Ecole Centrale de Nantes *Université de Nantes* *Ecole des Mines de Nantes*

MASTER AUTOMATIQUE ROBOTIQUE ET SYSTEMES DE PRODUCTION

SPECIALITE : TEMPS REEL CONDUITE ET SUPERVION (TRCS)

Année 2013 / 2014

Thèse de Master

Présenté et soutenu par :

Emna BEN ABDALLAH

Le
18/09/2014

à
L'École Centrale de Nantes

TITRE

Recherche exhaustive des propriétés dynamiques dans le Process Hitting en utilisant l'Answer Set Programming

JURY

Président :	**Didier LIME**	**Maître de Conférences HDR dans l'équipe temp réel, ECN**
Examinateurs :	**Olivier ROUX**	**Professeur et Responsable de l'équipe MeForBio, ECN**
	Maxime FOLSCHETTE	**Doctorant dans l'équipe MeForBio, ECN**
	Morgan MAGNIN	**Maitre de conférences dans l'équipe MeForBio, ECN**

Directeur de thèse : Pr. Olivier ROUX et Maxime FOLSCHETTE

Laboratoire : Équipe MeForBio, IRCCyN, ECN

Table des matières

Remerciements

Je tiens à remercier mes deux encadreurs, Pr. Olivier ROUX et Maxime FOLSCHETTE, pour leur soutien continu et leurs conseils tout au long de cette thèse de master. Je remercie aussi chaleureusement Morgan MAGNIN pour son suivi tout au long de mon stage. Je les remercie tous pour tout ce qu'ils m'ont apporté, pour leur présence, leur patience, pour m'avoir fait confiance et m'avoir laissé la liberté nécessaire à l'accomplissement de mes recherches, tout y gardant un œil critique et avisé. Je les remercie également pour l'excellente ambiance que j'ai vécue dans leur équipe pendant mon stage. Je remercie beaucoup M. Olivier H.ROUX, responsable du master ARIA à l'École Centrale de Nantes. J'adresse tous mes reconnaissances à tous mes professeurs pendant mes études de Master notamment de mes 3 années en cycle ingénieur informatique. Je tiens également à remercier tout le personnel et les professeurs de l'Ecole Centrale de Nantes et à l'Ecole Nationale des Ingénieurs à Sfax, ayant contribué de près ou de loin au bon déroulement de cette thèse de master. Finalement, un grand Merci chaleureux et de tout mon cœur à mes chers parents, sans qui je ne serais absolument pas où j'en suis aujourd'hui. Je les remercie pour leurs encouragements et soutien constants.

Résumé

Le Process Hitting (PH) est un formalisme récemment introduit pour modéliser les processus concurrents. Il est notamment apte à modéliser les réseaux de régulation biologiques et permet une analyse efficace de ces derniers. Dans cette thèse de master, nous expliquons les méthodes que nous avons développées en programmation logique (Answer Set Programming) pour trouver les points fixes, des états au bout desquels il n'est plus possible d'avoir des évolutions du modèle autrement dit aucun changement d'état n'est possible. Nous visons également à résoudre le problème d'atteingnabilité qui consiste à décider si, à partir d'un état initial donné, il est possible d'atteindre un objectif précis, cet objectif peut être un état partiel ou complet fixé. Enfin, nous illustrons les mérites de nos méthodes en les appliquant à des exemples biologiques avec une comparaison de certaines méthodes existantes.

Introduction

J'ai effectué mon stage dans l'équipe MeForBio (Méthodes Formelles pour la Bioinformatique) de l'IRCCyN (Institut de Recherche en Communications et Cybernétique à Nantes) qui est composée de trois permanents, deux doctorants. Elle est impliquée dans plusieurs projets d'envergure nationale dont CirClock (CNRS) et BioTempo (ANR), et possède des partenariats avec d'autres équipes de recherche (équipe Bio-Info de l'I3S a Sophia Antipolis) et des collaborations internationales sont en cours (notamment avec l'Allemagne et le Japon). En effet ils travaillent sur tout ce qui est en rapport avec les réseaux de régulation biologiques (RRBs). Les RRBs consistent en des ensembles de composants biologiques ayant des effets mutuellement positifs ou négatifs entre les composants. Puisque les phénomènes de régulation jouent un rôle capital dans les systèmes biologiques, ils doivent être étudiés avec précision. Dans le but de l'analyse de ces systèmes, ils sont souvent modélisés sous forme de graphes qui permettent de déterminer les évolutions possibles de tous les composants du système d'interaction. En effet, afin de répondre à la vérification formelle de propriétés dynamiques dans de très grands BRN, l'équipe MeForBio dans laquelle j'ai effectué mon stage a commencé à utiliser récemment un nouveau formalisme, appelé le « Process Hitting » (PH) (ou les frappes des processus) [9].

Ce formalisme permet de modéliser des systèmes concurrents ayant des composants avec plusieurs niveaux qualitatifs. Un PH décrit, d'une façon atomique, les évolutions possibles d'un processus (représentant un niveau d'un composant) ciblé par l'action d'au plus un autre processus du même système. Cette structure particulière permet notamment une analyse efficace des BRN avec des centaines de composants. Grâce à la précision de ces informations, le PH est bien adapté pour modéliser des RRBs avec différents niveaux d'abstraction en capturant les dynamiques les plus générales.

Les objectifs des travaux présentés dans ce mémoire sont les suivants. Nous montrons, tout d'abord, qu'à partir d'un modèle de PH, il est possible de trouver tous les états stables (points fixes [15]).

Nous définissons un état du réseau à un instant donné par l'ensemble des processus actifs à cet instant. Nous effectuons alors une recherche exhaustive des états possibles pour le réseau biologique étudié. La deuxième phase de mon travail consiste à calculer la dynamique d'évolution du réseau. Elle consiste à déterminer à partir d'un état initial connu les futurs états possibles. Enfin, je vérifie s'il est possible d'atteindre un état spécifique d'un ou plusieurs composants (gène ou protéine).

Nous avons eu des résultats, qui permettent de déterminer les états stables et vérifient si la dynamique du réseau satisfait nos objectifs. Pour cela nous avons utilisé l'Answer Set Programming (ASP) [3] afin d'effectuer ce calculer. ASP a émergé à la fin des années 1990 comme un nouveau paradigme de programmation logique, ayant ses racines dans le raisonnement non monotone, les bases de données déductives et la programmation logique avec négation par échec. Depuis sa création, il a été considéré comme l'incarnation du calcul de raisonnement non monotone et un outil efficace de représentation des connaissances. Ce point de vue a été stimulé par l'émergence de solveurs très efficaces pour ASP. Il semble désormais difficile de contester que l'ASP a apporté une nouvelle vie à la programmation la logique.

En effet ASP s'est montré efficace pour traiter des modèles avec un grand nombre de composants et de paramètres. Notre objectif ici est d'évaluer son potentiel c'est à dire le calcul de certaines propriétés dynamiques des modèles de PH.

Dans cette thèse de master, nous montrons qu'ASP s'avère être efficace pour ces recherches énumérative qui justifient son utilisation. Le bénéfice de notre approche est qu'elle permet d'obtenir des résultats exacts et assez rapides. Ainsi pour la dynamique il nous permet de vérifier s'il est possible d'atteindre des objectifs précis et si oui d'obtenir des chemins minimaux pour les atteindre.

1 État de l'art

Durant mon stage j'ai utilisé principalement deux outils. Le premier est un langage de développement programmation logique : l'Answer Set Programming (ASP). Précisément c'est un langage de programmation logique à propos duquel j'ai rédigé mon rapport de recherche bibliographique. Le deuxième framework est un *focus* des travaux de l'équipe, le Process Hitting (PH) un nouveau formalisme utilisé pour la représentation des réseaux biologiques. Nous verrons dans la suite de cette section plus de détails à propos de ces deux frameworks.

1.1 Answer Set Programming

L'ASP est un paradigme de programmation déclarative avec une sémantique connue comme la sémantique des ensembles de réponse (answer sets). Ce paradigme permet au programmeur de spécifier quel est le problème à résoudre et non pas comment le résoudre. Les programmes ASP, qui sont écrits dans AnsProlog* (l'abréviation de "Answer Set Programming in Logic" avec * en exposant), sont composés d'un ensemble de faits et d'un ensemble de règles à partir desquels d'autres faits peuvent être dérivés. Un ensemble de faits cohérents qui peut être dérivé à partir d'un programme à l'aide des règles est appelé « ensemble de réponse » pour le programme. Les ensembles de réponses possibles pour un programme d'AnsProlog* sont calculés avec un programme appelé un solveur.

1.1.1 Programme logique

On considère un programme logique d'AnsProlog tel que chaque règle est une paire ordonnée,[3] :

$$head \leftarrow body. \tag{1}$$

avec $Head$ et $Body$ des ensembles de littéraux. Cette règle est alors équivalente à :

$$L_0 \ or \ ... \ or \ L_k \ \leftarrow \ L_{k+1}, ..., L_m, \ \textbf{not} \ L_{m+1}, ..., \ \textbf{not} \ L_n. \tag{2}$$

Les L_i sont des littéraux dans le sens de la logique classique. Le **not** est utilisé pour exprimé la négation.

D'une manière intuitive, la règle 2 énonce que :

$$\left.\begin{array}{l}\text{Si } L_{k+1}, ..., L_m \text{ sont } \textbf{vrais} \\ \text{et } L_{m+1}, ..., L_n \text{ sont } \textbf{faux}\end{array}\right\} \text{Alors au moins l'un des littéreaux de } L_0 \text{ à } L_k est \ \textbf{vrai}$$

Particulièrement la règle 1 est aussi représentée sous la forme :

$$L_0 \leftarrow L_1, ..., L_m, \textbf{not} L_{m+1}, ..., \textbf{not} L_n. \tag{3}$$

Et cette règle énonce que :

$$\left.\begin{array}{l}\text{Si } L_1, ..., L_m \text{ sont } \textbf{vrais} \\ \text{et } L_{m+1}, ..., L_n \text{ sont } \textbf{faux}\end{array}\right\} \text{Alors } L_0 \text{ est } \textbf{vrai}$$

Dans la suite on se contente des règles de types (3) utilisée dans le développement de mes scripts pour la résolution des problèmes. Pour plus d'informations, je propose aux interessés de consulter mon rapport de recherche bibliographique [1] qui était à propos l'Answer Set Programming.

Ce langage simple a beaucoup d'avantages qui en font l'un des plus efficaces au niveau de la représentation des connaissances, du raisonnement et de la résolution de problèmes déclaratifs.

Pour commencer, les symboles non classiques "$\leftarrow$", "*not*", dans les règles (3) donnent au programme AnsProlog une structure et permettent de définir facilement des sous-classes syntaxiques et leurs propriétés. Il se trouve cette sous-classe a un niveau de complexité et d'expressivité importants. Ces avantages nous permettent ainsi de choisir les sous-classes appropriées pour des applications particulières.

1.1.2 La syntaxe et sémantique d'ASP

1. **L'alphabet**

   D'après [3] , l'alphabet de l'axiome (ou tout simplement l'alphabet) du framework de l'answer set se constitue de sept classes de symboles :

   - Variables,
   - Constantes d'objets (référencé aussi par constantes),
   - Symboles des fonctions,
   - Symboles des prédicats,
   - Connecteurs,
   - Symboles de ponctuation, et
   - Symbole spécial $\bot$

   Toutes ces classes varient d'un alphabet à un autre mais les ensembles des 5^{eme} et 6^{eme} classes (Connecteurs et Symboles de ponctuation) sont définis comme suit :

   - Les connecteurs par {$\neg$, or, $\leftarrow$, not, ‘,’}
   - Les symboles de ponctuation par {‘(’, ‘)’, ‘.’}

   Les autres classes restent constantes du fait qu'elles respectent une certaine convention informelle. En général, les variables commencent avec une lettre majuscule et contiennent des lettres et des chiffres. Les constantes, symboles et prédicats suivent la même règle, mais ils commencent par une minuscule. Parfois il y a l'addition d'une convention supplémentaire qui porte sur les lettres utilisées [3]] :

   - Variables : X,Y,Z,... Commencent par une lettre en **majuscule**
   - Prédicats : p,q,. . .
   - Fonctions : f,g,h,. . .
   - Constantes : a,b,c,. . .

