École Centrale de Nantes Université de Nantes École des Mines de Nantes

MASTER AUTOMATIQUE ET SYSTÈMES DE PRODUCTION

SPECIALITÉ SPIE

Année 2010 / 2011

Thèse de Master

Présentée et soutenue par :

MAXIME FOLSCHETTE

le 1er septembre 2011

à l'Institut de Recherche en Communications et Cybernétique de Nantes

TITRE

APPLICATION DE LA LOGIQUE DE HOARE AUX RESEAUX DE REGULATION GENETIQUE AVEC MULTIPLEXES

JURY

Président :	Jean-François LAFAY	Professeur
Examinateurs :	Olivier ROUX	Professeur
	Morgan MAGNIN	Maître de conférences
	Loïc PAULEVE	Doctorant

Directeur(s) de thèse : Olivier ROUX & Morgan MAGNIN

Laboratoire : Institut de Recherche en Communications et Cybernétique de Nantes – UMR CNRS 6597

N° B.U. :

Rapport de thèse de Master

Application de la logique de Hoare aux réseaux de régulation génétique avec multiplexes

Maxime Folschette

Version corrigée du 12 mars 2012

Remerciements :

à Olivier Roux et Morgan Magnin pour leur encadrement régulier et sérieux,

à Pierre Castéran et Yves Bertot pour leurs idées,

à Gilles Bernot, Jean-Paul Comet et Adrien Richard pour leurs retours,

à Loïc Paulevé pour ses conseils,

à mes anciens camarades d'option pour leur soutien ininterrompu.

Table des matières

Résumé

Parmi les nombreux domaines de la bio-informatique, l'étude des systèmes de gènes en interaction recourt au modèle de Thomas, qui se base sur l'utilisation des réseaux de régulation génétique. Cet outil prend en compte un certain nombre de paramètres biologiques afin de caractériser la dynamique d'un ensemble de gènes qui s'influencent mutuellement. Il est cependant sujet à l'explosion combinatoire, qui complique la recherche de paramètres adéquats pour des systèmes de grande taille. Pour pallier cela, il a été proposé d'appliquer une forme adaptée de la logique de Hoare afin d'inférer des paramètres sur ces réseaux. Le stage d'application relatif à ce sujet a consisté en l'implémentation de l'outil théorique présenté : deux approches ont été suivies pour répondre à cette problématique, et sont ainsi détaillées afin de comprendre leur fonctionnement et en saisir les différences. La première a consisté à utiliser les outils et le formalisme mathématiques de Coq. La seconde a consisté à utiliser OCaml afin de bénéficier de plus de souplesse. La discussion finale permettra de récapituler les avantages et les limites des deux approches.

Introduction

Cette thèse de Master a été réalisé à l'IRCCyN, au sein de l'équipe MeForBio [1]. Il s'inscrit dans une démarche d'application de méthodes et d'outils formels au domaine de la bio-informatique.

Contexte scientifique

La terme de *bio-informatique* est très générique : il inclut aujourd'hui tous les domaines de recherche utilisant les technologies de l'information dans le but d'étudier les systèmes biologiques [2]. Cela comprend des applications très variées, parmi lesquelles ont peut notamment citer le recensement du génome, le développement de médicaments ou encore l'étude des interactions entre gènes. C'est au sein de cette dernière discipline, dont le but est d'étudier des systèmes de gènes interagissant entre eux, que se situe le sujet du présent rapport. Plusieurs catégories d'outils ont vu le jour dans le but d'étudier ces systèmes, et nous nous intéresserons ici aux réseaux de régulation, qui en proposent une modélisation sous forme de graphes.

Déroulement du stage

Cette thèse se base sur un nouvel outil théorique permettant l'inférence de paramètres au sein des réseaux de régulation génétique à l'aide de la logique de Hoare. Afin d'en préciser les enjeux et le contexte, nous définirons et expliquerons les notions nécessaires à sa compréhension au chapitre 1. Le stage d'application de ce Master a consisté en l'implémentation de cette méthode, et la première piste suivie a été l'utilisation de l'assistant de preuve Coq, afin de profiter de la puissance de son formalisme ; son utilisation sera détaillée au chapitre 2. La seconde piste suivie a été l'utilisation du langage de programmation OCaml, qui a permis d'obtenir davantage de résultats grâce à une approche légèrement différente et à plus de souplesse. Cette seconde piste sera présentée au chapitre 3. Enfin, une discussion rappellera les raison du choix de ces deux outils et leurs limites, et introduira la conclusion.

1. Méthodes Formelles pour la Bio-informatique

Chapitre 1

Définitions et aspect théorique

Le présent travail a consisté en une application de la théorie développée dans [9]. Ce document propose d'utiliser la logique de Hoare pour effectuer de l'inférence de paramètres biologiques sur des réseaux de régulation génétique. Afin d'expliquer comment cette théorie a été mise en œuvre, il est au préalable nécessaire de rappeler les concepts qui la composent. Nous verrons tout d'abord dans ce chapitre comment sont définis les réseaux de régulation génétique avec multiplexes, puis nous étudierons les principes de la logique de Hoare et nous verrons comment elle s'applique au domaine des réseaux de régulation.

1.1 Réseaux de régulation génétique

Le modèle de Thomas permet l'étude des systèmes de gènes en interaction et repose sur l'utilisation d'un *réseau de régulation génétique*. Cet outil a été popularisé par René Thomas en 1973, puis complété en plusieurs occasions [11]. Il permet de représenter les interactions (de type activation ou inhibition) au sein d'un système de gènes, et de quantifier leur force. Mathématiquement, un système de gènes y est représenté sous la forme d'un graphe, appelé *graphe d'interaction*, où les nœuds représentent les gènes et les arcs leurs différentes interactions mutuelles. Afin d'améliorer l'expressivité des graphes d'interaction, la notion de multiplexe a été proposée dans [5], et sera réutilisée ici. Un multiplexe est un nœud particulier qui se trouve nécessairement sur le chemin d'influence liant un gène à un autre, et qui en précise les conditions d'interaction. La définition formelle du graphe d'interaction est la suivante :

Définition 1.1 (Graphe d'interaction) Un *graphe d'interaction avec multiplexes* est un quadruplet $G = (V; M; E_V; E_M)$ qui vérifie les propriétés suivantes :

- $(V \cup M; E_V \cup E_M)$ est un graphe orienté et étiqueté, dont les nœuds sont $V \cup M$ et les arcs sont $E_V \cup E_M$, vérifiant les contraintes suivantes :
  - les ensembles V et M sont finis et disjoints ; les éléments de V sont appelés *variables* et ceux de M sont appelés *multiplexes*,
  - les arcs de E_V ont pour source une variable et pour cible un multiplexe, tandis que les arcs de E_M ont pour source un multiplexe et pour cible une variable ou un autre multiplexe,
  - tout cycle de G contient au moins une variable,
- toute variable v de V est étiquetée par un entier positif b_v qui est appelé son *plafond*,

– tout arc de E_V est étiqueté par un entier positif s inférieur ou égal au plafond de sa variable source et appelé *seuil* ; on note un tel arc $v \xrightarrow{s} m$ où v est sa variable source et m est son multiplexe cible.
– tout multiplexe m de M est étiqueté par une formule φ_m appartenant au langage $\mathcal{L}_m$ défini inductivement par :
  – si $v \xrightarrow{s} m$ appartient à E_V alors $v \geq s$ est un atome de $\mathcal{L}_m$,
  – si $m' \to m$ appartient à E_M alors m' est un atome de $\mathcal{L}_m$,
  – si φ et ψ appartiennent à $\mathcal{L}_m$ alors $\neg\varphi$, $\varphi \wedge \psi$ et $\varphi \vee \psi$ sont des atomes de $\mathcal{L}_m$.

Un exemple de graphe d'interaction est donné à la figure 1.1. Cet exemple sera réutilisé plusieurs fois durant ce rapport afin d'illustrer les notions présentées.

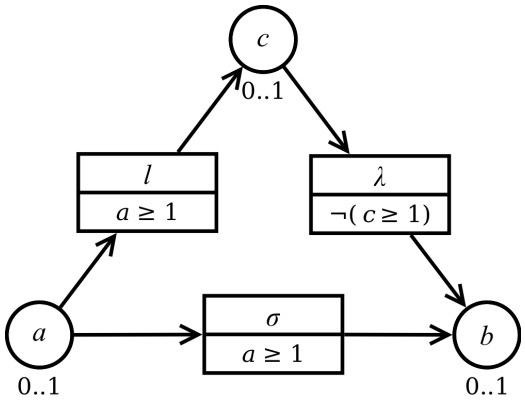


Figure 1.1 – Un exemple de graphe d'interaction simple [9] : les nœuds modélisent les gènes et les arcs leurs influences mutuelles. Ici, le gène a possède une influence sur les deux autres gènes, tandis que c n'influe que sur b. Les nœuds rectangulaires représentent les multiplexes tandis que les nœuds circulaires représentent les variables.

Chacune des variables présentes dans un graphe d'interaction peut être activée à un certain degré. Cette activité à un instant donné est représentée par un niveau d'expression discret qui peut être amené à évoluer. L'ensemble des niveaux d'expression d'un graphe d'interaction est appelé état du graphe :

Définition 1.2 (État d'un graphe d'interaction) Un *état* d'un graphe d'interaction $G = (V; M; E_V; E_M)$ est une fonction $\eta : V \to \mathbb{N}$ telle que pour toute variable v de V, on ait : $\eta(v) \leq b_v$. $\eta(v)$ est appelé le *niveau d'expression* de v.

À tout instant de l'évolution du graphe d'interaction (*i.e.* pour un état donné), chaque variable est influencée par un certain nombre de multiplexes parmi ses prédécesseurs. Seuls les multiplexes dont la formule est vraie sont actifs et peuvent exprimer leur influence sur des variables.

Définition 1.3 (Ressources d'une variable) Étant donnés un graphe d'interaction $G = (V; M; E_V; E_M)$ et un état η de G, l'ensemble des *ressources* d'une variable v de V pour l'état η, qu'on note $\rho(v, \eta)$, est l'ensemble des multiplexes m de $G^{-1}(v)$ tels que φ_m est satisfaite. L'interprétation de la formule φ_m d'un multiplexe m est inductivement définie par :

- si φ_m est réduite à un atome $v \geq s$, où $v \in G^{-1}(m)$ et $(v \xrightarrow{s} m) \in E_V$, alors φ_m est satisfaite ssi $\eta(v) \geq s$,
- si φ_m est réduite à un atome $m' \in M$, où $m' \in G^{-1}(m)$, alors φ_m est satisfaite ssi $\varphi_{m'}$,
- si φ_m est une formule constituée d'atomes et de connecteurs logiques, alors :
  - $\varphi_m \equiv \neg\psi$ est satisfaite ssi ψ n'est pas satisfaite,
  - $\varphi_m \equiv \psi_1 \wedge \psi_2$ est satisfaite ssi ψ_1 et ψ_2 sont satisfaites,
  - $\varphi_m \equiv \psi_1 \vee \psi_2$ est satisfaite ssi ψ_1 est satisfaite ou ψ_2 est satisfaite.

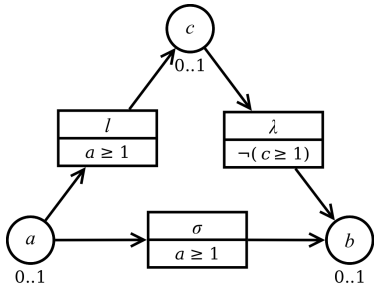
Le symbole $\equiv$ utilisé dans cette définition et dans les définitions à venir indique l'équivalence entre deux assertions. La notation $G^{-1}(m)$, pour un multiplexe m donné, représente l'ensemble de ses prédécesseurs.

Le graphe d'interaction donne la structure du système de gènes étudié, et permet d'en apprécier la dynamique par l'évolution des variables qui le composent. Cependant, il y manque encore une façon de quantifier la « force » des interactions qui entrent en jeu. La *paramétrisation* vient donc compléter le graphe d'interaction en spécifiant comment est subie chacune des interactions présentes dans le graphe ; ainsi, chaque variable pourra être influencée différemment selon le nombre et la nature de ses ressources.

Définition 1.4 (Réseau de régulation) Un *réseau de régulation génétique avec multiplexes* est un couple $(G; \mathcal{K})$ où :
- $G = (V; M; E_V; E_M)$ est un graphe d'interaction avec multiplexes,
- $\mathcal{K} = \{k_{v,\omega}\}$ est une famille de paramètres indexés par $v \in V$ et $\omega \subset G^{-1}(v)$ telle que chaque $k_{v,\omega}$ de $\mathcal{K}$ soit un entier vérifiant : $0 \leq k_{v,\omega} \leq b_v$.

$\mathcal{K}$ est appelée *paramétrisation* de N (ou de G).

La figure 1.2 représente un réseau de régulation génétique constitué du graphe d'interaction de la figure 1.1 et d'un exemple de paramétrisation.



