Rapport de stage de L3

Analyse de la dynamique des modèles biologiques par programmation logique

Léo-Paul Delsaux[†]

Encadrant du stage
Maxime Folschette[‡]

Juin-Août 2022

[†]ENS de Lyon
[‡]Équipe Bio-Computing, labortaoire CRIStAL, CNRS de Lille

Table des matières

1 Introduction

La biologie s'intéresse à l'étude des systèmes vivants ou composant le vivant (gènes, cellules, écosystèmes...). La bioinformatique est l'utilisation de méthodes informatiques pour aider la biologie. Un des domaines de la bioinformatique s'apelle la *biologie des systèmes* et consiste à proposer des modèles et des méthodes informatiques et mathématiques pour représenter et étudier de tels systèmes.

Ainsi, des études biologiques permettent de déterminer comment certains gènes interagissent entre eux, ce qui peut être représenté mathématiquement sous la forme d'un graphe. De plus, ces interactions peuvent être vues comme des transitions entre les sommets de notre graphe, ce qui se rapproche alors plus d'un automate. La représentation du modèle étudié lors de ce stage est explicité dans le chapitre Modèle.

L'étude de la dynamique de systèmes biologiques lève plusieurs problèmes tels que l'identification d'attracteurs, les bifurcations ou encore la connexité entre deux états globaux. Dans ce rapport je vais discuter de la recherche d'attracteurs dans des réseaux d'automates asynchrones : il s'agit d'un ensemble d'états duquel on ne peut pas s'échapper et minimal au sens de l'inclusion.

Ce qui sera étudié dans ce rapport ne le sera que partiellement : on n'entrera pas en détail dans le code présenté, mais on se penchera simplement sur les grandes lignes et sur les points techniques qu'il arbore.

2 Answer Set Programming

L'Answer Set Programming (ASP) est un paradigme de programmation logique comparable à Prolog. Ces dernières décennies, ASP s'est trouvé être puissant pour traiter des modèles biologiques, permettant de parcourir un grand nombre de configurations rapidement. ASP peut énumérer facilement les ensembles solutions à un problème qu'on lui encode : c'est un paradigme très efficace pour la combinatoire. Nous allons ici présenter brièvement son fonctionnement, et plus précisément, les outils qui ont été utiles pour mon stage.

2.1 Termes

On commence par définir un terme (ou atome) en ASP :

1. terme simple :
   - un entier relatif (0, 1, 42, -35, ...)
   - une constante qui démarre par un enderscore ou une lettre minuscule, et est suivie par des lettres/chiffres/enderscore (v0us, lis35, un, ex3mpl3, __sisi__, ...)
   - une chaîne de caractères sous la forme "ma_chaine_de_caracteres" ("gf43ha43gG", "42", ...)
   - une variable identique grammaticalement à la constante, mais dont la première lettre doit être majuscule (Nous, Sommes, Des, Variables, ...)
   - un enderscore symbolisant une variable sans nom
   - $\#sup$ et $\#inf$ qui sont des constantes définies par ASP
2. une fonction est de la forme : constante(terme, terme, ..., terme) avec un nombre de termes fini qui peut être nul, et qu'on appelle 'arité' (f(23), je(suis(une(fonction(42, true)), et(pas_moi))), ...)

3. un tuple de la forme : (terme, terme, ..., terme) avec un nombre de termes fini qui ne peut être nul, et qu'on appelle 'arité' ((37, oui), (), (1,h(-4),0,1), ...)

2.2 Règles et modèles d'un programme

Un *programme d'ensemble solutions (= answer set program)* est un nombre fini de règles de la forme :

$$a_0 \text{ :- } a_1, ..., a_m, not\ a_{m+1}, ..., not\ a_n.$$

avec $n \geq m \geq m \geq 0$. a_0 est un atome ou $\perp$ (Bottom/le Faux) et représente ici la **tête** de la règle. Les $a_1, ..., a_n$ sont des atomes et représente quant à eux le **corps** de la règle, et le symbole "*not*" représente la négation par l'échec. Cette règle se lit intuitivement : si les atomes $a_1, .., a_m$ sont tous vrais et qu'aucun des atomes $a_{m+1}, ..., a_n$ n'est vrai, alors a_0 est vrai.

Si $n = m = 0$, cela signifie que a_0 est vrai. Dans ce cas, on parle d'un fait, et on ne doit pas renseigner ":-". D'une autre part, si $a_0 = \perp$, on parle de contrainte : comme $\perp$ ne peut jamais être vrai, si le corps de la règle est vrai, cela invalide la solution actuelle. On ne renseigne pas $\perp$ pour la tête, on laisse une tête vide au niveau du code.