   Les variables doivent commencer par une majuscule alors que les prédicats, les fonctions et les constantes commencent toujours par une minuscule.

Ce qui semble un peu flou est le concept de prédicat. En effet le mot prédicat peut être une innovation de la nouvelle grammaire. Pourtant, ni le mot ni la réalité qu'il recouvre ne sont nouveaux. Utilisé en logique, le mot prédicat désigne le second terme d'un énoncé (1er terme : variable ou constante), terme qui dit quelque chose du sujet de l'énoncé. Cette définition dépend de (et s'oppose à) celle de sujet (dans l'acception classique), qui est le « quelque chose » en question, c'est-à-dire ce dont on parle dans une phrase. Soit la phrase suivante :

Socrate était athénien.

Socrate est le sujet et *était athénien* le prédicat. Cette phrase peut être notée en ASP par :

athenien(Socrate) $\leftarrow \top.$ avec $\top$ le symbole de vrai.

2. **Les règles** Comme nous l'avons déjà mentionné, une règle est de la forme :

$$head \leftarrow body. \qquad ((1))$$

$$A \leftarrow B^+, \textbf{ not } B^-. \qquad (4)$$

Équivalente à :

$$L_0 \leftarrow L_1, ..., L_m, \textbf{ not } L_{m+1}, ..., \textbf{ not } L_n. \qquad ((3))$$

avec L_i des littéraux et k ≥ 0, m $\geq$ k et n $\geq$ m.

Head est appelé aussi conclusion (tête de la règle)
Body est appelé aussi prémisse (le corps de la règle)

et si on nomme une règle de cette forme par r on aura les définitions suivantes :
$head(r) = A = L_0$
$body(r) = B^+, notB^- = \{L_1, ..., L_m, \textbf{not } L_{m+1}, \ldots, \textbf{not } L_n\}$,
$body^+(r) = pos(r) = B^+ = \{L_1, \ldots L_m\}$,
$body^-(r) = neg(r) = B^- = \{L_{m+1}, \ldots L_n\}$.

Des cas spéciaux des règles [8, 3] :

- Une règle est **constante** si tous ses littéraux sont des constantes (notée par **ground**, ground signifie constant) ;
- Règle **disjonctive** : c'est toute règle ayant plus qu'un littéral dans le head (c'est à dire k ≥ 1 dans les règles 2). Ce type de règles je ne m'en sers pas dans la suite.
- Une **réalité** ou un **fait** : c'est une règle telle que le body est vide et sans avoir une disjonction ($k = n = m = 0$). Elle peut être écrite même sans la flèche $\leftarrow$:

$$L_0. \ ou \ L_0 \leftarrow \top. \qquad (5)$$

- Une **contrainte** : c'est une règle telle que $L_0 = \bot$. Ce symbole faux à la tête (*head*) est souvent éliminé et la contrainte sera écrite d'une manière générale ainsi :

$$\leftarrow\ L_1, ..., L_m,\ \textbf{not}\ L_{m+1}, ...,\ \textbf{not}\ L_n. \tag{6}$$

  Nous disons qu'un ensemble X de littéraux viole la contrainte (6), si $\{L_1, ..., L_m\} \subseteq X$ et $\{L_{m+1}, ..., L_n\} \nsubseteq X$. Si nous avons un programme ayant des contraintes de type la règle 6, alors X est un answer set de ce programme π si et seulement si : X est un *answer set* de de $\pi \setminus \{r\}$, avec r une contrainte telle 6, et X ne viole pas la contrainte 6.

- Les contraintes de cardinalité :
  Il s'agit des littéraux étendus de la forme suivante :

$$l\ \{q_1,\ ...,\ q_m\}\ u. \tag{7}$$

  avec $m \geq 1$, l un entier et u peut être un entier ou l'infini par défaut s'il n'existe pas. l et u les limites inférieure et supérieure de la cardinalité des sous-ensembles de $\{q_1,\ ...,\ q_m\}$ qui satisfont les answer sets. Ces littéraux qui sont contraints (q_i) peuvent apparaitre dans la tête et le corps de la règle. Une contrainte de cardinalité est satisfaite dans un answer set X, si le nombre des atomes de $\{q_1,\ ...,\ q_m\}$ appartenant à X est compris entre l et u. Autrement dit :

  $l\ \leq\ |\{q_1, ...,\ q_m\}\ \cap\ X|\ \geq\ u$

  avec $\cap$ le symbole d'intersection et $|A|$ est le cardinal de l'ensemble A.

1.1.3 Modélisation d'un problème en ASP

On peut considérer que la construction de modèles est l'une des composantes fondamentales de la démarche scientifique. Elle concerne tout système que nous cherchons à maîtriser. Un modèle possède deux caractéristiques principales [2] :

- il est une simplification d'un système donné
- il permet une action sur ce système

On modélise pour apporter, d'une manière ou d'une autre, une solution à un problème identifié comme tel. Cette notion de modèle permet d'aborder sous un autre angle les questions liées aux processus de représentation.

1. **Les étapes de modélisation**

   Le processus de modélisation peut être considéré comme une forme particulière de représentation dont les opérations sont détaillées dans [3]. Les programmes logiques d'ASP suivent la stratégie « générer et tester ». Cette stratégie comprend quatre étapes :

   - Enumérer avec des faits ;
   - Expliquer avec des règles ;
   - Générer toutes les possibilités avec des cardinalités, et enfin ;

- Filtrer avec des contraintes.

2. **Exemple N-Queens**

   A titre d'exemple considérons le problème bien-connu des n-reines (ou n-queens) comme il a été résolu dans [2]. Le but est de placer n reines sur un échiquier de dimension $n \times n$ de sorte que deux reines n'apparaissent jamais sur la même ligne, colonne, ou diagonale. Pour simplifier ce problème on va le représenter en suivant les étapes énoncées ci-dessus :

   - **Enumérer avec des faits**
     On propose de représenter le positionnement des reines par des atomes de la forme $q(i, j)$, où 1 ≥ i, j ≥ n. Autrement dit, un atome $q(i, j)$ indique qu'une reine est à la position (i, j), un i^{eme} ligne et j^{eme} colonne.
     En outre, nous introduisons un atome auxiliaire $q'(i, j)$ indiquant qu'il n'y a **pas** de reine à la position (i, j). En fait, q' joue le rôle de $\neg q$, et sa déclaration est explicite car il ne fait pas partie de la sémantique (Nous préférons ne pas utiliser la négation dans nos programmes).
     Enfin, il nous faut un prédicat pour les dimensions des lignes et des colonnes. Soit d le prédicat indiquant qu'il s'agit d'une dimension de l'échiquier tel que $d(n)$ indique que l'entier n est une dimension dans l'échiquier (de la ligne ou de la colonne)
     Soit cet ensemble de faits qui permet de définir un échiquier de n cases :

     d(1).
     d(2).
     ...
     d(n).

   - **Expliquer avec des règles**
     Tout en tenant en compte de ces faits, le fonctionnement de la dualité q et q' est en conséquence représenté comme suit, Π :

     $$q(X, Y) \leftarrow d(X),\ d(Y),\ not\ q'(X, Y) \tag{8}$$

     $$q'(X, Y) \leftarrow d(X),\ d(Y),\ not\ q'(X, Y) \tag{9}$$

     Dans l'équation (8), "$q(X, Y)$" indique l'existence d'une reine dans la position (X, Y) et "**not** $q'(X, Y)$" l'absence d'information sur l'existence d'une reine à cette position avec X et Y des dimensions de l'échiquier "$d(X),\ d(Y)$", et inversement pour l'équation (9).

   - **Générer toutes les possibilités avec des cardinalités**
     Prenons le cas d'un échiquier de dimension 1×1, donc la dimension sera égale à 1 : $d(1)$ est le seul atome possible pour le prédicat "d". Donc de cet échiquier on n'a que 2 cas possibles, il existe ou n'existe pas une reine dans cette case unique. Ainsi on obtient 2 *answer sets* du programme Π :

     $X_1 = \{\ q(1, 1),\ d(1)\}$ *et* $X_2 = \{q'(1, 1),\ d(1)\}$

     Si on augmente les dimensions de l'échiquier tel que n=2, on aura 16 *answer sets* parmi lesquels :

Aucune reine dans toutes les cases :

$$\{\ q'(1,1),\ q'(1,2),\ q'(2,1),\ q'(2,2),\ d(1),\ d(2).\} \tag{10}$$

1 reine dans une seule case (1,1) :

$$\{\ q(1,1),\ q'(1,2),\ q'(2,1),\ q'(2,2),\ d(1),\ d(2).\} \tag{11}$$

2 reines dans les deux cases diagonales :

$$\{\ q(1,1),\ q'(1,2),\ q'(2,1),\ q(2,2),\ d(1),\ d(2).\} \tag{12}$$

4 reines dans toutes les cases :

$$\{\ q(1,1),\ q(1,2),\ q(2,1),\ q(2,2),\ d(1),\ d(2).\} \tag{13}$$

etc...

Toutes ces règles de l'équation (10) à l'équation (13) peuvent être présentées simplement par :

$$\{q(X,Y)\ :\ d(X)\ \}\ \leftarrow\ d(Y). \tag{14}$$

$$\{q(X,Y)\ :\ d(Y)\ \}\ \leftarrow\ d(X). \tag{15}$$

En effet, chaque ligne et chaque colonne peuvent avoir un ensemble de reines (ou *queens*). Ces règles nous permettent d'avoir toutes les combinaisons possibles des queens sur un échiquier. En revanche certaines de ces réponses ne sont pas satisfaisantes pour répondre au problème des N-Queens, ce qui demande une élimination des fausses rèponses.

- **Filtrer avec des contraintes**

  Maintenant on doit éliminer les candidats qui ne vérifient pas la condition "Jamais deux reines sur la même ligne, colonne ou diagonale". Cela sera mis en évidence par l'ajout des contraintes :

  - **Colonne :** L'existence de deux reines sur la même colonne (même Y) est faux, avec X, X' et Y des indices de dimensions dans l'échiquier :

$$\leftarrow\ q(X,Y),\ q(X',Y),\ ' \neq X,\ d(X),\ d(Y),\ d(X'). \tag{16}$$

  - **Ligne :** L'existence de deux reines sur la même ligne (même X) est faux, avec X, Y et Y' des indices de dimensions dans l'échiquier :

$$\leftarrow\ q(X,Y),\ q(X',Y),\ Y' \neq Y,\ d(X),\ d(Y),\ d(Y'). \tag{17}$$

  - **Diagonale :** L'existence de deux reines sur la même diagonale ($|X-X'| = |Y-Y'|$) est faux, avec X, X', Y et Y' des indices de dimensions de l'échiquier :

$$\leftarrow\ q(X,Y),\ q(X',Y'),\ |X\breve{}X'| = |Y\breve{}Y'|,\ X' \neq X,\ Y' \neq Y,\ d(X),\ d(Y),\ d(X'),\ d(Y'). \tag{18}$$

    En effet, toutes ces contraintes ne sont pas compatibles avec l'ensemble (13) qui sera par suite éliminé.

