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

Ce rapport fait écho au chapitre [Ben Abdallah u. a. (2022)]. Mon stage a démarré par la lecture complète de ce chapitre, nécessaire pour traiter le sujet. Mon rapport démarrera donc par introduire les notions majeurs de ce chapitre.

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.

Answer Set Programming (= ASP) est un paradigme de programmation logique particulièrement efficace pour résoudre des problèmes combinatoires. Je n'ai travaillé qu'avec ce langage (et un peu de scripting python) durant mon stage afin de chercher les attracteurs parmi le graphe des états globaux de notre réseau.

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

ASP fonctionne à l'aide de déclarations de faits, de règles, de prédicats contenant un ou plusieurs arguments tels que : *parentOf(jenny, charles)*, ce qui nécessite tout d'abord que j'introduise la notion de **terme** (ou **atome**). On définit un terme de la manière suivante :

1. terme simple :
   — un entier relatif (0, -35, 42)

— une constante qui démarre par une lettre minuscule(v0us, lis35, quatre, ex3mpl35)
— une chaîne de caractères encadrée de guillemets ("ma_chaine_de_caracteres", "42")
— une variable identique grammaticalement aux constantes, mais dont la première lettre est majuscule (N0u5, S0mm35, D35, V4r14bl35)
— un (ou plusieurs) underscore symbolisant une variable sans nom (_, __)

2. une fonction (qu'on appellera également prédicat) est de la forme : constante(t_1, t_2, ..., t_k) avec un nombre de termes (qui sont les arguments) fini qui peut être nul (f(23), je_suis_une_fonction(42, true), et_moi_aussi(oui)).
   Ces fonctions n'ont pas vocation à calculer, mais uniquement à déclarer des objets avec les arguments souhaités comme étant vrais.
3. un tuple de la forme : (t_1, t_2, ..., t_k) avec un nombre de termes fini qui ne peut être nul ((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 $\bot$ (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 $a_0 = \bot$, on parle de contrainte : comme $\bot$ ne peut jamais être vrai, si le corps de la règle est vrai, cela invalide la solution actuelle. On ne renseigne pas $\bot$ 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 et que I est maximale, 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 le corps. 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 de la déduction. 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 :

$0\ \{coloration(X, Y, Teinte) : couleur(Teinte)\}\ 1$:- abscisse(X), ordonnee(Y).

La **borne inférieure**, située à gauche des accolades (ici 0), et la **borne supérieure** (ici 1) 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). Dans ce petit exemple, on choisira donc une ou 0 couleur parmi celles possibles (telles qu'il existe un atome couleur(Teinte) avec la Teinte souhaitée qui soit vrai) pour chaque abscisse et ordonnée définies.
Les agrégats permettent de faire des disjonctions de cas.

2.5 Premier exemple : le 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
care le point virgule permet de déclarer plusieurs atomes en un

$1\,\{s(R,C,V) : val(V)\}\,1$:- $val(R); val(C).$

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.

2.6 Scripting Python

En ASP, on peut écrire des morceaux de script en Lua ou en Python. Pour cela, il suffit d'ajouter la commande '#script (mon_langage)', puis de taper son code dans le langage que l'on a choisi, et de finir le script par la commande '#end.'. Le sccripting admet trois atouts majeurs :

- L'appel au fonction en ASP via la commande '@ma_fonction(et, ses, arguments)', permettant d'effectuer des calculs sous Python et de mettre des variables à vrai en fonction de ce qui a été trouvé dans ces calculs.
- Un contrôle de la résolution permettant le filtrage des ensembles solutions, l'ajout de faits/règles, la demande de grounding et de solving. C'est ce dont je me suis le plus servi durant mon stage.
- Une méthode incrémentale efficace lorsque l'on a besoin d'avoir une notion de temporalité ou bien un calcul qui se fait dans un certain ordre : en fonction des résultats trouvés on peut partir sur une autre piste. La notion d'ordre étant totalement inexistante en ASP (les lignes de code peuvent être écrites dans n'importe quel ordre), cela est pratique.

2.7 Second exemple : le sokoban

Le sokoban a été le dernier jeu de logique que j'ai encodé en ASP, et le plus complexe à cause de la duplication très couteuse du nombre de coups nécessaires pour la résolution d'une grille.

