \{s' \in \mathcal{S}^{\mathcal{T}} \mid s' \subseteq \mathsf{Ccl}(s, P) \cup d(s, T \setminus \text{var}(\mathsf{Ccl}(s, P)))\}$ with $\mathsf{sp}_{\overline{\mathcal{F}} \to \overline{\mathcal{T}}}(s) \subseteq d(s, \emptyset)$. Thus, the same proof gives $\mathcal{T}_{gen}(P) = \mathcal{T}_{gen}(P_{\mathcal{O}}(\mathcal{T}_{gen}(P)))$ according to Theorem 2.

[Let us show that: $\mathcal{T}_{asyn}(P) = \mathcal{T}_{asyn}(P_{\mathcal{O}}(\mathcal{T}_{asyn}(P)))$.] Let p be a function from $\mathcal{S}^{\mathcal{F}} \times \wp(\mathcal{A}_{\mathcal{T}})$ to $\wp(\mathcal{S}^{\mathcal{T}}) \setminus \{\emptyset\}$ so that $\forall s \in \mathcal{S}^{\mathcal{F}}, \forall D \subseteq \mathcal{A}_{\mathcal{T}}$:

$$p(s, D) = \{s' \in \mathcal{S}^{\mathcal{T}} \mid s' \subseteq D \cup d(s, \mathcal{T} \setminus \text{var}(D)) \wedge \big(|s' \setminus \mathsf{sp}_{\overline{\mathcal{F}} \to \overline{\mathcal{T}}}(s)| - |\mathcal{T} \setminus \overline{\mathcal{T}}| = 1 \vee (D \cup d(s, \mathcal{T} \setminus \text{var}(D)))_{\overline{\mathcal{T}}} = \mathsf{sp}_{\overline{\mathcal{F}} \to \overline{\mathcal{T}}}(s)\big)\}$$

where $\mathcal{A}_{\mathcal{T}}$ and $D_{\overline{\mathcal{T}}}$ are restriction notations from Definition 12. From Definition 15, we have: $\mathcal{T}_{asyn} P = p(s, \mathsf{Ccl}(s, P))$.

[Let us show that: $\forall D \subseteq \mathcal{A}_{\mathcal{T}}, p(s, \bigcup_{s' \in p(s,D)} s') = p(s, D)$.] Let D in $\mathcal{A}_{\mathcal{T}}$.

- If $(D \cup d(s, \mathcal{T} \setminus \text{var}(D)))_{\overline{\mathcal{T}}} = \mathsf{sp}_{\overline{\mathcal{F}} \to \overline{\mathcal{T}}}(s)$, then $\bigcup_{s' \in p(s,D)} s' = D$ and thus $p(s, \bigcup_{s' \in p(s,D)} s') = p(s, D)$.
- If there exists $\mathrm{v}^{val} \in \mathcal{A}_{\overline{\mathcal{T}}}$ so that $\text{var}(D \cup d(s, \mathcal{T} \setminus \text{var}(D)) \setminus \mathsf{sp}_{\overline{\mathcal{F}} \to \overline{\mathcal{T}}}(s)) \cap \overline{\mathcal{T}} = \{\mathrm{v}\}$, then for all state $s' \in p(s, D)$, s' differs from s on the regular variable v and on variables in $\mathcal{T} \setminus \overline{\mathcal{T}}$. Thus, $\bigcup_{s' \in p(s,D)} s' = (D \cup d(s, \mathcal{T} \setminus \text{var}(D))) \setminus \{\mathrm{v}^{val'} \mid \mathrm{v}^{val'} \in s\}$. By construction of p, it comes: $p(s, \bigcup_{s' \in p(s,D)} s') = p(s, D)$ because $\mathrm{v}^{val'} \in s'$ would contradict the condition $|s' \setminus \mathsf{sp}_{\overline{\mathcal{F}} \to \overline{\mathcal{T}}}(s)| - |\mathcal{T} \setminus \overline{\mathcal{T}}| = 1$.
- Otherwise, $|\text{var}(D \cup d(s, \mathcal{T} \setminus \text{var}(D)) \setminus \mathsf{sp}_{\overline{\mathcal{F}} \to \overline{\mathcal{T}}}(s)) \cap \overline{\mathcal{T}}| > 1$ then there exists two states $s'_1, s'_2 \in p(s, D)$, so that they differ from s on a different regular variable each. Especially, by construction of p, $\mathsf{sp}_{\overline{\mathcal{F}} \to \overline{\mathcal{T}}}(s) \subseteq s'_1 \cup s'_2 \subseteq D \cup d(s, \mathcal{T} \setminus \text{var}(D))$. Therefore, $\bigcup_{s' \in p(s,D)} s' \subseteq D \cup d(s, \mathcal{T} \setminus \text{var}(D))$. Finally, and by definition of p, $D \cup d(s, \mathcal{T} \setminus \text{var}(D)) \subseteq \bigcup_{s' \in p(s,D)} s'$ because

for each atom in $D \cup d(s, \mathcal{T} \setminus \text{var}(D))$, it is possible to build a state s' containing it: either as the projection of the initial state s or as the only variable changing its value in s' compared to $\mathsf{sp}_{\overline{\mathcal{F}} \to \overline{\mathcal{T}}}(s)$. In conclusion: $D \cup d(s, \mathcal{T} \setminus \text{var}(D)) = \bigcup_{s' \in p(s,D)} s'$, which gives: $p(s, \bigcup_{s' \in p(s,D)} s') = p(s, D)$.

Thus, $\mathcal{T}_{asyn}(P) = \mathcal{T}_{asyn}(P_{\mathcal{O}}(\mathcal{T}_{asyn}(P)))$, according to Theorem 2. □

C Appendix: proofs of Section 4

Theorem 14 (Theorem 4: properties of the least revision) *Let R be a $\mathcal{M}$VL rule and $s \in \mathcal{S}^{\mathcal{F}}$ such that $R \sqcap s$. Let $S_R := \{s' \in \mathcal{S}^{\mathcal{F}} \mid R \sqcap s'\}$ and $S_{\text{spe}} := \{s' \in \mathcal{S}^{\mathcal{F}} \mid \exists R' \in L_{\text{spe}}(R, s, \mathcal{A}, \mathcal{F}), R' \sqcap s'\}$.*

Let P be a $\mathcal{DM}$VLP and $T, T' \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$ such that $|\text{first}(T)| = 1 \wedge \text{first}(T) \cap \text{first}(T') = \emptyset$. The following results hold:

1. $S_{\text{spe}} = S_R \setminus \{s\}$,
2. $L_{\text{rev}}(P, T, \mathcal{A}, \mathcal{F})$ *is consistent with* T,
3. $\overset{P}{\hookrightarrow} T' \implies \xhookrightarrow{L_{\text{rev}}(P,T,\mathcal{A},\mathcal{F})} T'$,
4. $\overset{P}{\hookrightarrow} T \implies \xhookrightarrow{L_{\text{rev}}(P,T,\mathcal{A},\mathcal{F})} T$,
5. P *is complete* $\implies L_{\text{rev}}(P, T, \mathcal{A}, \mathcal{F})$ *is complete.*

Proof.