    Par contre on ne doit pas oublier qu'il faut avoir toutes les n reines sur l'échiquier, cela peut être vérifié par les règles ci-dessous qui éliminent à leur tour les ensembles (10) et (11) :

– $hasq(X)$ est vrai quand il y a une reine dans la X^{eme} ligne :

$$hasq(X) \leftarrow d(X),\ d(Y),\ q(X,Y) \tag{19}$$

– Une ligne qui n'a pas une reine doit invalider la solution :

$$\leftarrow d(X),\ \textbf{not}\ hasq(X) \tag{20}$$

+ **Optimisation**
On peut améliorer encore ces règles. Pour être sûr qu'il n'y a qu'une seule reine dans chaque colonne j, on utilise l'expression :

$$1\ \{q(1,j),...,\ q(n,j)\}\ 1\ \leftarrow\ d(j) \tag{21}$$

En effet cette expression implique que dans l'ensemble $\{q(1,j),...,\ q(n,j)\}$ on ne peut avoir qu'une seule reine dans la j^{eme} colonne du moment que cette position $(q(.,j))$ est un sous-ensemble de ce grand ensemble qui est contraint par un minimum =1 et un maximum = 1.
Ou encore plus simplifié avec le littéral conditionné suivant qui contrainte la regle 2 , quel que soit X un numéro de ligne (dimension d'une ligne de l'échiquier), on a :

$$1\ \{q(X,Y)\ :\ d(X)\ \}\ 1\ \leftarrow\ d(X) \tag{22}$$

Ainsi, la problématique d'unicité sur les colonnes et les lignes du programme des n-queens pourra être codée comme ci-dessous. Ces 2 règles remplacent toutes les lignes de 2 à 20)

$$1\ \{q(X,Y)\ :\ d(X)\ \}\ 1\ \leftarrow\ d(Y) \tag{23}$$

$$1\ \{q(X,Y)\ :\ d(Y)\ \}\ 1\ \leftarrow\ d(X) \tag{24}$$

1.1.4 Résolution des programmes ASP

1. **Génération et élimination d'Answer set**
Soit π une collection des règles de la forme (3).
La sémantique d'un programme π est donnée dans les atomes des *answers sets* du programme $ground(\pi)$.
Définissons :

   – **Univers d'Herbrand et base d'Herbrand**
   Les *answer sets* d'un programme sont des sous-ensembles d'atomes de la base de Herbrand, qui satisfont certaines conditions :

   Définition 1. Soit L un Langage en ASP, on note alors :
   HU_L : L'univers d'Herbrand des langages de L : c'est l'ensemble de tous les termes constants.

   HB_L : La base d'Herbrand ensemble de tous les atomes constants qui peuvent être formés par des prédicats de L et les termes de HUL comme arguments des prédicats.

   Exemple 1. Considérons le langage L_1 pris de [3] :
   X et Y des variables, a et b des constantes, f symbole d'une fonction d'arité 1 et p symbole de prédicat d'arité 1.
   L'univers d'Herbrand de L_1 :
   C'est l'ensemble $\{a, b, f(a), f(b), f(f(a))etf(f(b)), f(f(f(b)))...\}$

La base d'Herbrand de L_1 :
C'est l'ensemble $\{p(a), p(b), p(f(a)), p(f(b)), p(f(f(a))), p(f(f(b)))...\}$
Le *grounder* (ou stabilisateur)est responsable de ce calcul pour les programmes en ASP.

- **La fermeture**

Définition 2. Soit π un programme stabilisé (qui ne contient donc que des constantes et aucune variable), et soit X un ensemble de littéraux. On dit que X est fermé sous π, si pour chaque règle r dans π, $head(r) \subset X$ si $body_+(r) \subset X$ et $body_-(r) \cap X = \varnothing$. On note par $C_n(\pi)$: le plus petit ensemble d'atomes qui est fermé sous π et on l'appelle l'*answer set* de π.

- **La réduction**

Définition 3. Soit π programme d'AnsProlog et X un ensemble des atomes de HB_π. π^X est la réduction de π par rapport à X obtenue par la transformation de Gelfond-Lifschitz qui consiste à supprimer :
- Toute règle r ayant un naf-littéral $(not\ L)$ dans le body tel que $L \in X$, et
- Les naf-littéraux dans le body des règles restante
On a alors :

$\pi^X = \{head(r) \ \leftarrow \ body^+(r)\ tel\ que\ r \in \pi\ avec\ body^-(r) \cap X = \varnothing\}$.

Si $C_n(\pi^X) = X$ autrement dit si X est un answer set de π^X alors on dit que X est aussi un answer set de π.
La notion d'*answer set* est d'abord définie pour les programmes qui ne contiennent pas la négation par échec "*not*" : $n = m$ dans chaque règle (3) du programme, ce qui signifie que pour toute règle r de π, $body^-(r) = \varnothing$.
D'après [3], l'answer set d'un programme π', la version positive de π : sans la négation dans les corps des règles $(m = n)$, est :
- Le plus petit modèle de l'ensemble d'Herbrand de ce programme, ou encore,
- Le plus petit ensemble fermé des atomes de la base d'Herbrand correspondante à ce programme : $C_n(\pi')$.
Nous allons étendre dans la suite la définition d'un *answer set* pour des programmes ayant la négation par échec "**not**". Dans l'exemple qui suit on va se référencer à ce que : si $C_n(\pi^X) = X$ alors X est un *answer set* de π.

Exemple 2. On considère le programme π :

$$p \leftarrow p \tag{25}$$

$$q \leftarrow not\ p \tag{26}$$

alors on a,

$head(25) = \{p\}$, $body^+(25) = \{p\}$, $body^-(25) = \varnothing$,
et $head(26) = \{q\}$, $body^+(26) = \varnothing$, $body^-(26) = \{p\}$.

Le tableau ci-dessous permet de calculer l'answer set du programme π :

X	$body^-(25) \cap X$	$body^-(26) \cap X$	Π^X	$C_n(\pi^X)$	$C_n(\pi^X) = X$?
$\varnothing$	$\varnothing$ $\Rightarrow$ Bon	$\varnothing$ $\Rightarrow$ Bon	$p \leftarrow p$ $q \leftarrow$	$\{p\}$	Non
$\{p\}$	$\varnothing$ $\Rightarrow$ Bon	$\{p\}$	$p \leftarrow p$	$\varnothing$	Non
$\{q\}$	$\varnothing$ $\Rightarrow$ Bon	$\varnothing$ $\Rightarrow$ Bon	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	Oui
$\{p, q\}$	$\varnothing$ $\Rightarrow$ Bon	$\{p\}$	$p \leftarrow p$	$\varnothing$	Non

FIGURE 1 – Calcul de l'Answer set du programme π

Parmi les ensembles proposés, on ne trouve qu'un seul ensemble qui est un answer set pour ce programme, celui où on a $C_n(\pi^X) = X$: c'est $\{q\}$.

2. **Les solveurs de l'Answer Set**
D'aprés [8] et [5], le couple grounder et solver fonctionnent généralement ensemble : le grounder sert à supprimer les variables pour obtenir un programme constant (seulement des constantes) et le solveur calcule l'ensemble des anwser sets pour le programmes stabilisé généré par le grounder.

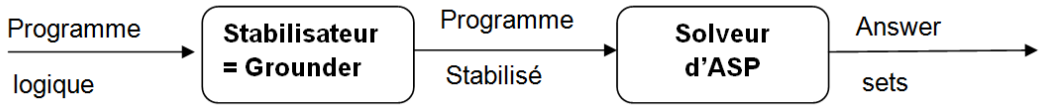

FIGURE 2 – Étape de résolution d'un programme en ASP

Exemple de Stabilisateur (Grounder) : `GRINGO` , `DLV`, `LPARSE`

Exemple Solveur d'ASP : `SMODELS`, `DLV`, `CMODELS`, `CLASP`...

L'utilisation conjointe du stabilisateur et solveur permet de spécifier de grands programmes dans un format compact, à l'aide de règles avec de variables schématiques et d'autres abréviations. Les deux systèmes emploient des algorithmes de stabilisation (grounding) sophistiqués qui fonctionnent rapidement et simplifient le programme. Durant mon stage nous avons travaillé avec `CLINGO` qui est une combinaison du grouder `GRINGO` et du solveur `clasp`.

1.2 Le Process Hitting

Dédié la modélisation de grands systèmes complexes, le Process Hitting permet de représenter les modèles possédant une structure simple dont nous prenons avantage pour le développement d'analyses statiques efficaces. Le Process Hitting regroupe un nombre fini de processus (chacun étant biologiquement équivalent à un niveau) séparés en un ensemble de sortes (dont chacune représente un composant biologique). À tout instant, un et un seul processus de chaque sorte est présent (autrement dit chaque composant biologique a un niveau à chaque instant). Un processus peut être remplacé par un autre processus de même sorte par la frappe d'un autre processus présent ce qui se traduit par

le changement de niveaux des composant après une certaine réaction. Particulièrement adapté à la modélisation des RRB, le Process Hitting permet une construction simple de la dynamique généralisée des RRB.

1.2.1 Réseau de régulation biologique

Les phénomènes de régulation jouent un rôle crucial dans de nombreux systèmes biologiques, tel la production des protéines au sein d'une cellule. La figure 3 schématise le phénomène de régulation génique : quand un gène est exprimé, il produit, à travers un processus de transcription et de traduction, une protéine, qui peut avoir un effet d'activation ou d'inhibition sur l'expression d'un autre gène, accélérant ou freinant ainsi la production d'autres protéines (ou sa propre production).

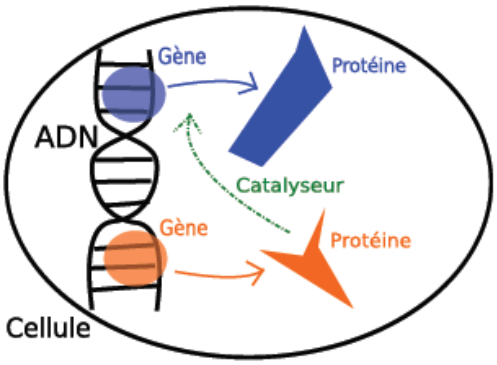

FIGURE 3 – Schéma d'un phénomène de régulation au sein d'une cellule [9] : un gène produit une protéine (à travers un processus biologique complexe) qui a un effet de catalyseur (accélération ou décélération) sur la production d'une autre protéine.

Ces informations qualitatives sur l'influence positive ou négative entre les composants d'un système biologique sont représentées par un graphe des interactions : On trouve des noeuds qui représentent des entités biologiques (gène, ARN, etc.), et sont reliés par des arcs orientés positifs ou négatifs dénotant respectivement une activation ou une inhibition. La figure 4 donne un exemple de graphe des interactions.

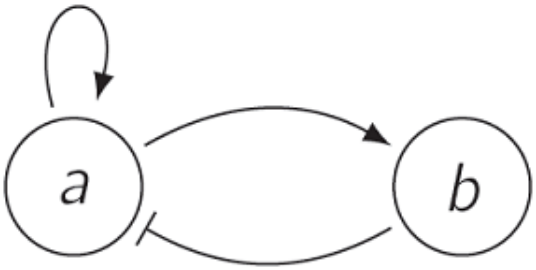

FIGURE 4 – Exemple pris de [9] de graphe des interactions d'un RRB ayant deux composants a et b avec un arc activateur de a vers b, une auto-activation de noeud a et un arc inhibiteur de b vers a.

Ce graphe des interactions constitue une spécification très abstraite d'un RRB et n'indique pas précisément l'évolution de la concentration des composants en fonction de leurs régulateurs. C'est pour cette raison qu'il y a eu le développement du nouveau formalisme appelé le Process Hitting que nous abordons par la suite.

