Les enjeux de l'inférence de modèles dynamiques des systèmes biologiques à partir de séries temporelles

Tony Ribeiro, Maxime Folschette, Laurent Trilling, Nicolas Glade, Katsumi Inoue, Morgan Magnin, Olivier Roux

Chapitre **1**

Les enjeux de l'inférence de modèles dynamiques des systèmes biologiques à partir de séries temporelles

Tony RIBEIRO[1], Maxime FOLSCHETTE[2], Laurent TRILLING[3], Nicolas GLADE[3], Katsumi INOUE[4,5], Morgan MAGNIN[4,6], Olivier ROUX[6]

[1]Chercheur Libre
[2]Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France
[3]TIMC-IMAG Laboratory, Université Grenoble Alpes - CNRS UMR 5525, La Tronche, France
[4]National Institute of Informatics, Tokyo 101-8430, Japan
[5]Tokyo Institute of Technology, Tokyo 152-8552, Japan
[6]Centrale Nantes, CNRS, LS2N, F-44000 Nantes, France

La modélisation des mécanismes de régulation biologique se décompose en deux principales tendances. La première, quantitative, repose sur les équations différentielles ordinaires mettant en jeu l'expression quantitative des composants en interaction. Cependant, ces équations sont généralement non-linéaires, ce qui empêche leur résolution analytique. De plus, les données biologiques obtenues expérimentalement sont généralement relativement bruitées, ce qui impose de les filtrer efficacement. A contrario, la deuxième tendance consiste à traiter le problème via une méthodologie discrète. Même si une telle modélisation discrète pourrait être considérée comme une abstraction moins fidèle de la réalité, elle s'est révélée efficace dans le traitement de nombreuses questions biologiques qualitatives (telles

que la compréhension des interactions entre plusieurs composants d'un système biologique, ou la détermination de l'accessibilité d'un état).

Grâce aux approches qualitatives, il devient plus abordable — bien que cela reste encore très compliqué — d'avoir une compréhension en profondeur des interactions impliquées dans un réseau génétique. Cette méthodologie vise à expliquer comment les composants d'un réseau de régulation génétique sont contrôlés les uns par les autres, ou à prédire un comportement sur la base des observations précédentes. Beaucoup de méthodes s'intéressent à ce problème d'apprentissage d'un point de vue statique, ce qui limite le pouvoir prédictif des modèles résultants. En effet, un modèle construit dans un certain contexte s'étend rarement à des conditions expérimentales légèrement différentes.

Cet enjeu de prédictabilité est pourtant d'une importance majeure, en particulier dans le domaine de la biologie de synthèse et de la conception de médicaments. Les difficultés rencontrées dans l'analyse et la prédiction peuvent être considérées comme relevant de l'une des quatre grandes catégories suivantes :

– L'inférence des interactions du système biologique étudié ;

– L'identification de paramètres du modèle ;

– L'identification des états stables et des attracteurs, propriétés clés de la plupart des systèmes biologiques ;

– Le contrôle du modèle.

Ces problématiques sont des points d'attaque cruciaux pour toute démarche de modélisation et d'analyse. Elles ont conduit à l'émergence de ce qu'on appelle désormais la biologie exécutable, dont le but est de fournir des méthodes formelles pour synthétiser automatiquement des modèles à partir d'expériences. Ces approches considèrent en entrée les données d'expression de gènes et en déduisent les régulations connexes. Une limite commune à nombre de méthodologies de ce type réside dans le fait qu'elles ne traitent pas toutes les informations contenues dans les données de séries temporelles considérées en entrée. Par exemple, les informations quantitatives de synchronisation sont abstraites, alors que c'est une caractéristique majeure pour comprendre le comportement dynamique de nombreux modèles.

Même si la plupart des approches existantes ne concernent que des données statiques, on note un intérêt croissant pour les algorithmes d'inférence qui intègrent *les aspects temporels*.

Notre objectif dans ce chapitre est de dresser un panorama des approches portant sur l'élaboration de modèles qualitatifs de réseaux de régulation, via des démarches s'apparentant au model-checking (pour l'analyse) et à la programmation logique (pour l'inférence). Nous nous attachons ici à étudier ce problème dans un *contexte*

large-échelle, autrement dit avec potentiellement plusieurs centaines de composants interagissant.

1.1. Les enjeux de l'apprentissage de séries temporelles

Diverses approches ont été récemment conçues pour s'attaquer à la rétro-ingénierie des réseaux de régulation génétiques à partir des données d'expression. Cela a conduit à l'émergence de la biologie dite exécutable, dont le but est de fournir des méthodes formelles pour automatiquement synthétiser des modèles à partir d'expériences (Koksal *et al*. 2013). La plupart d'entre elles ne sont que statiques. Mais il y a un intérêt croissant pour les algorithmes d'inférence qui intègrent les aspects temporels. Koh et al. ont récemment étudié la pertinence de ces différents algorithmes (Koh *et al*. 2009). Liu et al. ont proposé de déduire les réseaux de régulation génétiques avec retard via les réseaux bayésiens (Liu *et al*. 2004). Lopes et Bontempi ont montré que les algorithmes d'inférence comprenant des caractéristiques temporelles sont plus performants que leurs équivalents statiques (Lopes and Bontempi 2013). Le principal problème est alors d'être en mesure de déduire les délais temporels appropriés entre les influences en jeu. S'agissant d'un problème difficile, Zhang a affirmé que la question clé lors de l'analyse des données de séries temporelles consiste à segmenter les données de séries temporelles dans les différentes phases successives (Zhang 2008). Sa contribution s'est concentrée ensuite sur la résolution de ce problème de segmentation et il a montré le bien-fondé de leurs approches sur différentes études de cas.

Toutes ces approches considèrent en entrée les données d'expression génétiques et en déduisent les régulations associées. Un problème usuel des approches discrètes prenant les données d'expression en entrée réside dans la détermination d'un seuil pertinent pour définir les états actif et inactif de l'expression d'un gène. Parmi le panorama d'approches permettant de traiter les données biologiques brutes, citons les travaux de certains auteurs, tels Soinov *et al*. (2003), qui proposent une méthode originale qui ne considère pas un niveau de concentration, mais la façon dont la concentration est modifiée en présence/absence d'un régulateur. L'autre problème majeur en modélisation est la qualité des données d'expression. En d'autres termes, les données bruitées peuvent conduire à des erreurs dans le processus d'inférence. Par exemple, lorsqu'un gène est exprimé à un niveau faible, un faible rapport signal-bruit se traduirait par une mesure imprécise du comportement du gène.

Le pré-traitement des données est donc critique pour la pertinence des relations présumées entre les composants. Dans la suite de ce chapitre, nous mettons de côté ce problème. Nous supposons que les données en entrée ont déjà été pré-traitées et ont abouti à un système fiable de transitions d'états.

Nous nous concentrons donc ici sur les approches logiques pour l'apprentissage de systèmes biologiques dynamiques. Notre objectif porte sur l'élaboration (si possible

automatique) de modèles qualitatifs de réseaux de régulation, via des démarches de programmation logique.

Le caractère distinctif des démarches présentées ici réside dans la volonté de prendre en compte la dimension temporelle de ces interactions, et ce, dans le cadre de systèmes de grande dimension. Il s'agit là d'un objectif à long terme, qui passe par plusieurs étapes intermédiaires, à savoir la définition d'un formalisme logique et de méthodes efficaces pour l'inférence de réseaux, puis l'enrichissement progressif de ce formalisme et des algorithmes associés.

Dans ce chapitre, nous commencerons par nous focaliser sur des méthodes de programmation logique inductive permettant, à partir de données de séries temporelles, de construire un modèle logique d'un réseau de régulation. Nous discuterons des différentes hypothèses acceptables en termes d'apprentissage logique, intégrant notamment le non-déterminisme. Mais l'intégration de délais quantitatifs dans le modèle est parfois nécessaire pour une étude fine des propriétés dynamiques du système biologique étudié. C'est le cas, par exemple, des mécanismes en jeu dans l'horloge circadienne des mammifères. Ainsi, nous présenterons ensuite une approche, basée sur le paradigme ASP (Answer Set Programming), permettant d'inférer des réseaux de régulation avec délais à partir de données temporelles. Nous terminerons en faisant un bilan des mérites et limites des différentes méthodes d'apprentissage logique existantes, et dresserons des perspectives de recherche pour les années à venir.

1.2. Reconstruction d'un réseau de régulation (réseau booléen) et de ses règles logiques

La Programmation Logique Inductive (PLI, en anglais : *Inductive Logic Programming*) est une discipline qui étudie la construction par induction de programmes logiques à partir d'exemples et de connaissances préalables. La PLI se trouve à mi-chemin entre l'apprentissage artificiel inductif et la programmation logique. Dans le même esprit que l'apprentissage artificiel inductif, l'objectif de la PLI est le développement de méthodes de construction d'hypothèses à partir d'observations. Concrètement, cela consiste en l'extraction de connaissances générales depuis un ensemble d'exemples particuliers. Contrairement à la plupart des autres méthodes d'apprentissage inductif, la PLI se concentre surtout sur les propriétés des règles d'inférence, la convergence des algorithmes et la complexité des procédures de calcul. Depuis quelques années, certains travaux de la communauté PLI s'intéressent entre autres à l'apprentissage de la dynamique des systèmes, type machines à états, depuis leurs transitions d'état. Déterminer la dynamique d'un système a de nombreuses applications aussi bien dans les systèmes multi-agents que dans la robotique et, ou encore dans la bioinformatique. La connaissance de la dynamique d'un système peut être utilisée par les agents et les robots pour la

planification et l'ordonnancement de leurs tâches. En bioinformatique, l'apprentissage de la dynamique des systèmes biologiques peut correspondre à l'identification de l'influence des gènes et peut aider à mieux comprendre le fonctionnement de ces systèmes. Parmi ces travaux, certains représentent les systèmes à transition d'état par des programmes logiques, dans lesquels les dynamiques qui régissent les changements de l'environnement sont représentées par des règles logiques. S'inspirant de cette idée, nous avons proposé un framework permettant l'apprentissage de programmes logiques depuis les transitions d'état d'un système. Concrètement, ces transitions d'état sont nos observations du système et l'objectif est d'induire un programme logique qui réalise ces transitions. Ce faisant, les règles de ce programme capturent la dynamique du système observé. Pour résumer, depuis l'observation de leurs interactions locales, nous déterminons les influences entre les différents composants d'un système. Cette méthode permet, entre autres, d'apprendre un réseau booléen ou l'identification d'automates cellulaires en observant leurs différentes traces d'exécution. Cette technique peut être appliquée en bioinformatique, en particulier pour l'identification de réseaux de régulation génétique à partir de résultats d'expériences de laboratoire.

Dans la section 1.2.1 nous formalisons les notions de programmation logique nécessaires à notre modélisation des systèmes dynamiques. La section 1.2.2 formalise la révision de programme logique à des fins de modélisation automatique de la dynamique. La section 1.2.3 se focalise sur la formalisation des sémantiques synchrone, asynchrone et généralisée. La section 1.2.4 présente l'algorithme **GULA** (General Usage LFIT Algorithm) qui permet d'apprendre la dynamique d'un système depuis ses transitions d'état discret indépendamment de la sémantique en présence. La section 1.2.5 présente l'algorithme **PRIDE** (Polynomial Relational Inference of Dynamic Environnement) qui est une version approximative de GULA avec une complexité polynomiale. Les quatre premières sections de 1.2 sont une traduction de notre article Ribeiro *et al.* (2018), la dernière section est originale. Une implémentation python de ces deux algorithmes est disponible sous licence libre GPL à l'adresse suivante : `https://github.com/Tony-sama/pylfit`.