1. First, let us suppose that $\exists s'' \notin S_R \setminus \{s\}$ such that $\exists R' \in L_{\text{spe}}(R, s, \mathcal{A}, \mathcal{F}), R' \sqcap s''$. By definition of matching $R' \sqcap s'' \implies b(R') \subseteq s''$. By definition of least specialization, $b(R') = b(R) \cup \{\text{v}^{val}\}, \text{v}^{val'} \in s, \text{v}^{val''} \notin b(R), val \neq val'$. Let us suppose that $s'' = s$, then $b(R') \not\subseteq s''$ since $\text{v}^{val} \in b(R')$ and $\text{v}^{val} \notin s$, this is a contradiction. Let us suppose that $s'' \neq s$ then $\neg(R \sqcap s'')$, thus $b(R) \not\subseteq s''$ and $b(R') \not\subseteq s''$, this is a contradiction.
   Second, let us assume that $\exists s'' \in S_R \setminus \{s\}$ such that $\forall R' \in L_{\text{spe}}(R, s, \mathcal{A}, \mathcal{F}), \neg(R' \sqcap s'')$. By definition of S_R, $R \sqcap s''$. By definition of matching $\neg(R' \sqcap s'') \implies b(R') \not\subseteq s''$. By definition of least specialization, $b(R') = b(R) \cup \{\text{v}^{val}\}, \text{v}^{val'} \in s, val \neq val'$. By definition of matching $R \sqcap s'' \implies b(R) \subseteq s'' \implies s'' = b(R) \cup I, b(R) \cap I = \emptyset$ and thus $b(R') \not\subseteq s'' \implies \text{v}^{val} \notin I$. The assumption implies that $\forall \text{v}^{val'} \in I, \forall R' \in L_{\text{spe}}(R, s, \mathcal{A}, \mathcal{F}), \text{v}^{val} \in b(R'), val \neq val'$. By definition of least specialization, it implies that $\text{v}^{val'} \in s$ and thus $I = s \setminus b(R)$ making $s'' = s$, which is a contradiction.
   Conclusion: $S_{\text{spe}} = S_R \setminus \{s\}$
2. By definition of a consistent program, if two sets of $\mathcal{M}$VL rules SR_1, SR_2 are consistent with T then $SR_1 \cup SR_2$ is consistent with T. Let $R_P = \{R \in P \mid R \sqcap s, \forall (s, s') \in T, h(R) \notin s'\}$ be the set of rules of P that conflict with T. By definition of least revision $L_{\text{rev}}(P, T, \mathcal{A}, \mathcal{F}) = (P \setminus R_P) \cup \bigcup_{R \in R_P} L_{\text{spe}}(R, s, \mathcal{A}, \mathcal{F})$. The first part of the expression $P \setminus R_P$ is consistent with T since $\nexists R' \in P \setminus R_P$ such that R' conflicts with T. The second part of the expression $\bigcup_{R \in R_P} L_{\text{spe}}(R, s, \mathcal{A}, \mathcal{F})$ is also consistent with T: $\nexists R' \in L_{\text{spe}}(R, s, \mathcal{A}, \mathcal{F}), R' \sqcap s$ thus $\nexists R' \in L_{\text{spe}}(R, s, \mathcal{A}, \mathcal{F})$ that conflict with T and $\bigcup_{R \in R_P} L_{\text{spe}}(R, s, \mathcal{A}, \mathcal{F})$ is consistent with T. Conclusion: $L_{\text{rev}}(P, T, \mathcal{A}, \mathcal{F})$ is consistent with T.
3. Let $(s_1, s_2) \in T'$ thus $s_1 \neq s$. From definition of realization, $\text{v}^{val} \in s_2 \implies \exists R \in P, h(R) = \text{v}^{val}, R \sqcap s_1$. If $\neg R \sqcap s$ then $R \in L_{\text{rev}}(P, T, \mathcal{A}, \mathcal{F})$ and $\xrightarrow{L_{\text{rev}}(P,T,\mathcal{A},\mathcal{F})} (s_1, s_2)$. If $R \sqcap s$, from the first point $\exists R' \in L_{\text{spe}}(R, s, \mathcal{A}, \mathcal{F}), R' \sqcap s_1$ and since $h(R') = h(R) = \text{v}^{val}$, $\xrightarrow{L_{\text{rev}}(P,T,\mathcal{A},\mathcal{F})} (s_1, s_2)$. Applying this reasoning on all elements of T' implies that $\overset{P}{\hookrightarrow} T' \implies \xhookrightarrow{L_{\text{rev}}(P,T,\mathcal{A},\mathcal{F})} T'$.

4. Let $(s_1, s_2) \in T$, since $\overset{P}{\hookrightarrow} T$ by definition of realization $\forall \mathrm{v}^{val} \in s_2, \exists R \in P, R \sqcap s_1, h(R) = \mathrm{v}^{val}$. By definition of conflict, R is not in conflict with T thus $R \in L_{\text{rev}}(P, T, \mathcal{A}, \mathcal{F})$ and $\xrightarrow{L_{\text{rev}}(P,T,\mathcal{A},\mathcal{F})} T$.
5. Let $(s_1, s_2) \in \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$, if P is complete, then by definition of a complete program $\forall \mathrm{v} \in \mathcal{V}, \exists R \in P, R \sqcap s_1, \text{var}(h(R)) = v$. If $\neg(R \sqcap s)$ then $R \in L_{\text{rev}}(P, T, \mathcal{A}, \mathcal{F})$. If $R \sqcap s$, from the first point $\exists R' \in L_{\text{spe}}(R, s, \mathcal{A}, \mathcal{F}), R' \sqcap s_1$ and thus $R' \in L_{\text{rev}}(P, T, \mathcal{A}, \mathcal{F})$ and since $\text{var}(h(R')) = \text{var}(h(R)) = \mathrm{v}$, $L_{\text{rev}}(P, T, \mathcal{A}, \mathcal{F})$ is complete.