Cette notion de "solution actuelle" se formalise de la façon suivante : on parle d'une **interprétation** I en tant qu'ensemble fini d'atomes propositionnels. Une règle r définie comme ci-dessus est *vraie dans* I si et seulement si :

$$\{a_1, ..., a_m\} \subseteq I \wedge \{a_{m+1}, ..., a_n\} \cap I = \emptyset \Rightarrow a_0 \in I$$

Si toutes les règles d'un programme P sont vraies dans une même interprétation I, alors on dit que I est un **modèle** de P. Lorsque l'on déclare un problème P en ASP, le solveur nous renverra en sortie tout les modèles possibles pour P.

2.3 Variables

Si une variable apparaît dans un atome de la tête, elle doit également être dans la queue. Chaque instance d'une variable va être groundée par clingo afin de trouver tous les modèles à notre problème. Prenons un petit exemple, voici un problème encodé avec 2 règles :

parentOf(jenny, charles).
parentOf(mary, jenny).

Ce programme contient 2 faits : Charles est un parent de Jenny, et Jenny est un parent de Mary. C'est ainsi que l'on souhaite comprendre les termes "parentOf(jenny, charles)" et "parentOf(mary, jenny)". Si on souhaite désormais spécifier que Charles est un grand parent de Mary, on peut le faire en ajoutant le terme : "grandparentOf(Mary, Charles)". Cependant, ASP est capable de faire bien mieux pour cela. En effet, on peut généraliser la notion de grand parent comme étant le parent d'un parent de la façon suivante :

grandparentOf(X, Z) :- parentOf(X, Y), parentOf(Y, Z).

Pour chaque valeurs possible de X, Y, et Z, le grounding d'ASP va nous créer des règles associées. Ainsi, les 27 règles dont grandparentOf(mary, mary) :- parentOf(mary, mary), parentOf(mary, mary), ou encore grandparentOf(charles, mary) :- parentOf(charles, jenny), parentOf(jenny, mary) vont être créées par cette façon de procéder. Ceci n'est pas du tout un problème : le solveur d'ASP va simplement regarder si les atomes des queues sont vraies ou pas. Si ce n'est pas le cas, comme le faux implique tout, les règles seront vraies. Si c'est le cas, la tête sera alors mise à vraie. Sachant qu'une seule de ces 27 possibilités mène à la véracité d'un nouvel atome, on obtiendra en sortie d'ASP les faits suivants :

SATISFAIBLE
parentOf(jenny, charles)
parentOf(mary, jenny)
grandparentOf(mary, charles)

2.4 Agrégats

Les agrégats forment le dernier outil d'ASP que j'ai utilisé dans le cadre de mon stage. Il s'agit d'un moyen de sélectionner un certain nombre d'atomes parmi un ensemble, et de les mettre à vrai. Ils se structurent suivant l'exemple suivant :

$$0\ \{f(A,B) : g(A)\}\ 2.$$

La **borne inférieure**, située à gauche des accolades, et la **borne supérieure** définissent l'intervalle discret du nombre d'atomes que le solveur peut mettre à vrai. Si l'une de ces deux bornes est omise, ASP comprend qu'il s'agit de la borne maximale (ou minimale) possible (à savoir 0 pour la borne inf et cardinal de l'ensemble défini par les accolades pour la borne sup). Les agrégats permettent de faire des disjonctions de cas.

2.5 Exemple du sudoku

Durant les premières semaines de mon stage, j'ai encodé différents jeux de logique en ASP afin de me familiariser avec le langage. Le jeu le plus simple et connu que j'ai encodé a été le sudoku. Dans cette section, je vais briévement détailler le fonctionnement de ce code.
On codera une grille de sudoku en ASP en utilisant s, une fonction d'arité 3 prenant R, C et V comme argument, où R est la ligne, C la colonne et V la valeur de la case (R,C) dans la grille.

$val(1..9).$
1..9 est un raccourci pour dire itérer pour les entiers allant de 1 à 9. Ici, on déclare qu'il y a 9 valeurs dans une fonction d'arité 1 : 'val'
$border(1;4;7).$
On déclare également 3 autres faits pour mémoriser où sont situés les bordures de notre grille
$1\ \{s(R,C,V) : val(V)\}\ 1$:- $val(R); val(C).$
On ne prend qu'une seule valeur par carré
$1\ \{s(R,C,V) : val(R)\}\ 1$:- $val(C); val(V).$
Une valeur ne peut pas apparaître plusieurs fois dans la même colonne
$1\ \{s(R,C,V) : val(C)\}\ 1$:- $val(R); val(V).$
Ni dans la même ligne
$1\ \{s(R,C,V) : val(R), val(C), R1 <= R, R <= (R1+2), C1 <= C, C <= (C1+2)\}\ 1$:- $val(V); border(R1); border(C1).$
Une valeur ne peut pas apparaître plusieurs fois dans une sous-grille.