1.2.1. *Logique multi-valuée*

Soit $\mathcal{V} = \{\mathrm{v}_1, \ldots, \mathrm{v}_n\}$ un ensemble fini de $n \in \mathbb{N}$ variables et $\mathsf{dom} : \mathcal{V} \to \mathbb{N}$ une fonction qui associe une valeur maximale (et donc un domaine) à chaque variable. Les atomes d'une Logique Multi-Valuée ($\mathcal{L}$MV) sont de la forme v^{val} où $\mathrm{v} \in \mathcal{V}$ et $val \in [\![0; \mathsf{dom}(\mathrm{v})]\!]$. Pour un ensemble de variables $\mathcal{V}$ et une fonction de domaine dom, l'ensemble des atomes logiques correspondant est représenté par $\mathcal{A}^{\mathcal{V}}_{\mathsf{dom}}$.

Une règle $\mathcal{L}$MV est définie par :

$$R \;=\; \mathrm{v}_0^{val_0} \;\leftarrow\; \mathrm{v}_1^{val_1} \wedge \cdots \wedge \mathrm{v}_m^{val_m} \qquad [1.1]$$

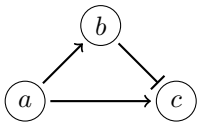

Figure 1.1 – Exemple de graphe d'interaction de réseau de régulation représentant une boucle de réaction positive vers l'avant (*feed-forward loop*) où a influence positivement b et c, tandis que b (et donc, indirectement, a) influence négativement c.

où $\forall i \in [\![0; m]\!], \mathrm{v}_i^{val_i} \in \mathcal{A}_{\mathsf{dom}}^{\mathcal{V}}$ sont des atomes de $\mathcal{L}\mathrm{MV}$, et où chaque variable est mentionnée au plus une fois dans la partie droite : $\forall j, k \in [\![1; m]\!], j \neq k \Rightarrow \mathrm{v}_j \neq \mathrm{v}_k$. Intuitivement, la règle R a le sens suivant : « si, pour chaque $i \in [\![1; m]\!]$, la variable v_i a la valeur val_i dans l'état actuel du système, alors la variable v_0 peut prendre la valeur val_0 dans le prochain état ».

L'atome dans la partie gauche de la règle, à gauche de la flèche, est appelé la *tête* de R et est noté $h(R) := \mathrm{v}_0^{val_0}$. La notation $\mathrm{var}(h(R)) := \mathrm{v}_0$ désigne la variable qui apparait dans $h(R)$. La conjonction dans la partie droite de la règle, à droite de la flèche, est appelée le *corps* ou les *conditions* de R, noté $b(R)$ et peut être assimilée à l'ensemble $\{\mathrm{v}_1^{val_1}, \ldots, \mathrm{v}_m^{val_m}\}$; les opérations ensemblistes tel que $\in$ et $\cap$ s'appliquent donc dessus. Un *programme logique multi-valué* ($\mathcal{P}\mathcal{L}\mathrm{MV}$) est un ensemble de règles $\mathcal{L}\mathrm{MV}$.

Dans ce qui suit, nous travaillerons sur un ensemble $\mathcal{V}$ et une fonction dom spécifique que nous omettrons de mentionner quand le contexte est sans ambiguïté, et l'ensemble des atomes logiques sera simplement désigné par $\mathcal{A}$.

Nous définissons dans la suite plusieurs notions sur les règles et programmes de $\mathcal{L}\mathrm{MV}$ qui seront utilisées dans l'apprentissage de la dynamique. La définition 1.1 introduit une relation de domination entre règles qui définit un ordre partiel antisymétrique, tel qu'explicité par la proposition 1.1.

EXEMPLE 1.1.– *La figure 1.1 donne un exemple de réseau de régulation comportant trois éléments a, b et c. L'information de ce réseau n'est pas complète ; notamment, la « force » relative des composants a et b sur le composant c n'est pas explicite. Plusieurs dynamiques sont donc possibles sur ce réseau, dont deux sont données par les programmes suivants définis sur $\mathcal{A} = \{a^0, a^1, b^0, b^1, c^0, c^1\}$. Dans le programme 2, les deux règles en rouge introduisent potentiellement du non-déterminisme dans la dynamique (voir l'exemple 1.7 pour plus de détails).*

Programme 1
$a^0 \leftarrow a^0$
$a^1 \leftarrow a^1$
$b^0 \leftarrow a^0$
$b^1 \leftarrow a^1$
$c^0 \leftarrow a^0$
$c^0 \leftarrow b^1$
$c^1 \leftarrow a^1 \wedge b^0$

Programme 2
$a^1 \leftarrow \emptyset$
$b^0 \leftarrow a^0$
$b^1 \leftarrow a^1$
$c^0 \leftarrow a^0$
$c^0 \leftarrow b^1$
$c^1 \leftarrow a^1$

DÉFINITION 1.1 (Domination de règles).– *Soient R_1, R_2 deux règles $\mathcal{L}$MV. La règle R_1* domine *R_2, noté $R_2 \leq R_1$, si $h(R_1) = h(R_2)$ et $b(R_1) \subseteq b(R_2)$.*

EXEMPLE 1.2 (Domination).– *Dans le programme 1 de l'exemple 1.1, la règle $a^1 \leftarrow a^1$ domine $a^1 \leftarrow a^1 \wedge b^0$ qui domine à la fois $a^1 \leftarrow a^1 \wedge b^0 \wedge c^0$ et $a^1 \leftarrow a^1 \wedge b^0 \wedge c^1$.*

PROPOSITION 1.1.– *Soient R_1, R_2 deux règles $\mathcal{L}$MV. Si $R_1 \leq R_2$ et $R_2 \leq R_1$ alors $R_1 = R_2$.*

Les règles avec les corps les plus simples dominent les autres règles. En pratique, ce sont ces règles qui nous intéressent vu qu'elles couvrent les cas les plus généraux.

Le système dynamique dont nous voulons apprendre les règles est représenté par une succession d'*états* telle que formellement décrite dans la définition 1.2. Nous définissons également la notion de « compatibilité » d'une règle avec un état à la définition 1.3, et avec une autre règle à la définition 1.4 ; la proposition 1.2 formalise une propriété utile sur cette dernière notion.

DÉFINITION 1.2 (État discret).– *Un* état discret *s est une fonction de $\mathcal{V}$ dans $\mathbb{N}$, c.-à-d. qui associe un entier à chaque variable dans $\mathcal{V}$. Il peut être représenté de manière équivalente par l'ensemble d'atomes $s = \{\mathrm{v}^{s(\mathrm{v})} \mid \mathrm{v} \in \mathcal{V}\}$ permettant d'y appliquer les opérations ensemblistes classiques. $\mathcal{S}$ dénote l'ensemble de tous les états discrets, une paire d'états $(s, s') \in \mathcal{S}^2$ est appelée une* transition*, et s' est appelé* successeur *de s.*

EXEMPLE 1.3.– *L'ensemble des états possibles d'un programme défini sur un ensemble d'atomes $\mathcal{A} = \{a^0, a^1, b^0, b^1, c^0, c^1\}$ est : $\{\{a^0, b^0, c^0\}, \{a^0, b^0, c^1\}, \{a^0, b^1, c^0\}, \{a^0, b^1, c^1\}, \{a^1, b^0, c^0\}, \{a^1, b^0, c^1\}, \{a^1, b^1, c^0\}, \{a^1, b^1, c^1\}\}$.*

DÉFINITION 1.3 (Couverture règle-état).– *Soit $s \in \mathcal{S}$. La règle $\mathcal{L}$MV R* couvre *s, noté $R \sqcap s$, si $b(R) \subseteq s$.*

EXEMPLE 1.4.– *Soit $\mathcal{A} = \{a^0, a^1, b^0, b^1, c^0, c^1\}$. La règle $c^0 \leftarrow a^1 \wedge b^1 \wedge c^1$ ne couvre que l'état $\{a^1, b^1, c^1\}$. La règle $c^0 \leftarrow a^0 \wedge b^1$ couvre $\{a^0, b^1, c^0\}$ et $\{a^0, b^1, c^1\}$. La règle $b^1 \leftarrow a^1$ couvre $\{a^1, b^0, c^0\}$, $\{a^1, b^0, c^1\}$, $\{a^1, b^1, c^0\}$, $\{a^1, b^1, c^1\}$.*

DÉFINITION 1.4 (Intersection de règles).– *Soient R et R' deux règles* $\mathcal{L}$MV. *Ces règles s'*intersectent*, ce qui est noté $R \sqcap R'$, quand il existe $s \in \mathcal{S}$ tel que $R \sqcap s$ et $R' \sqcap s$.*

PROPOSITION 1.2 (Intersection de règles).– *Soient R et R' deux règles* $\mathcal{L}$MV.

$$R \sqcap R' \textit{ ssi } \forall \mathrm{v} \in \mathcal{V}, \forall val, val' \in \mathbb{N}, (\mathrm{v}^{val}, \mathrm{v}^{val'}) \in b(R) \times b(R') \implies val = val'.$$

EXEMPLE 1.5.– *Les règles $a^0 \leftarrow a^0$ et $c^1 \leftarrow b^1$ s'intersectent, car elles couvrent toutes deux par exemple l'état $\{a^0, b^1, c^0\}$. Étant donné que l'union des corps des deux règles ne contient pas plus d'une valeur par variable, on peut déduire leur intersection directement et construire les états qu'elles couvrent mutuellement : ici, il s'agit de tous les états contenant $\{a^0\} \cup \{b^1\} = \{a^0, b^1\}$.*

Le programme final que nous voulons apprendre doit être complet et cohérent avec les transitions observées. Les définitions qui suivent formalisent ces propriétés requises. Dans la définition 1.5, nous caractérisons le fait qu'une règle d'un programme est utile pour décrire la dynamique d'une variable dans une transition ; cette notion est étendue à un programme et un ensemble de transitions, à condition qu'il existe une telle règle pour chaque variable et chaque transition. Une incohérence (définition 1.6) se caractérise par une règle qui décrit un changement qui n'apparaît pas dans l'ensemble de transitions considéré. Deux règles sont concurrentes (définition 1.8) si elles s'intersectent mais ont une tête différente pour la même variable. Enfin, les définitions 1.7 et 1.9 formalisent les caractéristiques d'un programme complet (l'ensemble de la dynamique est couverte) et cohérent (sans règle incohérente).