□

Proposition 6 (Proposition 2: optimal program of empty set) $P_{\mathcal{O}}(\emptyset) = \{\mathrm{v}^{val} \leftarrow \emptyset \mid \mathrm{v}^{val} \in \mathcal{A}_{\mathcal{T}}\}$.

Proof. Let $P = \{\mathrm{v}^{val} \leftarrow \emptyset \mid \mathrm{v}^{val} \in \mathcal{A}_{\mathcal{T}}\}$. The $\mathcal{M}$VLP P is consistent and complete by construction. Like all $\mathcal{M}$VLPs, $\overset{P}{\hookrightarrow} \emptyset$ and there is no transition in $\emptyset$ to match with the rules in P. In addition, by construction, the rules of P dominate all $\mathcal{M}$VL rules. □

Theorem 15 (Theorem 5: least revision is suitable) *Let $s \in \mathcal{S}^{\mathcal{F}}$ and $T, T' \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$ such that $|\text{first}(T')| = 1 \wedge \text{first}(T) \cap \text{first}(T') = \emptyset$. $L_{\text{rev}}(P_{\mathcal{O}}(T), T', \mathcal{A}, \mathcal{F})$ is a $\mathcal{DM}$VLP suitable for $T \cup T'$.*

Proof. Let $P = L_{\text{rev}}(P_{\mathcal{O}}(T), T')$. Since $P_{\mathcal{O}}(T)$ is consistent with T, by Theorem 4, P is also consistent with T and thus consistent with $T' \cup T$. Since $P_{\mathcal{O}}(T)$ realizes T by Theorem 4, $\overset{P}{\hookrightarrow} T$. Since $s \notin \text{first}(T)$, a $\mathcal{M}$VL rule R such that $b(R) = s$ does not conflict with T. By definition of suitable program $\exists R' \in P_{\mathcal{O}}(T), R \leq R'$, thus $\xrightarrow{P_{\mathcal{O}}(T)} T'$. Since $\xrightarrow{P_{\mathcal{O}}(T)} T'$ by Theorem 4 $\overset{P}{\hookrightarrow} T'$ and thus $\overset{P}{\hookrightarrow} T \cup T'$. Since $P_{\mathcal{O}}(T)$ is complete, by Theorem 4, P is also complete. To prove that P verifies the last point of the definition of a suitable $\mathcal{M}$VLP, let R be a $\mathcal{M}$VL rule not conflicting with $T \cup T'$. Since R is also not conflicting with T, there exists $R' \in P_{\mathcal{O}}(T)$ such that $R \leq R'$. If R' is not conflicting with T', then R' will not be revised and $R' \in P$, thus R is dominated by a rule of P. Otherwise, R' is in conflict with T', thus $R' \sqcap s$ and $\forall (s, s') \in T', h(R') \notin s'$. Since R is not in conflict with T' and $h(R) = h(R')$, since $R \leq R'$ then $b(R) = b(R') \cup I, \exists \mathrm{v}^{val} \in I, \mathrm{v}^{val} \notin s$. By definition of least revision and least specialization, there is a rule $R'' \in L_{\text{spe}}(R', s)$ such that $\mathrm{v}^{val} \in b(R'')$ and since $R'' = h(R') \leftarrow b(R') \cup \mathrm{v}^{val}$ thus $R \leq R''$. Thus R is dominated by a rule of P. □

Theorem 16 (Theorem 6: GULA Termination, soundness, completeness, optimality) *Let $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$.*
(1) *Any call to* **GULA** *on finite sets terminates,*
(2) $\mathbf{GULA}(\mathcal{A}, T, \mathcal{F}, \mathcal{T}) = P_{\mathcal{O}}(T)$,
(3) $\forall \mathcal{A}' \subseteq \mathcal{A}_{\mathcal{T}}, \mathbf{GULA}(\mathcal{A}_{\mathcal{F}} \cup \mathcal{A}', T, \mathcal{F}, \mathcal{T}) = \{R \in P_{\mathcal{O}}(T) \mid h(R) \in \mathcal{A}'\}$.

Proof. (1) The algorithm of **GULA** iterates on finite sets, and thus terminates.

(3) Let $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$. The algorithm iterates over each atom $\mathrm{v}^{val} \in \mathcal{A}'$, $\mathcal{A}' \subseteq \mathcal{A}_{\mathcal{T}}$ iteratively to extract all states s such that $(s, s') \in T \implies \mathrm{v}^{val} \notin s'$. This is equivalent to group the transitions by initial state: generate the set $TT = \{T'_s \subseteq T \mid s \in \mathcal{S}^{\mathcal{F}}, \text{first}(T'_s) = \{s\} \wedge \forall s' \in \mathcal{S}^{\mathcal{T}}, (s, s') \in T \implies (s, s') \in T'_s\}$.

To prove that $\forall \mathcal{A}' \subseteq \mathcal{A}_{\mathcal{T}}, \mathbf{GULA}(\mathcal{A}_{\mathcal{F}} \cup \mathcal{A}', T, \mathcal{F}, \mathcal{T}) = \{R \in P_{\mathcal{O}}(T) \mid h(R) \in \mathcal{A}'\}$ and thus $\mathbf{GULA}(\mathcal{A}, T, \mathcal{F}, \mathcal{T}) = P_{\mathcal{O}}(T)$, it suffices to prove that the main loop (Algorithm 3, lines 19–46) preserves the invariant $P^{val}_{\mathrm{v}} = \{R \in P_{\mathcal{O}}(T_i) \mid h(R) = \mathrm{v}^{val} \in \mathcal{A}'\}$ after the i^{th} iteration where T_i is the union of all set of transitions of TT already selected line 19 after the i^{th} iteration for all i from 0 to $|TT|$.

Line 18 initializes $P_{\mathrm{v}^{val}}$ to $\{\mathrm{v}^{val} \leftarrow \emptyset\}$. Thus by Proposition 2, after line 18, $P_{\mathrm{v}^{val}} = \{R \in P_{\mathcal{O}}(\emptyset) \mid h(R) = \mathrm{v}^{val}\}$.