Une fois que l'on a fait cela, il ne nous reste plus qu'à instancier notre problème sur une grille. Pour cela, on renseigne s(R,C,V) pour chaque case déjà pré-numérotée de la grille à notre code, et le solveur se chargera de nous renvoyer l'intégralité de la grille complétée.

3 Formalismes d'automates

Différents modèles existent pour représenter efficacement un système biologique et manipuler sa dynamique : les deux principaux sont les réseaux booléens synchrones de Stuart Kauffman, et les réseaux asynchrones de René Thomas. Dans le cadre de mon stage je ne parlerai que des réseaux asynchrones, et plus particulièrement des réseaux d'automates asynchrone (AAN)

[Folschette et al., 2015, Paulevé, 2016a], qui forment une extension d'une précédente structure appelé "Process Hitting" [Paulevé et al., 2014].

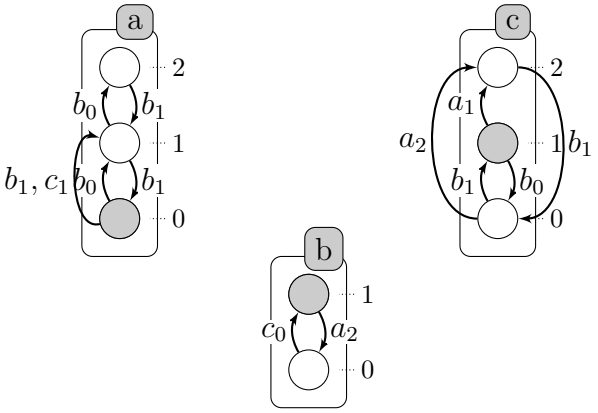
3.1 AAN

Un automate A, dans le contexte de mon stage, sera défini comme étant un ensemble d'états $q_0, q_1, ..., q_{|A|-1}$ avec des transitions dont les étiquettes seront un ou plusieurs états (qu'on appellera également *niveaux*) d'automates externes. Il n'y a donc ni état final, ni état initial. On ne lui donnera pas non plus de mot à lire en entrée puisqu'on va s'intéresser à la dynamique de nos ensembles d'automates. On parlera alors de transition locale : $t(q_i, q_j, l)$ symbolisera le fait que l'on peut passer de l'état q_i à l'état q_j si toutes les conditions de l sont vérifiés.
Un réseau d'automates asynchrone est un triplet (Σ, S, T) avec :

- $\Sigma = \{a, b, ...\}$ est un ensemble fini d'automates non vides.
- Si C_a est le nombre d'états d'un automate a, alors $S_a = \{a_0, a_1, ..., a_{C_a-1}\}$ est l'ensemble des **états locaux** de l'automate a. $S = \prod_{a\in\Sigma} S_a$ est l'ensemble fini des **états globaux**, et $LS = \bigcup_{a\in\Sigma} S_a$ représente l'ensemble de tous les états locaux.
- Pour chaque $a \in \Sigma$, $T_a \subseteq \{(a_i, a_j, l) \in S_a \times S_a \times \rho(LS/S_a) | t(a_i, a_j, l), a_i \neq a_j\}$ est l'ensemble des **transitions locales** d'un automate a, avec ρ qui désigne la puissance ensembliste. $T = \bigcup_{a\in\Sigma} T_a$ est l'ensemble des transitions locales du modèle.

Exemple : On représente l'AAN suivant de cette manière :

- $\Sigma = \{a, b, c\}$
- $S_a = \{a_0, a_1, a_2\}$, $S_b = \{b_0, b_1\}$ et $S_c = \{c_0, c_1, c_2\}$
- $T_a = \{t(a_0, a_1, [b_0]), t(a_0, a_1, [b_1, c_1]), t(a_1, a_0, [b_1]), t(a_1, a_2, [b_0]), t(a_2, a_1, [b_1])\}$
  $T_b = \{t(b_0, b_1, [c_0]), t(b_1, b_0, [a_2])\}$
  $T_c = \{t(c_0, c_1, [b_1]), t(c_0, c_2, [a_2]), t(c_1, c_0, [b_0]), t(c_1, c_2, [a_1]), t(c_2, c_0, [b_1])\}$



3.2 Traduction en ASP

En ASP, on définira un AAN en deux temps :

- On commencera par déclarer chacun de nos automates avec les niveaux (= états) qu'il contient :

automaton_level("a", 0..2).
automaton_level("b", 0..1).
automaton_level("c", 0..2).