(a)

ω	$k_{a,\omega}$
$\emptyset$	1

ω	$k_{b,\omega}$
$\emptyset$	0
$\{\lambda\}$	1
$\{\sigma\}$	0
$\{\lambda, \sigma\}$	1

ω	$k_{c,\omega}$
$\emptyset$	0
$\{l\}$	1

(b)

Figure 1.2 – Un exemple réseau de régulation génétique comprenant un graphe d'interaction (a) et une paramétrisation (b). [9]

La paramétrisation peut contenir un nombre important de valeurs, qui dépend du nombre de gènes, de leurs plafonds et de leur nombre de prédécesseurs. Ainsi, un problème courant est de trouver, parmi toutes les paramétrisations possibles pour un graphe d'interaction donné, quelle est la paramétrisation qui décrit le comportement correct du système qu'on souhaite étudier. Ce nombre étant le fruit d'une combinatoire importante, il peut devenir très important, ce qui requiert de tester un très grand nombre de possibilités. Pour l'exemple de graphe

d'interaction de la figure 1.1 (qui est assez simple car il ne comporte que trois gènes dont les plafonds sont tous à 1, et peu de multiplexes), le nombre de paramétrisations possibles s'élève à $2^7 = 128$. L'utilisation de la logique de Hoare présentée dans la section suivante propose une démarche pour déterminer certains paramètres en précisant les comportements que l'on souhaite possibles dans le système, tout en évitant de subir l'explosion combinatoire inhérente à une recherche exhaustive.

Une fois un réseau de régulation génétique entièrement défini à l'aide de son graphe d'interaction et de sa paramétrisation, il est possible d'en construire le *graphe d'états*, qui représentera toute la dynamique possible du système.

Définition 1.5 (Fonction de direction) Pour un réseau de régulation $N = (G; \mathcal{K})$ et un état η de $G = (V; M; E_V; E_M)$ donnés, la *fonction de direction* $d : V \to \{-1; 0; 1\}$ est définie par :

$$\forall v \in V, d(v) = \begin{cases} -1 & \text{si } \eta(v) > k_{v,\rho(v,\eta)} \\ 0 & \text{si } \eta(v) = k_{v,\rho(v,\eta)} \\ +1 & \text{si } \eta(v) < k_{v,\rho(v,\eta)} \end{cases}$$

Définition 1.6 (Successeur d'un état) Si $N = (G; \mathcal{K})$ est un réseau de régulation et η un état de G donnés, un état η' de G est un *successeur* de η ssi :

- il existe une variable u telle que $\eta'(u) = \eta(u) + d(u)$ et $d(u) \neq 0$,
- pour toute autre variable $v \neq u$, on a : $\eta'(u) = \eta(u)$.

Définition 1.7 (Graphe d'états asynchrone) Étant donné un réseau de régulation génétique $N = (G; \mathcal{K})$, on appelle *graphe d'états asynchrone* le graphe $\mathcal{S}_N$ vérifiant les propriétés suivantes :

- l'ensemble des nœuds de $\mathcal{S}_N$ est l'ensemble des états possibles de G (isomorphe au produit cartésien $\prod_{v \in V}[0; b_v]$),
- l'ensemble des arcs de $\mathcal{S}_N$ est l'ensemble des couples $(\eta; \eta')$ tels que η' est un successeur de η.

La figure 1.3 donne le graphe d'états du réseau de régulation génétique donné en exemple à la figure 1.2.

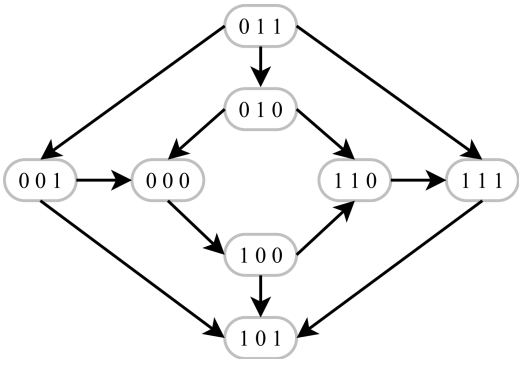


Figure 1.3 – Le graphe d'états obtenu à partir du réseau de régulation de la figure 1.2. Chaque état est un triplet représentant le niveau d'expression des gènes a, b et c, dans cet ordre.

Le graphe d'états constitue la solution la plus explicite pour étudier un réseau de régulation génétique, car il représente tous les états accessibles par un réseau et les chemins entre ces états. Cependant, sa construction est sujette à l'explosion combinatoire, et nécessite un temps de calcul et une consommation en mémoire exponentiels en fonction du nombre de variables, de leurs plafonds et de la paramétrisation. Il s'agit du principal obstacle dans la recherche de la paramétrisation adéquate, car cette recherche nécessite le calcul de tous les graphes d'états possibles. C'est pourquoi des méthodes alternatives sont étudiées afin de passer outre l'explosion combinatoire, et ne s'intéresser qu'aux éléments du réseau de régulation génétique qui sont dignes d'intérêt pour l'étude en cours. L'une de ces méthodes utilise la logique de Hoare, et est présentée dans la section suivante.

1.2 Logique de Hoare

La logique de Hoare est un outil de l'informatique théorique proposé par Hoare en 1969 [8]. Elle permet de caractériser l'exécution d'un programme informatique en formulant une assertion sur l'état de départ et l'état d'arrivée des variables. De cette manière, il est possible de prouver formellement le bon comportement d'un programme ou l'absence de comportement indésirable lors de son exécution.

Dans le contexte de l'étude des interactions entre gènes, la logique de Hoare peut être utilisée en effectuant l'analogie suivante entre un système informatique et un réseau de régulation génétique :

- un état η du réseau de régulation génétique (qui représente la configuration de tous les niveaux d'expression des gènes à un instant donné) est assimilé à la mémoire du système informatique contenant ses variables,
- la paramétrisation $\mathcal{K}$ du réseau de régulation (qui est donnée et constante pour un réseau) représente une mémoire constante du système informatique (dont les valeurs ne peuvent être modifiées par le programme),
- une évolution au sein du graphe d'états peut être représentée sous la forme d'un programme informatique.

Nous verrons dans la suite quelles conséquences cette analogie aura sur notre perception de la logique de Hoare, et comment nous allons l'utiliser.

1.2.1 Règles de la logique

La logique de Hoare repose sur l'utilisation de *triplets de Hoare*, généralement notés $\{P\}\ Q\ \{R\}$, qui regroupent les trois informations nécessaires à l'utilisation de cette logique :

- une pré-condition P, qui caractérise l'état de la mémoire avant exécution,
- un programme informatique Q, généralement exprimé sous forme algorithmique,
- une post-condition Q, qui caractérise l'état de la mémoire après exécution.

Ainsi, les deux conditions nous permettront d'exprimer des assertions sur le niveau d'expression des gènes et sur la paramétrisation, tandis que le programme décrira l'évolution du système de gènes via une série d'incrémentations et de décrémentations des niveaux d'expression.

Notons que nous utiliserons ici la forme dite faible de la logique de Hoare, qui ne présume en aucun cas de la terminaison du programme. Ainsi, dans le cas d'un programme qui ne se termine jamais du fait d'une boucle infinie, la post-condition n'a jamais besoin d'être véri-

fiée. Au niveau biologique, un tel scénario de non-terminaison peut posséder une signification naturelle, par exemple un cycle de régulation.

D'après les définitions concernant la fonction de direction et le successeur d'un état, données dans la section précédente, il existe certaines contraintes sur une variable pour qu'elle puisse évoluer d'un niveau d'expression vers un autre. Ces contraintes proviennent de la paramétrisation du réseau de régulation, qui impose le sens d'évolution des variables en fonction de leur ressources. Il est donc nécessaire de prendre en compte cette paramétrisation lors des incrémentations et décrémentations des variables. Considérons pour la suite un gène v, dont b_v est le plafond et $G^{-1}(v)$ l'ensemble des prédécesseurs, et ω un sous-ensemble de $G^{-1}(v)$; on rappelle de plus que pour tout multiplexe m, on note φ_m l'assertion qui lui est attachée. Posons alors la notation suivante :

$$\Phi_v^\omega \equiv (\bigwedge_{m\in\omega} \varphi_m) \wedge (\bigwedge_{m\in G^{-1}(v)\setminus\omega} \neg\varphi_m)$$

Si Φ_v^ω est vérifiée dans un état η donné, cela signifie que l'ensemble des ressources de v dans η est exactement l'ensemble ω. Définissons enfin :

$$\Phi_v^+ \equiv (0 \leq v) \wedge (v < b_v) \wedge (\bigwedge_{\omega\subset G^{-1}(v)} (\Phi_v^\omega \Longrightarrow k_{v,\omega} > v))$$

$$\Phi_v^- \equiv (0 < v) \wedge (v \geq b_v) \wedge (\bigwedge_{\omega\subset G^{-1}(v)} (\Phi_v^\omega \Longrightarrow k_{v,\omega} < v))$$

Si Φ_v^+ (resp. Φ_v^-) est vérifiée dans un état η donné, cela signifie qu'il est possible d'incrémenter v (resp. de décrémenter v) à partir de l'état η, car la paramétrisation et les ressources de v le permettent.

Enfin, les axiomes et les règles qui forment la logique de Hoare appliquée aux réseaux de régulation génétique sont données ci-dessous.

Règle 1.1 (Règle du programme vide)

$$\frac{P \Longrightarrow R}{\{P\}\ \texttt{skip}\ \{R\}}$$

Règle 1.2 (Règle d'incrémentation)

$$\frac{}{\{\Phi_v^+ \wedge R[v+1/v]\}\ v+\ \{R\}}$$

Règle 1.3 (Règle de décrémentation)

$$\frac{}{\{\Phi_v^- \wedge R[v-1/v]\}\ v-\ \{R\}}$$

Règle 1.4 (Règle de composition)

$$\frac{\{P\}\ Q_1\ \{R\} \qquad \{R\}\ Q_2\ \{S\}}{\{P\}\ Q_1; Q_2\ \{S\}}$$

Règle 1.5 (Règle de condition)

$$\frac{\{P \wedge C\}\ Q_1\ \{R\} \qquad \{P \wedge \neg C\}\ Q_2\ \{R\}}{\{P\}\ \texttt{If}\ (C)\ \texttt{Then}\ Q_1\ \texttt{Else}\ Q_2\ \texttt{End\_If}\ \{R\}}$$

Règle 1.6 (Règle d'itération)

$$\frac{\{P \wedge C\}\ Q\ \{P\}}{\{P\}\ \texttt{While}\ (C)\ \texttt{Do}\ Q\ \texttt{End\_While}\ \{P \wedge \neg C\}}$$

Règle 1.7 (Règle du quantificateur universel)

$$\frac{\{P_1\}\ Q_1\ \{R\} \qquad \{P_2\}\ Q_2\ \{R\}}{\{P_1 \wedge P_2\}\ \forall(Q_1, Q_2)\ \{R\}}$$

Règle 1.8 (Règle du quantificateur existentiel)

$$\frac{\{P_1\}\ Q_1\ \{R\} \qquad \{P_2\}\ Q_2\ \{R\}}{\{P_1 \vee P_2\}\ \exists(Q_1, Q_2)\ \{R\}}$$

1.2.2 Calcul de la plus faible pré-condition

Lors de l'écriture de triplets de Hoare, il peut être intéressant de chercher à maximiser leur expressivité. Considérons par exemple les deux triplets généraux suivants :

$\{x \geq 0\}\ x+\ \{x \geq 1\}$
$\{x = 0\}\ x+\ \{x \geq 1\}$

Ces deux triplets de Hoare portent sur le même programme et comportent la même post-condition, mais la pré-condition du second en réduit nettement l'expressivité. En effet, il ne s'applique qu'au cas où la variable x vaut 0, alors qu'il serait tout à fait possible d'étendre son expressivité à un plus large ensemble de cas, comme c'est le cas du premier triplet qui traite de façon globale tous les cas où x est positif.

Afin de travailler sur les triplets les plus expressifs possibles pour un programme et une post-condition donnés, on peut chercher à rendre la pré-condition la plus faible possible. Des règles permettent de calculer cette plus faible pré-condition ; dans la suite, on appellera WP l'opérateur donnant la plus faible pré-condition, et on connaît sa valeur pour les instructions données précédemment. La notion de plus faible pré-condition et certaines règles permettant de la calculer sont introduites par Dijkstra dans [7].

Règle 1.9 (Plus faible pré-condition de l'incrémentation)

$$\mathrm{WP}(v+, R) \equiv \Phi_v^+ \wedge R[v+1/v]$$

Règle 1.10 (Plus faible pré-condition de la décrémentation)

$$\mathrm{WP}(v-, R) \equiv \Phi_v^- \wedge R[v-1/v]$$

Règle 1.11 (Plus faible pré-condition de la composition)

$$\mathrm{WP}(Q_1; Q_2, R) \equiv \mathrm{WP}(Q_1, \mathrm{WP}(Q_2, R))$$

Règle 1.12 (Plus faible pré-condition de la structure conditionnelle)

$$\mathrm{WP}(\texttt{If } (C) \texttt{ Then } Q_1 \texttt{ Else } Q_2 \texttt{ End\_If}, R) \equiv C \Rightarrow \mathrm{WP}(Q_1, R) \wedge \neg C \Rightarrow \mathrm{WP}(Q_2, R)$$

Il est à noter qu'il existe deux formes pour exprimer la plus faible pré-condition de la structure itérative. La première nécessite de faire apparaître explicitement les environnements (*i.e.* les espaces de variables), ici représentés sous les noms x et y. C'est cette forme qui sera utilisée pour l'implémentation.