Let us assume that before the $(i+1)^{\text{th}}$ iteration of the main loop, $P_{\mathrm{v}^{val}} = \{R \in P_{\mathcal{O}}(T_i) \mid h(R) = \mathrm{v}^{val}\}$. Through the loop of lines 21–24, $P' = \{R \in P_{\mathcal{O}}(T_i) \mid R \text{ does not conflict with } T_{i+1} \wedge h(R) = \mathrm{v}^{val}\}$ is computed. Then the set $P'' = \bigcup_{R \in P_{\mathcal{O}}(T_i) \setminus P' \wedge h(R) = \mathrm{v}^{val}} L_{\text{spe}}(R, s, \mathcal{A}, \mathcal{F})$

is iteratively build through the calls to **least_specialization** at line 27 and the dominated rules are pruned as they are detected by the loop of lines 28–45. Each revised rule can be dominated by a rule in $\{R \in P_{\mathcal{O}}(T_i) \backslash P'\}$ or another revised rule and thus dominance must be checked from both. But only a revised rule ($R \in P''$) can be dominated by a revised rule: if a rule in $\{R \in P_{\mathcal{O}}(T_i) \backslash P'\}$ is dominated by a revised rule, then it was dominated by its original rule in $\{R \in P_{\mathcal{O}}(T_i)\}$ which is impossible since $P_{\mathrm{v}^{val}} = \{R \in P_{\mathcal{O}}(T_i) \mid h(R) = \mathrm{v}^{val}\}$. Thus it is safe to only check domination of the revised rules by previous rules ($P_{\mathcal{O}}(T_i) \setminus P'$) or by other revised rules (P''). Thus by Theorem 5 and Proposition 3, $P_{\mathrm{v}^{val}} = \{R \in P_{\mathcal{O}}(T_{i+1}) \mid h(R) = \mathrm{v}^{val}\}$ after the $(i+1)^{\text{th}}$ iteration of the main loop. By induction, at the end of all the loop lines 19–46, $P_{\mathrm{v}^{val}} = \{R \in P_{\mathcal{O}}(\bigcup_{T' \in TT} T') \mid h(R) = \mathrm{v}^{val}\} = \{R \in P_{\mathcal{O}}(T) \mid h(R) = \mathrm{v}^{val}\}$ since it has iterated on all elements of TT. Since the same operation holds for each $\mathrm{v}^{val} \in \mathcal{A}'$, $P = \bigcup_{\mathrm{v}^{val} \in \mathcal{A}'} P_{\mathrm{v}^{val}} = \{R \in P_{\mathcal{O}}(T) \mid h(R) = \mathrm{v}^{val} \wedge \mathrm{v}^{val} \in \mathcal{A}'\}$ after all iterations of the loop of line line 6. Finally: $\forall \mathcal{A}' \subseteq \mathcal{A}_{\mathcal{T}}, \mathbf{GULA}(\mathcal{A}_{\mathcal{F}} \cup \mathcal{A}', T, \mathcal{F}, \mathcal{T}) = \{R \in P_{\mathcal{O}}(T) \mid h(R) \in \mathcal{A}'\}$.

(2) Thus $\mathbf{GULA}(\mathcal{A}, T, \mathcal{F}, \mathcal{T}) = \mathbf{GULA}(\mathcal{A}_{\mathcal{F}} \cup \mathcal{A}_{\mathcal{T}}, T, \mathcal{F}, \mathcal{T}) = \{R \in P_{\mathcal{O}}(T) \mid h(R) \in \mathcal{A}_{\mathcal{T}}\} = P_{\mathcal{O}}(T)$. □

Theorem 17 (Theorem 7: GULA Complexity) *Let $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$ be a set of transitions, Let $n := max(|\mathcal{F}|, |\mathcal{T}|)$ and $d := \max(\{|\mathsf{dom}(\mathrm{v})| \in \mathbb{N} \mid \mathrm{v} \in \mathcal{F} \cup \mathcal{T}\}$. The worst-case time complexity of* **GULA** *when learning from T belongs to $\mathcal{O}(|T|^2 + |T| \times (2n^4 d^{2n+2} + 2n^3 d^{n+1}))$ and its worst-case memory use belongs to $\mathcal{O}(d^{2n} + 2nd^{n+1} + nd^{n+2})$. Proof.* Let $d_f := \max(\{|\mathsf{dom}(\mathrm{v})| \in \mathbb{N} \mid \mathrm{v} \in \mathcal{F}\})$ (resp. $d_t := \max(\{|\mathsf{dom}(\mathrm{v})| \in \mathbb{N} \mid \mathrm{v} \in \mathcal{T}\})$) be the maximal number of values of features (resp. target) variables. The algorithm takes as input a set of transition $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$ bounding the memory use to $O(d_f^{|\mathcal{F}|}) \times d_t^{|\mathcal{T}|}) = O(d^{2n})$. The learning is performed iteratively for each possible rule head $\mathrm{v}^{val} \in \mathcal{A}' \subseteq \mathcal{A}_{\mathcal{T}}$. The extraction of negative example requires to compare each transition of T one to one and thus has a complexity of $op_1 = O(|T|^2)$. Those transitions are stored in $Neg_{\mathrm{v}^{val}}$ which size is at most $|\mathcal{S}^{\mathcal{F}}|$ extending the memory use to $O(d_f^{|\mathcal{F}|} \times d_t^{|\mathcal{T}|} + d_f^{|\mathcal{F}|})$ which is bounded by $O(d^{2n} + d^n)$.

The learning phase revises a set of rule $P_{\mathrm{v}^{val}}$ where each rule has the same head v^{val}. There are at most $d_f^{|\mathcal{F}|} \leq d^n$ possible rule bodies and thus $|P_{\mathrm{v}^{val}}| \leq d_t^{|\mathcal{F}|} \leq d^n$, the memory use of $|P_{\mathrm{v}^{val}}|$ is then $O(d_t^{|\mathcal{F}|})$ extending the memory bound to $O(d_f^{|\mathcal{F}|} \times d_t^{|\mathcal{T}|} + d_f^{|\mathcal{F}|}) + d_f^{|\mathcal{F}|}) = O(d_f^{|\mathcal{F}|} \times d_t^{|\mathcal{T}|} + 2d_f^{|\mathcal{F}|}))$, which is bound by $O(d^{2n} + 2d^n)$.