1.2.2 Définition du PH

la figure 5 présente le Process Hitting(PH)[9] ,qui permet de modéliser un nombre fini de niveaux locaux, appelées processus, regroupés en un ensemble fini d'éléments, appelés sorte. Un processus est noté par a_i, où a est le nom de la sorte, et i est l'identifiant du processus dans la sorte. A tout moment, exactement un processus de chaque sorte est actif, et l'ensemble des processus actifs est appelé un état.
Les interactions concurrentes entre les processus sont définies par un ensemble d'actions. Ces actions décrivent le remplacement d'un processus par un autre de la même sorte conditionné par la présence d'au plus un autre processus dans l'état actuel. Une action est notée par $a_i \to b_j \not\Rsh b_k$, qui est lu comme "a_i frappe b_j pour le faire bondir à b_k", où a_i, b_j, b_k sont des processus de sortes a et b, appelées respectivement le *frappeur*, la *cible* et le *bond* de l'action. Nous notons également une *auto-frappe* toute action dont le frappeur et cible sont les mêmes, de la forme : $a_i \to a_i \not\Rsh a_k$.

Définition 4 (Process Hitting)**.** ([9])

Un *Process Hitting* est un triplet $(\Sigma, \mathcal{L}, \mathcal{H})$:

- $\Sigma = \{a, b, \ldots\}$ est l'ensemble fini de toutes les *sortes* ;
- $\mathcal{L} = \prod_{a \in \Sigma} \mathcal{L}_a$ est l'ensemble des états avec $\mathcal{L}_a = \{a_0, \ldots, a_{l_a}\}$ l'ensemble fini des *processus* de la sorte $a \in \Sigma$ et l_a un entier positif, avec $a \neq b \Rightarrow \mathcal{L}_a \cap \mathcal{L}_b = \emptyset$;
- $\mathcal{H} = \{a_i \to b_j \not\Rsh b_k \in \mathcal{L}_a \times \mathcal{L}_b^2 | (a, b) \in \Sigma^2 \wedge b_j \neq b_k \wedge a = b \Rightarrow a_i = b_j\}$ est l'ensemble fini d'*actions*.

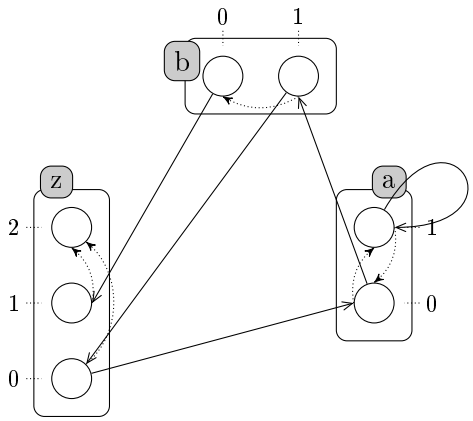

FIGURE 5 – Un réseau de PH ayant 3 sortes a, b et z. Chaque sorte a des niveaux présentés par des cercles qui sont les processus (sorte a a 2 processus a_0 et a_1, etc) et on trouve 4 actions qui assurent la dynamique du réseau (action $a_0 \to b_1 \not\Rsh b_0$).

Un état du réseau est un ensemble de processus actifs et on trouve un seul processus pour chaque sorte. Dans un état $s \in \mathcal{L}$, on dit qu'un processus $a_i \in s$ avec a le nom de la sorte et i le niveau du processus à cet état, et on note par : $s[a] = a_i$

Définition 5 (Acion jouable)**.** Soient $(\Sigma, \mathcal{L}, \mathcal{H})$ des Frappes de Processus et $s \in \mathcal{L}$ un état. On dit que l'action $h = a_i \rightarrow b_j \not\Uparrow b_k \in \mathcal{H}$ est jouable dans l'état s si et seulement si $frappeur(h) = a_i \ \in s$ et $cible(h) = b_j \in s$ (équivalent à dire que $s[a] = a_i$ et $s[b] = b_j$)
L'état résultant du jeu d'une action h dans s est dénoté par $(s \cdot h)$ ou $(s \cdot h)[b] = b_k$ et $\forall c \in \Sigma,\ c \neq b, (s \cdot h)[c] = s[c]$.

1.2.3 Propriétés dynamiques

1.2.4 Point fixe

L'étude des points fixes (et plus généralement des bassins d'attractions) de la dynamique apporte une compréhension importante des différents comportements d'un RRB [15]. Ce sont les états stables du réseau. Nous abordons la caractérisation des points fixes dans le Process Hitting. Il apparaît que les points fixes d'un PH composé de n sortes sont exactement les n-cliques dans une représentation complémentaire du PH appelée Graphe Sans-Frappe (où deux processus sont mis en relation si et seulement si aucun d'eux ne frappe l'autre). L'extraction des points fixes des Frappes de Processus revient alors à énumérer les n-cliques d'un graphe n-parti, pouvant être implémenté de manière efficace. La caractérisation des points fixes des Process Hitting permet l'étude des points fixes sur des RRB dont les paramètres discrets n'ont été spécifiés que partiellement : ces points fixes sont alors communs à toutes les dynamiques issues d'un raffinement de cette spécification partielle.

Définition 6 (Point Fixe du Process Hitting)**.** Soit $(\Sigma, \mathcal{L}, \mathcal{H})$ un Process Hitting. Un état $s \in \mathcal{L}$ est un point fixe si aucune action de $\mathcal{H}$ n'est jouable ($\forall h \in \mathcal{H},\ frappeur(h) \ \notin s \ \vee \ cible(h) \notin s$).

Étant données un Process Hitting, nous introduisons une représentation complémentaire à celle de graphe de la figure 5 que nous appelons Graphe Sans-Frappe (définition 7). Le Graphe Sans-Frappe de Frappes de Processus $(\Sigma, \mathcal{L}, \mathcal{H})$ met en relation deux processus de sortes différentes si et seulement si ils ne se frappent. Les sommets d'un Graphe Sans-Frappe peuvent être séparés en $n \leq |\Sigma|$ partitions, où aucun sommet d'une partition n'a de relation avec un autre sommet de la même partition.

Définition 7 (Graphe sans-frappe)**.** Le graphe *Sans-Frappe* des Frappes de Processus $(\Sigma, \mathcal{L}, \mathcal{H})$ est un graphe (V, F) non-orienté où les sommets V et les arcs E sont définis de la maniére suivante :
$V \ = \ \{a_i`|a \in \mathcal{L}_a \wedge \nexists h \in \mathcal{H}, frappeur(h) = a_i,\ cible(h) = a_i \ \},$
$E \ = \ \{\{a_i, b_j\} \subseteq V | \nexists h \in \mathcal{H}, \{frappeur(h), cible(h)\} = \{a_i, b_j\}\}.$

Définition 8 (n-clique)**.** Étant donné un graphe (V, E), $C \subseteq V$ est une $|C| - clique$ du graphe si et seulement si $\forall\ \{a_i,\ b_j\} \subseteq C,\ \{a_i,\ b_j\} \in E$.

Théorème 1. Soient $(\Sigma, \mathcal{L}, \mathcal{H})$ des Frappes de Processus. Un état $s \in \mathcal{L}$ est un point fixe des Frappes de Processus si et seulement si s correspond à une $|\Sigma| - clique$ du Graphe Sans-Frappe associé.

Les figures 5 et 6 illustrent le théorème 1 avec des Frappes de Processus ayant un seul point fixe. Le Graphe Sans-Frappe correspondant ne contient pas le processus a_1, car il exerce une frappe sur lui-même (une auto-frappe).

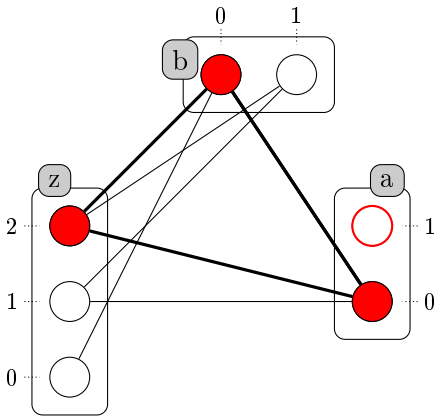

FIGURE 6 – Graphe sans-frappes correspondant à la figure 5, où chaque arc représente un non-frappe entre les 2 processus. Le processus a_1 était supprimé car il a une auto-frappe. Les processus colorés reliés par les arcs en gras constituent le 3-clique du graphe, ils forment alors un point fixe.

On remarque que ce réseau n'a qu'un seul point fixe $< a_0, b_0, z_2 >$. En effet c'est la seul combinaison de trois processus qui forment un 3-clique (n=3, le nombre des sortes du réseau)

1.2.5 Atteingnabilité

Dans cette section, nous nous concentrons sur l'accessibilité d'un réseau à un ou des processus Définition (9), en répondant à la question : "Est-il possible, à partir d'un état initial donné, de jouer un certain nombre d'actions afin qu'un processus donné soit actif dans l'état qui en résulte ?"

Comme déjà mentionné dans la définition 5, une fois qu'une action $h = a_i \to b_j \Rsh b_k$ est jouée dans un état s, elle sert à évoluer le réseau vers l'état résultant suivant s'. On trouve alors dans l'état s' tous les processus de s avec le remplacement du processus $b_j = cible(h)$ par $b_k = bond(h)$.
On appelle l'ensemble de successions d'actions $(h_0 \cdot h_1 \cdot h_2...)$ à partir de l'état s_0, par scénario noté $\mathbf{Sce}(s_0)$ correspondant à la succession d'état $(s_0 \mapsto s_1 \mapsto s_2...)$.

Définition 9 (Question d'atteignabilité)**.** Si $\varsigma \in \mathcal{L}$ est un état et $A \in \mathbf{Proc}$ est un ensemble de processus, on note $\mathcal{P}(\varsigma, A)$ pour *la question d'accessibilité* : $\exists ?\delta \in \mathbf{Sce}(\varsigma), A \subseteq ([\varsigma] \cdot \delta)$ (càd $\forall a_i \in A, (\varsigma \cdot \delta)[a] = a_i$). Avec $\mathbf{Sce}(\varsigma)$ est l'ensemble des actions successivement jouables à partir de l'état ς.

Pour mieux comprendre on peut prendre un exemple de réseau biologique ci–dessous et le faire évoluer en vérifiant si notre objectif pourrait être atteint.

La question qui se pose dans ce cas est : serait-il alors possible d'activer d_2 à partir de l'état initial $< a_0, b_0, c_0, d_0 >$?
On rappelle qu'une action peut être jouée si le frappeur et la cible correspondante sont à l'état actif dans un état donné ainsi le processus bond sera à l'état actif dans l'état suivant et la cible sera désactivée. Par exemple pour ce réseau il est possible de jouer l'action $a_0 \to c_0 \Rsh c_1$ et l'action $b_0 \to d_0 \Rsh d_1$. Il peut exister plusieurs évolutions possibles d'un réseau afin d'activer un/des process donné(s). Dans notre cas il s'avère qu'il existe deux

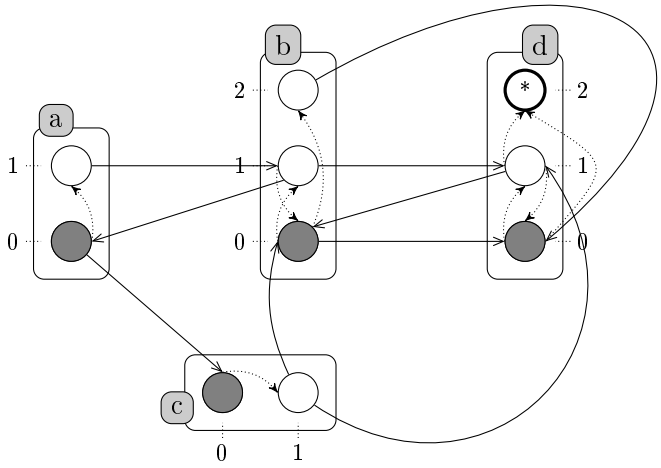

FIGURE 7 – Réseau PH ayant 4 sortes, l'état initial est $< a_0, b_0, c_0, d_0 >$, et le processus d_2 est l'objectif

chemins possibles qui peuvent activer d_2 et nous détaillons l'un des deux à la figure 8.