Règle 1.13 (Plus faible pré-condition de la structure itérative)

$$\begin{aligned}\mathrm{WP}(\texttt{While } (C) \texttt{ With } (I) \texttt{ Do } Q \texttt{ End\_While}, R) \equiv\ & I \\ & \wedge \forall x, (E(x) \wedge I(x)) \Rightarrow \mathrm{WP}(Q, I)(x) \\ & \wedge \forall y, (\neg E(y) \wedge I(y)) \Rightarrow R(y)\end{aligned}$$

La seconde forme de la plus faible pré-condition de la structure itérative ne nécessite pas d'expliciter les environnements mais est définie par récurrence. Celle-ci a l'inconvénient d'être potentiellement infinie (précisément dans le cas où la boucle ne termine pas), ce qui explique qu'elle ne sera pas utilisée pour l'implémentation.

Règle 1.14 (Alternative à la plus faible pré-condition de la structure itérative)

$$\mathrm{WP}(\texttt{While } (C) \texttt{ Do } Q \texttt{ End\_While}, R) \equiv \exists n \geq 0, H_n$$

$$\text{où :} \begin{cases} H_0 \equiv \neg C \wedge R \\ \forall k \in \mathbb{N}, H_{k+1} \equiv H_0 \vee \mathrm{WP}(Q, H_k) \end{cases}$$

Règle 1.15 (Plus faible pré-condition du quantificateur universel)

$$\mathrm{WP}(\forall(Q_1, Q_2), R) \equiv \mathrm{WP}(Q_1, R) \wedge \mathrm{WP}(Q_2, R)$$

Règle 1.16 (Plus faible pré-condition du quantificateur existentiel)

$$\mathrm{WP}(\exists(Q_1, Q_2), R) \equiv \mathrm{WP}(Q_1, R) \vee \mathrm{WP}(Q_2, R)$$

On voit clairement apparaître, dans la forme des plus faibles pré-conditions des instructions d'incrémentation et de décrémentation, les propositions de la forme Φ_v^+ et Φ_v^- pour une variable v donnée. De ces propositions proviennent les contraintes sur la paramétrisation nécessaires pour assurer les comportements que l'on cherche à faire apparaître. Pour obtenir de tels résultats, il suffit de se donner un programme et une post-condition qui reflètent un comportement que l'on souhaite possible pour le système en cours d'étude, puis de calculer la plus faible pré-condition associée : les conditions sur la paramétrisation apparaîtront directement dans le résultat.

Illustrons l'utilisation de cette logique à l'aide de l'exemple de graphe d'interaction donné à la figure 1.1. Nous cherchons à inférer certains éléments de la paramétrisation afin de faciliter sa recherche. Nous pouvons pour cela contraindre certains comportements dans l'évolution du

système. Posons par exemple la contrainte suivante : lorsque a est à 1, on souhaite que c puisse passer à 1. Pour cela, nous allons étudier le triplet de Hoare suivant :

$$\{a = 1 \wedge c = 0\}\ c+\ \{a = 1 \wedge c = 1\}$$

Calculons la plus faible pré-condition associée au programme et à la post-condition :

$$\mathrm{WP}(c+, a = 1 \wedge c = 1) \equiv \begin{cases} a = 1 \\ c = 0 \\ 0 \leq c \\ c < 1 \\ \varphi_l \Longrightarrow k_{c,\{l\}} > 0 \\ \neg\varphi_l \Longrightarrow k_{c,\emptyset} > 0 \end{cases}$$

Comme on a : $\varphi_l \equiv (a \geq 1) \equiv (a = 1)$, cette plus faible pré-condition se simplifie en :

$$\mathrm{WP}(c+, a = 1 \wedge c = 1) \equiv \begin{cases} a = 1 \\ c = 0 \\ k_{c,\{l\}} = 1 \end{cases}$$

Nous avons donc obtenu une contrainte sur la paramétrisation qui permet d'assurer le comportement souhaité. Notons au passage que b n'intervient pas dans ce calcul ; cette méthode permet en effet de ne pas prendre en compte les parties du réseau qui n'entrent pas en jeu, ce qui diminue considérablement la complexité de la résolution.

Vérifions enfin que l'exemple présenté au fil de la section précédente respecte le résultat que nous venons de montrer. En observant le graphe d'états de la figure 1.3, on constate clairement qu'il respecte la contrainte qu'on a souhaité poser : les deux états dans lesquels on a $(a = 1 \wedge c = 0)$ permettent bien d'accéder à un état où $(c = 1)$. Ainsi, la paramétrisation du réseau de régulation génétique dont il est tiré devrait respecter la règle exhibée ici ; on constate en effet en observant la figure 1.1 que la paramétrisation contient : $k_{c,\{l\}} = 1$. Le résultat montré ici est donc bien confirmé par l'exemple de la section précédente.

Ce premier chapitre a été l'occasion de rappeler les définitions essentielles au sujet traité ici. Ces définitions relèvent des domaines des mathématiques et de l'informatique fondamentale, et comportent des aspects très calculatoires, par exemple en ce qui concerne le calcul de plus faibles pré-conditions. C'est pourquoi il a été envisagé d'effectuer une implémentation de ce sujet, afin de pouvoir produire des résultats de façon automatique. Les deux chapitres qui suivent seront l'occasion de présenter les deux pistes qui ont été suivies dans ce but, de présenter leurs résultats et d'insister sur les avantages et inconvénients de chacune.

Chapitre 2

Utilisation de Coq

Coq [1] est un assistant de preuves formelles développé conjointement par l'INRIA, l'École Polytechnique, l'Université Paris-Sud 11 et le CNRS. Son développement a débuté à l'INRIA en 1984, et il est aujourd'hui écrit en OCaml et en C. Nous verrons dans ce chapitre comment fonctionne cet assistant, afin de comprendre de quelle manière il a été utilisé pour cette application.

2.1 Fonctionnement et utilisation de Coq

Le langage de programmation de Coq, appelé Gallina, utilise le paradigme de la programmation fonctionnelle. Le paradigme fonctionnel, très proche des mathématiques, consiste à déclarer des constantes et des fonctions, et à considérer le résultat de ces fonctions sans impacter le système ; il s'oppose en cela à la programmation impérative, qui voit un programme comme une suite d'instructions qui impactent le système. Gallina permet effectivement de déclarer et manipuler des fonctions, des éléments, des ensembles et des propriétés, le tout avec une rigueur très mathématique. Coq tire parti de l'aspect mathématique de ce langage, en permettant d'exprimer des assertions logiques et de les démontrer.

2.1.1 Définitions inductives

À l'instar des mathématiques, Gallina permet de formuler des définitions inductives (qui se satisfont à elles-mêmes) à l'aide de la commande `Inductive`. Une définition inductive engendre un ensemble qui contient exactement tous les éléments créés par la définition. Chacun de ces éléments est défini à partir d'un constructeur et éventuellement d'arguments. Par exemple, la définition des entiers naturels utilisée par Coq est la suivante :

```
Inductive nat : Type :=
  | O : nat
  | S : nat -> nat.
```

Cette définition crée le type « `nat` » et indique que :
- le constructeur « `O` » est un élément de `nat`,
- le constructeur « `S` » accompagné d'un élément de `nat`, est un élément de `nat`.

Ainsi, un entier naturel est défini comme étant zéro (représenté par `O`), ou le successeur d'un autre entier naturel (à l'aide du constructeur `S`). De plus, cette définition indique qu'aucun autre élément formé à l'aide d'un autre constructeur que `O` ou `S` ne peut être un élément de `nat`. Nous pouvons alors vérifier, à l'aide de la commande `Check`, qui vérifie le type d'un objet, que les éléments `O`, `(S O)`, `(S (S O))`, etc. sont des éléments de `nat`.

2.1.2 Définition de fonctions

À partir d'objets déjà définis (de façon inductive par exemple), il est possible d'en définir de nouveaux à l'aide de la commande `Definition`. On peut par exemple définir la constante `deux` de la façon qui suit :

```
Definition deux : nat := S (S O).
```

Dans cette ligne de définition, on a pris soin de préciser le type de l'objet défini (de type `nat`). Cependant, cette précision est généralement facultative car Gallina est un langage fortement typé, ce qui signifie que Coq est capable de reconnaître le type de n'importe quel objet par sa définition ou l'utilisation qu'on en fait.

On peut enfin définir des fonctions, qui à la différence des constantes prennent un certain nombre d'arguments en entrée. Considérons par exemple la fonction suivante, qui à un entier naturel n associe $n + 2$:

```
Definition plus_deux (n:nat) : nat := S (S n).
```

Cette définition est presque équivalente à la définition de `deux`. La seule différence consiste en l'utilisation d'un argument `n` de type `nat`, qui est réutilisé dans le corps de la fonction.

Il est intéressant de savoir que, comme de nombreux langages fonctionnels, Gallina propose une fonction anonyme (aussi appelée *fonction lambda*) qui peut être utilisée dans le corps d'une définition pour plus de clarté. Ainsi, la définition de `plus_deux` pourrait aussi s'écrire de façon équivalente :

```
Definition plus_deux' : nat -> nat :=
  fun (n:nat) => S (S n).
```

Le type de cette nouvelle entité est `nat -> nat`, ce qui signifie : « une fonction qui à un entier naturel associe un entier naturel », et sa valeur est la fonction anonyme qui à n associe $n + 2$. Si le résultat est identique, cette seconde possibilité d'écriture permet cependant une meilleure compréhension dans certains situations.

2.1.3 Filtrage et définition de points fixes

Lorsqu'un programme utilise un type défini inductivement, il est souvent intéressant de savoir à partir de quel constructeur il a été créé. Dans le cas d'un entier naturel, par exemple, on peut chercher à savoir si on manipule actuellement l'élément nul `O` ou un successeur créé avec `S`. On peut pour cela utiliser ce qu'on appelle le filtrage (ou *pattern-matching* en anglais), qui consiste à appliquer différents traitement selon le constructeur utilisé. Cela s'effectue avec

le mot-clef `match`, qui permettent d'emprunter différents embranchements de programme selon les cas. Voici par exemple la définition de prédécesseur d'un entier :

```
Definition pred (n:nat) : nat :=
  match n with
    | O => O
    | S m => m
  end.
```

Cette fonction associe, à tout entier naturel de la forme $n = m+1$, l'entier m, qui est donc son prédécesseur. Comme 0 n'a pas de prédécesseur, on lui associe une valeur par défaut, ici 0. On voit bien ici que la structure `match ... with ... end` permet d'adapter le comportement du programme selon la nature de l'élément considéré.

Enfin, il est possible de définir des fonctions récursives en Gallina à l'aide de la commande `Fixpoint`, comme dans l'exemple suivant, qui calcule la somme de deux entiers naturels :

```
Fixpoint plus (n:nat) (m:nat) : nat :=
  match n with
    | O => m
    | S n' => S (plus n' m)
  end.
```

Cette fonction est structurellement décroissante par rapport à son premier argument. En effet, lorsqu'elle s'appelle elle-même, son premier argument possède au moins un constructeur de moins (ici, `n` = `S n'` devient `n'`). De façon intuitive, cela se traduit par le fait que pour calculer $n + m$, il faut déjà calculer $(n - 1) + m$, puis $(n - 2) + m$, etc. jusqu'à arriver à $(n - n) + m$ dont on connaît le résultat : m, puis ajouter 1 pour chaque appel récursif. Cela assure que la récursivité prendra fatalement fin, et que la fonction possède toujours une valeur bien définie (ce qui est nécessaire à la définition d'une fonction mathématique) ; il s'agit donc bien du calcul d'un point fixe, qui est plus strict que l'exécution d'une fonction récursive pouvant potentiellement s'appeler à l'infini.

Il est possible de définir une notation afin de rendre l'appel à la fonction `plus` davantage ergonomique, en utilisant la commande `Notation` de la façon suivante :

```
Notation "x + y" := (plus x y) (at level 50, left associativity).
```

La fin de la commande précise les règles de priorités et d'associativité. De cette façon, écrire `a + b` est maintenant équivalent à `plus a b`.

Enfin, il est utile de noter que Coq propose une notation arabe afin d'alléger l'écriture des entiers naturels : il est donc possible d'écrire les nombres à l'aide de chiffres arabes. Ainsi, écrire `4` est équivalent à écrire `S (S (S (S O)))`.

2.1.4 Propriétés et preuves

La particularité de Coq, outre son langage très proche des mathématiques, est la possibilité de formuler des propriétés. Il ne s'agit pas d'expressions ayant une valeur booléenne, mais bien de propriétés mathématiques, pouvant éventuellement être prouvées. Ne pas pouvoir prouver une propriété peut signifier qu'elle est fausse, qu'elle ne peut pas être prouvée, ou que la démarche de preuve n'est pas la bonne et qu'il faut tenter une autre approche. Il est donc possible d'exprimer des propriétés dont on sait qu'elles sont fausses, mais il ne sera pas possible de les prouver par la suite.

Les propriétés possèdent le type `Prop` (qui est l'un des trois types de base de Coq, avec `Type` et `Set`). Voici quelques exemples de propriétés utilisant les définitions précédentes :

```
Definition p1 : Prop := 1 + 2 = 3.
Definition p2 : Prop := 2 + 2 = 22.
Definition p3 : Prop := forall (n:nat), 0 + n = n.
Definition p4 : Prop := forall (m n:nat), 0 + m = n -> m = n.
```

On peut noter plusieurs choses dans cette série de propriétés.