For each state s of $Neg_{\mathrm{v}^{val}}$, each rule of $P_{\mathrm{v}^{val}}$ that matches s are extracted into a set of rules R_m. This operation has a complexity of $op_2 = O(d_f^{|\mathcal{F}|} \times |\mathcal{F}|)$ bound by $O(nd^n)$. Each rule of R_m are then revised using least specialization, this operation has a complexity of $O(|\mathcal{F}|^2)$ bound by $O(n^2)$. $|R_m| \leq d_f^{|\mathcal{F}|} \leq d^n$ thus the revision of all matching rules is $op_3 = O(d_f^{|\mathcal{F}|} \times n^2)$ bounded by $O(d^n \times n^2)$. All revisions are stored in LS and there are at most $d_f \times |\mathcal{F}| \leq dn$ revisions for each rule, thus $|LS| \leq d_f^{|\mathcal{F}|} \times d_f |\mathcal{F}| \leq d^n \times dn$ extending the memory bound to $O(d_f^{|\mathcal{F}|} \times d_t^{|\mathcal{T}|} + 2d_f^{|\mathcal{F}|}) + d_f |\mathcal{F}| \times d_f^{|\mathcal{F}|})$ bounded by $O(d^{2n} + 2d^n + nd^{n+1})$.

Learning is performed for each $\mathrm{v}^{val} \in \mathcal{A}' \subseteq \mathcal{T}$, thus the memory usage of **GULA** is therefore $O(d_f^{|\mathcal{F}|} \times d_t^{|\mathcal{T}|} + |\mathcal{A}'|(2d_f^{|\mathcal{F}|} + d_f |\mathcal{F}| \times d_f^{|\mathcal{F}|}))$, bounded by $O(d_f^{|\mathcal{F}|} \times d_t^{|\mathcal{T}|} + td_t(2d_f^{|\mathcal{F}|}) + d_f |\mathcal{F}| \times d_f^{|\mathcal{F}|}))$ wich is bounded by $O(d^{2n} + dn(2d^n + nd^{n+1})) = O(d^{2n} + 2nd^{n+1} + nd^{n+2})$.

The worst-case memory use of **GULA** is thus $O(d^{2n} + 2nd^{n+1} + nd^{n+2})$.

All rules of LS are compared to the rule of $P_{\mathrm{v}^{val}}$ for domination check, this operation has a complexity of $op_4 = O(2 \times |LS| \times |P_{\mathrm{v}^{val}}| \times |\mathcal{F}|^2) = O(2 \times d_f^{|\mathcal{F}|} \times d_f |\mathcal{F}| \times d^n \times n^2) = O(2 \times |\mathcal{F}|^3 \times d_f^{2|\mathcal{F}|+1})$ which is bounded by $O(2 \times n^3 \times d^{2n+1})$.

Learning is performed for each $\mathrm{v}^{val} \in \mathcal{A}' \subseteq \mathcal{T}$, $|\mathcal{A}'| \leq |\mathcal{T}| d_t$, thus the complexity is bound by $O(op_1 + |T| \times |\mathcal{T}| \times d_t(op_2 + op_3 + op_4)) = O(|T|^2 + |T|\ times |\mathcal{T}| \times d_t(d_f^{|\mathcal{F}|} \times |\mathcal{F}| + d_f^{|\mathcal{F}|} \times n^2 + 2 \times |\mathcal{F}|^3 \times d_f^{2|\mathcal{F}|+1}))$ which is bounded by $O(|T|^2 + |T| \times nd(d^n \times n^2 + d^n \times n^2 + 2 \times n^3 \times d^{2n+1})) = O(|T|^2 + |T| \times nd(2n^3 d^{2n+1} + 2n^2 d^n)) = O(|T|^2 + |T| \times (2n^4 d^{2n+2} + 2n^3 d^{n+1}))$.

The computational complexity of **GULA** is thus $O(|T|^2 + |T| \times (2n^4 d^{2n+2} + 2n^3 d^{n+1}))$.
□

D Appendix: proofs of Section 5

Theorem 18 (Theorem 8: Optimal $\mathcal{DM}$VLP and constraints correctness under synchronous constrainted semantics) *Let $T \subseteq \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}$, it holds that $T = \mathcal{T}_{syn-c}(P_{\mathcal{O}}(T) \cup C'_{\mathcal{O}}(T))$.*

Proof. From Definition 9, $\forall (s, s') \in T, s' \subseteq \mathsf{Ccl}(s, P_{\mathcal{O}}(T))$ thus according to Definition 21, $s' \in \mathcal{T}_{syn-c}(P_{\mathcal{O}}(T))(s)$, thus $T \subseteq \mathcal{T}_{syn-c}(P_{\mathcal{O}}(T))$ (property 1).

By Definition 24, $\forall (s, s') \in T, \nexists C \in C_{\mathcal{O}}(T), C \sqcap (s, s')$, thus since $C'_{\mathcal{O}}(T) \subseteq C_{\mathcal{O}}(T), \nexists C \in C'_{\mathcal{O}}(T), C \sqcap (s, s')$ and then $T \subseteq \mathcal{T}_{syn-c}(P_{\mathcal{O}}(T) \cup C'_{\mathcal{O}}(T))$ (property 2).

Let us suppose $\exists (s, s') \in \mathcal{T}_{syn-c}(P_{\mathcal{O}}(T) \cup C'_{\mathcal{O}}(T)), (s, s') \notin T$. From Definition 21, $\forall \mathrm{v}^{val} \in s', \exists R \in P_{\mathcal{O}}(T), b(R) \sqcap s, h(R) = \mathrm{v}^{val}$. From Definition 24, $\exists C' \in C_{\mathcal{O}}(T), C' \sqcap (s, s')$ since $(s, s') \notin T$. But since $\exists (s, s') \in \mathcal{T}_{syn-c}(P_{\mathcal{O}}(T) \cup C'_{\mathcal{O}}(T))$, thus $C' \notin C'_{\mathcal{O}}(T)$. From Definition 25, it implies that $\exists \mathrm{v}^{val} \in s', \nexists R \in P_{\mathcal{O}}(T), h(R) = \mathrm{v}^{val}, \forall \mathrm{w} \in \mathcal{F}, \forall val', val'' \in \mathsf{dom}(\mathrm{w}), \mathrm{w}^{val'} \in b(R) \wedge \mathrm{w}^{val''} \in b(C) \implies val' = val''$. Since $b(C') \subseteq (s \cup s')$, $\nexists R \in P_{\mathcal{O}}(T), h(R) = \mathrm{v}^{val}, b(R) \subseteq s$, thus $s' \not\subseteq \mathsf{Ccl}(s, P_{\mathcal{O}}(T))$ and by Definition 21, $(s, s') \notin \mathcal{T}_{syn-c}(P_{\mathcal{O}}(T) \cup C'_{\mathcal{O}}(T))$, contradiction, thus $\mathcal{T}_{syn-c}(P_{\mathcal{O}}(T) \cup C'_{\mathcal{O}}(T)) \subseteq T$ (property 3).