Le jeu est composé d'une grille possédant des cases qui sont du sol ou du mur. Sur une case représentant le sol, il ya le joueur. Sur plusieurs autres cases de sol il y a des caisses. Sur autant de cases de sol qu'il y a de caisses, on trouve des cases d'arrivée (qui sont également du sol). Le but du jeu est de pousser toutes les caisses sur les cases d'arrivée sachant que le joueur ne peut se déplacer qu'en haut, à gauche, en bas ou à droite, et ne peut pas traverser les murs ni pousser une caisse s'il y a un mur ou une autre caisse derrière celle-ci.

Une grille de sokoban est encodée en spécifiant quelles cases sont des murs, et quelles cases sont des sols avec les prédicats $mur(X,Y)$ et $sol(X,Y)$. On rensigne de plus un prédicat $init(X,Y)$, ce qui nous donne la position initiale du joueur, et $caisse\_init(Numero\_caisse,X,Y)$ nous renseignant sur les positions initiales des caisses (le premier argument varie de 1 à n lorsque l'on a

n de ces caisses). Enfin, le prédicat $arrivee(X, Y)$ symbolisera une case d'arrivée en X, Y.

J'ai fait différentes versions pour ce jeu. L'une d'entre elles utilise une méthode incrémentale. Pour l'importer, il nous suffit d'ajouter la commande '#include <incmode>.'. On doit alors définir les sous-programmes *base*, *step(k)* et *check(k)*.

— **base :**

1. perso(0, R, C) :- init(R, C).
% prédicats pour mémoriser la position du joueur à l'étape 0
2. caisse_a_instant(0, R, C) :- caisse_init(_, R, C). % ainsi que celles des caisses

— **step(k) :**

Pour des raisons de lisibilité, je ne mettrai qu'une instruction pour une seule direction

3. acces_a_instant(k-1, R, C) :- perso(k-1, R, C).
% prédicats pour mémoriser toutes les cases accessibles

4. acces_a_instant(k-1, R+1, C) :- sol(R+1, C), acces_a_instant(k-1, R, C), not caisse_a_instant(k-1, R+1, C).
% on ajoute les cases à droite de celles accessibles (idem pour les 3 autres directions)

5. 1 { coup_a_instant(k, (R, C), (R+1, C)) : % on peut faire un coup à l'instant k
6. *sol(R, C), sol(R+2, C),% si l'on a deux cases de sol c_1 et c_2*
7. *acces_a_instant(k-1, R, C),% tels que l'on ait accès à c_1*
8. *caisse_a_instant(k-1, R+1, C),% qu'il y ait une caisse entre c_1 et c_2*
9. *not caisse_a_instant(k-1, R+2, C) ;% et qu'il n'y a pas de caisse en c_2*
10. *[...] % idem pour les 3 autres directions*
11. *} 1.*

12. perso(k, R2, C2) :- coup_a_instant(k, _, (R2, C2)). % Déplacement du personnage

*13. caisse_a_instant(k, 2*R2-R, 2*C2-C) :- % On pousse une seule caisse : elle bouge*
14. *sol(R2, C2), sol(R, C), sol(2*R2-R, 2*C2-C),*
15. *caisse_a_instant(k-1, R2, C2), coup_a_instant(k, (R, C), (R2, C2)).*

16. caisse_a_instant(k, R, C) :- % On ne pousse pas une caisse : elle ne bouge pas
17. *sol(R, C), sol(R2, C2),*
18. *caisse_a_instant(k-1, R, C), perso(k, R2, C2), R2 != R.*

19. caisse_a_instant(k, R, C) :- % Idem, mais pour les colonnes
20. *sol(R, C), sol(R2, C2),*
21. *caisse_a_instant(k-1, R, C), perso(k, R2, C2), C2 != C.*

22. caisse_pas_place(k) :- caisse_a_instant(k, R, C), not arrivee(R, C).
% On détermine s'il existe encore des caisses qui ne sont pas placées pour l'étape k

23. fini(k) :- not caisse_pas_place(k).
% Si toutes les caisses sont placées, on a fini la grille

— **check(k) :**

24. instant(k) :- k != 0. % si on est à un instant $k \geq 1$
25. :- not fini(k), query(k). % que l'on a fini à cette étape
26. :- query(k), k = 0. % et qu'on n'a pas $k = 0$

% Alors check(k) est vrai, ce qui a pour conséquence d'arrêter les appels successifs à clingo (le grounder et solveur d'ASP), ce qui fait arrêter le programme

On récupère alors la liste des coups de caisses à faire, et on peut résoudre notre grille en comblant les coups qui déplaçaient les caisses par ceux nécessaires pour atteindre la prochaine caisse à déplacer.