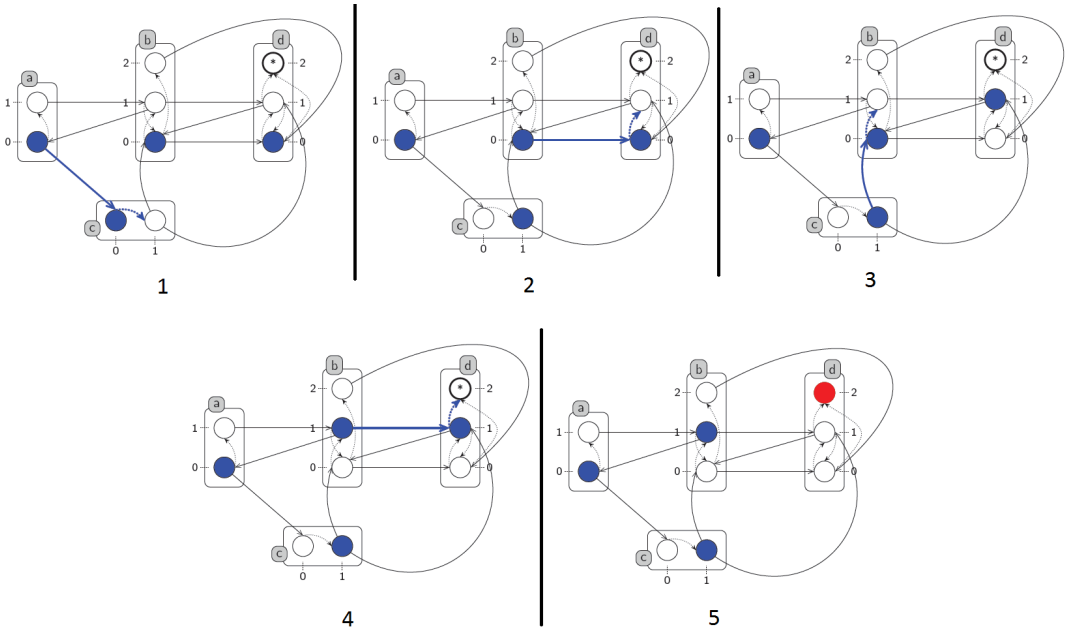

FIGURE 8 – Exemple d'évolution du réseau de la figure 7 vers le processus objectif d_2 aprés 4 changements. La successions des actions à partir de l'état initial $< a_0, b_0, c_0, d_0 >$ est : $a_0 \rightarrow c_0 \Rsh c_1 : : b_0 \rightarrow d_0 \Rsh d_1 : : c_1 \rightarrow b_0 \Rsh b_1 : : b_1 \rightarrow d_1 \Rsh d_2$. Ainsi l'état final $< a_0, b_1, c_1, d_2 >$ vérifie l'activation de d_2

2 Implémentation du point fixe en ASP

En plus de la partie bibliographique sur les deux *frameworks* cités ci-dessus, la deuxième partie du sujet de mon stage de master consiste à implémenter le PH ainsi que la vérification de certaines propriétés en ASP. Cette implémentation nécessite au début la traduction du réseau biologique avec tous ces composants (sortes, processus et actions) en ASP puis la recherche de la propriété statique : le point fixe. Ensuite une étude d'évolution du graphe dans le but de vérifier les propriétés dynamique : l'atteignabilité des processus cibles.

2.1 Représentation du réseau de régulation biologique

Dans le but de manipuler un réseau biologique écrit en PH, il fallait donc tout d'abord le présenter en ASP. Pour ce faire nous avons choisi les prédicats : *sort*, *process* et *action*. Ci-dessous un exemple de réseau PH présenté en ASP.

Exemple 3 (Exemple du point fixe)**.** Si on essaye de présenter le réseau de la figure 5 nous aurons :

```
1 sort("a"). sort("b"). sort("z").
2 process("a", 0..1). process("b", 0..1). process("z", 0..2).
3 action("a",0,"b",1,0). action("a",1,"a",1,0). action("b",1,"z",0,2).
4 action("b",0,"z",1,2). action("z",0,"a",0,1).
```

La ligne 1 présente chaque sorte du réseau avec le prédicat *sort* et entre parenthèses le nom de la sorte. Dans la ligne 2 on trouve la liste des processus correspondant à chaque sorte par exemple la sorte "z" a 3 processus numérotés de 0 à 2, cette numérotation est assurée par le 2^{eme} paramètre du prédicat $process("z", 0..2)$. Et enfin toutes les actions du réseau qui sont définies aux lignes 3 et 4, par exemple la première action $action("a", 0, "b", 1, 0)$ est une action de a_0 à b_1 pour faire bondir la sorte b vers b_0.

Cette génération d'un réseau biologique quelconque écrit en PH vers l'ASP se fait par l'intermédiaire d'un outil qui s'appelle PINT [1]. PINT est une bibliothèque écrite en OCaml par un ancien doctorant de l'équipe, Loïc Paulevé, durant sa thèse. En effet PINT est l'abréviation de Paulevé, PINT, Process HittINg related Tools. Elle permet de lire des modèles au format Process Hitting, et de réaliser sur ceux-ci certaines opérations d'analyse développées formellement.

Le logiciel PINT implémentant la spécification discrète, stochastique et temporelle des Process Hitting, ainsi que des analyses statiques. PINT apporte notamment :
– un langage de spécification pour les Frappes de Processus supportant le paramétrage stochastique et temporel ;
– un simulateur non-markovien implémentant la machine abstraite ;
– un énumérateur des points fixes ;
– un vérificateur statique de propriétés d'atteignabilité ;
– des outils permettant l'importation et l'exportation des Frappes de Processus pour divers outils existants.

La commande qui permet de traduire un fichier d'extension *ph* (reseau.ph) en un autre fichier d'extension *asp* ou *lp* (reseau.lp) manipulable en ASP est la suivante :

1. Pint est une bibliothéque disponible en ligne : `http://loicpauleve.name/pint/`

```
pint ph2asp.ml reseau.ph > reseau.lp
```

Le script *ph2asp.ml* écrit en *OCAML* a été développé par l'un de mes encadreurs et qui est un doctorant dans l'équipe MeForBio. J'ai adapté ce dernier à la présentation du PH que j'ai choisie (lignes de 1 à 4). Ainsi, tout réseau écrit en PH pourra être traduit en ASP automatiquement et rapidement (quelques millisecondes) et ce qui est très utile vu le nombre important de composants d'un réseau biologique (de l'ordre de cent).

2.2 La recherche du point fixe

La première idée était d'essayer de mettre en œuvre un algorithme qui vérifie intégralement le théorème 1 qui définit un point fixe en tant qu'un $|C| - clique$ dans un graphe sans-frappe. La première étape est alors la transformation du graphe avec des frappes (graphe orienté) à un graphe sans-frappes (graphe non-orienté). Elle consiste alors à commencer par éliminer les process ayant un auto-frappe notés en ASP par le prédicat "$hiddenProcess(A, I)$". Cela permet alors d'avoir une liste restreinte des process qui peuvent être traités, qui sont notés alors en ASP par : "$shownProcess(A, I)$" tels que :

```
5  hiddenProcess(A,I) ← action(A,I,B,J,K), A=B ,process(A,I), process(B,J), process(B,K).
6  shownProcess(A,I) ←not hiddenProcess(A,I), process(A,I).
```

Ensuite, il faut construire le nouveau graphe (graphe sans-frappe) avec des nouveaux arcs non orientés. Chaque arc relie obligatoirement deux processus de différentes sortes et il correspond à la non existence d'une frappe entre ces deux dernières dans le graphe initial (en PH). Dans le programme ASP ces arcs sont notés par noAction (B, J, A, I) dans la ligne 9 avec A et B des noms de sorts et I et J les indices de process :

```
7  hit (A,I,B,J) ← action(A,I,B,J,B,K).
8  noAction(B, J, A, I) ← not hit(A, I, B, J), not hit(B, J, A, I), A!=B, shownProcess(A, I),
9                          shownProcess(B, J).
```

Maintenant il faut parcourir ce nouveau graphe et extraire toutes les combinaisons possibles des shownProcess en sélectionnant un et un seul processus de chaque sorte. La règle 10, est une règle avec des contraintes de cardinalités (définition de telle règle dans la section 1.1.2 equation (7)). Elle a une borne inférieure qui est égale à 1 et une borne supérieur qui est égale à 1 exprimant bien la condition "un et un seul".

```
10  1 { selectProcess (A, I) : shownProcess(A, I)} 1 ← sort (A).
```

Cette règle 10 crée plusieurs *answer sets* (ensemble de réponses) dont chacun contient un seul atome "$selectProcess(A, I)$" pour chaque sorte A. Mais il reste à vérifier lesquelles de ces combinaisons satisfont les propriétés du point fixe autrement dit vérifier si dans cette combinaison chaque processus sélectionné est en relation avec tous les autres processus sélectionnés. Cette relation n'est qu'un arc ; ainsi tous deux processus sélectionnés doivent avoir un arc entre eux (équivalent à dire qu'il n'ont pas une action entre eux dans le graphe du PH). Nous avons alors défini le prédicat "$noExistFixPoint$" qui teste cette condition et élimine dans une contrainte les réponses non satisfaisantes :

```
11  noHit(A,I,B,J) ←noAction(A,I,B,J), selectProcess(A, I), selectProcess(B, J).
12  noExistFixPoint ←0{noHit(A,I,B,J)}0, selectProcess(A, I), selectProcess(B, J) .
13   ←noExistFixPoint.
```

Les *noHit* de la ligne 11 constituent un sous-ensemble de *noAction* qui est définit pour chaque nouvelle combinaison liant uniquement les processus sélectionnés. Ces derniers sont

des candidats et peuvent construire un point fixe. Enfin la combinaison des candidats qui vérifie bien toutes les conditions, sera prise et affichée comme réponse :

14 fixProcess(A, I) ←selectProcess(A, I).

Exemple 4. Si nous appliquons la méthode de recherche ci-dessus (de la ligne 5 à la ligne 14) au réseau du Process Hitting de la figure 5 présenté dans un script indépendamment (de la ligne 1 à la ligne 4), notre programme en ASP retourne l'unique réponse suivante :

```
Answer 1 : fixProcess(a, 0), fixProcess(b, 0), fixProcess(z, 2)
```

2.3 Implémentation optimale de recherche du point fixe

Généralement les réseaux biologiques se composent d'une centaine de composants (ADN, gène, protéine...), par la suite leur traduction en PH puis en ASP aboutira à des fichiers de très grande taille ayant une centaine de prédicats de sortes, des processus et d'actions. Dans ce cas l'éxcution du script ASP de la recherche du point fixe(par la méthode expliquée ci-dessus) sera (très) lourde en exécution. L'idée était d'essayer d'optimiser ce script en diminuant le nombre de prédicats dans le but d'avoir une exécution plus rapide.