- Enfin, les transitions seront encodés via des labels (t1, t2, ...), et on donnera chacune des conditions, ainsi que l'état d'arrivée, via un fait :

condition(t1, "a", 0). target(t1, "a", 1). condition(t1, "b", 0).
condition(t2, "a", 1). target(t2, "a", 2). condition(t2, "b", 0).
condition(t3, "a", 2). target(t3, "a", 1). condition(t3, "b", 1).
condition(t4, "a", 1). target(t4, "a", 0). condition(t4, "b", 1).
condition(t5, "b", 0). target(t5, "b", 1). condition(t5, "c", 0).
condition(t6, "b", 1). target(t6, "b", 0). condition(t6, "a", 2).
condition(t7, "c", 0). target(t7, "c", 1). condition(t7, "b", 1).
condition(t8, "c", 1). target(t8, "c", 0). condition(t8, "b", 0).
condition(t9, "c", 1). target(t9, "c", 2). condition(t9, "a", 1).
condition(t10, "c", 0). target(t10, "c", 2). condition(t10, "a", 2).
condition(t11, "c", 2). target(t11, "c", 0). condition(t11, "b", 1).
condition(t12, "a", 0). target(t12, "a", 1). condition(t12, "b", 1). condition(t12, "c", 1).

Nous avons ainsi défini l'automate de notre **exemple** en ASP.

4 Dynamique

Soit $R = (\Sigma, S, T)$ un AAN (défini ci-dessus). On définit quelques notions :

- On dit qu'une transition locale est **jouable** si toutes les conditions de celle-ci sont vérifiés
  On notera P_ζ l'ensemble des transitions locales jouables depuis un état global ζ
  L'état global (a_0, b_1, c_1) *de l'**exemple** admet une seule transition jouable :* $t(a_0, a_1, [b_1, c_1])$
- On appelle **état stable** un état global ne possédant aucune transition jouable.
  *Dans l'**exemple**, aucun état global n'est stable*

La dynamique d'un AAN se définit à l'aide de sa **sémantique**. La sémantique forme l'ensemble des propriétés définissant les **transitions globales jouables** : dans un AAN on s'intéresse à l'évolution globale du réseau.

4.1 Sémantiques

4.1.1 Asynchrone

Soit $R = (\Sigma, S, T)$ un AAN, et $\zeta \in S$ un état global. L'ensemble des transitions globales jouables depuis ζ pour la sémantique *asynchrone* est donnée par :

$$U^{asyn}(\zeta) = \{\{t(a_i, a_j, l)\} \,|t(a_i, a_j, l) \in P_\zeta\}$$

De manière informelle : chaque transition locale jouable est une transition globale.

En ASP, on commence donc par déclarer que deux états globaux sont différents sur un automate à l'aide de la règle suivante :

```
1. different_on(Gs1, Gs2, Automaton) :- % Cet atome est vrai
2.     global_state(Gs1), global_state(Gs2), Gs1 != Gs2, % si deux états globaux diffèrent
3.     automaton(Automaton), % qu'il existe un automate
```

4. *active_in_g(level(Automaton, LevelI), Gs1), % sur lequel Gs1 vaut I*
5. *active_in_g(level(Automaton, LevelJ), Gs2), % et sur lequel Gs2 vaut J*
6. *LevelI != LevelJ. % avec $I \neq J$*

Ensuite, on fait comprendre à ASP qu'une transition locale n'est pas jouable si l'une (au moins) de ses conditions n'est pas vérifiée de la façon suivante :

1. *unplayable(Transition, Gs) :- % Cet atome est vrai*
2. *local_transition(Transition), % s'il existe une transition locale*
3. *global_state(Gs), % un état global*
4. *automaton(Automaton), % et un automate*
5. *active_in_g(level(Automaton, LevelI), Gs), % sur lequel Gs vaut I*
6. *condition(Transition, Automaton, LevelJ), % alors qu'il devrait valoir $J \neq I$*
7. *LevelI != LevelJ. % pour que la transition soit jouable*

On déclare ensuite que deux états globaux différent sur au moins un autre automate qu'*Automaton* avec le terme $not\_equal\_except(Automaton, Gs1, Gs2)$ comme ceci :

1. *not_equal_except(Automaton, Gs1, Gs2) :- % Cet atome est vrai*
2. *automaton(Automaton), automaton(Automaton2), % s'il existe deux automates*
3. *Automaton != Automaton2, % qui différent*
4. *global_state(Gs1), global_state(Gs2), % deux états globaux*
5. *different_on(Gs1, Gs2, Automaton2). % tels que $Gs1 \neq Gs2$ sur Automaton2*