From property 2 and 3: $\mathcal{T}_{syn-c}(P_{\mathcal{O}}(T) \cup C'_{\mathcal{O}}(T)) = T$. □

Theorem 19 (Theorem 9: Synchronizer correctness) *Given any set of transitions T,* **Synchronizer**$(\mathcal{A}, T, \mathcal{F}, \mathcal{T})$ *outputs $P_{\mathcal{O}}(T) \cup C'_{\mathcal{O}}(T)$.*

Proof. Let $G1 = GULA(\mathcal{A}, T, \mathcal{F}, \mathcal{T})$ and $G2 = GULA(\mathcal{A}_{\mathcal{F} \cup \mathcal{T} \cup \{\varepsilon^1\}}, T', \mathcal{F} \cup \mathcal{T}, \{\varepsilon\})$. From Theorem 6, $P = G1 = P_{\mathcal{O}}(T)$ (property 1).

Let $P' = G2$. By definition of T': $\forall (s, s') \in T', s' = \{\varepsilon^0\}$. Thus $\forall R \in P'$, R is consistent with T' by Theorem 6, thus $\nexists (s, s') \in T', R \sqcap s$, since $h(R) = \varepsilon^1$ because $\forall (s, s') \in T', s' = \{\varepsilon^0\}$ (property 2).

From Theorem 6, $P' = \{R \in P_{\mathcal{O}}(T') \mid h(R) = \varepsilon^1\}$. From Definition 9, $P_{\mathcal{O}}(T')$ is complete thus $\forall (s, s') \in \mathcal{S}^{\mathcal{F}} \times \mathcal{S}^{\mathcal{T}}, ss' := s \cup s', ss' \notin \mathrm{first}(T'), \exists R \in P', R \sqcap ss'$ (property 3).

From definition of T', $(s, s') \in T \implies (s \cup s', \{\varepsilon^0\}) \in T'$, thus $\forall C \in P', C$ is a constraint (property 4).

- From property 2 and 4: $(s, s') \in T \implies (s \cup s', \{\varepsilon^0\}) \in T' \implies \nexists C \in P', C \sqcap (s, s')$, P' consistent with T.
- From property 3 and 4: $(s, s') \notin T \implies (s \cup s') \notin \mathrm{first}(T') \implies \exists R \in P', R \sqcap (s, s')$, P' is complete with T.
- If there exists a constraint consistent with T that is not dominated by a constraint in P' it implies that a rule consistent with T' whose head is ε^1 is not dominated by a rule in $G2$ wich is in contradiction with Theorem 6. All constraint consistent with T are dominated by a constraint in P'.
- From Theorem 6, the rules of $G2$ do not dominate eachover, thus the same hold for the constraint of P'.
- From Definition 24, $P' = C_{\mathcal{O}}(T)$ (property 5).

Let $P'' := \{C \in P' \mid \forall \mathrm{v}^{val} \in b(C), \mathrm{v} \in \mathcal{T}, \exists R \in P, h(R) = \mathrm{v}^{val} \wedge \big(\forall \mathrm{w} \in \mathcal{F}, \forall val', val'' \in \mathsf{dom}(\mathrm{w}), \mathrm{w}^{val'} \in b(R) \wedge \mathrm{w}^{val''} \in b(C) \implies val' = val''\big)\}$. Since $P = P_{\mathcal{O}}(T)$ and $P' = C_{\mathcal{O}}(T)$, thus $P'' = C'_{\mathcal{O}}(T)$, from Definition 25 (property 6).

Therefore, from property 1 and 6, $Synchronizer(\mathcal{A}, T, \mathcal{F}, \mathcal{T}) = P_{\mathcal{O}}(T) \cup C'_{\mathcal{O}}(T)$.
□

E Appendix: detailed pseudo-code of Section 4

Algorithms 3 and 4 provide the detailed pseudocode of **GULA**. Algorithm 3 learns from a set of transitions T the conditions under which each value val of each variable v may appear in the next state. Here, learning is performed iteratively for each value of variable to keep the pseudo-code simple. But the process can easily be parallelized by running each loop in an independent thread, bounding the run time to the variable for which the learning is the longest. In the case where we are not interested about the dynamics of some variables, the parameter $\mathcal{A}'$ and $\mathcal{T}'$ can be reduced accordingly. The algorithm starts by the pre-processing of the input transitions. Lines 7–16 of Algorithm 3 correspond to the extraction of $Neg_{\mathrm{v}^{val}}$, the set of all negative examples of the appearance of v^{val} in next state: all states such that v never takes the value val in the next state of a transition of T. For efficiency purpose, it is important that the negatives examples are ordered in a way that reduce the difference between nearby elements, for example lexicographically. Indeed, it increase the proportion of revised rules (produced to satisfy a previous example) still consistent with the following examples, reducing the average number of rules stored and thus checked in the following processes. Those negative examples are then used during the following learning phase (lines 17–46) to iteratively learn the set of rules $P_{\mathcal{O}}(T)$. The learning phase starts by initializing a set of rules $P_{\mathrm{v}^{val}}$ to $\{R \in P_{\mathcal{O}}(\emptyset) \mid h(R) = \mathrm{v}^{val}\} = \{\mathrm{v}^{val} \leftarrow \emptyset\}$ (see Proposition 2).

$P_{\mathrm{v}^{val}}$ is iteratively revised against each negative example neg in $Neg_{\mathrm{v}^{val}}$. All rules R_m of $P_{\mathrm{v}^{val}}$ that match neg have to be revised. In order for $P_{\mathrm{v}^{val}}$ to remain optimal, the revision of each R_m must not match neg but still matches every other state that R_m matches. To ensure that, the least specialization (see Definition 17) is used to revise each conflicting rule R_m. Algorithm 4 shows the pseudo code of this operation. For each variable of $\mathcal{F}'$ so that $b(R_m)$ has no condition over it, a condition over another value than the one observed in state neg can be added (lines 3–8). None of those revision match neg and all states matched by R_m are still matched by at least one of its revisions.