3 Formalismes d'automates et leur dynamique

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

3.1 AAN et leur traduction en ASP

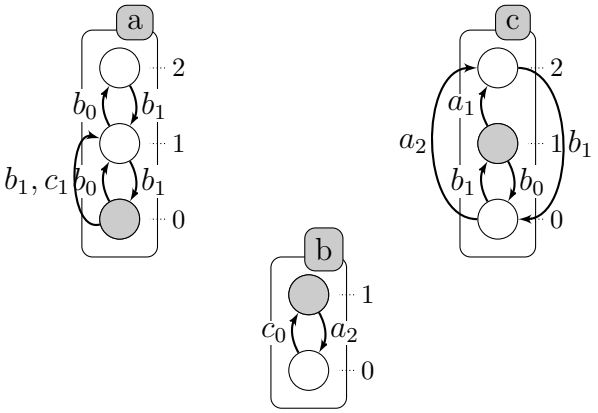
Un automate A, dans le contexte de mon stage, sera défini comme étant un ensemble d'états $q_0, q_1, ..., q_{|A|-1}$ avec des transitions dont les étiquettes seront un ou plusieurs états (qu'on appellera également *niveaux*) d'automates externes. Il n'y a donc ni état final, ni état initial. On ne lui 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 : $q_i \xrightarrow{l} q_j$ 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 $a \in \Sigma$, $T_a \subseteq \left\{ a_i \xrightarrow{l} a_j \in S_a \times \rho(LS/S_a) \times S_a | a_i \neq a_j \right\}$ 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 $S_c = \{c_0, c_1, c_2\}$
— $T_a = \left\{ a_0 \xrightarrow{b_0} a_1, a_0 \xrightarrow{b_1, c_1} a_1, a_1 \xrightarrow{b_1} a_0, a_1 \xrightarrow{b_0} a_2, a_2 \xrightarrow{b_1} a_1 \right\}$

$$T_b = \left\{ b_0 \xrightarrow{c_0} b_1, b_1 \xrightarrow{a_2} b_0 \right\}$$
$$T_c = \left\{ c_0 \xrightarrow{b_1} c_1, c_0 \xrightarrow{a_2} c_2, c_1 \xrightarrow{b_0} c_0, c_1 \xrightarrow{a_1} c_2, c_2 \xrightarrow{b_1} c_0 \right\}$$



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.

3.2 Sémantiques

Soit $R = (\Sigma, S, T)$ un AAN (défini ci-dessus). On définit la notion de transition jouable :

— 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 : $a_0 \xrightarrow{b_1, c_1} a_1$*

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.

3.2.1 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 :

$$U^{asyn}(\zeta) = \left\{ \left\{ a_i \xrightarrow{l} a_j \right\} | a_i \xrightarrow{l} a_j \in P_\zeta \right\}$$