DÉFINITION 1.5 (Réalisation de transitions par une règle et un programme).– *Soient R une règle* $\mathcal{L}$MV *et $(s, s') \in \mathcal{S}^2$. La règle R* réalise *la transition (s, s'), noté $s \xrightarrow{R} s'$, si $R \sqcap s \wedge h(R) \in s'$.*

Un $\mathcal{PL}$MV *P* réalise *une transition $(s, s') \in \mathcal{S}^2$, noté $s \xrightarrow{P} s'$, si $\forall \mathrm{v} \in \mathcal{V}, \exists R \in P, \mathrm{var}(h(R)) = \mathrm{v} \wedge s \xrightarrow{R} s'$. Il* réalise *un ensemble de transitions $T \subseteq \mathcal{S}^2$, noté $\overset{P}{\hookrightarrow} T$, si $\forall (s, s') \in T, s \xrightarrow{P} s'$.*

EXEMPLE 1.6.– *La règle $c^1 \leftarrow a^1 \wedge b^1$ réalise la transition $(\{a^1, b^1, c^0\}, \{a^1, b^1, c^1\})$ car elle couvre l'état initial et sa conclusion est contenue dans l'état suivant. La transition $(\{a^1, b^1, c^0\}, \{a^1, b^1, c^1\})$ est réalisée par le programme 2 de l'exemple 1.1, mais pas le programme 1.*

Dans ce qui suit, pour tout ensemble de transitions $T \subseteq \mathcal{S}^2$, on dénote : $\mathrm{fst}(T) := \{s \in \mathcal{S} \mid \exists (s_1, s_2) \in T, s_1 = s\}$. Notons que $\mathrm{fst}(T) = \emptyset \implies T = \emptyset$.

DÉFINITION 1.6 (Cohérence et incohérence).– *Une règle* $\mathcal{L}\mathrm{MV}$ R *est* incohérente *avec un ensemble de transitions* $T \subseteq \mathcal{S}^2$ *si* $\exists s \in \mathrm{fst}(T), \big(R \sqcap s \wedge \forall (s, s') \in T, h(R) \notin s'\big)$. *Sinon, la règle est dite* cohérente *avec* T.

Une règle est cohérente si pour tout état initial des transitions de T (fst(T)) couvert par la règle, il existe une transition de T dont elle vérifie la conclusion.

DÉFINITION 1.7 (Programme cohérent).– *Un* $\mathcal{PL}\mathrm{MV}$ P *est* cohérent *avec un ensemble de transitions* $T \subseteq \mathcal{S}^2$ *si* P *ne contient aucune règle incohérente avec* T.

DÉFINITION 1.8 (Règles concurrentes).– *Deux règles* $\mathcal{L}\mathrm{MV}$ R *et* R' *sont dites* concurrentes *si* $R \sqcap R' \wedge \mathrm{var}(h(R)) = \mathrm{var}(h(R')) \wedge h(R) \neq h(R')$.

Deux règles concurrentes sont donc deux règles qui s'intersectent et qui concluent différemment sur la même variable, causant alors potentiellement du non-déterminisme.

DÉFINITION 1.9 (Programme complet).– *Un* $\mathcal{PL}\mathrm{MV}$ P *est* complet *si* $\forall s \in \mathcal{S}, \forall \mathrm{v} \in \mathcal{V}, \exists R \in P, R \sqcap s \wedge \mathrm{var}(h(R)) = \mathrm{v}$.

Ainsi, un programme complet propose au moins un état suivant pour chaque variable et à partir de n'importe quel état.

EXEMPLE 1.7.– *Les programmes 1 et 2 de l'exemple 1.1 sont tous les deux complets. Le programme 2 contient les deux règles concurrentes indiquées en rouge :* $c^0 \leftarrow b^1$ *et* $c^1 \leftarrow a^1$. *En effet, ces règles couvrent notamment toutes deux l'état* $\{a^1, b^1, c^1\}$ *et donnent une valeur suivante différente pour c.*

1.2.2. *Opérations d'apprentissage*

Cette section se concentre sur la manipulation de programmes dans le processus d'apprentissage. Les définitions 1.10 et 1.11 formalisent les principales opérations utilisées sur une règle ou un programme par l'algorithme d'apprentissage, dont l'objectif est de produire une modification minimale d'un $\mathcal{PL}\mathrm{MV}$ donné afin qu'il soit cohérent avec un nouvel ensemble de transitions. De façon informelle, l'objectif est de considérer itérativement de nouvelles transitions (s, s'), et d'ajouter des atomes au corps de certaines règles pour que celles-ci ne couvrent plus l'état s, et ainsi conserver la cohérence du programme général.

DÉFINITION 1.10 (Spécialisation minimale de règle).– *Soient* R *une règle* $\mathcal{L}\mathrm{MV}$ *et* $s \in \mathcal{S}$ *tel que* $R \sqcap s$. *La* spécialisation minimale *de* R *par* s *est :*

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

La spécialisation minimale $L_{\text{spe}}(R, s)$ produit un ensemble de règles qui couvrent tous les états couverts par R exception faite de s. Donc $L_{\text{spe}}(R, s)$ réalise toutes les transitions que R réalise sauf celles commençant par s.

EXEMPLE 1.8.– *Soient* $R := c^1 \leftarrow a^1$ *et* $s := \{a^1, b^0, c^1\}$. *On note que* R *couvre* s. *La spécialisation minimale de* R *par* s *est :* $L_{\text{spe}}(R, s) = \{c^1 \leftarrow a^1 \wedge c^0, c^1 \leftarrow a^1 \wedge b^1\}$, *où les atomes en rouge sont les spécialisations minimales.*

DÉFINITION 1.11 (Révision minimale de programme).– *Soient* P *un* $\mathcal{PL}$MV, $s \in \mathcal{S}$ *et* $T \subseteq \mathcal{S}^2$ *tel que* $\text{fst}(T) = \{s\}$. *Soit* $R_P := \{R \in P \mid R \text{ incohérente avec } T\}$. *La* révision minimale *de* P *par* T *est* $L_{\text{rev}}(P, T) := (P \setminus R_P) \cup \bigcup_{R \in R_P} L_{\text{spe}}(R, s)$.

La révision minimale $L_{\text{rev}}(P, T)$ d'un programme P par un ensemble de transitions T, tel que toutes les transitions de T partent du même état s, réalise l'ensemble des transitions de P à l'exception des transitions $(s, s') \notin T$. En d'autres termes, un programme peut être ainsi revu de façon à ne pas produire d'autres comportements que ceux observés.

EXEMPLE 1.9.– *Soient un programme* P *et un ensemble de deux transitions* T *:*

$$P := \left\{ \begin{array}{l} a^1 \leftarrow a^1 \\ b^0 \leftarrow a^0 \\ b^1 \leftarrow a^1 \\ c^0 \leftarrow a^0 \\ c^1 \leftarrow a^1 \wedge b^0 \end{array} \right\}, \quad T := \left\{ \begin{array}{l} (\{a^1, b^0, c^0\}, \{a^0, b^1, c^0\}), \\ (\{a^1, b^0, c^0\}, \{a^0, b^0, c^0\}) \end{array} \right\} .$$

Dans le programme P, *seules les règles* $a^1 \leftarrow a^1$ *et* $c^1 \leftarrow a^1 \wedge b^0$ *sont incohérentes avec les transitions de* T. *Donc la révision minimale de* P *par* T *est l'union des révisions minimales de ces deux règles par chaque transition de* T *:*

$$L_{\text{rev}}(P, T) = \{a^1 \leftarrow a^1 \wedge b^1, a^1 \leftarrow a^1 \wedge c^1\} \cup \{c^1 \leftarrow a^1 \wedge b^0 \wedge c^1\}.$$

Le théorème 1.1 donne des propriétés de la spécialisation minimale, afin de justifier son utilisation dans l'algorithme d'apprentissage.

THÉORÈME 1.1.– *Soit* R *une règle* $\mathcal{L}$MV *et* $s \in \mathcal{S}$ *tel que* $R \sqcap s$. *Soit* $S_R := \{s' \in \mathcal{S} \mid R \sqcap s'\}$ *et* $S_{\text{spe}} := \{s' \in \mathcal{S} \mid \exists R' \in L_{\text{spe}}(R, s), R' \sqcap s'\}$.

Soient P *un* $\mathcal{PL}$MV *et* $T, T' \subseteq \mathcal{S}^2$ *tels que* $|\text{fst}(T)| = 1 \wedge \text{fst}(T) \cap \text{fst}(T') = \emptyset$. *Les résultats suivants sont vrais :*

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

2) $L_{\text{rev}}(P, T)$ *est cohérent avec* T,

3) $\overset{P}{\hookrightarrow} T' \implies \overset{L_{\text{rev}}(P,T)}{\hookrightarrow} T'$,

4) $\overset{P}{\hookrightarrow} T \implies \overset{L_{\mathrm{rev}}(P,T)}{\hookrightarrow} T$*,*

5) P *est complet* $\implies L_{\mathrm{rev}}(P,T)$ *est complet.*

Informellement, les différents points du théorème 1.1 ont les significations suivantes :

1) La spécialisation minimale d'une règle par un état ne retire que la couverture de cet état.

2) La révision d'un programme le rend cohérent avec l'ensemble de transitions donné.

3) Ce que réalise le programme depuis tout autre état de départ reste inchangé une fois le programme révisé.

4) La réalisation des transitions observées est conservée.

5) Si le programme est complet, sa révision l'est aussi.

ESQUISSE DE PREUVE. Les deux premiers points découlent des définitions 1.10 et 1.11. Le troisième point découle de la définition 1.5 et du premier point. Le quatrième point découle des définitions 1.5 et 1.11. Le dernier point découle de la définition 1.9 et du premier point. La preuve complète est donnée dans Ribeiro *et al.* (2018). □

La définition 1.12 regroupe toutes les propriétés que le programme appris doit posséder : pertinence et optimalité. Notre objectif est l'obtention d'un programme P permettant de reproduire et expliquer les observations T. Pour cela, le programme doit être cohérent avec ces observations T et les réaliser, ce qui signifie qu'à partir des états de départ de T ($\mathrm{fst}(T)$), il produit exactement les transitions de T. Il doit aussi être complet, ce qui signifie qu'il doit proposer un futur pour tout état possible, même ceux n'apparaissant pas dans T. Enfin, il n'existe pas de règle cohérente avec T qui ne soit pas dominée par une règle de P. Ces quatre propriétés définissent la pertinence d'un programme P ; un tel programme est également optimal lorsqu'aucune de ses règles n'est dominée par une autre.

DÉFINITION 1.12 (Programme pertinent et optimal).– *Soit* $T \subseteq \mathcal{S}^2$*. Un* $\mathcal{PL}\mathrm{MV}$ P *est* pertinent *pour* T *quand :*

1) P *est cohérent avec* T*,*

2) P *réalise* T*,*

3) P *est complet*

4) pour toutes les règles $\mathcal{L}\mathrm{MV}$ R *qui sont cohérentes avec* T*, il existe* $R' \in P$ *tel que* $R \leq R'$*.*

De plus, P *est appelé* optimal *quand :*

5) pour toute règle $R \in P$, toutes les règles $\mathcal{L}\mathrm{MV}$ R' appartenant à un $\mathcal{PL}\mathrm{MV}$ pertinent pour T sont telles que $R \leq R'$ implique $R' \leq R$, i.e. $R' = R$.

La proposition 1.3 montre que le programme optimal d'un ensemble de transitions est unique.

