Gubs, un langage de description comportementale pour la biologie de synthèse

Adrien Basso-Blandin

Thèse présentée pour obtenir le grade de docteur

Université d'Evry Val d'Essonne

Laboratoire IBISC, Equipe COSMO
École doctorale du Génome Aux Organismes

Discipline : Informatique

Gubs, un langage de description comportementale pour la biologie de synthèse

Conception d'un langage dédié à la conception de fonctions biologiques de synthèse par compilation de spécifications comportementales

---

Par : **Adrien Basso-Blandin**

Sous la direction de Franck Delaplace, Professeur des Universités

Membres du jury:

- **Examinateur :** Franck Delaplace, Professeur, IBISC
- **Examinateur :** François Fages, Dir. de recherche, INRIA Paris-Rocquencourt
- **Examinateur :** Jean-Louis Giavitto, Dir. de recherche, IRCAM
- **Examinateur :** Russell Harmer, Ch. de recherche, ENS Lyon
- **Rapporteur externe :** Cédric Lhoussaine, HDR, LIFL
- **Examinateur :** Franck Molina, Dir. de recherche, SysDiag CNRS
- **Rapporteur externe :** Olivier Roux, Professeur, IRCCyN

Remerciements

Je tiens à remercier tout d'abord Franck Delaplace pour son encadrement sans faille durant ces trois années, nos discussions de recherches, son avis éclairé et nos débats scientifiques constants. Je tiens aussi à le remercier pour m'avoir appris que « chose » et « truc » n'étaient pas des mots scientifiques ainsi qu'à nommer et définir chaque concept avec rigueur.

Je tiens ensuite à remercier mes rapporteurs Olivier Roux et Cédric Lhoussainne pour le temps passé en relecture ainsi que pour leur commentaires et remarques qui m'ont permis d'améliorer mon manuscrit et rendre certaines parties plus pédagogiques.

Je remercie les membres du jury d'avoir accepté de juger mes travaux.

Je remercie Amélie, Anaïs et ma mère pour leur relecture attentive qui leur a pris des heures.

Je remercie aussi Amélie pour toutes nos discussions au bureau et m'avoir servi de binôme ces deux dernières années et pour relire encore ces remerciements.

Je souhaite remercier les personnes suivantes auprès desquelles j'ai beaucoup appris : Serenella pour nos discussions sur la logique et son soutien dans les coups de blues, Hanna et Franck Pommereau pour leurs avis et commentaires éclairés, Laurent et techstaff pour m'avoir fait installer archLinux, Guillaume pour m'avoir entrainé dans la fête de la science que j'ai du coup refais les 3 années et tous les membres du laboratoire pour leur accueil et l'excellente ambiance qui y règne ainsi que pour les discussions scientifiques et enseignements.

Je tiens aussi à remercier Stan et Vincent avec qui j'ai commencé et fini cette thèse et partagé de nombreux bons moments au bureau, au bar ou à l'escalade. Merci aussi à mes alumni Lukasz, Romain, Arnaud et Céline pour l'ambiance au sein de l'équipe doctorante et les soirées jeux de sociétés/films au labo.

Je remercie Murielle pour sa gestion de l'administration du laboratoire qui m'a bien aidé plus d'une fois ainsi que Georgia pour m'avoir fait découvrir les coulisses d'une conférence en biologie.

Pour finir je tiens à remercier mes parents et Anaïs pour m'avoir soutenu tout ce temps.

Résumé

La biologie de synthèse est un domaine émergent en quête d'outils afin de formaliser et d'automatiser la caractérisation et la conception de systèmes biologiques. Dans ce cadre, nous proposons un langage de spécification comportementale des systèmes biologiques, ainsi que la conception d'un compilateur traduisant cette spécification en un assemblage de composants biologiques.

La première partie sera dédiée à un langage de description comportementale nommé **Gubs** (Genetic Unified Behaviour Specification) pour la spécification de composants biologiques en les décrivant comme des systèmes ouverts dynamiques et discrets [1]. **Gubs** est un langage déclaratif dont la syntaxe se fonde sur une description des comportements par un ensemble de relations causales. Contrairement à un système fermé, un programme est toujours une description partielle du comportement du système. La sémantique a été conçue afin de prendre en compte la présence d'actions non spécifiées qui pourraient potentiellement altérer le comportement des composants programmés en l'exprimant sous forme d'une formule de logique hybride.