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 ≠ 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 ≠ 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 ≠ 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
```

3.2.2 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)*.

3.2.3 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, et au moins l'un d'entre eux doit 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)*.

3.3 Dynamique

Soit U une sémantique. On introduit les dernières notions nécessaires pour la définition d'un attracteur :

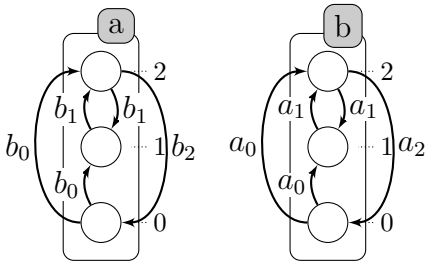
— On appelle **état stable** un état global ne possédant aucune transition globale jouable pour U.
*Dans l'**exemple**, aucun état global n'est stable pour la sémantique asynchrone (et donc cela est également vrai pour la sémantique synchrone et généralisée)*
— Pour $Gs_1, Gs_2 \in S$, on notera $Gs_1 \rightarrow_U Gs_2$ pour symboliser le fait qu'il existe $u \in U(Gs_1)$ tel que si l'on joue toutes les transitions locales de u, alors Gs_1 devient Gs_2.
— Un **chemin** est une famille finie $(A_k)_{1 \leq k \leq n}$ et ordonnée d'états globaux tels que pour tout $1 \leq k \leq n-1$, $A_k \rightarrow_U A_{k+1}$.
— Un **cycle** est un chemin $(A_k)_{1 \leq k \leq n}$ tel que $A_1 = A_n$.
— Un **domaine de piège** est un ensemble non vide d'états globaux $S_E \in S$ tel que toute transition globale jouable depuis S_E arrive dans un état global de S_E. Plus formellement : $\forall \zeta_1 \in S_E \wedge \forall \zeta_2 \in S, \zeta_1 \rightarrow_U \zeta_2 \Rightarrow \zeta_2 \in S_E$

Enfin, on peut définir ce qu'est un attracteur.
Un ensemble $A \subseteq S$ d'états globaux, avec $|A| \geq 2$ est appellé **attracteur** si et seulement si c'est un domaine de piège minimal en terme d'inclusion ensembliste.
Un travail précédent ([Ben Abdallah u. a. (2022)]) a démontré le résultat mathématique suivant dont on va se servir pour trouver les attracteurs au sein d'un AAN :

Lemme : Les attracteurs d'un AAN sont exactement les domaines de piège cycliques.

4 Contribution personnelle

La version de laquelle j'ai commencé à travailler est assez proche du travail que j'ai effectué. Cette version était fonctionnelle et efficace pour la sémantique synchrone, mais ne fonctionnait pour la sémantique synchrone que lorsque les attracteurs étaient exactement des cycles simples (chaque état global du graphe produit doit avoir un degré sortant égal à 1), et ne fonctionnait pas pour les autres AAN comme celui-ci :



En effet, l'ensemble $\{(0,0),(1,1),(1,2),(2,1),(2,2)\}$ est un attracteur de cet AAN pour la sémantique synchrone. De plus, il existe deux (même quatre ici) transitions globales jouables depuis l'état global $(0,0)$: on peut aller en $(1,1)$ ou encore en $(2,2)$.

Après avoir identifié le problème au sein du code, j'ai trouvé deux alternatives pour corriger cela : la première consiste à reprendre tout le code, enlever le problème ciblé et corriger cela avec un script Python, et la seconde consiste à considérer les états globaux et à définir les transitions globales jouables dans des fichiers dédiés aux sémantiques.
Lors de mon exploration de la seconde alternative, je me suis confronté à de nombreux problèmes, et la complexité finale de mon algorithme étant piteuse, je ne parlerai dans ce rapport que de la première alternative.
Je vais donc détailler comment nous arrivons à trouver les attracteurs au sein des AAN en ASP avec la sémantique synchrone.

4.1 Code pré-existant

On commence par renseigner quelques faits utiles pour pouvoir manipuler aisèment les informations que l'on a à disposition.

```
%%% Initialisation (faits étendus)
% Noms des automates
automaton(Automaton) :- automaton_level(Automaton, _).
% Noms des transitions locales
local_transition(Transition, Automaton) :- target(Transition, Automaton, _).
local_transition(Transition) :- target(Transition, _, _).

%%% Etapes
% Etapes dans tout le chemin
step(0..n). % Ici n est une constante qu'on définit lors de la compilation (souvent 10)
% Longueur du cycle principal (i.e., un sous-chemin cyclique)
1 {main_cycle_length(N) : step(N), N >0 } 1.
% Etapes dans le cycle principal
cycle_step(0..N) :- main_cycle_length(N).
% Etapes après le cycle principal
after_cycle_step(N+1..n) :- main_cycle_length(N).
```

On choisit ensuite un état initial, on définit les transitions non jouables, et la sémantique.

```
%%% Etat initial
% On sélectionne aléatoirement un état inital (step "0")
1 { active(level(Automaton, Level), 0) : automaton_level(Automaton, Level) } 1 :-
        automaton(Automaton).

%%% Préparation pour le calcul des transitions jouables par la sémantique
% Calcule les transitions locales non jouables pour chaque étape
unplayable(Transition, Step) :-
        active(level(Automaton, LevelI), Step),
        condition(Transition, Automaton, LevelJ),
        LevelI != LevelJ, step(Step).

%%% Sémantique synchrone
% On renseigne les automates qui ont au moins une transition locale jouable
has_playable(Automaton, Step) :-
        not unplayable(Transition, Step),
        local_transition(Transition, Automaton),
        step(Step).