PROPOSITION 1.3.– *Soit $T \subseteq \mathcal{S}^2$. Le $\mathcal{PL}\mathrm{MV}$ optimal pour T est unique. On le dénote $P_{\mathcal{O}}(T)$.*

Démonstration. En raisonnant par contradiction, s'il existe deux programmes optimaux P, P', ils diffèrent par au moins une règle R qui apparaît dans P mais pas P'. Par le 4$^{\text{e}}$ point, une règle R' dans P' doit la dominer. D'après le 5$^{\text{e}}$ point, $R \leq R' \implies R' \leq R \implies R = R'$, donc $R \in P'$. □

Les prochaines propriétés sont directement utilisées dans l'algorithme d'apprentissage. La proposition 1.4 donne une définition explicite du programme optimal pour un ensemble vide de transitions, ce qui est le point de départ de l'algorithme. L'intuition derrière $P_{\mathcal{O}}(\emptyset)$ est qu'en l'absence d'observation sur un état donné, tout est possible depuis cet état.

PROPOSITION 1.4.– $P_{\mathcal{O}}(\emptyset) = \{\mathrm{v}^{val} \leftarrow \emptyset \mid \mathrm{v}^{val} \in \mathcal{A}\}$.

ESQUISSE DE PREUVE. Par construction. La preuve complète est donnée dans Ribeiro *et al*. (2018). □

Le théorème 1.2 garantit que toute révision minimale conserve la pertinence d'un programme.

THÉORÈME 1.2.– *Soient $s \in \mathcal{S}$ et $T, T' \subseteq \mathcal{S}^2$ tels que $|\mathrm{fst}(T')| = 1 \wedge \mathrm{fst}(T) \cap \mathrm{fst}(T') = \emptyset$. $L_{\mathrm{rev}}(P_{\mathcal{O}}(T), T')$ est un $\mathcal{PL}\mathrm{MV}$ pertinent pour $T \cup T'$.*

ESQUISSE DE PREUVE. La cohérence est prouvée par contradiction. La complétude et la réalisation découlent du théorème 1.1. Le dernier point est prouvé en exhibant pour chaque règle R cohérente avec $T' \cup T$ la règle dans $L_{\mathrm{spe}}(P_{\mathcal{O}}(T'), T)$ qui la domine. La preuve complète est donnée dans Ribeiro *et al*. (2018). □

Enfin, la proposition 1.5 donne une méthode pour obtenir le programme optimal depuis n'importe quel programme pertinent, simplement en retirant les règles dominées.

PROPOSITION 1.5.– *Soit $T \subseteq \mathcal{S}^2$. Si P est un $\mathcal{PL}\mathrm{MV}$ pertinent pour T, alors $P_{\mathcal{O}}(T) = \{R \in P \mid \forall R' \in P, R \leq R' \implies R' \leq R\}$*

Ces trois derniers points donnent l'idée générale du processus d'apprentissage. L'ensemble vide de la proposition 1.4 sert de point de départ au processus, et le

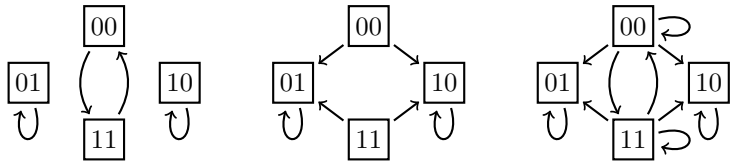

Figure 1.2 – Représentation graphique des comportements permis par les trois sémantiques présentées dans cette section pour le programme : $\{a^0 \leftarrow b^1, a^1 \leftarrow b^0, b^0 \leftarrow a^1, b^1 \leftarrow a^0\}$. Ici, $\boxed{01}$ représente l'état discret $\{a^0, b^1\}$.

programme initial produit est par définition pertinent. Le théorème 1.2 permet de calculer itérativement des révisions de programme à partir du programme initial, tout en conservant sa pertinence. Enfin, il suffit d'appliquer la proposition 1.5 pour obtenir l'optimalité à partir de la pertinence.

1.2.3. *Sémantiques de dynamiques*

On appelle « sémantique » la description du mode de mise à jour choisi pour un programme donné ; autrement dit, la façon dont on interprète un programme logique pour calculer l'ensemble des états suivants possibles pour un état donné. Les notions développées précédemment sont indépendantes de la sémantique choisie pour l'exécution ou l'apprentissage d'un programme logique donné. Dans cette section, nous cherchons à formaliser la notion de sémantique, puis nous en donnons des exemples courants (synchrone, asynchrone, généralisée) qui sont compatibles avec notre méthode d'apprentissage. La figure 1.2 donne un exemple d'application des trois sémantiques qui seront décrites dans la suite.

La définition 1.13 commence par formaliser la notion de sémantique, en tant que politique de mise à jour basée sur un programme. De façon plus formelle, une sémantique est une fonction qui, à un programme complet, associe un ensemble de transitions où chaque état apparaît au départ d'au moins une transition. Cet ensemble de transitions peut lui-même être considéré comme une fonction qui, à un état, associe un ensemble non-vide d'états vus comme autant de possibles ramifications dynamiques.

DÉFINITION 1.13 (Sémantiques).– *Soient $\mathcal{A}^{\mathcal{V}}_{\mathsf{dom}}$ un ensemble d'atomes et $\mathcal{S}$ l'ensemble d'états correspondant. Une* sémantique *(sur $\mathcal{A}^{\mathcal{V}}_{\mathsf{dom}}$) est une fonction qui associe, à chaque $\mathcal{PL}$MV complet P, un ensemble de transitions $T \subseteq \mathcal{S}^2$ tel que : $\mathrm{fst}(T) = \mathcal{S}$. De façon équivalente, une sémantique peut être vue comme une fonction*

de $\left(\text{c-}\mathcal{PL}\text{MV} \rightarrow (\mathcal{S} \rightarrow \wp(\mathcal{S}) \setminus \emptyset)\right)$ *où* c-$\mathcal{PL}$MV *est l'ensemble de* $\mathcal{PL}$MV*complets pour* $\mathcal{S}$ *et* $\wp(\mathcal{S})$ *est l'ensemble des parties de* $\mathcal{S}$.

Dans ce qui suit, nous présentons à la fois une définition formelle et une caractérisation de trois sémantiques particulières qui sont répandues dans le domaine des systèmes dynamiques discrets complexes : la sémantique (purement) synchrone, la sémantique (purement) asynchrone et la sémantique généralisée. Nous traitons également le cas particulier de la sémantique synchrone déterministe. Pour la suite de cette section, nous considérerons un ensemble d'atomes $\mathcal{A}$ et un ensemble d'états discrets $\mathcal{S}$ donnés. De plus, il sera sous-entendu que les sémantiques présentées respectent la définition 1.13, autrement dit, qu'il s'agit de fonctions qui, à tout élément de c-$\mathcal{PL}$MV (l'ensemble des $\mathcal{PL}$MV complets pour $\mathcal{S}$), associent un élément $T \subseteq \mathcal{S}^2$ tel que $\text{fst}(T) = \mathcal{S}$.

Notons que certains aspects dans ces définitions sont arbitraires et pourraient être discutés en fonction du paradigme de modélisation. Cela concerne par exemple les règles R telles que $\exists s \in \mathcal{S}, s \sqcap R \wedge h(R) \in s$, modélisant la stabilité dynamique de certains états, qu'on peut choisir d'inclure (comme pour les sémantiques synchrone et généralisée) ou d'exclure (comme pour la sémantique asynchrone) des dynamiques possibles. La méthode d'apprentissage présentée ici est indépendante de la sémantique considérée tant que celle-ci respecte la définition 1.13.

La définition 1.14 introduit la sémantique synchrone, qui consiste à mettre à jour toutes les variables du modèle, en fonction des règles du programme, afin de calculer le prochain état. Cependant, mise à jour n'implique pas nécessairement changement de valeur : les règles faisant changer la valeur de la variable n'ont pas priorité sur les règles qui la conservent. De plus, si plusieurs règles statuant sur la même variable couvrent l'état actuel mais possèdent une conclusion différente, alors plusieurs transitions différentes sont possibles, en fonction de la règle appliquée. Ainsi, pour que l'auto-transition (s, s) se produise, il est nécessaire d'avoir, pour chaque atome $\text{v}^{val} \in s$, une règle qui couvre s et dont la conclusion est v^{val}. Notons cependant qu'une telle boucle n'est pas nécessairement un état stable (aussi appelé attracteur singleton) car d'autres règles peuvent permettre de sortir de l'état ; c'est uniquement le cas si toutes les règles de P qui couvrent s ont leur conclusion dans s.

DÉFINITION 1.14 (Semantique synchrone).– *La* sémantique synchrone $\mathcal{T}_{syn}$ *est définie par :*

$$\mathcal{T}_{syn} : P \mapsto \{(s, s') \in \mathcal{S}^2 \mid s' \subseteq \{h(R) \in \mathcal{A} \mid R \in P, R \sqcap s\}\}$$

On note que si deux transitions (s, s') et (s, s'') sont observées depuis le même état s, alors tous les états qui sont des combinaisons de s' et s'' sont aussi des successeurs de s. Cette propriété est utilisée dans la proposition 1.6 comme une caractérisation de la sémantique synchrone.

Dans ce qui suit, si $s \in \mathcal{S}$ est un état et $X \subseteq \mathcal{A}$ est un ensemble d'atomes tel que $\forall \mathrm{v}_1^{val_1}, \mathrm{v}_2^{val_2} \in X, \mathrm{v}_1 = \mathrm{v}_2 \implies val_1 = val_2$, on dénote : $s \backslash\backslash X := \{\mathrm{v}^{val} \in s \mid \mathrm{v} \notin \{\mathrm{w} \mid \mathrm{w}^{val'} \in X\}\} \cup X$. En d'autres termes, $s \backslash\backslash X$ est l'état discret s où toutes les variables mentionnées dans X voient leur valeur remplacée par leur valeur dans X.

PROPOSITION 1.6 (Transitions synchrones).– *Soit $T \subseteq \mathcal{S}^2$ tel que* $\mathrm{fst}(T) = \mathcal{S}$. *Les transitions de T sont* synchrones*, c.-à-d. $\exists P$ un $\mathcal{PL}$MV tel que $\mathcal{T}_{syn}(P) = T$, si et seulement si* $\forall (s, s_1), (s, s_2) \in T, \forall s_3 \in \mathcal{S}, s_3 \subseteq s_1 \cup s_2 \implies (s, s_3) \in T$.

ESQUISSE DE PREUVE. ($\Rightarrow$) Par définition de $\mathcal{T}_{syn}$. ($\Leftarrow$) Considérer le programme le plus naïf P qui réalise T ; appliquer itérativement la caractérisation permet de conclure que $\mathcal{T}_{syn}(P) \subseteq T$, tandis que $T \subseteq \mathcal{T}_{syn}(P)$ découle par définition de P et de $\mathcal{T}_{syn}$. La preuve complète est donnée dans Ribeiro *et al.* (2018). □

Dans la définition 1.15, nous formalisons la sémantique asynchrone qui impose qu'au plus une variable peut changer de valeur à chaque transition. Contrairement à la précédente, cette sémantique priorise le changement. Ainsi, pour que l'auto-transition (s, s) se produise, il est nécessaire que toutes les règles de P qui couvrent s aient leur conclusion dans s, ce qui n'arrive que lorsque (s, s) est un état stable. La proposition 1.7 caractérise la sémantique asynchrone en statuant que depuis un état s, soit le seul successeur est s lui-même, soit tous les successeurs diffèrent de s par exactement un atome.