- Les propriétés sont bien des entités à part entière, pouvant être affectées à des noms (ici, `p1`, `p2`, etc.) au même titre que des constantes ou des fonctions.
- Il est possible de définir des propriétés non démontrables : c'est le cas de la propriété `p2` qui est clairement erronée.
- Le mot-clef `forall`, présent dans les propriétés `p3` et `p4`, représente le quantificateur universel $\forall$, et permet de faire appel à des éléments génériques pour produire des propriétés génériques.
- Le symbole `->` représente l'implication logique.

Il est naturellement possible de définir des fonctions ayant pour valeur une propriété.

On peut démontrer une propriété à l'aide de la commande `Theorem` (d'autres commandes comme `Axiom` ou `Example` sont équivalentes). Cette commande a pour effet de démarrer un environnement de preuve. Cet environnement permet d'utiliser des tactiques, qui sont des commandes spécifiques aux preuves de Coq. De plus, un environnement est constitué d'un objectif courant à atteindre (et éventuellement de plusieurs autre objectifs en attente) et d'un contexte regroupant les hypothèses formulées spécialement pour la preuve en cours. Les tactiques permettent alors de faire évoluer l'objectif courant à l'aide du contexte et des théorèmes déjà prouvés, afin de parvenir à une évidence ou à un résultat déjà acquis. Lorsque tous les objectifs sont complétés de cette manière, le théorème est enregistré et l'environnement de preuve est fermé.

Prenons l'exemple de théorème suivant :

```
Theorem plus_null : forall (n:nat), 0 + n = n.
Proof.
```

Nous avons ici démarré l'environnement de preuve. Coq liste alors le contexte (ci-après, au-dessus de la double barre verticale : il est vide au début d'une preuve) et l'ensemble des objectifs, qui n'est ici constitué que de l'énoncé du théorème :

```
1 subgoal

  ============================
   forall n : nat, 0 + n = n
```

Nous pouvons introduire l'entier `n` dans le contexte courant afin de pouvoir le manipuler plus tard, puis simplifier l'objectif courant (afin de transformer `0 + n` en `n`) :

```
  intro n.
  simpl.
```

Nous obtenons donc l'environnement suivant, où on voit bien que `n` est apparu dans le contexte, et que l'objectif a évolué :

```
1 subgoal

  n : nat
  ============================
   n = n
```

Pour terminer cette preuve, il ne nous reste qu'à constater l'évidence de l'égalité `n = n` et à fermer l'environnement de preuve :

```
  reflexivity.
Qed.
```

Coq confirme alors la terminaison correcte de la preuve, et enregistre le théorème. Celui-ci pourra être réutilisé plus tard pour d'autres preuves.

2.2 Implémentation

L'objectif de cette section est de présenter les outils développés et les choix de conception effectués pour mener à bien ce travail. Nous verrons tout d'abord comment les différents aspects des réseaux de régulation génétique ont été implémentés. Afin de pouvoir travailler avec la logique de Hoare, la bibliothèque *Hoare Logic Tutorial*, développée par Sylvain Boulmé, a été utilisée ; nous verrons comment elle fonctionne et les ajouts qui y ont été faits. Enfin, nous verrons les résultats de ce travail ainsi que les pistes à suivre pour l'améliorer et le compléter.

2.2.1 Aspects caractéristiques des réseaux de régulation génétique

La majorité des outils développés pour l'informatique fondamentale utilisent uniquement des variables traditionnelles. Or, dans le cas de l'étude des réseaux de régulation génétique, les multiplexes rentrent aussi en compte, et les variables sont plus contraintes (du fait de leur plafond, de la paramétrisation, et de l'influence de leurs prédécesseurs). Il est donc nécessaire de définir ces nouvelles notions afin de créer un environnement complet pour leur étude. Cette sous-section fait écho à la section 1.1, et son développement a été inspiré des travaux déjà

effectués dans ce domaine, dont la bibliothèque présentée dans la section suivante.

Les variables ont été définies comme des constructeurs sans argument du type `var`. Ainsi, elles peuvent être manipulées comme des objets abstraits comme nous le verrons par la suite, et ne portent aucune information. Les états du réseau de régulation génétique (aussi appelés environnements par analogie avec les environnements de variables informatiques) indiquent le niveau d'expression de chaque variable à un instant donné ; ils prennent la forme de listes qui contiennent les niveaux d'expression de chaque variable. Deux fonctions ont ensuite été définies afin de manipuler ces environnements :

- `get` permet d'obtenir le niveau d'expression d'une variable donnée dans un environnement donné,
- `upd` permet de modifier le niveau d'expression d'une variable donnée à partir d'un environnement donné.

De la même manière, les multiplexes ont été définis comme des constructeurs sans argument du type `mult`. Ainsi, les variables et multiplexes du graphe d'interaction donné en exemple à la figure 1.1 sont représentés de la façon qui suit :

```
Inductive var : Type :=
  | a : var
  | b : var
  | c : var.

Inductive mult : Type :=
  | l : mult
  | lambda : mult
  | sigma : mult.
```

Voici de plus un exemple d'environnement où la variable a est au niveau d'expression 1 et les deux autres sont au niveau d'expression 0 (l'ordre des valeurs indique à quelle variable elles se réfèrent) :

```
Definition env_init := [1 ; 0 ; 0].
```

Afin de stocker les informations concernant le plafond des différentes variables, on peut s'inspirer de la forme des environnements. Ainsi, une liste est créée pour contenir le plafond de chaque variable :

```
Definition vb := [1 ; 1 ; 1].
```

Cette représentation permet aussi de définir simplement les prédécesseurs de chaque variable sous la forme d'une liste :

```
Definition tabpredec : predec :=
  [[] ; [sigma ; lambda] ; [l]].
```

Les différentes formules qui seront utilisées par la suite (*i.e.* les formules des multiplexes et les expressions booléennes utilisées dans les algorithmes vus plus loin) sont définies en deux

temps :

- Une définition syntaxique : les différents opérateurs sont définis comme des constructeurs avec les arguments adéquats. Cela permettra d'écrire des expressions abstraites sous forme d'arbres, à l'aide des opérateurs suivants : atomes sur une variable, opérations sur les entiers, comparaison d'entiers et opérations sur les booléens.
- Une définition sémantique : des fonctions évaluant les expressions sous forme d'arbres permettent d'attribuer une valeur à chaque expression.

Cette distinction permet de manipuler des expressions sans contexte particulier, et de les évaluer uniquement lorsque cela est nécessaire et qu'un environnement d'évaluation est donné.

Les formules des multiplexes sont aussi mémorisées sous la forme d'une liste qui contient la formule de chacun d'entre eux. Pour notre exemple, ladite liste a la forme suivante :

```
Definition mf : multformula :=
  [ATOMV a 1 ; NEG (ATOMV c 1) ; ATOMV a 1].
```

La dernière étape a consisté à écrire les formules du type Φ_v^+ et Φ_v^- qui sont utilisées dans les règles d'incrémentation et de décrémentation de la logique de Hoare (cf. sous-section 1.2). Pour cela, il a été nécessaire d'écrire toute une théorie concernant les notions d'ensembles et sous-ensembles de prédécesseurs qui sont utilisées dans ces définitions. Un ensemble est ici en réalité une liste triée (afin de rendre compte du fait qu'un ensemble n'est pas ordonné, et pour rendre les comparaisons plus simples) et débarrassée de ses doublons. En plus de la fonction de tri, d'autres fonctions ont été codées afin de tester l'appartenance d'un élément, l'inclusion et l'égalité de deux ensembles, de calculer l'union de deux ensembles, et de calculer l'ensemble des parties d'un ensemble.

2.2.2 Utilisation de la logique de Hoare

Hoare Logic Tutorial [6] est une bibliothèque écrite pour Coq par Sylvain Boulmé. Elle a pour but de mettre en œuvre une implémentation de la logique de Hoare qui soit modulable. Cette bibliothèque a été choisie pour base car elle rend possible l'utilisation des outils développés pour les réseaux de régulation génétiques présentés dans la sous-section précédente, et permet le calcul de plus faibles pré-conditions. Elle permet de plus d'effectuer les preuves de correction et complétude afin de prouver que la logique établie est correcte et renvoie exactement les résultats recherchés. Il a cependant fallu modifier certaines parties de la bibliothèque afin de l'adapter au travail en cours.

Cette bibliothèque modélise un langage de programmation impératif simple, au sein duquel il a été nécessaire de séparer l'instruction d'affectation pour former les deux instructions d'incrémentation et de décrémentation, et d'ajouter les deux instructions indéterministes représentées par les quantificateurs universel et existentiel. La définition syntaxique de chaque instruction s'effectue à l'aide de constructeurs pour former le type `ImpProg`, à la façon des opérateurs présentés dans la section précédente, ce qui permet de créer un arbre représentant un programme impératif. Cette définition syntaxique ne présume en rien de la sémantique qui se trouve derrière chaque instruction, mais elle est suffisante pour distinguer les instructions et faire fonctionner le processus de calcul de plus faible pré-condition.

Un système de notations a de plus été utilisé afin de donner plus de clarté aux programmes

impératifs écrits de cette manière. Il est à noter qu'une instruction particulière (représentée par des doubles points-virgule grâce au système de notation) représente la composition de deux instructions, ce qui permet de créer un programme entier. De plus, l'instruction qui représente la boucle répétitive, notée `WHILE`, nécessite d'en expliciter l'invariant. Considérons l'exemple suivant, tiré de [9], et qui s'applique au graphe d'interaction donné à la figure 1.1 :

```
Definition prog : ImpProg := b++ ;; c++ ;; b--.
```

Ce programme simple se comprend aisément comme représentant une évolution dans le graphe d'états. Il signifie que b augmente, puis que c augmente, et enfin que b diminue.

Les définitions de pré-condition et de post-condition de cette bibliothèque profitent directement du caractère fonctionnel du langage Gallina. En effet, elles sont représentées sous la forme de prédicats, à savoir des fonctions qui à un environnement associent une propriété. Cela permet d'exprimer des propriétés qui dépendent directement d'un environnement abstrait, mais aussi de les manipuler avec facilité ; ainsi, le calcul de plus faible pré-condition s'effectue facilement en réutilisant la post-condition fournie et en l'adaptant selon les besoins. Considérons l'exemple de prédicat suivant, tiré de la même source :

```
Definition post : Pred :=
  fun (e:env) => (get b e = 0).
```

La fonction `get`, qui avait été présentée dans la section précédente, renvoie le niveau d'expression d'une variable donnée dans un environnement donné. L'entité `post` définie ici est bien un prédicat, car elle est définie grâce à une fonction anonyme, qui à un environnement associe une propriété. Ce prédicat représente le fait que dans l'environnement `e`, la variable b est au niveau d'expression nul.

Les deux définitions précédentes nous ont permis d'établir un programme et une post-condition, à partir desquels nous souhaitons maintenant calculer la plus faible pré-condition. Ce calcul prend la forme d'une fonction appelée `synt_wp` qui prend un programme et un prédicat (considéré comme une post-condition), et renvoie un prédicat (qui est la plus faible la pré-condition recherchée). Cette fonction effectue un filtrage selon l'instruction qui constitue la racine de l'arbre représentant le programme, et adapte la post-condition en fonction des règles données à la sous-section 1.2.2. Ainsi, pour l'exemple qui nous concerne, la pré-condition se calcule simplement de la façon suivante :

```
Definition pre : Pred := synt_wp prog post.
```

Le résultat de ce calcul consiste en un nombre important de conjonctions et d'implications, du fait des règles de plus faibles préconditions des instructions d'incrémentation et de décrémentation, et de la forme des propositions Φ_v^+ et Φ_v^-. Il s'agit donc d'un prédicat très largement simplifiable, dont seule une fraction apporte de véritables informations, car une grande partie de son contenu est de la forme « $Faux \Longrightarrow P$ », ce qui se simplifie en « $Vrai$ ». Or Coq ne parvient pas à simplifier une telle expression de façon naturelle, aussi faut-il effectuer cette simplification manuellement, par exemple en utilisant l'environnement de preuves qui permet l'utilisation de plusieurs tactiques utiles.

Une fois cette simplification effectuée, Coq apporte cependant la possibilité de vérifier le

résultat trouvé à la main. Il est en effet possible d'utiliser une fois encore l'environnement de preuve en cherchant à montrer que le résultat calculé par Coq est équivalent au résultat trouvé à la main.

2.3 Exemples et résultats

Dans cette section seront présentés les deux exemples principaux traités avec cet outil. Les résultats obtenus et les problèmes rencontrés y seront développés.

2.3.1 Exemple principal

Le premier exemple est celui du graphe d'interaction présenté à la figure 1.1. Plusieurs exemples de résultats sont donnés dans [9] et nous verrons dans la suite dans quelle mesure ils ont pu être reproduits formellement à l'aide de Coq. Le but de ces résultats est de reproduire certains comportements présents sur le graphe d'états de la figure 1.3, afin de retrouver des éléments de la paramétrisation du réseau de régulation génétique de la figure 1.2.

Le premier exemple traite du programme Q et de la post-condition R suivants :

$$\begin{cases} Q = b+; c+; b- \\ R \equiv (b = 0) \end{cases}$$

Le but de l'étude de ce programme est de trouver quelles sont les conditions pour que b signale, *via* une production transitoire, que a est à 1. La conclusion du document sur cet exemple est que la condition d'un tel comportement est la propriété P' suivante :

$$P' \equiv k_{b,\{\sigma\}} = 0 \wedge k_{b,\{\sigma,\lambda\}} = 1 \wedge k_{c,\{l\}} = 1$$

sous la condition que $a = 1$, $b = 0$ et $c = 0$ au début du programme.