Each revised rule can be dominated by a rule in $P_{\mathrm{v}^{val}}$ or another revised rules and thus dominance must be checked from both. But only revised rule can be dominated by a revised rule: if a rule in $P_{\mathrm{v}^{val}}$ is dominated by a revised rule, then it was dominated by its original rule and thus could not be part of $P_{\mathrm{v}^{val}}$ since it would have been discard in a previous step. Thus we can safely only check the revised rules to discard the ones dominated by the new current revised rule. The non-dominated revised rules are then added to $P_{\mathrm{v}^{val}}$.

Once $P_{\mathrm{v}^{val}}$ has been revised against all negatives example of $Neg_{\mathrm{v}^{val}}$, $P_{\mathrm{v}^{val}} = \{R \in P_{\mathcal{O}}(T) \mid h(R) = \mathrm{v}^{val}\}$, that is, $P_{\mathrm{v}^{val}}$ is the subset of rules of the final optimal program having v^{val} as head. Finally, $P_{\mathrm{v}^{val}}$ is added to P and the loop restarts with another atom. Once all values of each variable have been treated, the algorithm outputs P which is then equal to $P_{\mathcal{O}}(T)$.

Algorithm 3 GULA($\mathcal{A}'$,T,$\mathcal{F}'$,$\mathcal{T}'$)

INPUT: A set of atoms $\mathcal{A}'$, a set of transitions $T \subseteq \mathcal{S}^{\mathcal{F}'} \times \mathcal{S}^{\mathcal{T}'}$, two sets of variables $\mathcal{F}'$ and $\mathcal{T}'$
OUTPUT: $P_{\mathcal{O}}(T)$

$T := \{(s_1, \{s_2 \mid (s_1, s_2) \in T\}) \mid s_1 \in \text{first}(T)\}$ // Group transitions by initial state
$T := sort(T)$ // Sort the transitions in Lexicographical order over initial state
$P := \emptyset$
for each $\mathrm{v}^{val} \in \mathcal{A}'$ such that $\mathrm{v} \in \mathcal{T}'$ **do**
  // 1) Extraction of negative examples, (states where no successor contains v^{val})
  $Neg_{\mathrm{v}val} := \emptyset$
  for each $(s_1, S) \in T$ **do**
    $negative\_example := true$
    for each $s_2 \in S$ **do**
      if $\mathrm{v}^{val} \in s_2$ **then**
        $negative\_example := false$
        break
    if $negative\_example == true$ **then**
      $Neg_{\mathrm{v}val} := Neg_{\mathrm{v}val} \cup \{s_1\}$

  // 2) Revision of the rules of v^{val} to avoid matching of negative examples
  $P_{\mathrm{v}val} := \{\mathrm{v}^{val} \leftarrow \emptyset\}$
  for each $neg \in Neg_{\mathrm{v}val}$ **do**
    $M := \emptyset$ // Set of rules of $P_{\mathrm{v}val}$ that are in conflict
    for each $R \in P_{\mathrm{v}val}$ **do** // Extract all rules that conflict and remove them from P
      if $b(R) \subseteq neg$ **then**
        $M := M \cup \{R\}$
        $P^{val}_{\mathrm{v}} := P^{val}_{\mathrm{v}} \setminus \{R\}$

    $LS := \emptyset$
    for each $R_m \in M$ **do** // Revise each conflicting rule
      $P' := least\_specialization(R_m, neg, \mathcal{A}', \mathcal{F}')$

      for each $R_{ls} \in P'$ **do**
        $dominated := false$
        for each $R_p \in P_{\mathrm{v}val}$ **do** // Check if the revision is dominated by $P_{\mathrm{v}val}$
          if $b(R_p) \subseteq b(R_{ls})$ **then**
            $dominated := true$
            break
        if $dominated == true$ **then**
          continue

        for each $R_p \in LS$ **do** // Check if the revision is dominated LS
          if $b(R_p) \subseteq b(R_{ls})$ **then**
            $dominated := true$
            break
        if $dominated == true$ **then**
          continue

        for each $R_p \in LS$ **do**// Remove previous specialization that are now dominated
          if $b(R_{ls}) \subseteq b(R_p)$ **then**
            $LS := LS \setminus \{R_p\}$

        $LS := LS \cup \{R_{ls}\}$ // Add the revision
      $P_{\mathrm{v}val} := P_{\mathrm{v}val} \cup LS$ // Add non-dominated revisions
  $P := P \cup P_{\mathrm{v}val}$
return P

Algorithm 4 **least_specialization**(R, s, $\mathcal{A}'$, $\mathcal{F}'$) : specialize R to avoid matching of s

INPUT: a rule R, a state s, a set of atoms $\mathcal{A}'$ and a set of variables $\mathcal{F}'$
OUTPUT: a set of rules LS which is the least specialization of R by s according to $\mathcal{F}'$ and $\mathcal{A}'$.

$LS := \emptyset$
// Revise the rules by least specialization
for each $\mathrm{v}^{val} \in s$ **do**
  if $\mathrm{v} \notin \mathrm{var}(b(R))$ **then** // Add condition for all values not appearing in s
    for each $\mathrm{v}^{val'} \in \mathcal{A}', \mathrm{v} \in \mathcal{F}', val' \neq val$ **do**
      $R' := h(R) \leftarrow (b(R) \cup \{\mathrm{v}^{val'}\})$
      $LS := LS \cup \{R'\}$
return LS

F Synchronizer Scalability

Benchmark	size	synchronous	asynchronous	general
n6s1c2	6	0.2s/0.3s/0.2s/0.1s/64	2.5s/4.4s/3.6s/1s/230	9s/6s/2.9s/0.5s/1, 039
n7s3	7	1.6s/3.1s/2.5s/0.3s/128	32s/35s/26s/5s/451	139s/68s/21s/6s/2, 243
randomnet_n7k3	7	5.9s/16s/19s/6.6s/128	25s/47s/32s/5.4s/394	133s/93s/45s/9.9s/1, 580
xiao_wnt5a	7	0.96s/1.4s/1s/0.2s/128	11s/21s/12s/3s/324	25s/14s/7s/1.1s/972
arellano_rootstem	9	86s/83s/40s/2.6s/512	-/-/-/145s/1, 940	-/-/-/41s/11, 472
davidich_yeast	10	-/796s/363s/28s/1, 024	-/-/-/622s/4, 364	-/-/-/-/38, 720
faure_cellcycle	10	-/-/558s/31s/1, 024	-/-/-/865s/4, 273	-/-/-/-/30, 971
fission_yeast	10	-/-/478s/36s/1, 024	-/-/-/662s/4, 157	-/-/-/-/33, 727
mammalian	10	-/-/598s/33s/1, 024	-/-/-/841s/4, 273	-/-/-/-/30, 971

Table 2: Run time of **Synchronizer** for Boolean network benchmarks from 9 to 23 nodes for the three semantics: run time in seconds for 25%/50%/75%/100% of the transitions as input / total number of transitions with the semantics.