% On sélectionne une transition à jouer pour chaque automate si possible
1 {played(Transition, Step) :
        not unplayable(Transition, Step),
        local_transition(Transition, Automaton)
} 1 :- has_playable(Automaton, Step).
```

```
% Contrainte : on doit jouer au moins une transition locale à chaque étape
22.  :- 0 { played(_, Step) } 0, step(Step).
```

Maintenant que l'on a choisi notre coup, on en déduit l'état suivant.

```
%%% Calcule l'état suivant, le successeur de chaque S
% A Step, Automaton utilisent les Transition pour changer du niveau LevelI à LevelJ
23. change(Transition, Automaton, LevelI, LevelJ, Step) :-
24.         played(Transition, Step),
25.         target(Transition, Automaton, LevelJ),
26.         condition(Transition, Automaton, LevelI).

% On change le niveau actif s'il y a un changement dans Automaton
27. active(level(Automaton, LevelK), Step + 1) :-
28.         change(_, Automaton, _, LevelK, Step),
29.         Step <n.

% On garde le niveau actif s'il n'y a pas de changement dans Automaton
30. active(level(Automaton, LevelK), Step + 1) :-
31.         not change(_, Automaton, _, _, Step),
32.         active(level(Automaton, LevelK), Step),
33.         step(Step), Step <n.
```

Je viens ici d'expliquer comment faire pour générer tous les chemins de longueur n avec la sémantique synchrone. Cependant, pour qu'un de ses chemins nous intéressent, il faut qu'il respecte les 3 contraintes suivantes :

- avoir un cycle de longueur N (avec main_cycle_length(N))
- tout les états globaux du chemin visités après l'étape N (avec main_cycle_length(N)) doivent être des éléments du cycle
- toutes les transitions globales jouables depuis chacun des éléments du cycle doivent arriver dans un autre élément de ce cycle (= domaine piège)

Les deux premières conditions se vérifient assez aisèment.

Pour la première, on crée un prédicat *different_states_on(Step1, Step2, Automaton)* nous permettant de déduire si deux états globaux atteints à *Step*1 et *Step*2 sont différents ou pas. On en déduit alors un prédicat *same_state(Step1, Step2)*, et il ne nous reste qu'à ajouter la contrainte *:- not same_state(0, N), main_cycle_length(N).*

Quant à la seconde, il nous suffit de créer un prédicat *valid_state_after_main_cycle(Step2)* vrai lorsqu'il existe une étape *Step*1 dans le cycle principal et une étape *Step*2 en dehors du cycle principal telles que l'on ait l'atome *same_state(Step1, Step2)* qui soit vrai. Ce prédicat devant être toujours vrai pour chaque *Step*2, on rajoute finalement la contrainte suivante : *:- not valid_state_after_main_cycle(Step1), after_cycle_step(Step1).*

La troisième contrainte contenait la raison du bug du code. Comme la modélisation actuelle des choses ne mémorise pas quels sont les coups jouables pour la sémantique choisie (initialement l'idée était d'avoir un code qui fonctionne pour toutes les sémantiques en scindant le fichier ASP en deux : recherche des attracteurs, et sémantique, ce qui permettait d'avoir un seul solveur d'attracteur, universel pour toutes les sémantiques), cela n'était pas possible.

Deux options étaient alors envisageables pour résoudre ce problème : faire un script ASP dans laquelle je rajoute la dernière contrainte en introduisant une notion d'état global ; ou bien utiliser Python pour résoudre ce problème lors du solving de clingo. Comme dit précédemment, je ne parlerais que de la seconde option.

4.2 Résolution de la troisième contrainte en Python

Je ne vais pas rentrer en détail dans la manière dont on peut réussir à récupérer des atomes lors du grounding et du solving d'ASP en Python, mais sachez seulement que cela est faisable. Nous avons donc en Python une boucle for qui va, à chaque instance, avoir un ensemble d'atomes tous vrais tel que l'interprétation qu'il forme satisfasse toutes les règles de notre programme ASP.

Lors de la première itération de cette boucle, il est nécessaire de mémoriser toutes les transitions, les conditions, ainsi que les cibles de ces transitions.

On commence par créer des ensembles vides, ainsi qu'une variable booléenne pour savoir si nous avons déjà mémorisé ce que l'on souhaite.