En seconde partie, nous introduisons un système formel décrivant les principes de compilation d'une spécification en **Gubs** en un ensemble de composants biologiques synthétisables. Ce système est implémenté par **Ggc**, un compilateur permettant de sélectionner automatiquement les composants possédant les propriétés adéquates pour qu'une fois assemblés ils simulent le comportement décrit. La compilation d'une spécification **Gubs** s'appuie sur le principe d'ACI-Unification en utilisant un schéma similaire au système de preuve automatique afin de sélectionner les composants dont l'assemblage est correct par rapport à la spécification. Dans le cadre d'une unification avec une base de données de grande taille, l'algorithme d'ACI-Unification bascule sur un algorithme évolutionnaire d'optimisation permettant la recherche des composants en adéquation avec le programme afin d'obtenir une solution.

Finalement, cette thèse se conclut sur un ensemble d'optimisations permettant de sélectionner des composants selon des propriétés biologiques afin d'obtenir une sélection plus fine dans le but d'assurer une synthèse des éléments in-silico en systèmes biologiques viables in-vivo. Nous concluons aussi sur un traitement automatique des bases de données à disposition des chercheurs afin de les traduire en un ensemble de composants Gubs.

1. Un système discret se décrit informellement comme un ensemble de composants dont la somme des comportements n'est pas égale au comportement de leur assemblage

Sommaire

Introduction

La biologie de synthèse possède de nombreuses définitions [95, 27, 17], néanmoins la plupart sont axées sur plusieurs points clés :

- Synthétiser des systèmes biologiques complexes fondés ou inspirés de systèmes existants ;
- introduire dans des organismes biologiques des fonctions n'existant pas à l'état naturel ;
- spécifier des composants normalisés et interchangeables tels des modules assemblables ;
- formaliser et spécifier le comportement de systèmes biologiques existant en appliquant des méthodes d'ingénierie ;
- minimiser la taille des organismes vivants pour obtenir des organismes minimaux ;
- synthétiser de-novo des organismes existants.

Dans ce cadre, nous définirons ici la biologie de synthèse comme une science amenant à la compréhension, la synthèse d'organismes existants ou nouveaux ainsi que de leur modification en utilisant des outils et un formalisme introduits par les sciences de l'ingénierie, notamment dans le but d'automatiser ces procédés.

Comme dans tout domaine traitant des systèmes critiques [62], « manipuler » le vivant impose une ingénierie sûre et formalisable telle que celle utilisée en aéronautique, en aérospatiale ou en robotique médicale.

Ce chapitre introduit un bref historique de la biologie de synthèse, ses domaines d'applications, ses outils et les problématiques inhérentes au domaine, afin d'en donner une vision globale et de placer le contexte de notre travail.

La biologie de synthèse est un domaine de recherche scientifique en pleine émergence alliant la dimension investigatrice de la biologie avec celle constructive de l'ingénierie [86] afin de concevoir des systèmes biologiques de synthèse possédant de nouvelles fonctionnalités ou comportements qui n'existeraient pas dans la nature.

Elle est à la recherche de principes et d'outils pour rendre les dispositifs biologiques interopérables et programmables [78] dans la perspective d'en faciliter son ingénierie. Les projets de biologie synthétique ont d'abord mis l'accent sur la conception et l'amélioration de petits dispositifs génétiques comparables aux portes logiques utilisées dans les circuits électroniques [88, 39]. Récemment, des projets ont développé de grands biosystèmes [82, 5] composés de différents composants avec comme objectif à long terme, la conception de génome synthétique *De-novo* [61]. A cette fin, les environnements de conception assistée par ordinateur (CAO) joueront un rôle central en fournissant les fonctionnalités nécessaires pour la conception des systèmes : spécification, analyse et optimisation [19, 83, 110, 41]. Des applications pionnières ont d'ailleurs montré le potentiel de ces environnements dans la compétition Igem [64, 35, 50, 87] (International Genetically Engineered Machine).