Il fallait alors réfléchir autrement à la recherche du point fixe et éviter d'appliquer intégralement la définition. Si on a toutes les combinaisons possibles des processus on peut voir autrement le problème et dire « Si pour une combinaison des processus il existe deux processus ayant une frappe entre eux alors cette combinaison est à éliminer ». En effet cette réflexion nous offre la possibilité de profiter de la force d'ASP. Dans ce cas il n'est plus nécessaire de passer par l'intermédiaire du graphe sans-frappes et nous éliminerons ainsi les prédicats suivantes :

- noAction(A,I,B,J) arc entre deux processus dans le graphe sans-frappe et qui n'ont pas une action entre eux dans le graphe du PH.
- noHit(A,I,B,J) entre deux processus sélectionnés pour une combinaison donnée.
- noExistFixPoint est vrai si la combinaison ne forme pas un point fixe.

Tous ces prédicats sont supprimés et toutes les règles de la ligne 5 à la ligne 14 sont remplacées par une seule ligne qui est la contrainte suivante :

15 ←{hit(A,I,B,J)}, selectProcess(A, I), selectProcess(B, J), A!= B.

On rappelle qu'une contrainte est une règle avec la tête vide c'est-à-dire : si les prédicats dans le corps sont vrais alors cette réponse n'est pas une solution du problème et ne doit pas être retournée comme résultat. Dans notre cas si nous avons dans un ensemble de candidats (combinaison de processus) deux processus sélectionnés qui ont une action entre eux (hit(A,I,B,J)) alors cette combinaison n'est pas un point fixe, donc elle n'est pas une solution et est éliminée des ensembles des réponses grâce à cette contrainte 15.

Les résultats prouvent que la deuxième méthode est plus optimale en terme de temps d'exécution comme on le verra dans la section suivante, celle des résultats sur des modèles de réseaux biologiques, des exemples d'exécutions avec une comparaison entre ces deux méthodes. En effet le gain est de l'ordre de 50 fois plus rapide.

2.4 Résultats et discussion

Les deux méthodes développées pour la recherche du point fixe (L'implémentation directe du théorème et la définition de N-clique et la nouvelle vision de recherche) donnent les mêmes résultats mais non pas dans les même délais. En effet dans la deuxième méthode nous avons éliminé le calcul de plusieurs prédicats d'où le gain du temps de l'exécution. Le tableau suivant décrit la comparaison entre ces deux méthodes ainsi qu'une comparaison avec la bibliothèque PINT appliqués sur des modèles [2] de réseaux biologiques.

Model	#sorts	#procs	#actions	#etats	#PF	mthd1	mthd2	PINT
mvbrn	3	7	9	12	1	0.000	0.000s	0.006s
ERBB [13]	42	152	399	2^{70}	3	0.220s	0.000s	0.017s
tcrsig40 [7]	54	156	305	2^{73}	1	0.220s	0.020s	0.021s
tcrsig94 [12]	133	488	1124	2^{194}	0	2.540s	0.060s	0.027s
egfr104 [14]	193	748	2356	2^{320}	0	8.220s	0.140s	0.074s

FIGURE 9 – Comparaison des temps d'exécution sur les études de cas entre les 2 méthodes d'ASP et PINT appliquées sur des modèles de réseaux biologiques sur un ordinateur de bureau classique (processeur core i5 avec 4GB de mémoire vive). Pour chaque modéle on trouve le nombre de ses sorts (#sorts), le nombre de ses process (#procs), le nombre de ses actions (#actions), le nombre de ses états (#etats) et le nombre des points fixes trouvés (#PF). Puis le temps d'éxecution mis pour l'exécution de chacune des méthodes en ASP et celle existante en PINT avec (mthd1) correspond à la 1^{ere} qui implémente directement la définition d'un point fixe (section 2.2) et (mthd2) à celle plus optimale (section 2.3).

Le tableau 2.4 résume les temps d'exécution obtenus lors de l'analyse de plusieurs modèles à l'aide des méthodes présentées dans cette section sur un ordinateur portable classique (processeur core i5 avec 4GB de mémoire vive). Concernant les 2 méthodes en ASP, comme prévu, il est bien clair que la 2^{eme} fournit des résultats plus rapidement avec un rapport de l'ordre de 50 fois. Ce qui justifie que nous garderons la deuxième. Par contre les résultats montrent que cette méthode retenue n'est pas toujours plus performante que PINT. En effet on remarque qu'elle l'est pour des modèles ayant un nombre de sortes à l'ordre de 60 ou inférieur, mais pour les gros modèles PINT est plus rapide avec une différence de quelques millisecondes (de l'ordre de 2 fois plus rapide). Nous rappelons que nos méthodes traitent les réseaux traduit en ASP et PINT ceux écrit en PH et que la méthode de PINT se base sur du SAT qui traite les problèmes de satisfiabilité booléenne. Comparant avec ASP qui est un langage plus familier en écriture, SAT est plus complexe à représenter mais il est plus puissant en terme de rapidité.

2. Ces modéles sont disponibles en ligne sous format PH : `http://loicpauleve.name/pint/`

3 Implémentation de l'atteignabilité en ASP

Dans cette partie nous présentons comment il est possible, à partir d'un état initial connu, d'estimer les évolutions possibles grâce aux actions. Puis nous répondons à la question d'atteignabilité : « Est-il possible d'avoir des processus cibles à l'état actif à partir d'un état connu du réseau ?». L'étude de cette propriété dynamique du PH en ASP nous a conduit à avoir deux méthodes. La première est implémentée en ASP (CLINGO) et la deuxième qui est plus rapide en temps de réponse implémentée en ASP itératif (ICLINGO).

3.1 Calcul de la dynamique du réseau

A partir d'un état initial connu et après un nombre fini d'étapes un réseau PH peut évoluer vers plusieurs nouveaux états. Ma tâche était alors de détecter en utilisant ASP si ces évolutions pourraient avoir lieu.

Dans le cas d'un nombre d'étapes fini et connu, nous avons proposé un prédicat "$time(0..n)$" avec n le nombre total des étapes. Ainsi le réseau évolue vers la n^{eme} étape et les réponses seront toutes les possibilités d'évolution après n étapes. Par exemple si un biologiste cherche à savoir quels sont les états accessibles de son réseau après 10 étapes il n'a qu'à remplacer n par 10 :

```
16  time(0..10).
```

Abordons maintenant le problème d'initialisation. Pour cela nous avons déclaré le prédicat "$init$" dans la ligne (17) qui signifie que la sorte a est initialisée au niveau 0 (le processus a_0 est à l'état actif). Cette initialisation se fait automatiquement lors de la génération d'un nouveau réseau en ASP à partir du PH correspondant et toutes les sortes sont prises par défaut au niveau 0. Des modifications de niveaux pourront être faites directement sur le code si c'est nécessaire.

```
17  init(activeProcess("a",0)).
```

La dynamique du réseau est réalisée uniquement par les actions car ce sont les actions qui assurent le changement de niveau de la sorte. Rappelons qu'une action ne peut être jouée que si ses processus, frappeur et cible, sont à l'état actif. Ainsi, pour connaitre quelles sont les actions jouables à un état quelconque il faudra vérifier cette condition par le prédicat "$playableAction$" défini dans les lignes 18 et 19. On trouve aussi deux nouveaux prédicats "$inState$" et "$activeProcess$" dans "$inState(activeProcess(A,I),T)$". Ce dernier indique qu'à l'étape T, le processus A_I est à l'état actif. A chaque étape, les processus actifs sont calculés en fonction des nouveaux changements du graphe (voir les lignes 22 et 23). Sachant que le nombre des étapes a été déjà borné par N par le prédicat "$time(0..N)$", alors quand nous écrivons "$time(T)$" nous visons à dire que le numéro de l'étape courante doit être compris entre 0 et N $(0 \geq T \leq N)$).

```
18  playableAction(A,I,B,J,K, T) ←action(A,I,B,J,K, T), inState(activeProcess(A,I), T),
19                                  instate(activeProcess(B,J),T), time(T).
```

Après avoir identifié les actions jouables à l'instant courant T, il reste à en déduire l'ensemble des changements possibles par cette dynamique dans le prédicat $activeFromTo(B,J,K,T)$ (ligne 20). Nous avons ajouté les accolades dans la tête de la règle, autour du prédicat $activeFromTo(B,J,K,T)$, afin que la règle se transforme en une règle avec cardinalité mais sans contraintes des bornes (borne minimale = 0 et borne maximale = infinie). Cette règle nous permet d'avoir enfin un ensemble du prédicat concerné

avec différents arguments mais au même instant T, autrement dit l'ensemble de tous les changements possibles pour chaque instant T.

```
20  {activeFromTo(B,J,K, T)} ←playableAction(A,I,B,J,K, T), J!=K, time(T).
```

Sachant que la dynamique du Process Hittnig impose qu'au plus un seul changement entre deux états successifs ne pourra se produire, nous avons alors ajouté une règle (une contrainte) avec une contrainte de cardinalité (ligne 21) pour la borne minimale du nombre des "$activeFromTo(B, J, K, T)$" (égale à 2). En effet si une solution offre 2 changements ou plus elle va être éliminée et ne sera pas affichée comme réponse.

```
21  ←2{activeFromTo(B,J,K, T)}, time(T).
```

Grâce à "$activeFromTo(B, J, K, T)$" nous avons à chaque étape T le nouveau processus qui sera actif à l'étape suivante $(T + 1)$ qui est B_K. Ce nouveau processus actif sert après à trouver le nouvel état du réseau. En effet tous les processus qui étaient à l'état actif en T restent identiques en $T + 1$ (règles 23 et 24) à l'exception du processus B_K (règle 22).

```
22  instate(activeProcess(B,K), T+1) ←activeFromTo(B,J,K, T), time(T).
23  instate(activeProcess(A,I),  T+1) ←instate(activeProcess(A,I),  T), activeFromTo(B,J,K, T),
24                                          A!=B, time(T).
```

De cette façon que nous étudions l'évolution des réseaux alors d'un état à un autre jusqu'à atteindre le N^{eme} et dernier état (l'entier N indique le nombre maximal d'étapes, il est fixé dés le début dans le prédicat $time(0..N)$).

Exemple 5. Nous testons notre implémentation pour connaitre les évolutions possible du réseau de la figure 7 dans la section 1.2.5. Nous avons choisi 4 comme étant le nombre des étapes maximum, et $< a_0, b_0, c_0, d_0 >$ l'état initial du réseau. Le résultat affiché contenait 54 réponses possibles parmi lesquelles on cite :

```
Answer: 1 activeFromTo("d",0,1,0) activeFromTo("c",0,1,1) activeFromTo("b",0,2,2)
Answer: 2 activeFromTo("d",0,1,0) activeFromTo("c",0,1,1) activeFromTo("b",0,1,2)
Answer: 3 activeFromTo("d",0,1,0) activeFromTo("b",0,2,1)
Answer: 4 activeFromTo("d",0,1,0) activeFromTo("c",0,1,1) activeFromTo("d",1,0,2)
activeFromTo("b",0,1,3)
```

etc...

Dans la suite nous étudions la propriété d'atteignabilité, alors il ne sera plus intéressant de savoir toutes les évolutions possibles du réseau mais plutôt seulement celles qui vérifient l'objectif.