```
   #Ensemble Python pour les conditions des transitions ([transition][automata] = level).
1. condition_transitions = {}

   #Ensemble Python pour l'automate cible des transitions ([transition] = automaton_name).
2. target_transitions_automaton_name = {}

   #Ensemble Python pour le niveau cible des transitions ([transition] = level).
3. target_transitions_automaton_level = {}

   #Une variable booléenne pour dire si les 3 ensembles précédents sont déjà créés ou pas.
4. transitions_already_done = False
```

Boucle for magique
Ensuite, dans notre boucle for magique, on introduit une variable associé à l'attracteur courant, qu'on remplit, ainsi que les 3 autres variables.

```
5. cur_dict = {}
6. for x in cur_model.symbols(shown = True) :
            #On considère les atomes 'active(level(AUTOMATA, LEVEL), STEP)'
7.          if x.name == 'active' and x.arguments[0].name == 'level' :
8.              step_number = x.arguments[1].number
9.              automata_name = x.arguments[0].arguments[0].string
10.             active_level = x.arguments[0].arguments[1].number
                #Si c'est un nouveau numéro d'étape
```

```
11.                 if step_number not in cur_dict :
                        #On le rajoute dans le dictionnaire
12.                     cur_dict[step_number] = {}
                    #A priori il n'y a pas plusieurs niveaux pour un automate à une étape ( ?)
13.                 assert automata_name not in cur_dict[step_number]
14.                 cur_dict[step_number][automata_name] = active_level

                #Création des 3 ensembles pour les transitions si ce n'est pas déjà fait
15.             elif not transitions_already_done :
                    #On considère les atomes 'condition(TRANSITION, AUTOMATA, LEVEL)'
16.                 if x.name == 'condition' :
17.                     transition = x.arguments[0].name
18.                     automata_name = x.arguments[1].string
19.                     level = x.arguments[2].number
                        #Si c'est une nouvelle transition
20.                     if transition not in condition_transitions :
                            #On la rajoute dans le dictionnaire
21.                         condition_transitions[transition] = {}
22.                     assert automata_name not in condition_transitions[transition]
23.                     condition_transitions[transition][automata_name] = level
                    #On considère les atomes 'target(TRANSITION, AUTOMATA, LEVEL)'
24.                 elif x.name == 'target' :
25.                     transition = x.arguments[0].name
26.                     automata_name = x.arguments[1].string
27.                     level = x.arguments[2].number
                        #A priori il n'y a qu'un atome target par transition
28.                     assert transition not in (target_transitions_automaton_name
                            ↪ or target_transitions_automaton_level)
                        #Donc on vient rajouter ces informations dans nos ensembles
29.                     target_transitions_automaton_name[transition] = automata_name
30.                     target_transitions_automaton_level[transition] = level
    #Cette dernière ligne nous certifie qu'on ne crée qu'une fois les 3 ensembles
31. transitions_already_done = True
```

On introduit deux variables.

```
    #La liste des noms d'automates triée par ordre alphabétique.
32. sorted_automata_names == []
33. if sorted_automata_names == [] :
34.     sorted_automata_names = sorted(cur_dict[0])

    #Ensemble (non ordonné) d'états dans l'attracteur courant.
35. cur_set = set()
    #Pour chaque étape
36. for step_number in cur_dict :
        #On ajoute l'état correspondant à cur_set
37.     cur_set.add(tuple(cur_dict[step_number][automata_name]
                    ↪ for automata_name in sorted_automata_names))
```

On commence par créer et compléter l'ensemble des transitions locales jouables depuis chacun des états globaux de l'attracteur courant.

#playable est l'ensemble de toutes les transitions locales jouables depuis chaque état global de l'attracteur courant et chaque automate de celui-ci. Il est de la forme playable[global_state][automata_name][transition] = level où 'level' est le niveau de l'automate après la transition.