A l'heure actuelle, la conception de génomes de synthèse repose sur une description structurelle de l'assemblage de séquences d'ADN (Biobrick) comme par exemple dans l'outil Genocad[25]. Bien que cette description soit indispensable afin de faire une spécification complète des composants, le niveau d'abstraction actuel semble inapproprié pour appréhender de plus grands biosystèmes. En effet, la taille des programmes requis pour la description de telles séquences augmente fortement les risques d'erreurs et rend la tâche irréalisable. De la même façon qu'un programme informatique complexe ne peut être codé en binaire, de grands biosystèmes ne peuvent être décrits comme un ensemble de séquences d'ADN. Afin de passer à l'échelle en terme de complexité des systèmes biologiques, nous devons compléter cette description structurelle par une couche de programmation abstraite fondée sur un langage de haut niveau et mettre en place une conversion automatique d'une spécification dans ce langage du composant biologique en un ensemble de séquences d'ADN, telle que le font les compilateurs informatiques. De tels langages de haut niveau pour la biologie de synthèse sont annoncés comme les pierres angulaires de la nouvelle génération de recherche en biologie de synthèse afin de répondre à la complexité des

systèmes synthétiques de grande taille [86]. Néanmoins, les langages développés dans ce domaine en sont encore à leur balbutiement, et implémenter une telle solution est un des défis à relever en biologie de synthèse.

Une première question à se poser est : quelle est la taille des systèmes à programmer ? Et comment évolue cette taille ? en effet, la biologie de synthèse a fait des progrès significatifs ces dernières années comme le montre la figure 2. Une telle évolution est à rapprocher du coût de la synthèse et de la progression de la qualité et de la fiabilité de cette synthèse pour des génomes de synthèse de plus en plus longs, comparable à l'évolution de la puissance et du coût des processeurs en informatique (loi de Moore), comme le montre l'économiste R.Carlson [31] dans les courbes de la figure 1. De telles courbes sont indicatrices d'une révolution technologique. En ce sens, elle place la nécessité de concevoir des environnements adaptés à la conception comme l'une des phases importantes de ce domaine. Comme cela a été le cas pour l'informatique afin de permettre plus de performance et un meilleur usage de ce potentiel.

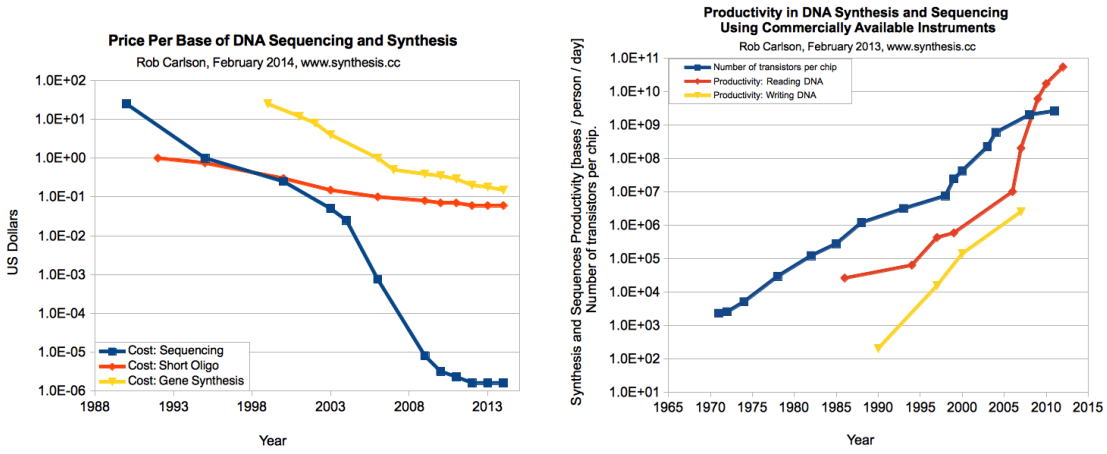


Figure 1 – Cette étude faite par l'économiste R.Carlson assimile cette croissance à celle connue pour les processeurs (Loi de Moore)

De tels progrès nécessitent donc de prendre en compte dans un futur proche, la conception de systèmes biologiques de grande taille, or comme l'a montré l'informatique avec l'augmentation en taille et en complexité des programmes développés, un tel changement d'échelle nécessite un changement d'approche et une évolution des langages utilisés pour les décrire.