3.2 La vérification de l'atteignabilité

Dans cette section nous nous concentrons sur le problème de l'accessibilité qui correspond à la question suivante : « Est-il possible, à partir d'un état initial donné, de jouer un certain nombre d'actions afin d'avoir comme résultat des processus donnés actifs ? »

L'idée est alors d'adapter le code de l'évolution dynamique du réseau donné à la section précédente afin de résoudre cette problématique d'atteignabilité. Pour commencer il faut déclarer un prédicat qui définit nos objectifs (l'ensemble des processus à l'état actif) :

```
25  goal(activeProcess("a",  1)).
```

La règle 25 donne un exemple d'un objectif à activer : le process a_1 de la sorte a dans le prédicat "$goal$". On peut définir aussi bien plusieurs processus à activer et non pas un seul tel que chacun soit dans une règle indépendante.
Soit $< a_1, b_3, z_0 >$ un exemple d'un état objectif pour le réseau de la figure 5 , section 1.2.2. Il est traduit en ASP par :

```
26 goal(activeProcess("a", 1)).
27 goal(activeProcess("b", 3)).
28 goal(activeProcess("z", 0)).
```

Pour faciliter l'utilisation de nos scripts, nous avons ajouté ce prédicat responsable de la définition des objectifs "*goal*", dans le script qui permet la traduction du PH en ASP (`ph2asp.ml`). Le prédicat sera généré pour chaque sorte du réseau en question, et il sera mis en commentaire (précédé par "%") avec un niveau ou un processus cible inconnu noté par "xx" :

```
29 \% goal(activeProcess("a", xx )).
30 \% goal(activeProcess("b", xx )).
31 \% goal(activeProcess("c", xx )).
```

Lors de la définition des objectifs, l'utilisateur doit enlever le "%" pour chaque sorte cible puis remplacer le "xx" par le niveau qu'il cherche à activer pour la sort concernée. Enfin il aura des ligne de codes comme celles écrites de 26 à 28 définissant ses *goals*.

Après avoir fixé les objectifs il faut vérifier si le réseau pourrait évoluer dans un sens offrant un état qui satisfait tout les objectifs. Nous avons dans la règle 32 un nouveau prédicat "$satisfiable$" qui est vrai au même instant T au bout duquel tous les objectifs sont atteints. Toute réponse ne satisfait pas les *goals* sera éliminée des ensembles des réponses par la contrainte définie dans la ligne 33.

```
32  satisfiable (F, T) ←goal(F),  instate(F, T).
33 ←not satisfiable (F, T).
```

Cette méthode retourne des résultats corrects pour les cas ou le N (nombre maximal des étapes) choisi est suffisant. En effet l'utilisateur doit prévoir le nombre des étapes pour que le réseau puisse atteindre ses objectifs et ce nombre N, défini dans "$time(0..N)$", doit être alors supérieur ou égal au numéro de l'étape qui vérifie le problème. Si non la réponse affichée sera insatisfait (`Unsatisfiable`) alors qu'en réalité c'est satisfiable à l'étape $N + 1$.

Exemple 6. Si nous prenons à titre d'exemple simple la figure 8, on remarque que notre objectif n'est atteint qu'aprés 4 changements (étape 3). Si nous définissons pour ce réseau le nombre maximal d'étapes pour l'évolution ($N = 2$), le résultat retourné est `Unsatisfiable` alors que par contre il est réellement satisfiable si $N \geq 3$

Il fallait une alternative pour résoudre ce problème et qu'il soit plus optimal. L'idée était alors d'éliminer le prédicat "$time(0..N)$" et de le remplacer par un mode de calcul incrémental. Effectivement ICLINGO introduit dans [4] est un mode de calcul en ASP avec une variable cumulative qui s'incrémente automatiquement jusqu'à atteindre une condition d'arrêt.

3.3 Résolution de l'atteignabilité en mode d'incrémentation

D'après le guide de l'utilisateur [4], le système ICLINGO s'étend de CLINGO par un mode de calcul avec incrémentation automatique qui intègre à la fois la stabilisation et la résolution. Par conséquent, sa langue d'entrée comprend toutes les constructions décrites tout au long de ce rapport.
ICLINGO comporte de plus les directives suivantes :

```
#base.
... (du code) ...
#cumulative constante.
... (du code) ...
#volatile constante.
... (du code) ...
```

#cumulative constante. et *#volatile constante.* sont utilisés pour déclarer une constante (symbolique) comme un symbol qui sera remplacé par les numéros d'étapes incrémentales. Notre constante est notée par "t" dans la suite indiquant le numéro d'une étape lors de l'évolution du réseau.
En définissant *#base.* au début, la suite du programme logique déclaré avant *#cumulative* sont considérés comme statique, c'est à dire les règles de cette partie sont traitées une seule fois au début d'un calcul incrémental (typiquement les données et les règles qui ne dépendent pas de l'étape en cours)
En revanche, les lignes du programme entre *#cumulative t.* et *#volatile t.* constituent la partie dans laquelle la constante "t" est en mode *cumulative.* Ces lignes sont traitées pour chaque valeur de "t" qui est remplacé dans toutes les règles par le numéro de l'étape courante. De plus, les règles résultantes, les faits et les contraintes d'intégrité sont accumulés sur l'ensemble du calcul incrémental. Pour notre problème cette partie traite chaque état du réseau à part, après chaque changement elle recalcule les prédicats nécessaires qui assurent l'évolution du système (tels les actions jouables par "*playableAction*").
La partie du programme logique au-dessous de *#volatile t* est locale à l'étape courante, c'est-à dire toutes les règles, les faits et les contraintes d'intégrité sont calculées en une seule étape puis rejetées avant l'étape suivante. Notons que dans cette partie le remplacement de la constante "t" est aussi similaire et que dans cette partie que nous définissons la condition d'arrêt du calcul incrémantal. Cette condition est l'aboutissement à l'état vérifiant les objectifs.
Dans la suite nous fournissons un exemple type dans lequel ces conditions tiennent naturellement et avec lequel nous résolvons notre problématique d'accessibilité.

L'initialisation (à $t = 0$) du réseau pour ce programme incrémental se fait dans la partie du code sous "*#base*" puisque cette phase est fixe :

```
34  instate(F,0) ←init(F).
```

Nous avons à peu près le même programme qui se répète à chaque étape mais avec des nouvelles données. En effet à chaque instant (ou étape) "t", le programme calcule les actions jouables "$playableAction(A, I, B, J, K, t)$", le changement possible "$activeFromTo(B, J, K, t + 1)$" et les processus actifs du prochain état "$instate(activeprocess(A, I), t + 1)$". Ce calcul répétitif se fait dans la partie en mode cumulatif au dessous de "*#cumulative t*", les mêmes règles des lignes de 18 à 24 seront copiées en remplaçant T par t. La condition d'arrêt pour ce calcul incrémental est écrite après *#volatile t* dans le prédicat *notSatisfiable* puis mise dans une contrainte (37) :

```
35  # volatile t.
36  notSatisfiable(t) ←goal(F), not instate(F, t).
37  ←notSatisfiable(t).
```

Effectivement, cette contrainte (37) présente la condition d'arrêt : elle contrainte l'arrêt du calcul et l'incrémentation de "t" tant qu'elle est vraie. En plus elle sert aussi à éliminer les réponses qui ne satisfont pas les objectifs et à n'afficher comme résultat que celles qui les satisfont.

Exemple 7. Considérons le même exemple de la figure 7 dans la section 1.2.5.
Son état initial est $< a_0, b_0, c_0, d_0 >$ et on cherche à ce que le processus d_2 soit à l'état actif. Ainsi notre objectif sera écrit en ASP comme suit :

```
goal(activeProcess("d",2)).
```

PINT retourne simplement "Oui"; par contre notre implémentation en ASP retourne aussi l'un des chemins d'activation (On peut les afficher tous aussi).
Dans cet exemple il y a eu 4 changements (figure 8) donc 4 étapes (0, 1, 2 et 3) tout au long du chemin pour avoir à la 4_{eme} étape le processus objectif d_2 actif :

```
Answer 1 : activeFromTo(c,0,1,0), activeFromTo(d,0,1,1), activeFromTo(b,0,1,2), activeFromTo(d,1,2,3).
```

Cette réponse signifie qu'il y a eu une succession d'actions qui ont évolué le réseau d'un état à un autre. La succession correspondante à cet exemple et cette réponse (Answer 1) est alors :

$< a_0, b_0, c_0, d_0 > \mapsto < a_0, b_0, c_1, d_0 > \mapsto < a_0, b_0, c_1, d_1 > \mapsto < a_0, b_1, c_1, d_1 > \mapsto < a_0, b_1, c_1, d_2 >$

On rappelle que le prédicat "$activeFromTo(c, 0, 1, 0)$" signifie que la sorte c a changé son niveau de 0 à 1 (bondir de c_0 à c_1) à la première étape 0. La réponse décrit bien le même chemin de la figure 8. En effet il y a changement du niveau des sorts comme suit ; à l'instant 0 la sorte c passe de 0 en 1 ($activeFromTo("c", 0, 1, 0)$), à l'instant 1 la sorte d passe de 0 en 1 ($activeFromTo("d", 0, 1, 1)$), à l'instant 2 la sorte b passe de 0 en 1 ($activeFromTo("b", 0, 1, 2)$) et à l'instant 3 la sorte d passe de 1 en 2 ($activeFromTo("d", 1, 2, 3)$). Effectivement notre objectif est atteint, d_2 est activé et le calcul est arrêté à l'état $< a_0, c_1, b_1, d_2 >$.

Traitement des boucles : Parfois, dans les cas où les objectifs ne sont pas atteints et que lors de la recherche le programme rencontre des boucles, le programme risque de tourner indéfiniment dans l'une de ces boucles puisque ce n'est pas possible ni d'atteindre les objectifs ni d'arrêter le calcul. Nous avons proposé alors dans ces cas de limiter l'incrémentation à un nombre maximal qui peut être atteint dans le cas des boucles infinies et qui peut ne pas l'être dans le cas où les objectifs sont vérifiés.

```
iClingo <nom_réseau>.lp <nom_fichier>.lp -imax=n
```

Il faudrait donner des valeurs intéressantes pour cet entier n qui fixe le nombre maximal des étapes. Par exemple, le nombre total d'états est un maximum possible (il ne sera jamais dépassé par un chemin minimal). On peut cependant réduire au nombre de sortes si on suppose qu'un seule sorte sera modifiée par étape (ce qui est souvent le cas des réseaux booléens) ou au nombre d'actions avec la même hypothèse.
Si au cours d'un chemin l'objectif est atteint au bout d'un instant $m \leq n$, le calcul s'arrête et affiche ce chemin comme solution.

3.4 Résultats et discussion

Il convient de noter que la seule analyse d'atteingnabilité précédemment développée sur le formalisme du Process Hitting a été mise en œuvre dans le logiciel PINT et qu'il s'agit d'une abstraction [10]. L'analyse d'atteignabilité de ce logiciel termine toujours mais peut être parfois non concluante même si cela arrive rarement. En outre, il ne donne pas le chemin grâce auquel on peut activer nos processus objectifs. Il répond par "Oui" ("True") dans le cas où il pourrait être atteint et par "Non" ("False") si non et "Non-conclusif" ("Inconc") si l'abstraction est trop large pour pouvoir conclure. Notre nouvelle méthode donne comme réponse les différents chemins qui peuvent aboutir à l'activation des objectifs, c'est à dire les processus à activer successivement jusqu'à avoir un état du réseau dans-lequel les processus concernés sont à l'état actif. De plus notre méthode est conclusive ("Oui" ou "Non") dans tous les cas où elle termine.
Nous avons testé notre implémentation sur des exemples pris des articles dans le domaine biologique.