Table 2 shows the run time of **Synchronizer** when learning from transitions of Boolean networks from Boolenet [9] and PyBoolnet [23] with same settings as in the experiements of Table 1. For the synchronous and general semantics, its only when we are given a subset of all possible transitions that the algorithm output constraints. Those constraint at least prevent transitions from unseen states and also some combination of atoms that are missing in next states but that are observed individually. Even when it outputs an empty set of constraint, the learning process needs to produce and revises constraint until its no more possible, so run time of full set of transitions is also considered. In the asynchronous case, given a set of transitions T, it needs to learn the constraints ensuring at most one change per transitions, i.e., $\{\leftarrow a_t^i, b_t^j, a_{t-1}^{i'}, b_{t-1}^{j'} \mid a, b \in \mathcal{A}_{\overline{\mathcal{T}}}, i \neq i' \wedge j \neq j'\}$ and the ones preventing the projection when only one variable can be updated: $\{C \mid \{a_t^i, a_{t-1}^i\} \in b(C), a \in \mathcal{A}_{\overline{\mathcal{T}}}, \nexists (s, s') \in T, b(C) \subseteq s \cup s'\}$. Those second kind of constraint will be specific to the few states this limitation occurs and show the limits of propositional representation for the explanation of the dynamics.

Learning constraints is obviously more costly than learning regular rules since both features and targets variables can appear in the body, i.e., number of features becomes $|\mathcal{F}| + |\mathcal{T}|$. The algorithm reached the time out of $1,000$ seconds with benchmarks of 9 nodes. Scalability of the algorithm can be greatly improved by using the approximated version of **GULA** for learning both rules and constraints. If learning rules can be done in polynomial time, learning constraint remains exponential. Since we do not present this approximated algorithm in this paper we will not go into the details. In short, this approximated version needs positives examples and thus require to generate the Cartesian product of all applicable rules heads for each initial state observed which is exponential. Scalability, readability and applicability could be improved by considering first order generalization of both rule and constraints but those generalization are application dependant and thus remains as future work. Such generalization is required to perform proper prediction from unseen states, thus application of the synchronizer output for prediction from unseen states are out of the scope of this paper.

G Information About this Paper

G.1 History of the paper

This paper is a substantial extension of [35] where a first version of **GULA** was introduced. In [35], there was no distinction between feature and target variables, i.e., variables at time step t and $t+1$. From this consideration, interesting properties arise and allow to characterize the kind of semantics compatible with the learning process of the algorithm (Theorem 2). It also allows to represent constraints and to propose an algorithm (**Synchronizer**, Section 5) to learn programs whose dynamics can mimic any given set of transitions with optimal properties on both rules and constraints. It also allows to use **GULA** to learn human readable explanations in form of rules on static classification problems (as long as all variables are discrete), which will be one of the focus of our future works.

G.2 Main contributions of the paper

The main contributions of this paper are:

- A modeling of discrete memory-less dynamics system as multi-valued propositional logic. This modeling is independent of the dynamical semantics the system relies on, as long as it respects some given properties we provided in this paper. The main contributions of this formalism is the characterization of optimality and the study of which semantics are compatible with this formalism (which includes notably synchronous, asynchronous and general semantics).
- A first algorithm named **GULA**, to learn such optimal programs.
- The formalism is also extended to represent and use constraints. This allows to reproduce any discrete memory-less dynamical semantics behaviors inside the logic program when the original semantics is unknown.
- A second algorithm named **Synchronizer**, that exploits **GULA** to learn a logic program with constraints that can reproduce any given set of state transitions. The method we proposed is able to learn a whole system dynamics, including its semantics, in the form of a single propositional logic program. This logic program not only explains the behavior of the system in the form of human readable propositional logic rules but also is able to reproduce the behavior of the observed system without the need of knowing its semantics. Furthermore, the semantics can be explained, without any previous assumption, in the form of human readable rules inside the logic program. In other words, the approach allows to learn all the previously cited semantics, as well as new ones.

G.3 What evidence is provided

We show through theoretical results the correctness of our approach for both modeling and algorithms (see above contribution for details). Empirical evaluation is performed on benchmarks coming from biological literature. It shows the capacity of **GULA** to produce correct models when all transitions are available. Also, we observe that learned models generalize to unseen data when given a partial input in those experiments.

G.4 Related work

The paper refers to relevant related work. As we discussed in the related work section, our approach is quite related to Bain and Srinivasan [3], Evans *et al.* [10,11], Katzouris *et al.* [19], Fages [12].

The techniques we propose in this paper are a continuation of the works on the LFIT framework from [16,37,35].

In [15,17], state transitions systems are represented with logic programs, in which the state of the world is represented by an Herbrand interpretation and the dynamics that rule the environment changes are represented by a logic program P. The rules in P specify the next state of the world as an Herbrand interpretation through the *immediate consequence operator* (also called the T_P *operator*) [44,2] which mostly corresponds to the synchronous semantics we present in Section 3. In this paper, we extend upon this formalism to model multi-valued variables and any memory-less discrete dynamic semantics including synchronous, asynchronous and general semantics.

[16] proposed the LFIT framework to learn logic programs from traces of interpretation transitions. The learning setting of this framework is as follows. We are given a set of pairs of Herbrand interpretations (I, J) as positive examples such that $J = T_P(I)$, and the goal is to induce a *normal logic program* (NLP) P that realizes the given transition relations. As far as we know, this concept of *learning from interpretation transition* (LFIT) has never been considered in the ILP literature before [16]. In this paper, we propose two algorithms that extend upon this previous work: **GULA** to learn the minimal rules of the dynamics from any semantics states transitions that respect Theorem 2 and **Synchronizer** that can capture the dynamics of any memory-less discrete dynamic semantics.

H Declarations

H.1 Funding

This work was supported by JSPS KAKENHI Grant Number JP17H00763 and by the "Pays de la Loire" Region through RFI Atlanstic 2020.

H.2 Conflicts of interest/Competing interests

None

H.3 Availability of data and material

Experiments data and sources code is available at `https://github.com/Tony-sama/pylfit` under GPL-3.0 License.

H.4 Code availability

Algorithms and experiments sources code is available at `https://github.com/Tony-sama/pylfit` under GPL-3.0 License.