Une telle approche nécessite de se poser une seconde question : que signifie programmer le vivant ? Et si c'est possible, comment le programmer ? Programmer le vivant peut signifier : programmer l'organisme en le modifiant complètement, ou lui ajouter des fonctionnalités par insertion d'*ADN*. La connaissance actuelle des organismes, bien qu'évoluant rapidement comme le montre la figure 2, ne nous permet pas d'envisager à l'heure actuelle de les programmer directement.

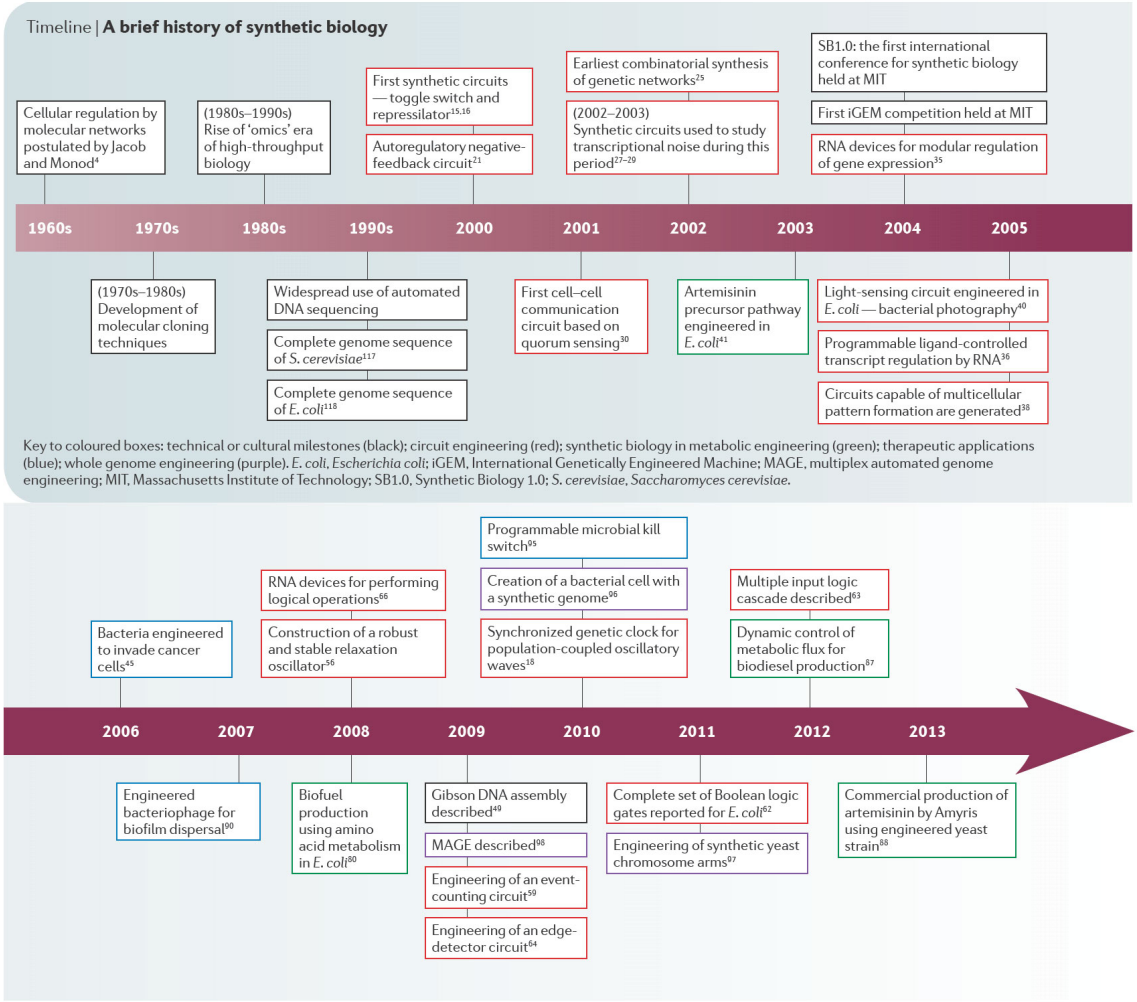


FIGURE 2 – Cette frise chronologique proposée par Cameron[29] propose de récapituler les grandes découvertes en biologie de synthèse. A cette chronologie s'ajoute en 2014 la synthèse d'un chromosome de S. Cerevisiae[5].

Nous nous orientons donc vers l'ajout de fonctionnalités, comme le font les recherches actuelles en biologie de synthèse. Finalement programmer le vivant revient ici à ajouter un ensemble de fonctionnalités à un organisme existant, ce que l'ont peut

traduire en informatique par l'ajout d'un ensemble de sous programmes (plug-in) à un programme existant. Néanmoins, une difficulté supplémentaire est que nous ne connaissons pas tout du programme existant, et que l'environnement externe à notre système peut agir sur les fonctionnalités que nous ajoutons. En effet, les organismes vivants ne sont pas des systèmes fermés comme ceux utilisés classiquement en ingénierie et plus particulièrement en informatique. Un système biologique possède une interaction forte avec son environnement, ainsi que des propriétés qui lui sont propres, servant à sa survie et à son fonctionnement et dépendant de l'organisme utilisé. Programmer le vivant ne peut donc pas consister à simplement assembler des composants afin d'obtenir un comportement souhaité, mais à prendre en compte un ensemble de facteurs, et parmi un ensemble de composants, sélectionner ceux permettant à la fois d'obtenir le comportement souhaité mais aussi de prendre en compte tous les facteurs environnementaux et propres au système. On passe donc d'une approche consistant à assembler des composants pour obtenir un comportement à une démarche consistant à définir un comportement et sélectionner des composants adéquats afin d'obtenir ce comportement. Nous passons donc d'une approche structurelle à une approche comportementale.