```
playable = {}
number_automaton = len(sorted_automata_names)
#Pour chaque automate de l'attracteur courant
for global_state in cur_set :
    playable[global_state] = {}
    #Pour chaque transition
    for transition in condition_transitions :
        #On teste si elle est jouable pour l'état global courant
        can_be_played = True
        #En regardant pour chaque condition de la transition
        for automata_name in condition_transitions[transition] :
            if automata_name not in playable[global_state] :
                playable[global_state][automata_name] = {}
            #Si elle est vérifiée ou pas
            if global_state[sorted_automata_names.index(automata_name)] !=
                ↪ condition_transitions[transition][automata_name] :
                can_be_played = False
        #Si elle est bien jouable
        if can_be_played :
            automata_name = target_transitions_automaton_name[transition]
            level = target_transitions_automaton_level[transition]
            #On l'ajoute au dictionnaire
            playable[global_state][automata_name][transition] = level
```

#list_of_list est une liste de listes de listes d'entiers. Pour faire simple, chacune de ses sous-listes représente un état global de l'attracteur courant. Chacune des sous-listes de ces états globaux représente un automate, et chacune des sous-listes de cet automate correspond à l'ensemble des niveaux dans lesquels il peut se retrouver après une transition locale jouable depuis l'état global. Si aucune transition locale n'est jouable, la sous-liste de l'automate en question sera vide.

```
list_of_list = []
#Pour chaque état global de l'attracteur courant
for global_state in playable :
    #On crée une liste pour l'état global courant
    list_for_current_state = []
    #Pour chaque automate
    for automaton_name in sorted_automata_names :
        #On crée une liste pour l'automate courant
        list_for_automaton = []
        #Si aucune transition locale n'est jouable pour l'automate, on le laisse à son niveau
        if playable[global_state][automaton_name] == {} :
            list_for_automaton.append(global_state[sorted_automata_names.index(
```

```
                    ↪ automaton_name)])
            #Sinon
60.         else :
                #Pour chaque transition locale jouable
61.             for transition in playable[global_state][automaton_name] :
                    #On détermine le niveau après transition
62.                 level = playable[global_state][automaton_name][transition]
                    #Et on l'ajoute à la liste des niveaux atteignables pour l'automate
63.                 list_for_automaton.append(level)
64.         list_for_current_state.append(list_for_automaton)
65.     list_of_list.append(list_for_current_state)

    # list_of_product est la liste des produits cartésiens de chaque coup possible pour un état
global. Il s'agit donc d'une liste d'états globaux atteignables depuis nos états présents dans le
cycle principal.
66. list_of_product = []
67. for l in list_of_list :
        # product est une fonction réalisant un produit cartésien de listes
68.     list_of_product += list(product(tuple(l)))

    # Enfin, on s'assure que chaque élément de list_of_product est bien un élément du cycle
principal. De cette façon, notre attracteur vérifiera bien la troisième condition, et sera valide.
69. neighbours_are_main_states = True
70. for global_state in list_of_product :
71.     if global_state not in cur_set :
72.         neighbours_are_main_states = False

Manipulation pour filtrer proprement les attracteurs et pas leur trace
→ On ne renvoie un attracteur que s'il n'a pas déjà été renvoyé (puisque jusqu'ici, si on parcourt
notre cycle depuis un autre point de départ, on renvoie un second ensemble solution alors qu'en
réalité il s'agit du même attracteur).

Fin boucle for magique
```

4.3 Légère modification pour sémantique généralisée

En reprenant le code présenté ci-dessus, j'ai également fait une version pour la sémantique généralisée (la version de laquelle je suis parti fonctionnant très bien pour la sémantique asynchrone, je n'ai rien à dire dessus). En effet, la différence avec la sémantique synchrone est que l'on peut faire changer nos automates ou pas, mais qu'au moins un de tous les automates doit changer de niveau lorsque l'on recherche quelles sont les transitions globales jouables. Concrètement, il faut modifier la sémantique :

```
[...]
% On sélectionne 0 ou 1 transition à jouer pour chaque automate
18. 0 {played(Transition, Step) :
19.         not unplayable(Transition, Step),
```

```
20.         local_transition(Transition, Automaton)
21. } 1 :- has_playable(Automaton, Step).
[...]
```

On doit ensuite modifier dans le code python le fait que list_fo_automaton soit initialement vide : pour chaque automate, on peut soit rester dans cet automate, soit faire une des transitions trouvés. Pour cela, on modifie la ligne 57 et les suivantes comme ceci :

```
[...]
57.         list_for_automaton = [global_state[sorted_automata_names.index(automaton_name)]]
            #Pour chaque transition locale jouable
58.         for transition in playable[global_state][automaton_name] :
                #On détermine le niveau après transition
59.             level = playable[global_state][automaton_name][transition]
                #Et on l'ajoute à la liste des niveaux atteignables pour l'automate
60.             list_for_automaton.append(level)
61.         list_for_current_state.append(list_for_automaton)
62.     list_of_list.append(list_for_current_state)