Les objets représentant Q et R ont déjà été définis à titres d'exemples dans la section précédente, sous les noms `prog` et `post`, et la plus faible pré-condition associée a été calculée et stockée sous le nom `pre`. Cette pré-condition n'étant pas simplifiée, il n'est pas trivial de vérifier qu'elle est équivalente au résultat trouvé sur papier. Cependant, nous pouvons utiliser l'environnement de preuve pour démontrer formellement cela. Pour ce faire, il faut définir un prédicat `P` représentant la propriété P', puis montrer :

$$(a = 1 \wedge b = 0 \wedge c = 0) \Longrightarrow (\texttt{pre} \Leftrightarrow \texttt{P})$$

Une telle preuve, bien que fastidieuse, est tout à fait possible, et consiste, pour les deux sens de l'équivalence, à simplifier le prédicat `pre` pour se ramener à `P`. Elle a ainsi été réalisée, permettant de confirmer formellement que le résultat trouvé est correct.

Le second exemple porte sur l'impossibilité d'exécuter le programme $Q_2 = b+; b-$. Ce programme est impossible pour ce réseau de régulation génétique de façon intrinsèque : en effet, pour qu'une variable change de direction d'évolution, il faut que ses ressources changent, ce qui n'est pas le cas ici (en effet, b ne s'influence pas elle-même). En utilisant une démarche similaire à la démarche présentée précédemment pour définir les objets dont nous avons besoin,

nous parvenons à calculer la plus faible pré-condition pour le programme Q_2 associé à la post-condition R précédente. Il a enfin été prouvé que cette plus faible pré-condition est fausse (et qu'elle ne peut donc être vérifiée pour aucun environnement et aucune paramétrisation). Cette preuve s'effectue simplement, en parvenant à montrer les contradictions portées par la plus faible pré-condition obtenue.

Le troisième exemple consiste en l'étude du chemin $Q_3 = c+$. À nouveau, on peut utiliser l'environnement de preuve pour démontrer que le résultat obtenu par Coq est équivalent à celui fourni dans l'exemple. Cette démonstration est très proche de celle du premier exemple, et peut être menée à terme de la même façon.

Le dernier exemple pose un problème qui n'a pas été résolu, et qui empêche de produire un résultat. Il cherche à prouver l'impossibilité, lorsque a et c ont atteint le niveau d'expression 1, que b puisse à son tour monter à 1. On peut vérifier cela en étudiant le programme Q_4 et la post-condition R' suivants :

$$\begin{cases} Q_4 = \texttt{While}\ (b < 1)\ \texttt{With}\ (I)\ \texttt{Do}\ \exists(b+, b-, c+, c-)\ \texttt{End\_While} \\ R' \equiv (b = 1) \end{cases}$$

Le calcul de la plus faible pré-condition de ce couple devrait donner un prédicat équivalent à $Faux$. Or la preuve est rendue impossible à cause de la forme des environnements. Ce problème est expliqué plus en détail dans la sous-section 2.4.2.

2.3.2 Exemple du phage λ

Une autre série d'exemples a porté sur l'étude du phage λ, à partir de l'étude qui en a été faite dans [10]. Cet exemple fait entrer en jeu un réseau de régulation génétique à quatre gènes (CI, Cro, CII et N) et huit multiplexes (m_1 à m_8). Pour produire des résultats, le graphe d'états représentant le comportement de ce système a été directement utilisé, dans le but de rechercher les conditions nécessaires pour reproduire certains des comportements observés. Il a ainsi été possible de retrouver certains résultats sur la paramétrisation, qui coïncident avec la paramétrisation proposée dans ce document, en s'intéressant aux deux comportements principaux du système (appelés réponse lytique et réponse lysogénique). Cependant, bien que l'un de ces deux comportements termine sur une boucle infinie, il n'a pas été possible de représenter celle-ci au sein d'un programme comportant une boucle `While`.

Les chemins et post-conditions utilisés pour obtenir des résultats sur cet exemple sont présentés dans la suite, accompagnés des conditions sur les paramètres qu'ils ont permis d'obtenir. Dans la suite, les Q_i représentent les programmes et les R_i les post-conditions utilisées, et les P_i représentent la portion concernant la paramétrisation extraite du calcul de la plus faible pré-condition. Ces résultats sont le fruit d'un calcul de plus faible pré-condition suivi d'une simplification, puis d'une vérification formelle du résultat. Les chemins ont été obtenus en s'inspirant directement du graphe d'états donné dans [10], en utilisant les chemins qualifiés de « biologiquement réalistes » dans le document. Les résultats sont tous compatibles avec la paramétrisation proposée dans ce même document, et qui a permis de réaliser le graphe d'états.

Réponse lysogénique : Cette réponse consiste en un chemin linéaire dans le graphe d'états, qui part de l'état $(0; 0; 0; 0)$ et se termine dans un état stable (*i.e.* un état sans successeur).

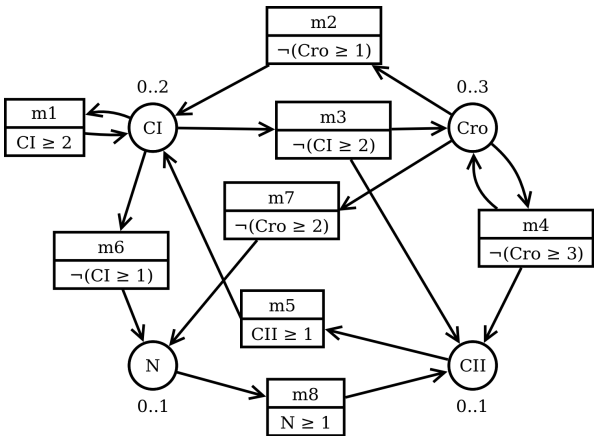


Figure 2.1 – Le graphe d'interaction avec multiplexes de l'exemple du phage λ. [10]

Les données utilisées sont les suivantes :

$$Q_a = N+; CII+; CI+; CI+; N-; CII-$$
$$R_a \equiv (CI = 2 \wedge Cro = 0 \wedge CII = 0 \wedge N = 0)$$

Le résultat obtenu est la conjonction de contraintes suivante :

$$P_a \equiv \begin{cases} k_{CI,\{m_2,m_5\}} = 2 \\ k_{CII,\{m_4\}} = 0 \\ k_{CII,\{m_3,m_4,m_8\}} \geq 1 \\ k_{N,\{m_7\}} = 0 \\ k_{N,\{m_6,m_7\}} = 1 \end{cases}$$

Réponse lytique : Cette réponse est constituée de plusieurs embranchements possibles débutant du même état $(0;0;0;0)$, et débouchant sur une boucle de régulation entre deux états. Seule une partie des comportements sera traitée ici, et deux des embranchements sont représentés par le quantificateur universel du programme suivant :

$$Q_b = Cro+; \forall(Cro+, (N+; Cro+; N-)); Cro+; Cro-$$
$$R_b \equiv (CI = 0 \wedge Cro = 2 \wedge CII = 0 \wedge N = 0)$$

Le résultat du calcul donne ici :

$$P_b \equiv \begin{cases} k_{Cro,\{m_3,m_4\}} = 3 \\ k_{Cro,\{m_3\}} \leq 2 \\ k_{N,\{m_6\}} = 0 \\ k_{N,\{m_6,m_7\}} = 1 \end{cases}$$

Après cet embranchement, le système arrive dans une boucle. Elle est ici représentée simplement comme une séquence revenant à sa situation initiale, plutôt que comme une boucle `While` infinie :

$$Q_c = Cro+; Cro-$$
$$R_c \equiv (CI = 0 \wedge Cro = 2 \wedge CII = 0 \wedge N = 0)$$

Elle a pour conséquence le résultat :

$$P_c \equiv \begin{cases} k_{Cro,\{m_3,m_4\}} = 3 \\ k_{Cro,\{m_3\}} \leq 2 \end{cases}$$

Cet exemple montre qu'il est bien possible d'inférer des paramètres en imposant des comportements au système. Les résultats présentés dans cette sous-section proviennent uniquement du calcul de plus faibles pré-conditions, en spécifiant certains des chemins que peut emprunter le système. De cette façon, il est donc possible de réduire le nombre de paramétrisations possibles grâce à la méthode présentée. Cependant, les programmes possibles ne peuvent comporter de boucle répétitive, pour des raisons qui seront expliquées dans la section suivante.

2.4 Pistes de développement

Malgré de premiers résultats prometteurs, plusieurs obstacles ont empêché la réalisation complète de cette implémentation. Tout d'abord, les preuves de complétude et de correction n'ont pas pu être réalisées, bien qu'elles constituent le principal intérêt de l'utilisation de Coq pour cette application. De plus, si des résultats peuvent effectivement être obtenus, les programmes entrant en jeu ne peuvent pas comprendre de boucle répétitive sans quoi le résultat devient inutilisable. Enfin, l'ajout de la composante temporelle qui faisait partie du sujet ne peut être traité de façon simple en Coq.

2.4.1 Preuves de correction et complétude

Effectuer les preuves de correction et complétude permet de s'assurer que la logique utilisée renvoie exactement tous les résultats corrects. Cela s'applique ici au calcul des plus faibles pré-conditions : il est nécessaire de s'assurer que les pré-conditions renvoyées sont toutes correctes, et qu'elles sont bien les plus faibles possibles. Ces démonstrations avaient été effectuées par Sylvain Boulmé dans la version originale de sa bibliothèque. Cependant, l'ajout des quantificateurs universel et existentiel à la liste des instructions change la donne et nécessite de prendre en compte l'indéterminisme qu'ils introduisent. Il a donc été nécessaire de revoir la sémantique du langage impératif, or l'expression de cette sémantique a posé problème, sa traduction en Gallina n'ayant jamais abouti. Le problème s'est posé au niveau de l'instruction de composition : si sa sémantique pouvait s'écrire facilement sur le papier, son écriture en Gallina posait de nouveaux problèmes. Il est pourtant très probable que si la sémantique parvenait à être exprimée correctement, les preuves de correction et complétude pourraient elles aussi être menées à bien.

2.4.2 Environnements sous forme de listes

Le calcul des plus faibles pré-conditions n'a cependant pas besoin que la sémantique du langage soit exprimée pour fonctionner. En effet, ce sont alors les règles de production de plus faibles pré-conditions qui donnent leur sens aux programmes étudiés. Cependant, un autre problème provient de la forme des environnements, et notamment lors de l'utilisation de la règle de plus faible pré-condition pour la boucle répétitive.

Les environnements (ou états) doivent contenir le niveau d'expression de chaque gène du système étudié. Comme ce nombre change d'un système à l'autre, l'utilisation des listes

semblait naturelle. Or les listes telles que définies en Gallina n'ont pas de longueur définie ; il n'est donc pas possible de contraindre la longueur d'une liste manipulée. Ce problème peut être contourné en travaillant autant que possible avec une liste de valeurs opaques. Il est en effet possible de créer en Coq des entités d'un type défini mais dont la valeur n'est pas connue. Ainsi, il suffit de définir un nombre adéquat d'entiers opaques et de créer un environnement à partir de ces entiers. De cette manière, certaines formules effectuant un accès à un environnement se simplifient automatiquement car l'environnement fourni est de la bonne taille. Or, la forme de la plus faible pré-condition de l'instruction `WHILE` fait appel à un quantificateur sur un environnement :

```
(IwhileInv cond inv p) =>
  (inv e)
  /\ (forall e', (inv e')
    -> (E.eval cond e')=false -> (post e'))
  /\ (forall e', (inv e')
    -> (E.eval cond e')=true -> (synt_wp p inv e'))
```

Cette formule dépend d'un environnement `e'` quelconque, qui est donc une liste de taille quelconque pouvant potentiellement avoir une taille non adaptée, et donc ne pas contenir toutes les informations requises (notamment dans le cas d'une liste plus courte que le nombre de variables). Cette taille indéfinie empêche de développer correctement les pré-conditions faisant intervenir cette formule, et donc de traiter correctement les programmes utilisant des boucles.

Une solution envisagée pour pallier ce défaut a été la création d'un type de liste de longueur fixée. Couplée à une définition plus expressive de la notion d'entier naturel, cette nouvelle forme de liste avait l'avantage de contraindre la taille des environnements, et d'offrir des fonctions plus performantes. En effet, accéder au n^e élément d'une liste traditionnelle peut se solder par un échec : si la liste est trop courte, l'élément recherché n'existe pas. La nouvelle définition développée palliait ce défaut en couplant une liste de longueur donnée, et un entier contraint par un plafond ; il n'était ainsi plus nécessaire de préciser une valeur par défaut lors de la recherche d'un élément dans une liste car cet élément existait toujours.

Cependant, cette solution superposait à la théorie déjà codée une nouvelle couche de complexité. La difficulté venait du fait que la taille de ce nouveau type de liste était intrinsèque à son type même, ce qui posait problème, par exemple, lors de la vérification de l'égalité de deux listes. En effet, écrire une égalité entre deux éléments nécessite que ceux-ci soient du même type ; ainsi, écrire une égalité entre deux listes de tailles m et n nécessite auparavant de vérifier que $m = n$. Si ces preuves sont réalisables, elles ajoutaient une importante part de complexité aux définitions qui l'utilisaient, et cette complexité se répercutait au final sur le calcul des plus faibles pré-conditions. Enfin, il arrivait que pour une raison inexpliquée, Coq ne parvienne plus à développer certaines expressions trop complexes utilisant ces listes. Ces problèmes ont montré que malgré la cohérence de l'idée d'un type de liste de longueur fixe, sa mise en œuvre devenait malgré tout trop complexe et bien souvent inutilisable.