Cette idée nous amène à penser qu'un langage de haut niveau est nécessaire, permettant une description des systèmes biologiques non pas en terme de séquences mais de fonctionnalités, permettant ainsi de programmer leurs comportement au lieu de la structure permettant ce comportement. En effet, la spécification de comportement contribue à documenter avec précision le dispositif en ajoutant une description comportementale pour évaluer les fonctionnalités automatiquement et formellement. Ceci permet notamment la génération de tests de cette spécification afin d'obtenir une indépendance par rapport à la technologie car différentes structures biologiques peuvent réaliser la même fonctionnalité. Dans ce cadre, les composants sont sélectionnés et assemblés automatiquement ou semi automatiquement pour générer une description structurelle du système lors de la phase de compilation dont le comportement est conforme à la fonction spécifiée. Une telle approche a déjà été utilisée dans le domaine des composants matériels en utilisant des langages tels que Vhdl [6] ou Verilog [107] afin de répondre à l'augmentation en complexité des circuits électroniques. Cependant, la différence majeure en biologie synthétique concerne la dimension ouverte des systèmes biologiques. Schématiquement l'ouverture introduit

deux problématiques supplémentaires : l'interaction avec l'environnement et les perturbations de l'environnement sur le système. L'interaction avec l'environnement nécessite la prise en compte des variations environnementales et de la réponse à fournir. Les perturbations de l'environnement quant à elles signifient que le langage doit prendre en compte des altérations du système biologique dues à l'environnement qui ne sont pas définies explicitement comme des modes d'interaction. Ce cas survient notamment lors de l'introduction d'une voie biologique dans un système et est connu comme le phénomène de Cross-talk [80].

Pour répondre à ces enjeux, nous proposons de définir un langage dédié à la biologie synthétique fondé sur une spécification comportementale qui prend en compte cette dimension ouverte du système et permettant de définir des systèmes de grande taille.