- **Exemple des modèles biologiques** Si nous prenons l'exemple ERBB ayant 42 sortes [3], nous initialisons toutes ses sortes au niveau 0 à l'exception de 5 sortes (EGF en 1, __coop0 en 3, __coop1 en 3, __coop3 en 1 and __coop4 en 1). Puis nous fixons notre objectif en cherchant que la sorte pRB soit au niveau 1 $goal(activeProcess("pRB", 1))$., `pint` retournera la réponse "True", notre implémentation en ASP, retourne de plus le chemin suivant :

  activeFromTo("ERBB1",0,1,0) activeFromTo("ERBB1ERBB1_2ERBB1_3",0,4,1)
  activeFromTo("__coop1",3,1,2) ... activeFromTo("CDK2CDK4CDK6",0,2,15)
  activeFromTo("CDK2CDK4CDK6",2,3,16) activeFromTo("pRB",0,1,17)).

3. Disponible en ligne à : `http://loicpauleve.name/pint/`

Model	#sorts	#procs	#actions	#etats	#etapes	ASP	ASPi	PINT
figure 7	4	10	9	36	4	0.000s	0.000s	0.000s
ERBB [14]	42	152	399	2^{70}	18	10.620s	5.020s	0.022s
tcrsig40 [7]	54	156	305	2^{73}	26	156.500s	127.250s	0.012s

FIGURE 10 – Comparaison des temps d'exécution sur les études de cas entre les 2 méthodes d'ASP (CLINGO et ICLINGO) et PINT appliquées sur des modèles de réseaux biologiques sur un ordinateur de bureau classique (processeur core i5 avec 4GB de mémoire vive). Pour chaque modéle on trouve le nombre de ses sorts (#sorts), le nombre de ses process (#procs), le nombre de ses actions (#actions), le nombre de ses états (#etats). Puis le temps d'éxecution mis pour l'éxecution de chacune des méthodes en ASP et celle existante en PINT.

De même pour l'exemple du réseau biologique *tcrsig40* ayant 54 sorts, nous initialisons toutes ses sortes à 0 (comme généré automatiquement lors de la traduction du PH en ASP) et les 3 sorts suivants initialisés à 1 : TCRlig, CD8 et CD45. Notre objectif est que le sort NFAT soit au niveau 1.

Nous concluons avec la comparaison du tableau suivant qui montre bien que la méthode itérative en ASP est moins efficace en terme de rapidité que PINT, en quelques secondes le résultat est affiché pour un gros réseau. Par contre elle est exacte et donne les chemins d'exécution (d'activation des goals). Il apparait que la méthode en ASP itérative est plus optimale que celle en ASP (standard) non seulement pour la rapidité mais aussi pour l'utilisation (pas besoin d'estimer le nombre d'étapes).

- **Comparaison avec la méthode de Rocca et al. [11] (simulation de model checking temporel en ASP)**

  Rocca et al. ont développé une méthode en ASP qui vérifie les propriétés CTL par *model cheking* (AF, EF, AG, EG...) pour les graphes d'états de transition. Puisque notre méthode et celle de Rocca et al. sont développées en ASP, nous avons proposé alors de comparer la propriété EF.
  Il fallait effectuer le test sur un exemple réel et qui n'est pas de grande taille, vu que la méthode de Rocca doit être testée sur des réseaux de graphe de transition. L'exemple de résorption de la queue du têtard pris de [6], ayant 12 sortes, 42 processus, 139 actions, 524.288 états est le candidat pour cette comparaison.
  A partir de l'état initial I_0 nous vérifions la propriété `prop1 = EF(`I_0`,` *goal*`)` avec I_0 un état tel que les sortes ci dessous sont initialisées ainsi :
  $(T4 = 0 \wedge T3 = 0 \wedge D2 = 0 \wedge D3 = 1 \wedge GI = 0 \wedge GP = 0 \wedge TR = 0 \wedge GT = 0)$
  et *goal* vérifie un état ayant les sortes dans les niveaux suivants :
  $(T3 = 3 \wedge T4 = 1 \wedge D2 = 1 \wedge D3 = 0 \wedge GI = 1 \wedge GP = 1 \wedge TR = 1 \wedge GT = 0)$.
  La traduction de ce réseau en graphe de transition écrit en ASP a pris **3 min 6 s** puis le résultat pour la vérification de la prop1 a été affiché après **7 min 17 s** avec la méthode de Roccaet al. Par contre notre méthode donne un résultat au bout de **1.9s** avec une traduction instantanée du modèle du PH en ASP.
  La méthode du *modèle checking* retourne les états à partir desquels la propriété sera vérifiée, tandis que notre méthode elle donne le chemin pour activer les objectifs à partir d'un état initial connu. De ce fait, on peut ne pas seulement conclure que notre nouvelle implémentation est plus efficace en terme de temps de réponse, mais que les deux méthodes pourront être complémentaires. En effet, notre méthode nécessite un état initial qui est donné par la méthode de Rocca et al. De plus, la méthode de Rocca et al. traite tous les prédicats de la logique temporelle, tandis que nous ne traitons que les prédicats de la forme EF.

4 Conclusion et perspectives

Nous avons montré dans cette thèse de Master une nouvelle analyse dynamique développée. Cette analyse est applicable à une classe de modèles dits PH et vise à déterminer des propriétés des réseaux modélisés.

La première propriété est la recherche des états stables du réseau qu'on appelle les points fixes. Ces points représentent les états du réseau pendant lesquels, le modèle ne peut plus évoluer. Il est intéressant à les connaitre car ils bloquent l'évolution des systèmes biologiques. Nous avons développé deux méthodes, sachant que la deuxième est plus optimale, qui retournent l'ensemble de tous les points fixes du réseau. Ce point fixe n'est qu'un état du réseau traduit en ASP par un niveau pour chaque composant, autrement un processus pour chaque sorte.
La deuxième propriété est une propriété qui se base sur la dynamique du réseau : l'atteignabilité. Un réseau biologique évolue dans plusieurs sens qui peuvent aboutir ou pas à des états-objectifs. Notre nouvelle approche retourne les chemins exactes aboutissant à atteindre un niveau cible d'un composant du système. Ce chemin se traduit par un ensemble de changements successives des niveaux. En PH cela se traduit par de changements successifs de processus et c'est ce que nos méthodes retournent comme résultat. Il s'avére que la méthode itérative en ASP est plus optimale que celle en ASP normal. En effet il n'est pas nécessaire de prévoir le nombre de changements pour la méthode itérative et elle retourne le résultat plus rapidement (quelques seconde pour des réseau moyennement grand).

Une comparaison a été faite par rapport à l'existant, PINT et la méthode de Rocca et al. Les résultats montrent que, par rapport à PINT, la méthode de recherche des points fixes est efficace, mais que pour l'accessibilité, elle l'est moins que prévu. Cependant aussi notre méthode retourne de plus le chemin d'atteignabilité. Elle offre la possibilité de poser des questions plus générales par rapport à PINT portant sur plusieurs sortes de plus.
Sachant que la présentation d'un réseau biologique en PH est simplifie le traitement et la traduction des modèles, il s'avère que notre méthode qui se base sur ce formalisme est plus efficace que d'autres méthodes développées en ASP aussi mais pour des réseau de graphes de transitions. Le cas de la méthode de Rocca qui est gourmande en temps par rapport à la notre, résultat retourné en des minutes contre un résultat affiché en quelques secondes.

Nous pensons que cette approche peut également être utilisée et adaptée avec d'autres modèles tels que le modèle de Thomas, les réseaux de Petri et les modèles synchrones. Cela nécessite une traduction propre au modèle étudié ainsi qu'un traitement approprié.
Parmi nos perspectives qui font partie de mon sujet de thèse, c'est d'essayer d'améliorer cette méthode en éliminant les cycles de la méthode itérative. Cela évite de tourner indéfiniment dans des boucles sans avoir un résultat affiché.

Ensuite, nous souhaitons étendre le programme pour chercher les attracteurs. Un attracteur est un ensemble d'états à partir desquels il n'est plus possible de sortir, et donc tel que le réseau tourne indéfiniment dans ces états. Le point fixe est un cas spécial des attracteur, en effet c'est un attracteur de dimension une. Par contre, la caractérisation des attracteurs de la dynamique,de dimension n, requiert une analyse des dynamiques possibles bien plus poussée que pour les points fixes.
Nous visons aussi à implémenter une recherche dynamique dans le sens inverse de l'atteignabilité et poser la question : "*Quels sont les états initiaux qui nous permettent d'atteindre nos objectifs ?*". La réponse à cette question est l'ensemble des états à partir des quels il existe des chemins qui activent le ou les objectif(s). C'est vrai que la méthode de Rocca et al. résolve cette problématique mais nous estimons à avoir une approche plus efficace en terme de temps et qui utilise le réseau en process hitting en non pas les graphes de transitions.

Tous ces problématiques constituent une perspective intéressante dans le cadre du développement de techniques d'analyse statique et dynamiques des propriétés du Process Hitting.

Références

[1] Emna Ben Abdallah. La programmation logique et l'answer set programming (ASP). Rapport bibliograhique, École Centrale de Nantes, 2014.

[2] Christian Anger, Kathrin Konczak, Thomas Linke, and Torsten Schaub. A glimpse of answer set programming. *KI*, 19(1) :12, 2005.

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

[4] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A user's guide to gringo, clasp, clingo, and iclingo, 2008.

[5] Robert Ibbotson. Extending and continuing the development of an integrated development environment for answer set programming. Master's thesis, University of Bath, 2010.

[6] Zohra Khalis, Jean-Paul Comet, Adrien Richard, and Gilles Bernot. The smbionet method for discovering models of gene regulatory networks. *Genes, Genomes and Genomics*, 3(1) :15–22, 2009.

[7] Steffen Klamt, Julio Saez-Rodriguez, Jonathan Lindquist, Luca Simeoni, and Ernst Gilles. A methodology for the structural and functional analysis of signaling and regulatory networks. *BMC Bioinformatics*, 7(1) :56, 2006.

[8] Vladimir Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1) :39–54, 2002.

[9] Loïc Paulevé. *Modélisation, Simulation et Vérification des Grands Réseaux de Régulation Biologique*. PhD thesis, École centrale de nantes, 2011.

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

[11] Alexandre Rocca, Nicolas Mobilia, Éric Fanchon, Tony Ribeiro, Laurent Trilling, and Katsumi Inoue. Asp for construction and validation of regulatory biological networks. In Luis Fariñas del Cerro and Katsumi Inoue, editors, *Logical Modeling of Biological Systems*, pages 167–206. Wiley-ISTE, 2014.

[12] Julio Saez-Rodriguez, Luca Simeoni, Jonathan A Lindquist, Rebecca Hemenway, Ursula Bommhardt, Boerge Arndt, Utz-Uwe Haus, Robert Weismantel, Ernst D Gilles, Steffen Klamt, and Burkhart Schraven. A logical model provides insights into t cell receptor signaling. *PLoS Computational Biology*, 3(8) :e163, 08 2007.

[13] Ozgur Sahin, Holger Frohlich, Christian Lobke, Ulrike Korf, Sara Burmester, Meher Majety, Jens Mattern, Ingo Schupp, Claudine Chaouiya, Denis Thieffry, Annemarie Poustka, Stefan Wiemann, Tim Beissbarth, and Dorit Arlt. Modeling ERBB receptor-regulated G1/S transition to find novel targets for de novo trastuzumab resistance. *BMC Systems Biology*, 3(1), 2009.

[14] Regina Samaga, Julio Saez-Rodriguez, Leonidas G Alexopoulos, Peter K. Sorger, and Steffen Klamt. The logic of egfr/erbb signaling : Theoretical properties and analysis of high-throughput data. *PLoS Computational Biology*, 5(8) :e1000438, 2009.

[15] Andrew Wuensche. Genomic regulation modeled as a network with basins of attraction. In R. B. Altman, A. K. Dunker, L. Hunter, and T. E. Klien, editors, *Pacific Symposium on Biocomputing*, volume 3, pages 89–102. World Scientific, 1998.