2.4.3 Ajout de la composante temporelle

L'un des aspects du sujet de la présente application de Master était l'ajout de la composante temporelle à l'implémentation réalisée. Le but de cette démarche était de faire entrer le temps dans les résultats fournis par le programme. Pour cela, il a été proposé d'assigner à chaque évolution du système (incrémentation ou décrémentation d'une variable) une durée dépendant uniquement du sens de l'évolution et de la variable concernée. Cela conduit à la définition d'une table de délais d'évolution (à la façon de la paramétrisation présentée à la section 1.1). Or, si la définition d'une telle table ne pose pas d'autre problème que la connaissance effective des durées entrant en jeu, le calcul du délai d'exécution d'un programme à l'aide de cet outil est plus difficile. En effet, il n'est pas possible de définir directement une fonction qui exécuterait le programme et calculerait ce délai, car une telle fonction (nécessairement récursive) pourrait ne pas prendre fin dans le cas où le programme comporterait une boucle infinie. Le fait que Coq n'ait pas l'assurance de la terminaison des programmes entraîne son refus de définir une telle fonction.

Afin de pouvoir néanmoins calculer de tels délais d'exécution à l'aide de Coq, il faudrait pouvoir définir, pour chaque boucle écrite en langage impératif, un variant dont on pourrait prouver la décroissance stricte, et qui aurait le rôle de témoin de terminaison. Il serait aussi nécessaire, avec cela, de définir toute une théorie concernant les variants, afin que Coq puisse les prendre en compte et accepte de produire un résultat pour les programmes se terminant effectivement. Une autre solution serait de se tourner vers un autre outil que Coq, par exemple un langage de programmation plus classique, qui accepte d'exécuter n'importe quel type de programme sans considération de sa terminaison, moyennant le risque que le calcul n'aboutisse pas. Bien que le calcul du délai d'exécution d'un programme infini ne terminerait alors pas, la petite taille des programmes actuellement étudiés permettrait de limiter le nombres d'étapes de calcul nécessaires pour reconnaître que le programme comporte une boucle infinie.

2.5 Conclusion

Coq a été choisi pour ce travail pour son côté proche des mathématiques et pour l'existence de bibliothèques déjà développées pour l'un des aspects du sujet. Il a été nécessaire de programmer la théorie requise pour le maniement des réseaux de régulation génétique, puis il a été possible de réutiliser, moyennant quelques modifications, une bibliothèque permettant l'utilisation de la logique de Hoare. Des résultats ont pu être produits, confirmés par [9] et [10]. Cependant, plusieurs obstacles ont empêché de développer entièrement cette application. Les preuves concernant la logique utilisée n'ont pas pu être menées, et le type de programmes utilisés est limité à des instructions simples. Enfin, l'ajout de la composante temporelle n'a pas pu être traité.

Afin de produire davantage de résultats et au vu des problèmes qui se posent, il a été décidé d'effectuer une seconde implémentation utilisant un autre langage de programmation moins porté sur la rigueur mathématique et offrant plus de liberté. Du fait de ses similitudes et de son influence sur Gallina, le langage OCaml a été choisi pour cette tâche, dans le but de pouvoir réutiliser et traduire rapidement les avancées déjà effectuées avec Coq, et de fournir de nouveaux résultats, potentiellement sous une forme différente.

Chapitre 3

Utilisation d'OCaml

OCaml [4] est un langage de programmation principalement fonctionnel et orienté objet. Il consiste en une version enrichie du langage Caml, dont le développement a débuté à l'INRIA en 1985. Il possède de nombreuses similitudes avec Gallina, le développement de celui-ci en ayant d'ailleurs été fortement inspiré. Dans ce chapitre, nous étudierons les fondamentaux d'OCaml afin de comprendre pourquoi et de quelle manière il a été utilisé.

3.1 Fonctionnement et utilisation d'OCaml

Le langage Gallina utilisé par Coq (cf. chapitre 2) a été largement inspiré par le langage OCaml, ce qui explique leurs nombreuses similitudes. Cependant, l'objectif d'OCaml n'est pas de fournir un langage proche des mathématiques, mais un langage de programmation permettant d'effectuer des tâches plus classiques (accès aux ressources de l'ordinateur, rapidité des calculs...). C'est pourquoi le langage est plus souple que Gallina et intègre un grand nombre d'objets et de fonctions standards, mais ne possède pas tout son bagage mathématique.

3.1.1 Définitions

Afin de définir une fonction ou une constante en OCaml, on utilise le mot-clef `let`. Le nom de l'objet défini doit commencer par une minuscule afin de le différencier des constructeurs qui seront vus dans la sous-section suivante. Il n'est nécessaire de préciser aucun des types des éléments qui entrent en jeu, car OCaml est un langage fortement typé, et procède par déduction pour définir le type de chaque entité. Voici un exemple de définition simple :

```
let plus_deux n = n + 2 ;;
```

Cette fonction, nommée `plus_deux`, prend un argument `n` et le renvoie incrémenté de deux ; elle produit donc le même résultat que son homonyme en Gallina donné à la sous-section 2.1.2. On peut constater que ni le type de la fonction ni celui de son argument `n` ne sont précisés ici car OCaml déduit leurs types en fonction de leur utilisation. Cela est confirmé par le retour d'OCaml *via* la console lorsque la fonction est définie :

```
val plus_deux : int -> int = <fun>
```

Cette ligne signifie que `plus_deux` est bien définie, et qu'il s'agit d'une fonction qui à un entier associe un entier. Dans le cas où rien ne permettrait de déduire le type d'un élément, OCaml peut cependant utiliser un type générique, comme cela serait le cas pour la fonction identité :

```
let id x = x ;;
```

Cette fonction associe à un élément quelconque ce même élément. Lors de sa définition, OCaml renvoie :

```
val id : 'a -> 'a = <fun>
```

Le type générique apparaît ici sous la forme du `'a` qui désigne à la fois le type de l'argument et du résultat de la fonction.

3.1.2 Types et filtrage

Il est possible de définir en OCaml des constructeurs similaires à ceux des définitions inductives de Gallina (cf. sous-section 2.1.1). Les constructeurs rentrent dans le cadre de la définition de types, qui s'effectue à l'aide du mot-clef `type`, et leur nom doit nécessairement débuter par une majuscule. Voici un exemple reprenant la définition des entiers naturels présentée au chapitre précédent :

```
type nat =
  | O
  | S of nat ;;
```

Les deux constructeurs utilisés pour ce type sont `O` qui ne prend pas d'argument, et `S` qui prend un élément de type `nat`.

À l'instar de Gallina, il est possible d'effectuer un filtrage sur une valeur d'un type défini par des constructeurs. Cela s'effectue à l'aide d'une structure `match ... with`, comme dans l'exemple suivant, qui renvoie le prédécesseur d'un entier naturel :

```
let pred n =
  match n with
    | O -> O
    | S m -> m
  ;;
```

Enfin, une forme plus compacte de la structure `match` dans le cas où l'élément à analyser est de type booléen est la structure `if ... then ... else`. Elle fonctionne de façon intuitive : lorsque l'expression booléenne donnée après le `if` est égale à `true`, le résultat placé après le `then` est retourné ; dans le cas contraire, le résultat retourné est celui de la clause `else`. Les deux clauses `then` et `else` doivent nécessairement comporter un résultat du même type. Ainsi, on peut définir la fonction valeur absolue de la façon suivante :

```
let abs n =
  if n < 0 then -n else n ;;
```

Il est à noter enfin que, contrairement au langage Gallina, tous les symboles de comparaison (sur les entiers, notamment) sont en réalité des opérateurs renvoyant un booléen. Ainsi, l'expression « `n < 0` » de l'exemple précédent renverra une valeur booléenne (`true` ou `false`) selon la valeur de `n`, à l'inverse du langage Gallina, qui aurait considéré cette expression comme une proposition et l'aurait conservée telle quelle.

3.1.3 Éléments de programmation impérative

Comme toute fonction doit retourner une valeur, OCaml est doté d'un type particulier, appelé `unit`, dont l'unique élément se note `()` et ne porte pas d'information, et qui est destiné aux fonctions dont le but n'est pas de fournir un résultat. Ce type est donc renvoyé par exemple par les fonctions d'affichage ou de manipulation des fichiers ; ainsi, la fonction `print_string`, qui permet d'afficher une chaîne de caractères, prend en entrée une chaîne de caractères donnée, et renvoie `()`.

OCaml propose quelques éléments permettant d'adoucir le formalisme fonctionnel parfois trop rigoureux. En effet, si le formalisme fonctionnel est toujours suffisant pour calculer un résultat, il ne l'est pas toujours pour traiter le résultat, et peut aussi manquer d'optimisation. Pour pallier cela, le langage permet, à l'intérieur d'une instruction fonctionnelle (terminée par `;;`), d'exécuter une série d'instructions séparées par des `;` simples. Ces instructions doivent nécessairement retourner le type `unit`, sauf la dernière qui fournira le résultat de tout le bloc impératif. Naturellement, le bloc impératif ne sera exécuté que s'il est atteint (il est donc possible de le placer dans une structure `match` ou `if`). Ainsi, l'instruction fonctionnelle suivante affichera la valeur de l'objet `a` puis renverra sa valeur absolue :

```
let a = -5 ;;
print_string "Valeur relative : " ;
print_int a ;
print_newline () ;
abs a ;;
```

3.2 Implémentation

3.2.1 Environnement des réseaux de régulation génétique

S'il a été possible de réutiliser une partie du travail effectué avec Coq pour cette seconde implémentation, les structures de données sont en revanche différentes, afin de s'adapter aux différences et aux possibilités nouvelles d'OCaml. Il a ainsi été possible de programmer le cadre théorique des réseaux de régulation génétique en utilisant des outils plus puissants. C'est notamment le cas des variables et des multiplexes, qui utilisent la gestion plus simple des chaînes de caractères en OCaml : au lieu d'être définis comme des entités abstraites à l'aide d'opérateurs comme expliqué à la section 2.2.1, ils sont représentés par une chaîne de caractères indiquant leur nom. Cette modification nécessite davantage de vérifications, mais

permet d'utiliser les outils de manipulation des chaînes de caractères d'OCaml (comme l'égalité de deux chaînes) qui permettent plus de simplicité.

L'une des premières conséquences de ce choix est la possibilité d'utiliser des listes d'association. Les éléments de ces listes sont des couples dont la première composante est la clef, et la seconde la valeur associée. Pour une clef donnée, on peut alors accéder à ou modifier la valeur associée, à l'aide des fonctions prédéfinies dans la bibliothèque d'OCaml relative aux listes. L'ordre des éléments d'une liste d'association n'a pas d'importance à condition que chaque clef n'apparaisse qu'une fois. Cette forme de liste permet une représentation simple et aisément manipulable de toutes les formes de données faisant correspondre une valeur à une clef, comme les environnements ou la paramétrisation (moyennant plusieurs éléments par clef), à condition d'effectuer certaines vérifications pour détecter la présence de clefs en double ou non définies. Ainsi, on peut aisément coder les fonctions `get` et `upd`, qui ont le même comportement que celles définies pour Coq, à savoir l'accès et la modification du niveau d'expression d'une variable.

Les valeurs de plafonds ont tout d'abord été définies sous la forme d'une liste d'association :

```
let vb = [("a", 1) ; ("b", 1) ; ("c", 1)] ;;
```

Cette liste servira par la suite à vérifier qu'une variable ne dépasse pas son plafond, mais pourra aussi être utilisée comme témoin afin de vérifier qu'une variable est définie (une variable n'y figurant pas étant considérée comme non définie, car n'ayant pas de plafond). Les environnements sont définis de la même façon ; ainsi, dans l'exemple d'environnement suivant, la variable a est au niveau d'expression 1 tandis que les deux autres sont au niveau 0 :

```
let env_init = [("a", 1) ; ("b", 0) ; ("c", 0)] ;;
```

La carte des prédécesseurs des variables peut aussi être définie à l'aide d'une liste d'association, chaque clef étant une variable et la valeur associée étant la liste de ses prédécesseurs. Ainsi, pour note exemple, la carte des prédécesseurs est :

```
let tabpredec =
  [("a", []) ;
   ("b", ["sigma" ; "lambda"]) ;
   ("c", ["l"])] ;;
```

Enfin, la liste des formules des multiplexes est aussi définie à l'aide d'une liste d'association dont les clefs sont les multiplexes du réseau et les valeurs sont leurs formules. Cette liste servira de témoin pour les multiplexes, à l'instar de la carte des plafonds pour les variables. Les formules de multiplexes sont représentées sous la forme d'arbres, chaque nœud étant un constructeur complété des arguments nécessaires, comme pour l'implémentation en Coq. La carte des formules pour l'exemple courant est donc :

```
let mf =
  [("l", Fatom (Atomv ("a", 1))) ;
   ("sigma", Fatom (Atomv ("a", 1))) ;
   ("lambda", FboolOP1 (BNEG, Fatom (Atomv ("c", 1))))] ;;
```

3.2.2 Logique de Hoare

L'une des différences cruciales entre le Gallina et OCaml est l'absence des propriétés qui pouvaient être manipulées par Coq. Il a donc été nécessaire de trouver un substitut simple à mettre en œuvre en OCaml. C'est pourquoi les prédicats de cette implémentation tirent parti du paradigme fonctionnel et prennent la forme d'une fonction qui à un environnement et une paramétrisation associent un booléen. Cette forme est proche de celle développée en Gallina et permet une expressivité suffisante à condition de définir l'opérateur d'implication booléen : $A \Rightarrow B \equiv \neg A \vee B$. Elle a l'avantage de pouvoir être d'évaluée simplement (la fonction renvoie un résultat une fois un environnement et une paramétrisation fournis, contrairement aux prédicats écrits en Gallina qui nécessitaient d'être prouvés), mais nécessite de tester exhaustivement toutes les possibilités si on souhaite trouver l'ensemble de celles qui conviennent. Les pré-conditions et post-conditions de la logique de Hoare sont représentées de cette manière, et cela permet d'effectuer toutes les opérations nécessaires au calcul de la plus faible pré-condition.