Plus précisément, **Gubs** est un langage déclaratif fondé sur des règles dédiées à la spécification comportementale des *systèmes dynamiques discrets ouvertes* dans le cadre de la biologie synthétique en interaction avec son environnement. **Gubs** définit symboliquement les comportements afin de fournir une indépendance vis-à-vis des structures en reportant la sélection de composants biologiques à la phase de compilation. Dans ce cadre, le compilateur traduit la spécification comportementale en une description structurelle d'un système dont le comportement correspond aux caractéristiques fonctionnelles définies par un programme.

Domaines d'applications

La biologie de synthèse voit son application dans de multiples domaines tels que la médecine, l'énergie, les bio-senseurs capables de détecter certains composés chimiques considérés comme toxiques, la conception de matériaux : qu'il s'agisse de matériaux conçus par des organismes génétiquement modifiés ou qu'il s'agisse de matériaux organiques composés d'organismes modifiés à ce dessein. On notera aussi l'utilisation d'organismes modifiés connus en agriculture tels que le maïs[116]. Cette multitude d'applications entraine une interaction entre de nombreux agents de ces domaines, qu'il s'agisse de biologistes, de chimistes, d'informaticiens, de bio-informaticiens ou de mathématiciens. Mais aussi de sociologues et de philosophes afin d'étudier les risques et impacts autant humains qu'environnementaux de telles recherches.

Dans cette section nous proposons de reprendre quelques applications de la biologie de synthèse dans les trois grands domaines précédemment décrits : Médecine, Énergie et Environnement.

Médecine

Weber conclut que « *L'émergence de la biologie synthétique porte de grands espoirs dans le domaine de la découverte de médicaments en fournissant de nouvelles opportunités tout au long du processus de découverte de médicaments. Les outils de la biologie de synthèse permettent d'analyser les mécanismes d'infection et les cibles de la maladie et fournissent également des pistes pour découvrir des molécules utiles en chimiothérapie ou encore la conception de nouveaux produits biopharmaceutiques. En outre, la biologie de synthèse permet de concevoir des processus de production microbienne rentables pour les produits naturels complexes, ce qui pourrait aider à surmonter la pénurie mondiale de médicaments. Ces progrès impressionnants ont été réalisés en seulement quelques années, et sont un indicateur du potentiel de la biologie synthétique* » [114].

Par exemple, le paludisme est actuellement un fléau touchant 216 millions de personnes dans le monde, causant 655 000 morts par an [93] dont le seul traitement connu est l'artémésinine, un composé chimique extrait de l'Armoise annuelle, une plante cultivée en Chine. Or le rendement de cette plante est faible, et celle-ci est extrêmement sensible au climat, empêchant son importation dans les pays principalement touchés. À cette fin, en 2006, Jay Keasling a proposé une solution [42] consistant à modifier génétiquement une bactérie (ici : *Saccharomyces cerevisiae*) afin que celle-ci synthétise l'artémésinine, permettant ainsi que la production ne soit plus sujette aux aléas du climat et de diviser par trois le coût des doses du médicament.

Énergie

Savage note que « *Bien que les ressources pétrolières soient en quantité limitée, il est raisonnable de supposer que les sources de combustibles fossiles non conventionnelles peuvent continuer à répondre aux demandes croissantes d'énergie de la société pour les décennies à venir. Le vrai défi est la durabilité : stabiliser et inverser le changement climatique mondial, en minimisant l'impact politique et économique, et faciliter la transition des combustibles fossiles dans un avenir proche. En réponse*

à ce défi, les biotechnologies cherchent à développer des carburants, tels que l'éthanol, le butanol, le biodiesel, et de l'hydrogène (H2), en les faisant synthétiser par des organismes vivants tirant leur énergie de la photosynthèse par la lumière du soleil pour le produire. » [97].

Une des applications est la synthèse de bioéthanol par des algues [115]. Plus précisément, il s'agit d'extraire du bio-carburant de la biomasse des algues en utilisant des cellules microbiennes génétiquement modifiées pour transformer ce bio-carburant par fermentation du sucre présent dans les algues. Un autre exemple d'application est la conception de batteries optimisées grâce à l'utilisation de virus modifiés [94]. Plus précisément, il s'agit d'utiliser une souche de *virus de la mosaïque du tabac* modifiée afin d'encoder des résidus de Cystéine s'auto-assemblant afin de créer des électrodes avec une surface de contact doublant leur capacités.