DÉFINITION 1.15 (Sémantique asynchrone).– *La* sémantique asynchrone $\mathcal{T}_{asyn}$ *est définie par :*

$$\mathcal{T}_{asyn} : P \mapsto \{(s, s \backslash\backslash \{h(R)\}) \in \mathcal{S}^2 \mid R \in P \wedge R \sqcap s \wedge h(R) \notin s\}$$
$$\cup \{(s, s) \in \mathcal{S}^2 \mid \forall R \in P, R \sqcap s \implies h(R) \in s\}.$$

PROPOSITION 1.7 (Transitions asynchrones).– *Soit $T \subseteq \mathcal{S}^2$ tel que* $\mathrm{fst}(T) = \mathcal{S}$. *Les transitions de T sont* asynchrones*, c.-à-d. $\exists P$ un $\mathcal{PL}$MV tel que $\mathcal{T}_{asyn}(P) = T$, si et seulement si* $\forall s, s' \in \mathcal{S}, s \neq s', \big((s, s) \in T \implies (s, s') \notin T\big) \wedge \big((s, s') \in T \implies |s \setminus s'| = 1\big)$.

ESQUISSE DE PREUVE. Considérer séparément les cas $s = s'$ et $s \neq s'$. ($\Rightarrow$) Par contradiction, en se basant sur la définition de $\mathcal{T}_{asyn}$. ($\Leftarrow$) En considérant le programme le plus naïf P qui réalise T ; depuis la caractérisation, il découle : $\mathcal{T}_{syn}(P) = T$. La preuve complète est donnée dans Ribeiro *et al.* (2018). □

La définition 1.16 formalise la sémantique généralisée comme une version plus permissive de la sémantique synchrone : n'importe quel sous-ensemble de variables peut changer sa valeur dans une transition. Une auto-transition (s, s) existe donc pour chaque état s puisqu'un ensemble vide de variables peut toujours être sélectionné pour

la mise à jour. Cependant, comme pour la sémantique synchrone, cette auto-transition n'est un attracteur que si toutes les règles de P qui couvrent s ont leur conclusion dans s. La proposition 1.8 est une caractérisation de la sémantique généralisée. Elle est similaire à la caractérisation du synchrone, mais la combinaison des deux états successeurs est aussi combinée avec l'état d'origine.

DÉFINITION 1.16 (Semantique généralisée).– *La* sémantique généralisée $\mathcal{T}_{gen}$ *est définie par :*

$$\mathcal{T}_{gen} : P \mapsto \{(s, s \backslash\backslash r) \in \mathcal{S}^2 \mid r \subseteq \{h(R) \in \mathcal{A} \mid R \in P \wedge R \sqcap s\} \wedge \\ \forall \mathrm{v}_1^{val_1}, \mathrm{v}_2^{val_2} \in r, \mathrm{v}_1 = \mathrm{v}_2 \implies val_1 = val_2\}.$$

PROPOSITION 1.8 (Transitions généralisées).– *Soit* $T \subseteq \mathcal{S}^2$ *tel que* $\mathrm{fst}(T) = \mathcal{S}$. *Les transitions de* T *sont* généralisées*, c.-à-d.* $\exists P$ *un* $\mathcal{PL}\mathrm{MV}$ *tel que* $\mathcal{T}_{gen}(P) = T$*, si et seulement si :* $\forall (s, s_1), (s, s_2) \in T, \forall s_3 \in \mathcal{S}, s_3 \subseteq s \cup s_1 \cup s_2 \implies (s, s_3) \in T$.

ESQUISSE DE PREUVE. Similaire au cas synchrone. La preuve complète est donnée dans Ribeiro *et al*. (2018). □

Enfin, la définition 1.17 formalise la notion de dynamique déterministe, qui est un ensemble de transitions sans « ramifications » dynamiques. Nous donnons une caractérisation particulière des dynamiques déterministes dans le cas synchrone dans la proposition 1.9.

DÉFINITION 1.17 (Transitions déterministes).– *Un ensemble de transitions* $T \subseteq \mathcal{S}^2$ *est* déterministe *si* $\forall (s, s') \in T, \nexists (s, s'') \in T, s'' \neq s'$. *Un* $\mathcal{PL}\mathrm{MV}$ P *est* déterministe vis-à-vis d'une sémantique *si l'ensemble de toutes les transitions* T_P *obtenues par l'application de cette sémantique sur* P *est déterministe.*

PROPOSITION 1.9 (Programme synchrone déterministe).– *Un* $\mathcal{PL}\mathrm{MV}$ P *produit des transitions déterministes avec la sémantique synchrone s'il ne contient pas de règle concurrente, c.-à-d.* $\forall R, R' \in P, \big(\mathrm{var}(h(R)) = \mathrm{var}(h(R')) \wedge R \sqcap R'\big) \implies h(R) = h(R')$.

Jusqu'à présent, les algorithmes de **LFIT** ne permettaient que l'apprentissage de programmes synchrones déterministes. En exploitant le formalisme introduit dans les sections précédentes, la méthodologie peut maintenant être étendue à l'apprentissage de systèmes depuis des transitions produites par les trois sémantiques définies ci-dessus, ce qui inclut aussi bien des systèmes déterministes que non-déterministes.

Enfin, avec le théorème 1.3, nous montrons que les définitions et méthodes développées dans les sections précédentes sont indépendantes de la sémantique choisie.

THÉORÈME 1.3 (Exactitude indépendamment de la sémantique).– *Soit P un $\mathcal{PL}$MV tel que P est complet.*

– $\mathcal{T}_{syn}(P) = \mathcal{T}_{syn}(P_{\mathcal{O}}(\mathcal{T}_{syn}(P)))$,

– $\mathcal{T}_{asyn}(P) = \mathcal{T}_{asyn}(P_{\mathcal{O}}(\mathcal{T}_{asyn}(P)))$,

– $\mathcal{T}_{gen}(P) = \mathcal{T}_{gen}(P_{\mathcal{O}}(\mathcal{T}_{gen}(P)))$.

ESQUISSE DE PREUVE. En utilisant les propriétés d'un programme optimal (définition 1.12) et par contradiction. La preuve complète est donnée dans Ribeiro *et al*. (2018). □

1.2.4. *GULA*

Dans cette section, nous présentons l'algorithme multi-usage de **LFIT** : **GULA** (General Usage LFIT Algorithm), une extension de l'algorithme **LF1T** qui capture aussi bien les dynamiques des sémantiques synchrone, asynchrone et généralisée.

GULA apprend un programme logique depuis les observations de ses transitions d'états. Étant donné un ensemble de transitions T, **GULA** construit itérativement un modèle du système correspondant en appliquant la méthode formalisée dans les sections précédentes. L'algorithme est basé sur le constat suivant : on peut obtenir toutes les règles optimales concluant sur un atome simplement en identifiant les états depuis lesquels il est impossible d'obtenir cet atome (illustré dans la figure 1.3), et en révisant itérativement $P_{\mathcal{O}}(\emptyset)$ par ces états.

L'algorithme complet est détaillé dans la suite. Ses étapes principales sont les suivantes :

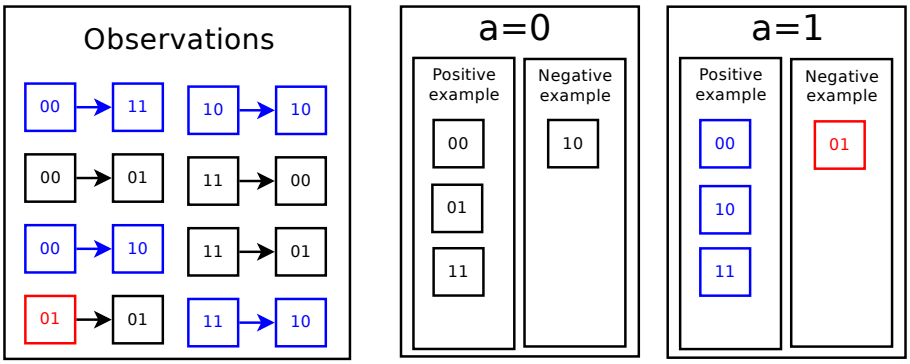

Figure 1.3 – Extraction des états pour l'apprentissage des règles de la variable a. Les transitions en bleu sont des exemples positifs pour l'apparition de a^1 dans l'état suivant, tandis que les rouges des exemples négatifs.

GULA — Version résumée
ENTRÉE : un ensemble d'atomes $\mathcal{A}$, un ensemble de transitions $T \subseteq \mathcal{S}^2$ et $A' \subseteq A$. Pour chaque atome $\mathrm{v}^{val} \in \mathcal{A}'$

– Extraire tous les états depuis lesquels aucune transition vers v^{val} n'existe : $Neg_{\mathrm{v}^{val}} := \{s \mid \nexists (s, s') \in T, \mathrm{v}^{val} \in s'\}$

– Initialiser $P_{\mathrm{v}^{val}} := \{\mathrm{v}^{val} \leftarrow \emptyset\}$

– Pour chaque état $s \in Neg_{\mathrm{v}^{val}}$

- Extraire chaque règle R de $P_{\mathrm{v}^{val}}$ qui couvre s : $M_{\mathrm{v}^{val}} := \{R \in P \mid b(R) \subseteq s\}$; $P_{\mathrm{v}^{val}} := P_{\mathrm{v}^{val}} \setminus M_{\mathrm{v}^{val}}$.
- Pour chaque règle $R \in M_{\mathrm{v}^{val}}$
  - Calculer sa spécialisation minimale $P' = L_{\mathrm{spe}}(R, s)$.
  - Retirer toutes les règles de P' dominées par une règle de $P_{\mathrm{v}^{val}}$.
  - Retirer toutes les règles de $P_{\mathrm{v}^{val}}$ dominées par une règle de P'.
  - Ajouter toutes les règles restantes de P' dans $P_{\mathrm{v}^{val}}$.

– $P := P \cup P_{\mathrm{v}^{val}}$

SORTIE : $P_{\mathcal{O}}(T) := P$.

Les algorithmes 1 et 2 donnent le pseudo-code détaillé de l'algorithme. Étant donnés un ensemble d'atomes $\mathcal{A}$, un ensemble de transitions T et un sous-ensemble d'atomes $\mathcal{A}' \subseteq \mathcal{A}$, l'algorithme 1 apprend les conditions dans lesquelles chaque valeur val de chaque variable v dans $\mathcal{A}'$ peut apparaître dans le prochain état. Lorsque $\mathcal{A}' = \mathcal{A}$, l'algorithme retourne le programme optimal réalisant T, noté $P_{\mathcal{O}}(T)$. En choisissant un $\mathcal{A}' \subsetneq \mathcal{A}$, ce paramètre permet de réduire l'apprentissage aux valeurs de variables qui nous intéressent. Ici, l'apprentissage est effectué itérativement pour chaque valeur et chaque variable afin de présenter un pseudo-code simple. Mais le

processus peut facilement être parallélisé en lançant sur $|\mathcal{A}|$ unités de calcul différentes un appel à **GULA**($\mathcal{A}$,T,$\{\mathrm{v}^{val}\}$) pour chaque $\mathrm{v}^{val} \in \mathcal{A}$, réduisant le temps de calcul de **GULA**($\mathcal{A}$,T,$\mathcal{A}$) à celui de l'appel **GULA**($\mathcal{A}$,T,$\{\mathrm{v}^{val}\}$) le plus long.