On peut enfin définir ce qu'est une transition globale jouable à l'aide de nos trois précédentes règles :

1. *playable(Gs1, Gs2) :- % Cet atome est vrai*
2. *global_state(Gs1), global_state(Gs2), % s'il existe deux états globaux Gs1 et Gs2*
3. *automaton(Automaton), local_transition(Transition), % un automate et une transition*
4. *target(Transition, Automaton, LevelJ), % qui fait changer le niveau de l'automate*
5. *not unplayable(Transition, Gs1), % et avec la transition (locale) qui est jouable*
6. *not not_equal_except(Automaton, Gs1, Gs2), % et $Gs1 = Gs2$ excepté sur Automaton*
7. *active_in_g(level(Automaton, LevelI), Gs1), % tel que le niveau d'Automaton dans Gs1*
8. *active_in_g(level(Automaton, LevelJ), Gs2), % et celui dans Gs2*
9. *LevelI != LevelJ. % sont différents*

4.1.2 Synchrone

Soit $R = (\Sigma, S, T)$ un AAN, et $\zeta \in S$ un état global. L'ensemble des transitions globales jouables depuis ζ pour la sémantique *synchrone* est donnée par :

$$U^{syn}(\zeta) = \{u \subseteq T | u \neq \emptyset \wedge \forall a \in \Sigma, (P_\zeta \bigcap T_a = \emptyset \Rightarrow u \bigcap T_a = \emptyset) \wedge (P_\zeta \bigcap T_a \neq \emptyset \Rightarrow |u \bigcap T_a| = 1\}$$

De manière informelle : tous les automates possédant au moins une transition locale jouable doivent changer de niveau.

Pour coder cela en ASP, on peut se servir des atomes *different_on* et *unplayable*.
On ajoute de plus une autre règle : *has_playable(Automaton, Gs, LevelI, LevelJ)* spécifiant que l'on peut faire changer *Automaton* dans *Gs* du niveau I vers le niveau J.
Enfin, il ne nous reste plus qu'a déclarer (par la négation) quelles sont les transitions globales qui ne sont pas jouables. Pour cela, on essaie pour toutes les paires d'états globaux si on peut faire la transition de l'un à l'autre en créant un terme *not_playable(Gs1,Gs2)* si la transition ne peut pas avoir lieu. On énumère alors les différents cas de figure :

1. si *Automaton* n'admet aucune transition locale jouable dans $Gs1$ et que $Gs1 \neq Gs2$ sur *Automaton*

2. si *Automaton* admet une (ou plusieurs) transition locale jouable dans $Gs1$ et qu'il n'existe pas de transition pour faire passer le niveau d'*Automaton* sur $Gs1$ à celui de $Gs2$

3. si aucune transition locale n'est jouable depuis $Gs1$

alors la transition $Gs1 \rightarrow Gs2$ n'est pas jouable.
Enfin, il nous suffit de tester pour toutes les paires d'états globaux distincts si on a *not_playable(Gs1,Gs2)* ou pas. Dans le cas contraire, alors on crée l'atome *playable(Gs1,Gs2)*.

4.1.3 Généralisée

Soit $R = (\Sigma, S, T)$ un AAN, et $\zeta \in S$ un état global. L'ensemble des transitions globales jouables depuis ζ pour la sémantique *généralisée* est donnée par :

$$U^{gen}(\zeta) = \{u \subseteq T | u \neq \emptyset \wedge \forall a \in \Sigma, (P_\zeta \bigcap T_a = \emptyset \Rightarrow u \bigcap T_a = \emptyset) \wedge (P_\zeta \bigcap T_a \neq \emptyset \Rightarrow |u \bigcap T_a| \leq 1\}$$

De manière informelle : tous les automates possédant au moins une transition locale jouable peuvent changer de niveau.

Pour coder cela en ASP, on prend exactement le code de la sémantique synchrone, en remplaçant le second cas de figure par ceci :

2. si $Gs1$ différe de $Gs2$ sur *Automaton* et qu'il n'existe pas de transition pour faire passer le niveau d'*Automaton* sur $Gs1$ à celui de $Gs2$

Cela permet bien de ne pas prendre une transition par automate qui est jouable puisqu'on va tester des paires d'états globaux qui ont peut-être 3 automates jouables, et si l'on ne fait qu'une seule transition locale cela ne sera pas filtré par notre atome *not_playable(Gs1,Gs2)*.

5 Révision du code

6 Conclusion

7 Bibliographie