[...]
```

4.4 Quelques mots sur la seconde manière de procéder

L'autre solution que je voyais pour résoudre le problème qu'on rencontrait mon maître de stage et ses collègues était de déclarer des prédicats *global_state(Gs)* représentant les états globaux du réseau. Cela permet de déclarer dans des fichiers pour les sémantiques quels sont les transitions locales jouables en utilisant un prédicat *playable(Gs1, Gs2)* signifiant que l'on peut aller depuis $Gs1$ en $Gs2$.

Le principal défaut de cette manière de procéder est que cela crée de nombreux atomes et demande à ASP de les garder en mémoire pendant tout son solving. De plus, déterminer si pour toute paire d'états globaux il existe une transition se fait en $O(k^2)$, avec $k = \prod_{a \in \Sigma} C_a$, alors que si l'on calcule les transitions jouables à chaque étape, on ne surcharge pas la mémoire du solveur. C'est à priori ce qui rend mon programme obsolète en terme de complexité.

5 Conclusion et pistes pour la suite

Durant mon stage j'ai élaboré deux versions pour résoudre le problème rencontré avec l'ajout de la troisième contrainte. J'ai présenté dans ce rapport la solution qui fait appel à un script Python, ainsi que la version modifiée pour traiter la sémantique généralisée. J'ai très brièvement parlé de la solution nécessitant la création d'états globaux en ASP.

Mes performances sur la batterie de test qu'à utiliser Maxime pour la version de laquelle je suis parti se sont avérés tout aussi bonne que celles de la sémantique asynchrone. Les résultats de ces tests sont dans les annexes. Les performances de ma seconde version y sont également. On voit notamment à quel point elle n'est pas efficace en temps.

Concernant cette seconde version, je pense que l'on peut grandement améliorer son efficacité en combinant mon code avec de l'incrémental. L'idée serait de ne considérer qu'un certains nombres d'états globaux : au début on les considère tous, puis, lorsque l'on trouve un attracteur, on regroundre le programme en enlevant tout les états globaux de l'attracteur (en les mettant tous à faux).

Une autre idée que j'avais en tête était, lorsque l'on trouvait un cycle qui n'était pas un attracteur, de remplacer tout les états de ce cycle par un seul représentant, dont les transitions globales entrantes et sortantes seront celles de chacun des états globaux du cycle. On travaillerait ensuite avec des représentants et des états globaux comme s'il s'agissait du même objet (si on a un représentant dans un autre cycle, on l'agrandira en rajoutant les autres éléments du cycle). Je pense que combiner cette idée avec de l'incrémental pourrait également être intéressant : à chaque cycle trouvé on effectue de nouveau un grounding en mettant à jour nos représentants.

Je ne suis pas sûr de ces pistes, mais c'est ce sur quoi j'essaierais personnellement de me pencher pour poursuivre le travail.

Enfin, je souhaite remercier l'ENS de Lyon qui m'a proposé ce stage, Maxime F. pour son encadrement, les personnes au sein de l'équipe BioComputing, mes quelques collègues stagiaires de bureau (même si le stage a été en semi-distanciel) ainsi que mes quelques relecteurs anonymes.

Références

[Ben Abdallah u. a. 2022] Ben Abdallah, Emna ; Folschette, Maxime ; Morgan, Magnin : *Analyzing Long-Term Dynamics of Biological Networks with Answer Set Programming*. Juli 2022. – URL `https://hal.archives-ouvertes.fr/hal-03735849`. – working paper or preprint

[Folschette u. a. 2015] Folschette, Maxime ; Paulevé, Loïc ; Magnin, Morgan ; Roux, Olivier : Sufficient conditions for reachability in automata networks with priorities. In : *Theoretical Computer Science* 608 (2015), S. 66–83

[Paulevé 2016] Paulevé, Loïc : Pint, a static analyzer for dynamics of Automata Networks. In : *14th International Conference on Computational Methods in Systems Biology (CMSB 2016)*, 2016

[Paulevé u. a. 2014] Paulevé, Loïc ; Chancellor, Courtney ; Folschette, Maxime ; Magnin, Morgan ; Roux, Olivier : Analyzing Large Network Dynamics with Process Hitting. In : *Logical Modeling of Biological Systems* (2014), S. 125 – 166. ISBN 978-1-4020-4222-5