L'algorithme commence par le pré-traitement des transitions d'entrée. Les lignes 2–11 de l'algorithme 1 correspondent à l'extraction de $Neg_{\mathrm{v}^{val}}$, l'ensemble de tous les exemples négatifs de l'apparition de v^{val} dans l'état suivant, c'est-à-dire l'ensemble de tous les états tels que v ne prend jamais la valeur val dans l'état suivant d'une transition de T. Ces exemples négatifs sont alors utilisés pendant la phase d'apprentissage qui suit (lignes 12–30) afin d'inférer itérativement l'ensemble de règles $P_{\mathcal{O}}(T)$. La phase d'apprentissage commence par l'initialisation d'un ensemble de règles $P_{\mathrm{v}^{val}}$ par $\{\mathrm{v}^{val} \leftarrow \emptyset\}$ correspondant à une règle du programme optimal pour l'ensemble vide (voir proposition 1.4). $P_{\mathrm{v}^{val}}$ est révisé itérativement par confrontation avec chaque exemple négatif neg de $Neg_{\mathrm{v}^{val}}$. Toute règle R_m de $P_{\mathrm{v}^{val}}$ qui couvre neg est incohérente et doit donc être révisée. Afin que $P_{\mathrm{v}^{val}}$ reste optimal, la révision de chaque R_m ne doit pas couvrir neg mais doit couvrir tout autre état couvert par R_m. Pour garantir cela, on utilise la spécialisation minimale (voir définition 1.10) pour réviser chaque règle R_m. L'algorithme 2 donne le pseudo code de cette opération. Pour chaque variable de $\mathcal{V}$ telle que $b(R_m)$ n'a pas de condition correspondante, une condition sur une autre valeur que celle observée dans l'état neg peut être ajoutée (lignes 1–6). Aucune de ces révisions ne couvre neg et tous les états couverts par R_m le sont par au moins l'une d'elles. Les règles révisées sont alors ajoutées à $P_{\mathrm{v}^{val}}$ après suppression des règles dominées. Une fois que $P_{\mathrm{v}^{val}}$ a été révisé par tous les exemples négatifs de $Neg_{\mathrm{v}^{val}}$, $P_{\mathrm{v}^{val}} = \{R \in P_{\mathcal{O}}(T) \mid h(R) = \mathrm{v}^{val}\}$ est ajouté à P. Une fois chaque valeur de chaque variable traitée, l'algorithme retourne $P = \{R \in P_{\mathcal{O}}(T) \mid h(R) \in \mathcal{A}'\}$ et si $\mathcal{A}' = \mathcal{A}$ alors $P = P_{\mathcal{O}}(T)$: il s'agit du programme optimal qui réalise les transitions de T.

Le théorème 1.4 donne les bonnes propriétés de l'algorithme, le théorème 1.5 montre que **GULA** peut apprendre aussi bien depuis des ensembles de transitions provenant d'une sémantique synchrone, qu'asynchrone ou généralisée. Enfin, le théorème 1.6 caractérise sa complexité en temps et en mémoire.

THÉORÈME 1.4 (Terminaison, exactitude, complétude et optimalité de GULA).– *Soit* $T \subseteq \mathcal{S}^2$. *L'appel* **GULA**$(\mathcal{A}, T)$ *termine et* **GULA**$(\mathcal{A}, T) = P_{\mathcal{O}}(T)$.

THÉORÈME 1.5 (Indépendance de la sémantique).– *Soit* P *un* $\mathcal{PL}$MV *tel que* P *est complet.*

– **GULA**$(\mathcal{A}, \mathcal{T}_{syn}(P)) = P_{\mathcal{O}}(\mathcal{T}_{syn}(P))$
– **GULA**$(\mathcal{A}, \mathcal{T}_{asyn}(P)) = P_{\mathcal{O}}(\mathcal{T}_{asyn}(P))$
– **GULA**$(\mathcal{A}, \mathcal{T}_{gen}(P)) = P_{\mathcal{O}}(\mathcal{T}_{gen}(P))$

Algorithme 1 GULA($\mathcal{A}$, T, $\mathcal{A}' = A$)

ENTRÉE : Un ensemble d'atomes $\mathcal{A}$, un ensemble de transitions $T \subseteq \mathcal{S}^2$ et un ensemble $\mathcal{A}' \subseteq \mathcal{A}$ optionnel, par défaut égal à $\mathcal{A}$.
SORTIE : $P = P_{\mathcal{O}}(T)$

for each $\mathrm{v}^{val} \in \mathcal{A}'$ **do**
  // 1) Extraction des exemples négatifs
  $Neg_{\mathrm{v}val} := \emptyset$
  for each $(s_1, s_1') \in T$ **do**
    $negative\_example := true$
    for each $(s_2, s_2') \in T$ **do**
      if $s_1 == s_2$ and $\mathrm{v}^{val} \in s_2'$ **then**
        $negative\_example := false$
        Break
    if $negative\_example == true$ **then**
      $Neg_{\mathrm{v}val} := Neg_{\mathrm{v}val} \cup \{s_1\}$
  // 2) Révision des règles de v^{val} pour éviter la couverture des exemples négatifs
  $P_{\mathrm{v}val} := \{\mathrm{v}^{val} \leftarrow \emptyset\}$
  for each $neg \in Neg_{\mathrm{v}val}$ **do**
    $M := \emptyset$
    for each $R \in P_{\mathrm{v}val}$ **do** // Extraction des règles incohérentes
      if $b(R) \subseteq neg$ **then**
        $M := M \cup \{R\}$; $P := P \setminus \{R\}$
    for each $R_m \in M$ **do** // Révision de chaque règle incohérente
      $LS := least\_specialization(R_m, neg, \mathcal{A})$
      for each $R_{ls} \in LS$ **do**
        for each $R_p \in P_{\mathrm{v}val}$ **do** // Suppression des révisions dominées
          if $b(R_p) \subseteq b(R_{ls})$ **then**
            $dominated := true$
            break
        if $dominated == false$ **then** // Suppression des anciennes règles à présent dominées
          for each $R_p \in P$ **do**
            if $b(R_{ls}) \subseteq b(R_p)$ **then**
              $P_{\mathrm{v}val} := P_{\mathrm{v}val} \setminus \{R_p\}$
          $P_{\mathrm{v}val} := P_{\mathrm{v}val} \cup \{R_{ls}\}$ // Ajout des révisions
  $P := P \cup P_{\mathrm{v}val}$
return P

Algorithme 2 least_specialization(R, s, $\mathcal{A}$) : spécialise R pour éviter la couverture de s, en vue de corriger une potentielle incohérence

ENTRÉE : une règle R, un état s et un ensemble d'atomes $\mathcal{A}$
SORTIE : un ensemble de règles LS qui est la spécialisation minimale de R par s basée sur $\mathcal{A}$.

$LS := \emptyset$
// Révision de la règle par spécialisation minimale
for each $\mathrm{v}^{val} \in s$ **do**
  if $\nexists \mathrm{v}^{val'} \in b(R)$ **then** // Ajout d'une condition pour chaque valeur qui n'apparaît pas dans s
    for each $\mathrm{v}^{val''} \in \mathcal{A}, val'' \neq val$ **do**
      $R' := h(R) \leftarrow (b(R) \cup \{\mathrm{v}^{val''}\})$
      $LS := LS \cup \{R'\}$
return LS

ESQUISSE DE PREUVE. D'après les théorèmes 1.3 et 1.4. La preuve complète est donnée dans Ribeiro *et al*. (2018). □

THÉORÈME 1.6 (Complexité de GULA).– *Soient $T \subseteq \mathcal{S}^2$ un ensemble de transitions, $n := |\mathcal{V}|$ le nombre de variables dans le système et $d := \max(\mathsf{dom}(\mathcal{V}))$ la plus haute valeur prise par ses variables. La complexité de calcul de* **GULA** *dans le pire cas lorsque qu'il apprend depuis T est de l'ordre de $\mathcal{O}(|T|^2 + 2n^3d^{2n+1} + 2n^2d^n)$ et son utilisation mémoire dans le pire cas est de l'ordre de $\mathcal{O}(d^{2n} + 2d^n + nd^{n+2})$.*

1.2.5. *PRIDE*

Dans cette section nous présentons un algorithme appelé **PRIDE** (Polynomial Relational Inference of Discrete Events) pouvant servir d'alternative à **GULA**. Celui-ci bénéficie d'une complexité polynomiale au lieu d'exponentielle, au prix de manquer des explications alternatives à certaines observations. En pratique, ces explications manquantes sont toujours des cas redondants avec les règles effectivement retournées par **PRIDE**. Formellement, pour un ensemble de transitions T, **PRIDE** retourne un programme P tel que $P \subseteq P_{\mathcal{O}}(T)$ et $\overset{P}{\hookrightarrow} T$. Enfin, à la différence de **GULA**, la complétude du programme retourné par **PRIDE** n'est garantie que si l'ensemble de transitions observé couvre tous les états possibles : $\mathrm{fst}(T) = \mathcal{S}$.

PRIDE est basé sur l'exploitation de deux propriétés formalisées dans la suite : le théorème 1.7 caractérise les règles les plus spécifiques réalisant une transition, tandis que le théorème 1.8 permet de caractériser une règle appartenant au programme optimal comme ne pouvant pas être simplifiée sans devenir incohérente. Ainsi, en partant des règles les plus spécifiques de chaque transition et par simplifications successives conservant la cohérence avec T, on finit donc par obtenir des règles de $P_{\mathcal{O}}(T)$; ce processus possède une complexité polynomiale.

THÉORÈME 1.7 (Règle cohérente la plus spécifique).– *Soient $T \subseteq \mathcal{S}^2$, $(s, s') \in T$ et $\mathrm{v}^{val} \in s'$. La règle $R = \mathrm{v}^{val} \leftarrow s$ est cohérente avec T et réalise (s, s').*

Démonstration. D'après la définition 1.3, étant donné que $b(R) = s, b(R) \subseteq s$ et donc $R \sqcap s$. Comme $h(R) \in s'$, d'après la définition 1.5, R réalise (s, s'). Et d'après la définition 1.6, R est cohérente avec T . □

THÉORÈME 1.8 (Transitivité de non-dominance).– *Soit R une règle $\mathcal{L}$MV cohérente avec un ensemble de transitions $T \subseteq \mathcal{S}^2$. Si $\forall R' \in \{h(R) \leftarrow b(R) \setminus \{c\} \mid c \in b(R)\}$, R' est incohérente avec T, alors il n'existe pas de règle $R'' \neq R$ cohérente avec T telle que $R \leq R''$. Dans ce cas, $R \in P_{\mathcal{O}}(T)$.*

Démonstration. 1) D'après la définition 1.1, $\{R''$ règle $\mathcal{L}\mathrm{MV} \mid R'' \neq R \wedge R \leq R''\} = \{R''$ règle $\mathcal{L}\mathrm{MV} \mid R' \leq R'' \wedge R' \in \{h(R) \leftarrow b(R) \setminus \{c\} \mid c \in b(R)\}\}$.