Une méthode élémentaire a été développée pour déterminer l'ensemble des environnements et paramétrisations vérifiant un prédicat : elle se contente de calculer l'ensemble exhaustif de tous les environnements et de toutes les paramétrisations possibles, puis de filtrer cet ensemble en fonction du prédicat fourni. Cette méthode fonctionne parfaitement en théorie, mais possède des défauts pratiques que nous verrons par la suite.

Afin de pouvoir utiliser la logique de Hoare par la suite, il est tout d'abord nécessaire de pouvoir exprimer les conditions du type Φ_v^+ et Φ_v^-. Pour cela, des outils de manipulation des ensembles ont été développés afin de voir les listes de multiplexes comme des ensembles et d'effectuer sur celles-ci les opérations nécessaires. Cette étape est assez proche de son équivalent en Gallina, et s'effectue encore plus simplement du fait de la nature des prédicats.

Enfin, il est nécessaire de définir les outils relatifs à la logique de Hoare en soi. Comme la logique de Hoare est simplifiée à son maximum car il n'y est plus question d'utiliser les environnements de preuve, cette définition se réduit à deux étapes : la définition du langage impératif et la définition de la fonction de calcul de plus faible pré-condition, qui font écho aux définitions en Gallina. Le langage impératif est défini comme un type dont les constructeurs sont les instructions du langage (ainsi qu'une instruction permettant de créer une composition de deux instructions). La fonction de calcul de faible pré-condition est identique à celle définie pour Coq, adaptée à la forme booléenne des prédicats. Sa définition n'aurait cependant pas été possible si l'invariant de l'instruction de la boucle répétitive n'avait pas été explicite (comme c'est le cas dans la définition originale de la logique de Hoare [8]), et s'il n'avait pas été possible d'énumérer exhaustivement les environnements et les paramétrisations.

Arrivé à ce point, il est possible d'utiliser les outils développés et d'en tirer des résultats. Pour cela, il suffit de définir un programme impératif et une post-condition qui reflètent la situation à étudier, puis de calculer la plus faible pré-condition associée, qu'il sera possible d'affiner (ce qui revient à calculer sa conjonction avec un autre prédicat afin d'affiner le cas d'étude, pour préciser par exemple les contraintes de l'environnement de départ). La dernière étape consiste alors à utiliser une méthode de résolution, comme celle qui a été expliquée plus haut, afin de calculer tous les couples environnement-paramétrisation qui vérifient le prédicat final.

3.3 Résultats

En suivant la démarche présentée ci-dessus, il est possible de produire certains résultats. Nous nous intéresserons pour commencer au réseau de régulation génétique qui nous a servi d'exemple dans ce chapitre, puis nous verrons quels autres résultats ont été produits ou se sont soldés par un échec.

3.3.1 Exemple principal

Cet exemple se base sur le graphe d'interaction présenté à la figure 1.1 ; les éléments permettant de le définir en OCaml ont été développés dans les exemples de la section précédente. Nous pouvons alors chercher à vérifier l'un des résultats de [9] : on cherche à savoir quelles sont les conditions pour que b signale, *via* une production transitoire, que a passe de 0 à 1. Pour cela, nous allons étudier le programme Q, avec la pré-condition P et la post-condition R suivantes :

$$\begin{cases} Q = b+; c+; b- \\ P \equiv (a = 1 \wedge b = 0 \wedge c = 0) \\ R \equiv (b = 0) \end{cases}$$

Définissons pour cela le programme et la post-condition, à l'aide des outils développés en OCaml, de la façon qui suit :

```
let prog = Iseq (Iseq (Iincr "b", Iincr "c"), Idecr "b") ;;
let post = fun p e -> get "b" e = 0 ;;
```

Le programme défini ici est composé de deux instructions de composition (`Iseq`) imbriquées, de façon à reproduire la séquence des trois instructions. La post-condition est une fonction qui renvoie $Vrai$ uniquement lorsque le niveau d'expression de b dans l'environnement `e` fourni est nul.

Une fois ces deux éléments définis, nous sommes en mesure de calculer la plus faible pré-condition du programme pour la post-condition donnée :

```
let pre_wp = synt_wp prog post ;;
```

Cette pré-condition peut ensuite être affinée pour ne prendre en compte que les cas acceptés par la pré-condition P donnée ci-dessus. Il suffit pour cela d'effectuer une conjonction entre la pré-condition calculée et les contraintes supplémentaires à apporter :

```
let pre = fun p e -> (pre_wp p e) &&
  (get "a" e = 1 && get "b" e = 0 && get "c" e = 0) ;;
```

Une fois ces étapes franchies, il ne reste qu'à trouver tous les cas qui satisfont la pré-condition `pre` obtenue. Pour cela, il suffit d'utiliser l'outil de recherche exhaustive présenté précédemment :

```
let solution = solvevp pre ;;
```

Ce calcul retourne en tout 16 solutions (sur les 1024 possibilités au total), détaillées dans la

table de la figure 3.1. Ces solutions font clairement apparaître les trois contraintes suivantes sur la paramétrisation : $k_{b,\{\sigma\}} = 0$, $k_{b,\{\sigma,\lambda\}} = 1$ et $k_{c,\{l\}} = 1$; ce résultat correspond bien à la conclusion de l'exemple du document de référence.

a	b	c	$k_{a,\emptyset}$	$k_{b,\emptyset}$	$k_{b,\{\lambda\}}$	$k_{b,\{\sigma\}}$	$k_{b,\{\lambda,\sigma\}}$	$k_{b,\emptyset}$	$k_{c,\{l\}}$
1	0	0	0	0	0	0	1	0	1
1	0	0	1	0	0	0	1	0	1
1	0	0	0	1	0	0	1	0	1
1	0	0	1	1	0	0	1	0	1
1	0	0	0	0	1	0	1	0	1
1	0	0	1	0	1	0	1	0	1
1	0	0	0	1	1	0	1	0	1
1	0	0	1	1	1	0	1	0	1
1	0	0	0	0	0	0	1	1	1
1	0	0	1	0	0	0	1	1	1
1	0	0	0	1	0	0	1	1	1
1	0	0	1	1	0	0	1	1	1
1	0	0	0	0	1	0	1	1	1
1	0	0	1	0	1	0	1	1	1
1	0	0	0	1	1	0	1	1	1
1	0	0	1	1	1	0	1	1	1

Figure 3.1 – Solutions du premier exemple de [9].

Nous pouvons chercher à traiter les autres exemples proposés dans [9] et vérifier que les résultats concordent une fois encore. Le second exemple concerne l'exécution du programme $Q_2 = b+; b-$. Si on utilise la même démarche que pour l'exemple précédent, avec le programme Q_2 et la même post-condition R, on n'obtient aucune solution, ce qui est bien le résultat attendu.

Un autre possibilité de chemin étudiée dans ce document est celle du programme $Q_3 = c+$. La valeur de la plus faible pré-condition qui y est donnée est :

$$P_{\mathrm{WP}} \equiv \mathrm{WP}(Q_3, R) \equiv \begin{cases} c = 0 \\ a = 0 \Longrightarrow k_{c,\emptyset} = 1 \\ a = 1 \Longrightarrow k_{c,l} = 1 \\ b = 0 \end{cases}$$

Afin de vérifier que le résultat obtenu à l'aide du présent travail est le même, il suffit d'exprimer d'une part le résultat P_{WP} sous la forme d'une fonction booléenne, et de vérifier que cette fonction et la plus faible pré-condition calculée d'autre part grâce à OCaml produisent exactement les mêmes solutions. Ce calcul amène à 128 solution, et à nouveau les résultats coïncident.

Le dernier exemple de [9] cherche à prouver l'impossibilité, lorsque a et c ont atteint le niveau d'expression 1, de permettre à b de monter à 1. On peut vérifier cela en étudiant le programme Q_4 suivant, avec la pré-condition P' et la post-condition R' suivantes, et en

cherchant à montrer que tout calcul de plus faible pré-condition mène à un prédicat qui vaut $Faux$:

$$\begin{cases} Q_4 = \texttt{While } (b < 1) \texttt{ With } (I) \texttt{ Do } \exists(b+, b-, c+, c-) \texttt{ End\_While} \\ P' \equiv (a = 1 \wedge b = 0 \wedge c = 0 \wedge k_{b,\{\sigma\}} = 0 \wedge k_{b,\{\sigma,\lambda\}} = 1 \wedge k_{c,\{l\}} = 1) \\ R' \equiv (b = 1) \end{cases}$$

La méthode employée pour résoudre ce problème sur papier consistait à montrer que, pour tout invariant I, la pré-condition calculée était équivalente à $Faux$. Cette méthode n'est pas envisageable ici (car nous ne pouvons dénombrer tous les invariants possibles), mais nous pouvons obtenir des résultats en choisissant un invariant convenable, et en calculant la plus faible pré-condition pour ce programme. Deux possibilités d'invariant sont $I \equiv Vrai$, qui est très général, et $I \equiv (a = 0)$, qui se rapproche davantage des contraintes du système dans la boucle. Avec ces deux invariants, le programme ne trouve aucune solution, ce qui recoupe bien les résultats originaux.

L'exécution du programme entier, comprenant les définitions nécessaires à son fonctionnement et le calcul des solutions, et après compilation, s'effectue en moyenne en 140 ms sur un ordinateur portable comportant un processeur double-cœur Intel Core 2 Duo T5550 à 1,83 GHz, et 3,9 Gio de mémoire vive DDR2 à 667 MHz.

3.3.2 Exemple annexe

Enfin, une tentative a été faite de produire des résultats sur le cas du phage λ. Cependant, la combinatoire de cet exemple est bien plus importante, et l'outil développé a finalement montré ses limites. Aucun résultat ne peut être produit sur cet exemple, car le calcul exhaustif de tous les environnements et toutes les paramétrisations possibles nécessaires à la résolution débouche toujours sur un dépassement de capacité.

Pour comprendre l'étendue de cette explosion combinatoire, effectuons le calcul du nombre de possibilités à générer pour la recherche exhaustive. Pour un graphe d'interaction donné, posons N_p ne nombre de paramétrisations et N_e le nombre d'environnements possibles. On a :

$$N_p = \prod_i (b_i + 1)^{2^{|G^{-1}(i)|}}$$

$$N_e = \prod_i (b_i + 1)$$

où la variable muette i parcourt l'ensemble des variables, b_i est le plafond de la variable i et $G^{-1}(i)$ est l'ensemble de ses prédécesseurs. Le nombre de possibilités de couples environnement-paramétrisation pour un graphe d'interaction donné s'élève donc au produit de ces deux valeurs, soit :

$$N = \prod_i (b_i + 1)^{2^{|G^{-1}(i)|}+1}$$

Ainsi, le nombre de possibilités à produire pour la recherche exhaustive pour ce second exemple s'élève à $N = 330\ 225\ 942\ 528$, ce qui explique le problème de dépassement de capacité. Pour comparaison, cette même valeur pour l'exemple principal n'était que de $N = 1\ 024$.

Cette consommation de mémoire limite les usages possibles pour ce programme, et des pistes pour la contourner seront discutées dans la section suivante.

3.4 Pistes de développement

La critique principale qu'on pourrait formuler à l'égard de ce second travail est l'explosion combinatoire rencontrée lors de la résolution finale. Deux pistes permettraient de contourner ce problème.

Il est tout d'abord envisageable de réduire la consommation de mémoire lors de la résolution en effectuant un filtrage à la volée. Ainsi, plutôt que d'attendre la fin de la génération exhaustive de l'ensemble des environnements et des paramétrisations pour appliquer le filtre relatif au prédicat à résoudre, on pourrait envisager de filtrer chacun de ces résultats potentiels au moment de leur création. Cette solution est simple à mettre en œuvre et permettrait d'éviter le dépassement de capacité dans une partie des cas, mais comporte néanmoins deux inconvénients :

- Si la solution comporte un grand nombre de couples environnement-paramétrisation, le dépassement de capacité sera tout de même atteint. Cette méthode peut donc ne pas retourner de solution dans tous les cas.
- Cette méthode ne règle pas le problème du coût en temps du calcul de la solution, car l'ensemble total de tous les environnements et de toutes les paramétrisations doit dans tous les cas être calculé.

Il peut donc être intéressant de tester cette possibilité afin de fournir davantage de résultats, mais elle ne sera pas suffisante pour tous les cas.