2) L'incohérence est transitive : si R' est incohérente avec T, alors toute règle R'' telle que $R' \leq R''$ est incohérente avec T. Si R' est incohérente avec un ensemble

de transitions $T \subseteq \mathcal{S}^2$, d'après la définition 1.6, $\exists (s, s') \in T$, $R' \sqcap s$ et $\nexists (s, s'') \in T, h(R') \in s''$. D'après la définition 1.3, $R' \sqcap s \implies b(R') \subseteq s$ et pour une règle R'', $b(R'') \subseteq b(R') \implies b(R'') \subseteq s \implies R'' \sqcap s$ et donc $h(R'') = h(R')$ implique que R'' est incohérente avec T.

En utilisant 1) et 2) on peut déduire la non-dominance de R depuis l'incohérence de toutes ses généralisations directes R'. □

La complexité de ce nouvel algorithme est polynomiale et il est possible de le paralléliser sur $|T \times \mathcal{A}|$ unités de calcul contre $|\mathcal{A}|$ pour **GULA**. L'idée est de dédier chaque unité de calcul à l'apprentissage d'une règle optimale pour chacune des variables, cette règle réalisant au moins une transition de T. En faisant ceci pour chaque transition de T, il suffirait alors de faire l'union des sorties de chaque unité de calcul pour obtenir un $P \subseteq P_{\mathcal{O}}(T)$ qui réalise T. Dans un souci de simplicité du pseudo code et de la discussion, la version de l'algorithme que nous présentons ici n'est pas parallélisée.

Comme **GULA**, **PRIDE** prend également en entrée un sous-ensemble optionnel d'atomes $\mathcal{A}' \subseteq \mathcal{A}$ qui est égal à $\mathcal{A}$ par défaut. Pour chaque valeur de variable, c'est-à-dire chaque atome de $\mathcal{A}'$, on extrait l'ensemble des exemples positifs et négatifs (ligne 1). Un exemple positif de l'apparition de v^{val} est un état d'où *au moins* une transition donne v^{val} dans l'état suivant. Un exemple négatif de l'apparition de v^{val} est un état d'où *aucune* transition ne donne v^{val} dans l'état suivant. Apprendre la dynamique de v^{val} devient alors un problème de classification classique consistant à chercher les hypothèses les plus simples, qui dans notre cas sont des règles $\mathcal{L}\mathrm{MV}$, donnant les conditions d'apparition de v^{val}.

Tant qu'il reste au moins un exemple positif pos non couvert par une règle, on cherche à inférer une règle optimale le réalisant (ligne 5). On part pour cela de la règle non-dominée $R = \mathrm{v}^{val} \leftarrow \emptyset$ qu'on révise itérativement pour chaque exemple négatif couvert. Ici, la révision est limitée à l'ajout des conditions qui apparaissent dans pos mais pas dans l'exemple négatif couvert (ligne 12). Cela donne une règle qui fait partie de la spécialisation minimale de R et qui couvre pos. Un tel ajout est toujours possible car pos n'est jamais un exemple négatif : dans le pire des cas, R devient la règle la plus spécifique qui couvre pos (voir théorème 1.7).

Une fois la cohérence de la règle obtenue, elle est itérativement simplifiée en retirant les conditions non nécessaires à ladite cohérence (ligne 16). En effet, l'ajout de conditions permettant d'assurer la cohérence avec un nouvel exemple négatif peut amener à assurer aussi la cohérence avec d'autres exemples précédemment traités ; la règle peut alors être simplifiée. Il suffit de tester la cohérence de la règle obtenue après retrait d'une condition atomique : si cette nouvelle règle est encore cohérente, elle sert de nouvelle référence sur laquelle on peut répéter le processus. De cette façon, on finit par obtenir une règle irréductible qui, d'après le théorème 1.8, est optimale.

On peut alors retirer tous les exemples positifs couverts par cette règle et reprendre la même procédure jusqu'à la réalisation de tous les exemples positifs (ligne 26). Au final, un ensemble de règles optimales formant un programme $P \subseteq P_{\mathcal{O}}(T)$ est retourné, réalisant T et constituant donc une explication suffisante pour T dans le cas pratique. En théorie, certaines règles redondantes peuvent manquer car la complétude du programme n'est pas assurée pour un ensemble de transitions dont les états de départ ne couvrent pas tous les états possibles.

Algorithme 3 PRIDE($\mathcal{A}$, T, $\mathcal{A}' = \mathcal{A}$)

ENTRÉE : Un ensemble d'atomes $\mathcal{A}$, un ensemble de transitions $T \subseteq \mathcal{S}^2$ et un ensemble $\mathcal{A}' \subseteq \mathcal{A}$ optionnel, par défaut égal à $\mathcal{A}$.
SORTIE : $P \subseteq P_{\mathcal{O}}(T)$ tel que P réalise T

1: **for** each $\mathrm{v}^{val} \in \mathcal{A}'$ **do**
2: // 1) Extraction des exemples positifs et négatifs
3: $Pos_{\mathrm{v}val} := \{s \mid \exists (s, s') \in T, \mathrm{v}^{val} \in s'\}$
4: $Neg_{\mathrm{v}val} := \{s \mid \nexists (s, s') \in T, \mathrm{v}^{val} \in s'\}$
5: // 2) Génération des règles de v^{val} qui sont dans $P_{\mathcal{O}}(T)$ et qui couvrent $Pos_{\mathrm{v}val}$
6: **while** $Pos_{\mathrm{v}val} \neq \emptyset$ **do**
7: $R = \mathrm{v}^{val} \leftarrow \emptyset$
8: pick $pos \in Pos_{\mathrm{v}val}$
9: // Cohérence vis-à-vis des exemples négatifs
10: **for** each $neg \in Neg_{\mathrm{v}val}$ **do**
11: **if** $R \sqcap neg$ **then**
12: pick $c \in (pos \setminus neg)$
13: $R = h(R) \leftarrow b(R) \cup \{c\}$
14: // Minimaliser en gardant seulement les conditions nécessaires
15: generalizable = true
16: **for** each $c \in b(R)$ **do** // Test de chaque condition
17: $R' = h(R) \leftarrow b(R) \setminus \{c\}$
18: $conflict = false$
19: **for** each $neg \in Neg_{\mathrm{v}val}$ **do**
20: **if** $R \sqcap neg$ **then** // Condition nécessaire
21: $conflict = true$
22: **BREAK**
23: **if** $conflict == false$ **then** // Simplification valide
24: $R = R'$
25: // On retire les exemples positifs maintenant couverts
26: $Pos_{\mathrm{v}val} = Pos_{\mathrm{v}val} \setminus \{s | s \in Pos_{\mathrm{v}val}, R \sqcap s\}$
27: $P = P \cup \{R\}$
28: **return** P

	Mammalian (10)	Fission (10)	Budding (12)	Arabidopsis (15)
GULA	$1,84$s	$1,55$s	$34,48$s	2066s
PRIDE	$0,038$s	$0,04$s	$0,457$s	$2,99$s

Tableau 1.1 – Temps d'exécution de **GULA** et **PRIDE** en secondes pour des cas d'études de réseaux booléens comprenant jusqu'à 15 variables pour la sémantique synchrone.

Le tableau 1.1 donne les temps de calcul de **GULA** et **PRIDE** pour des réseaux booléens de Dubrova and Teslenko (2011) qui travaillent sur les modèles suivants : le contrôle de la morphogenèse de la fleur de Arabidopsis thaliana (Chaos *et al*. 2006),

la régulation du cycle cellulaire de la levure bourgeonnante (*budding yeast*) (Li *et al*. 2004), la régulation du cycle cellulaire de la levure à fission (*fission yeast*) (Davidich and Bornholdt 2008) et la régulation du cycle cellulaire chez les mammifères (Fauré *et al*. 2006). Ici nous reproduisons les expériences de (Inoue *et al*. 2014) : toutes les transitions de chaque benchmark sont générées et l'algorithme doit apprendre les règles originales les ayant produites. Pour chaque benchmark, le nombre de transitions générées est de deux à la puissance du nombre de variables.

Les expériences sont effectuées à l'aide d'une implémentation optimisée en C++ en ce qui concerne **GULA**, la même que celle utilisée dans (Inoue *et al*. 2014), et une implémentation Python naïve de **PRIDE**, disponible en open source sur le dépôt Github suivant : `https://github.com/Tony-sama/pylfit`. Le but de ce dépôt est d'offrir une implémentation facile à comprendre de chaque algorithme du framework **LFIT**. Toutes les expériences ont été effectuées sur un CPU Intel Core I7 (6700, 3.4 GHz) avec 32 Gb de RAM.

Ces expériences montrent clairement que **PRIDE** est plus efficace que **GULA**, et que sur ces benchmarks, leurs sorties sont identiques. **PRIDE** est également plus performant que **LF1T** (Inoue *et al*. 2014), notre précédent algorithme dédié à l'apprentissage de systèmes synchrones.

En pratique, on observe donc que **GULA** est limité par rapport au nombre de variables du modèle, à cause de l'explosion combinatoire : la méthode deviendra trop coûteuse en temps sur un ordinateur de bureau pour des modèles au-delà de 15 variables. En revanche, **PRIDE** est beaucoup moins calculatoire (de complexité polynomiale) ce qui permet de repousser cette limite, au prix d'une sortie moins complète mais couvrant néanmoins toutes les données fournies. Enfin, il serait possible de paralléliser ces deux méthodes afin d'augmenter la taille des modèles traités car les étapes d'apprentissage sont majoritairement indépendantes. Par ailleurs, il est aussi possible de contraindre la taille des règles pour accélérer l'apprentissage.

1.3. Promesses de l'apprentissage automatique pour la biologie

Avec l'essor concomitant des sciences et technologies de l'information et de la communication et de nouvelles techniques de mesures, il est désormais possible d'accéder à un large volume de données biologiques (Marx 2013).

Différentes approches ont d'ores et déjà vu le jour pour traiter la rétro-ingénierie des réseaux de régulation à partir de profils d'expression génétiques (Silvescu and Honavar 2001, Li *et al*. 2006, Guziolowski *et al*. 2013). Mais la plupart des méthodologies existantes ne traitent que des modèles statiques. Cela veut ainsi dire, par exemple, que toute l'information contenue dans des données de séries

temporelles n'est pas encore exploitée. Dans ce contexte, nous souhaitons notamment traiter toute la richesse et la qualité des données biologiques et donc nous attaquer notamment à la richesse des données de séries temporelles. Les informations temporelles quantitatives qu'elles contiennent peuvent en effet être utiles pour comprendre la dynamique de nombreux systèmes.

Le traitement des données de séries temporelles pose plusieurs défis auxquels nous avons proposé, dans ce chapitre, de nous attaquer.

1.3.1. *Apprentissage de réseaux de régulation biologiques modélisant des comportements complexes*

D'abord, l'inférence automatique de modèles dynamiques à partir des données biologiques est un enjeu majeur pour étudier finement des systèmes où le temps joue un rôle critique, telle l'horloge circadienne des mammifères (Comet *et al*. 2012) ou le mécanisme de réparation de l'ADN par p53 (Abou-Jaoudé *et al*. 2009).