Une seconde possibilité peut être envisagée, afin de prévenir davantage les problèmes d'explosion combinatoire. Cette possibilité repose sur l'utilisation de prédicats sous la forme d'arbres de propriétés, afin de se rapprocher du formalisme utilisé avec Coq. Les prédicats ne seraient plus des fonctions booléennes, mais des arbres représentant une propriété abstraite, qu'il serait ensuite possible d'évaluer pour un environnement et une paramétrisation donnés. Cette solution, outre le fait qu'elle est plus proche de la formulation initiale de la logique de Hoare, comporte plusieurs avantages :

- Elle travaille avec des arbres de propriétés qui peuvent être étudiés sans être nécessairement évalués sur un environnement et une paramétrisation, contrairement aux fonctions booléennes utilisées pour le moment. Cela signifie qu'il est possible de travailler directement sur un prédicat pour en tirer des informations.
- Elle fournit un premier résultat qui évite le problème de l'explosion combinatoire, tant au niveau du temps de calcul que de la taille en mémoire.

Il reste cependant possible, après calcul de la plus faible pré-condition, d'effectuer une recherche exhaustive sur tous les environnements et toutes les paramétrisations possibles afin de déterminer l'ensemble des solutions. Cette recherche reste cependant soumise aux problèmes d'explosion combinatoire (bien qu'on puisse envisager de la faciliter par une étude préliminaire du prédicat étudié afin d'écarter d'office certaines possibilités).

Enfin, la notion de délais de production et de dégradation des protéines n'est actuellement pas abordée. Cependant, une fonction de calcul de ces délais pour un programme et un environnement de départ donnés, par exemple, est tout à fait envisageable en OCaml. Une telle fonction était difficile à mettre en œuvre en Coq à cause du problème des programmes ne terminant pas ; cette question ne se pose plus en OCaml car la fonction sera toujours exécutée, même s'il est possible qu'elle ne se termine pas dans le cas d'un programme comportant

une boucle infinie. Il serait aussi envisageable de créer une liste des états visités pour chaque boucle en cours, et de vérifier que l'état courant n'a pas déjà été visité ; une réponse positive signifierait que l'exécution du programme a rencontré une boucle infinie et que le calcul ne doit pas être continué.

En complément, il serait intéressant d'effectuer techniquement le lien entre les données d'entrée du programme réalisé et les données contenues dans un fichier de type GINML [3]. Le format GINML est un format XML développé pour le programme GINsim, qui permet de stocker des données relatives aux réseaux de régulation génétique. Des analyseurs XML sont disponibles en OCaml, ce qui permettrait d'inclure l'ouverture d'un fichier directement dans le programme OCaml.

3.5 Conclusion

OCaml a été choisi comme deuxième piste pour ce travail d'application à la fois pour ses similitudes avec Gallina et pour sa plus grande souplesse. En effet, ce langage a permis une traduction très rapide du travail déjà réalisé pour une réutilisation efficace, renforcée par les outils proposés par certaines bibliothèques. Cependant, comme OCaml n'est pas doté des mêmes outils mathématiques que Coq, et par manque de temps, il a été nécessaire d'effectuer quelques simplifications des outils utilisés. Cela a permis au final d'obtenir des résultats rapides mais pour le moment incomplets car les outils développés se heurtent à des problèmes d'explosion combinatoire.

Afin de compléter ce travail à l'occasion d'un début de thèse, il est proposé d'enrichir l'implémentation actuelle en OCaml en la complétant des pistes proposées à la fin de ce chapitre. Le but de ce travail est d'offrir un outil complet fournissant efficacement des résultats de calculs de plus faibles pré-conditions ou de délais d'exécution sur des réseaux de régulation génétique. Si cet outil bénéficie de suffisamment de temps de développement, il pourrait devenir assez ergonomique pour être utilisable par quiconque, sans connaissance préliminaire d'OCaml ou de programmation.

Discussion et conclusion

Discussion

Avant de clore ce rapport, il semble utile de justifier à nouveau les raisons qui ont motivé les deux pistes de cette application, et d'en rappeler les résultats.

Implémentation à l'aide de Coq

Coq a été choisi en premier lieu pour sa capacité à gérer des notions mathématiques abstraites (programmation fonctionnelle proche des mathématiques, formulation de propriétés, démonstrations), et pour la présence de plusieurs solutions déjà développées permettant l'utilisation de la logique de Hoare. Son utilisation a effectivement permis de se reposer sur des bases déjà existantes et a offert des outils qui n'existent pas naturellement dans d'autres langages, notamment la manipulation directe de propriétés mathématiques (et non de fonctions booléennes).

Son utilisation a cependant rencontré plusieurs obstacles, à commencer par la difficulté de sa prise en main et l'absence du support au sein de l'IRCCyN. De même, l'outil étant en développement et dédié à la recherche scientifique, et son cadre d'utilisation (la bio-informatique) étant peu répandu, il était parfois difficile de trouver de l'aide pour des questions précises. Cette difficulté liée au manque de maturité vis-à-vis du langage Gallina a eu pour conséquence un ralentissement du développement et l'impossibilité de mener à terme certains aspects de l'implémentation.

Si les résultats obtenus en suivant cette piste sont incomplets, cette implémentation offre néanmoins dans sa version finale la possibilité de calculer la plus faible pré-condition de n'importe quel programme impératif, et sa simplification dans le cas de programmes ne comportant pas de boucle répétitive est possible (bien que manuelle et par conséquent souvent laborieuse). Il est enfin possible d'utiliser judicieusement l'environnement de preuve afin, par exemple, de prouver l'équivalence du résultat calculé et du résultat simplifié, et de s'assurer de l'absence d'erreurs.

La principale lacune de cette implémentation est son échec dans les preuves de correction et de complétude, qui auraient permis de prouver la robustesse de la logique développée et l'exactitude des résultats obtenus. Le blocage se situe au niveau de l'expression de la sémantique du langage impératif, qu'un manque de maturité dans l'utilisation du langage Gallina a empêché de mener correctement à terme. Si la forme correcte de cette sémantique était obtenue, il est probable que les preuves de correction et complétude parviendraient finalement à confirmer ou infirmer la cohérence de la logique.

La simplification laborieuse des pré-conditions calculées reste le second point négatif sérieux

de cette piste. L'absence de possibilités d'automatisation de cette tâche nécessite de l'effectuer à la main. De plus, cette simplification est parfois rendue impossible à cause des couches de complexité nécessaires pour le formalisme des réseaux de régulation génétique, qui rendent certains développements impossibles ou mènent à un blocage inexpliqué du programme. L'utilisation d'un solveur capable de communiquer avec Coq ou adapté à cette utilisation permettrait de contourner le problème en simplifiant les résultats de façon plus efficace, avant de les renvoyer à Coq pour vérification. Les résultats simplifiés de cette manière seraient aussi fiables (car vérifiés formellement après simplification).

Ces résultats indiquent que Coq n'est pas adapté à l'utilisation qui en a été faite, ou qu'il a été mal utilisé. Si la première possibilité est peu probable (bien que continuellement en développement et malgré l'absence de solveur, Coq a acquis une certaine maturité et dispose de nombreux outils), la seconde implique cependant qu'une plus grande maturité soit acquise dans l'utilisation de Gallina, afin de fournir une implémentation plus proche de son utilisation habituelle ; cela nécessiterait alors davantage de temps consacré au développement, ou l'appel à une personne experte dans ce domaine.

Implémentation à l'aide d'OCaml

Face aux problèmes non négligeables posés par le développement à l'aide de Coq et au manque de maturité dans l'utilisation du langage Gallina, il a été décidé en second lieu de se tourner vers un autre langage de programmation, si possible plus conventionnel, afin d'offrir des résultats utilisables plus facilement. Du fait de sa proximité au langage Gallina qui s'en inspire fortement, OCaml était un bon candidat. OCaml offrait la possibilité de traduire rapidement depuis Gallina les morceaux de programme déjà développés afin de les réutiliser aisément. De plus, OCaml étant un langage de programmation plus classique et moins regardant en matière de rigueur mathématique, son utilisation était plus simple. Enfin, sa gestion plus souple de certaines formes de données (dont les listes et les chaînes de caractères) a permis d'écarter certaines lourdeurs incontournables avec Gallina.

Cette implémentation a pu déboucher sur des résultats concrets. Elle propose en sortie la liste exhaustive des résultats admissibles par les contraintes posées, et ces contraintes ne sont pas plus complexes à formuler qu'en Gallina.

Le principal obstacle qui s'est posé lors de la traduction du programme en OCaml a été l'impossibilité d'écrire des propositions sous la forme de propriétés brutes, qui étaient le support des pré-conditions et post-conditions en Gallina. Il a été décidé, afin d'arriver rapidement à des résultats concrets, de les traduire sous la forme de fonctions booléennes. Cela pose néanmoins un inconvénient majeur : la condition sous-jacente d'une fonction booléenne n'est pas accessible facilement et nécessite un test exhaustif pour produire un résultat, qui est soumis à l'explosion combinatoire que cette nouvelle théorie cherchait justement à éviter.

Deux pistes sont envisageables pour contourner le problème de l'explosion combinatoire. La première, et la plus simple à mettre en œuvre, est de filtrer les résultats à la volée (en même temps que leur génération) ; cette solution permettrait d'écarter dans certains cas le problème du manque de mémoire. La seconde, plus puissante, consisterait à reproduire la possibilité d'exprimer des propriétés en OCaml, à l'aide d'arbres de propriétés ; cette solution aurait le double avantage de fournir directement la condition associée et de diminuer le problème du temps de calcul. Le problème de la simplification du résultat se poserait à nouveau, mais il serait possible ici aussi d'utiliser ou de programmer un solveur adéquat.

Ces résultats font de l'utilisation d'OCaml une piste plus intéressante que prévue face à Coq, malgré la rigueur mathématique de ce dernier. En effet, la production de résultats semble plus simple et plusieurs pistes sont à nouveau envisageables. Cependant, son développement ne fait que débuter et mérite d'être achevé.

Conclusion

Dans ce rapport ont été présentées les deux pistes d'implémentation qui ont fait l'objet de ma thèse de Master, et dont le but était l'utilisation de la logique de Hoare pour l'inférence de paramètres biologiques au sein des réseaux de régulation génétique.

La première piste d'implémentation a consisté en l'utilisation de l'assistant de preuves formelles Coq. Malgré le travail effectué, elle reste incomplète du fait de nombreux obstacles, et elle ne permet pas de s'en servir pour produire des résultats complexes ou en grand nombre. Mais bien que n'ayant pas conduit à autant de résultats qu'espéré initialement, elle a néanmoins permis d'offrir une première base informatique au sujet, et a servi de point de départ à la seconde piste. De plus, elle a ouvert la voie à l'utilisation des assistants de preuves formelles et aux langages fonctionnels.

La seconde piste d'implémentation a été une traduction et une adaptation en OCaml de la première, dans une tentative de contourner les défauts de celle-ci. Le caractère plus conventionnel et moins rigoureux de ce langage permet davantage de libertés, mais restreint aussi ses possibilités naturelles. Bien que cette piste mène à des résultats plus généraux, elle fait actuellement face au problème de l'explosion combinatoire inhérent aux réseaux de régulation génétique. Cependant, certaines pistes de développement permettraient de contourner ce problème.

Bien qu'encore incomplet, ce travail consiste en la première implémentation de la théorie développée dans [9]. Cette application a déjà fourni des résultats en accord avec la théorie, mais nécessite d'être encore développée et complétée pour permettre l'obtention de davantage de matière.

Au niveau scientifique, ce travail m'a permis de m'assurer de l'importance du développement informatique dans les domaines qui s'y rattachent. De plus, j'ai été confronté de façon motivante à des problèmes propres à la limite ténue entre informatique et mathématiques, et à l'interprétation et la traduction pratique de résultats théoriques. Cette expérience a été enrichissante tant au niveau scientifique, par ce que j'y ai appris, que technique, par ce que j'ai été amené à manipuler, et m'a motivé à poursuivre ces travaux en début de doctorat.

Bibliographie

[1] *Coq Reference Manual*, 2 septembre 2011. http://coq.inria.fr/doc/.

[2] Entrée « Bioinformatics » de Wikipedia, 23 août 2011. http://en.wikipedia.org/wiki/Bio-informatics.

[3] *GINML format*, 2 septembre 2011. http://gin.univ-mrs.fr/GINsim/ginml.html#format.

[4] *The OCaml system*, 2 septembre 2011. http://caml.inria.fr/pub/docs/manual-ocaml/index.html.

[5] G. Bernot, J.-P. Comet, and Z. Khalis. Gene regulatory networks with multiplexes. In *European Simulation and Modelling Conference Proceedings*, pages 423–432, France, Oct. 2008.

[6] S. Boulmé. *Tutorial on Hoare Logic*, 2 septembre 2011. http://www-verimag.imag.fr/~boulme/HOARE_LOGIC_TUTORIAL/.

[7] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18 :453–457, Aug. 1975.

[8] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12 :576–580, Oct. 1969.

[9] Z. Khalis, G. Bernot, J.-P. Comet, A. Richard, O. Roux, and H. Siebert. A hoare logic to identify parameter values of discrete models of gene regulatory networks. Document de travail.

[10] A. Richard. *Modèle formel pour les réseaux de régulation génétique et influence des circuits de rétroaction*, chapter 5.4. Université d'Evry, Sept. 2006.

[11] A. Richard, J.-P. Comet, and G. Bernot. R. Thomas' logical method, Apr. 2008. Invited at Tutorials on modelling methods and tools : Modelling a genetic switch and Metabolic Networks, Spring School on Modelling Complex Biological Systems in the Context of Genomics.