Nous avons montré dans ce chapitre comment tirer profit de l'information chronologique des données de séries temporelles. Ainsi, nous avons introduit une approche logique pour apprendre des réseaux qualitatifs à partir de données de séries temporelles. En représentant ces données sous forme de séquences d'états-transitions, cette méthode permet de construire une modélisation sous forme de règles logiques. Le programme final résulte d'un apprentissage de la dynamique indépendamment de la sémantique en présence. Toutefois, cette approche est discrète, au sens chronologique, et ne tient pas encore compte des informations temporelles quantitatives entre les changements d'états du modèle. Dès lors, partant de la connaissance de la structure statique des gènes et de leurs interactions, la deuxième moitié du chapitre nous a permis de mettre en exergue une méthode permettant d'apprendre la chronométrie intrinsèque aux systèmes étudiés. Pour ce faire, la méthodologie repose sur une modélisation logique, implantée en ASP, pour déterminer les paramètres cinétiques du système.

1.3.2. *Révision de modèles*

Des expériences additionnelles peuvent apporter des précisions sur un modèle (rendant ainsi nécessaire un raffinement de celui-ci) ou permettre de discriminer les modèles acceptables parmi un ensemble de modèles considérés, pourvu que le bruit associé ne remette pas en cause la précision et la pertinence de l'information correspondante.

La production intensive de nouvelles données biologiques, ainsi que le nombre croissant de données de séries temporelles, pose un autre défi : comment réviser

efficacement un modèle existant (construit par exemple à partir de données précédemment obtenus et/ou de l'expertise de collaborateurs biologistes) avec de nouvelles données ?

Cela rend impératif la conception de méthodes de révision efficaces, capables de mettre à jour la connaissance préalable sur le système avec des informations additionnelles sur la dynamique. Autrement dit, il y a un besoin fort pour des méthodes automatiques de mise à jour d'un modèle de manière à ce que celui-ci soit cohérent avec les observations et un ensemble de critères donnés (par exemple, minimiser le nombre de modifications sur le modèle).

La révision et la complétion (le premier s'entend par l'ajout ou la suppression d'éléments dans le modèle, tandis que le second exclut toute suppression) ont fait l'objet de nombreux travaux récents. Dans (Akutsu *et al*. 2009), les auteurs se sont attaqués à la complétion de réseaux booléens stationnaires. Ils ont raffiné leur méthode au fil des années. Leurs derniers travaux (Nakajima and Akutsu 2013) se concentrent sur la complétion dans les *Time Varying Genetic Networks*. Dans ces réseaux, la topologie reste constante au fil du temps, mais la nature des interactions entre les composants (activation, inhibition, ou aucune interaction) peut changer en un nombre (fini) de points. Ces approches de complétion (qui, spécifiquement chez ces auteurs, englobent à la fois la suppression et l'addition d'interactions) ont été appliquées avec succès à des cas d'étude biologiques, par exemple les données du challenge DREAM4 (Nakajima and Akutsu 2014*b*). De plus, l'implantation logicielle a été améliorée grâce à des heuristiques (Nakajima and Akutsu 2014*a*). Ces méthodes souffrent toutefois de certaines limites : elles bornent généralement le nombre d'interactions entrantes sur un composant biologique (afin de limiter l'explosion combinatoire) et se restreignent aux réseaux acycliques.

Des travaux dans le domaine de la logique ont été également menés pour traiter le problème de la révision des réseaux. C'est ainsi que des approches logiques ont été mises au point dans le cas de réseaux de causalité (Inoue *et al*. 2013) et de réseaux moléculaires représentés avec le langage SBGN-AF (Yamamoto *et al*. 2014).

Même si elle constitue un champ de recherche à part entière, la révision est évidemment fortement liée à l'inférence de modèles. Partant d'un modèle vide, il est en effet possible d'utiliser la révision pour construire incrémentalement un modèle pertinent. En particulier, Answer Set Programming — qui a, on l'a vu, fait ses preuves dans différents contextes de représentation des connaissances et de raisonnement automatique (Niemelä 1999, Baral 2003, 2008) — s'est avéré utile pour la reconstruction de réseaux (Durzinsky *et al*. 2011) et l'inférence de réseaux métaboliques (Videla *et al*. 2014).

À notre connaissance, il n'y a pas eu, à l'heure actuelle, de travaux portant sur la révision de modèles temporisés sans faire d'hypothèse restreignant la structure du réseau.

Des champs d'investigation également prometteurs portent sur le traitement d'une part de données de cellules uniques (*single cell*), d'autre part de données qui seraient incohérentes avec des connaissances précédentes. C'est-à-dire qu'étant donné un modèle existant et une (ou plusieurs) nouvelle(s) observation(s), nous souhaiterions construire automatiquement l'ensemble minimal d'actions manquantes qui peuvent capturer l'observation tout en préservant la dynamique du modèle.

Enfin, quand plusieurs modèles sont compatibles avec des données de séries temporelles, un défi consiste à proposer des algorithmes semi-automatiques permettant de décider, d'un point de vue du modèle, quel type d'expériences pourraient être accomplies pour discriminer efficacement les modèles. Cette question est cruciale car, d'un point de vue pratique, les expériences biologiques sont coûteuses et/ou ne permettent pas d'observer toutes les variables du système. La notion d'observabilité, importante au sein de la communauté de l'automatique, est là aussi primordiale.

1.4. Bibliographie

Abou-Jaoudé, W., Ouattara, D. A., Kaufman, M. (2009), From structure to dynamics : frequency tuning in the p53–mdm2 network : I. logical approach, *Journal of theoretical biology*, 258(4), 561–577.

Akutsu, T., Tamura, T., Horimoto, K. (2009), Completing networks using observed data, *in* Algorithmic Learning Theory, Springer, pp. 126–140.

Baral, C. (2003), *Knowledge Representation, Reasoning and Declarative Problem Solving*, Cambridge University Press.

Baral, C. (2008), Using answer set programming for knowledge representation and reasoning : Future directions, *in* ICLP, pp. 69–70.

Chaos, A., Aldana, M., Espinosa-Soto, C., de León, B. G. P., Arroyo, A. G., Alvarez-Buylla, E. R. (2006), From genes to flower patterns and evolution : dynamic models of gene regulatory networks, *Journal of Plant Growth Regulation*, 25(4), 278–289.

Comet, J.-P., Bernot, G., Das, A., Diener, F., Massot, C., Cessieux, A. (2012), Simplified models for the mammalian circadian clock, *Procedia Computer Science*, 11, 127–138.

Davidich, M. I., Bornholdt, S. (2008), Boolean network model predicts cell cycle sequence of fission yeast, *PloS one*, 3(2), e1672.

Dubrova, E., Teslenko, M. (2011), A SAT-Based Algorithm for Finding Attractors in Synchronous Boolean Networks, *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 8(5), 1393–1399.

Durzinsky, M., Marwan, W., Ostrowski, M., Schaub, T., Wagler, A. (2011), Automatic network reconstruction using asp, *Theory and Practice of Logic Programming*, 11(4-5), 749–766.

Fauré, A., Naldi, A., Chaouiya, C., Thieffry, D. (2006), Dynamical analysis of a generic boolean model for the control of the mammalian cell cycle, *Bioinformatics*, 22(14), e124–e131.

Guziolowski, C., Videla, S., Eduati, F., Thiele, S., Cokelaer, T., Siegel, A., Saez-Rodriguez, J. (2013), Exhaustively characterizing feasible logic models of a signaling network using answer set programming, *Bioinformatics*, p. btt393.

Inoue, K., Doncescu, A., Nabeshima, H. (2013), Completing causal networks by meta-level abduction, *Machine learning*, 91(2), 239–277.

Inoue, K., Ribeiro, T., Sakama, C. (2014), Learning from interpretation transition, *Machine Learning*, 94(1), 51–79.

Koh, C., Wu, F.-X., Selvaraj, G., Kusalik, A. J. (2009), Using a state-space model and location analysis to infer time-delayed regulatory networks, *EURASIP Journal on Bioinformatics and Systems Biology*, 2009, 14.

Koksal, A. S., Pu, Y., Srivastava, S., Bodik, R., Fisher, J., Piterman, N. (2013), Synthesis of biological models from mutation experiments, *ACM SIGPLAN Notices*, 48(1), 469–482.

Li, F., Long, T., Lu, Y., Ouyang, Q., Tang, C. (2004), The yeast cell-cycle network is robustly designed, *Proceedings of the National Academy of Sciences of the United States of America*, 101(14), 4781–4786.

Li, X., Rao, S., Jiang, W., Li, C., Xiao, Y., Guo, Z., Zhang, Q., Wang, L., Du, L., Li, J. *et al*. (2006), Discovery of time-delayed gene regulatory networks based on temporal gene expression profiling, *BMC bioinformatics*, 7(1), 26.

Liu, T.-F., Sung, W.-K., Mittal, A. (2004), Learning multi-time delay gene network using bayesian network framework, *in* Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on, IEEE, pp. 640–645.

Lopes, M., Bontempi, G. (2013), Experimental assessment of static and dynamic algorithms for gene regulation inference from time series expression data., *Frontiers in genetics*, 4(December), 303.

Marx, V. (2013), Biology : The big challenges of big data, *Nature*, 498(7453), 255–260.

Nakajima, N., Akutsu, T. (2013), Network completion for time varying genetic networks, *in* Complex, Intelligent, and Software Intensive Systems (CISIS), 2013 Seventh International Conference on, IEEE, pp. 553–558.

Nakajima, N., Akutsu, T. (2014*a*), Exact and heuristic methods for network completion for time-varying genetic networks, *BioMed research international*, 2014.

Nakajima, N., Akutsu, T. (2014*b*), Network completion for static gene expression data, *Advances in bioinformatics*, 2014.

Niemelä, I. (1999), Logic programs with stable model semantics as a constraint programming paradigm, *Ann. Math. Artif. Intell.*, 25(3-4), 241–273.

Ribeiro, T., Folschette, M., Magnin, M., Roux, O., Inoue, K. (2018), Learning dynamics with synchronous, asynchronous and general semantics, *in* F. Riguzzi, E. Bellodi, R. Zese, (eds), Inductive Logic Programming, Springer International Publishing, Cham, pp. 118–140.

Silvescu, A., Honavar, V. (2001), Temporal boolean network models of genetic networks and their inference from gene expression time series, *Complex Systems*, 13(1), 61–78.

Soinov, L. A., Krestyaninova, M. A., Brazma, A. (2003), Towards reconstruction of gene networks from expression data by supervised learning, *Genome biology*, 4(1), 6.

Videla, S., Guziolowski, C., Eduati, F., Thiele, S., Gebser, M., Nicolas, J., Saez-Rodriguez, J., Schaub, T., Siegel, A. (2014), Learning boolean logic models of signaling networks with asp, *Theoretical Computer Science*, .

Yamamoto, Y., Rougny, A., Nabeshima, H., Inoue, K., Moriya, H., Froidevaux, C., Iwanuma, K. (2014), Completing sbgn-af networks by logic-based hypothesis finding, *in* Formal Methods in Macro-Biology, Springer, pp. 165–179.

Zhang, Z.-y. (2008), Time Series Segmentation for Gene Regulatory Process with Time-Window-Extension Technique, *Proceedings of the 2nd International Symposium on Optimization and Systems Biology*, pp. 198–203.