Environnement

Le terme *bio-senseur* se réfère à une large variété de composants permettant de « sentir » des composés considérés comme toxiques ou dangereux. Le plus commun est un composant biologique possédant une reconnaissance hautement spécifique d'un certain composé cible. Cette détection va induire un signal aisément détectable, de préférence facilement convertible en un signal électrique afin que le résultat puisse être traité par un dispositif électronique de traitement du signal, de stockage de données, etc. Une autre classe de *bio-senseur*, aussi appelé rapporteur biologique, utilise des cellules vivantes en tant que composant. Ces cellules détectent le composé cible par l'intermédiaire de certains récepteurs plus ou moins spécifiques et génèrent une réponse détectable, le plus souvent par l'induction d'un gène rapporteur [32].

Un exemple d'application est la détection de la présence de l'arsenic [104] dans l'eau. Dans le cas présent, une souche d'*Escherichia coli* est modifiée afin de produire de la GFP en présence d'arsenic. Une autre application environnementale est la conception de biofilm qui consiste à faire agréger des métaux lourds par des bactéries afin de les retraiter par la suite [111].

Les composants de la biologie de synthèse

La construction de systèmes biologiques artificiels se fonde sur un découpage hiérarchique des composants. Bien que n'étant pas totalement codifié, l'objectif est d'introduire une structuration par couche des systèmes biologiques, allant du composant atomique au système. Actuellement, les recherches portent sur, d'une part les biobricks [74] et d'autre part sur le support des composants : les châssis cellulaires. Ils constituent les deux extrêmes de cette organisation : composants atomiques et support d'intégration du système. L'objectif de **Gubs** étant d'assembler des composants biologiques afin de reproduire un comportement abstrait défini par un programme, une telle approche de standardisation des composants est nécessaire. Nous allons donc introduire les standards présents en biologie de synthèse.

Biobrick

La *biobrick* est une unité biologique définie suivant un standard universel ayant idéalement une structure permettant de les composer facilement entre elles. Les *briobrick* (Figure 3) sont décrites au sein d'une base de données nommée « parts registry » [2] introduite par Drew Endy [30] permettant de les réutiliser facilement dans le cadre de projet scientifique.

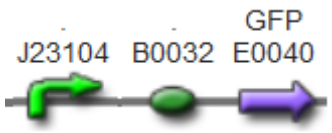


Figure 3 – Cet assemblage se compose de trois biobricks, un promoteur à gauche, un RBS au milieu et une séquence codante à droite. Cet assemblage est lui-même une biobrick qui peut donc être lui-même assemblé.

Les *biobricks* sont assemblées pour former des systèmes biologiques de synthèse, mais peuvent aussi être assemblées pour créer des sous-modules d'un système plus complexe, ces modules sont appelés *cassettes*. Une cassette, permet de décrire un système biologique, comme un ensemble de modules, sans avoir à en connaître la composition. En effet, les cassettes étant composées de *biobrick*, elles sont elles-mêmes composables. Il suffit donc de connaître l'entrée et sortie du module pour pouvoir

2. http ://partsregistry.org/Main_Page

l'utiliser sans en connaître le contenu.

On peut rapprocher cette notion de celle d'objet en informatique, ce qui amène l'idée d'une structure en couches d'abstractions des systèmes biologiques : le système étant composé de modules, eux-mêmes composés de *biobrick*.

Une partie des nouvelles *biobricks* pour la biologie synthétique sont spécifiées lors du concours IGEM. IGEM (International Genetic Engineering Machine) est un concours international de biologie synthétique organisé par le MIT qui regroupe des étudiants de tous les horizons : informatique, biologie, physique, médecine, afin de développer en équipe de recherche un système biologique synthétique original ou répondant à une problématique que l'équipe doit choisir, telle que prendre des clichés avec une boîte de Petri [3] ou encore le traitement des eaux usées [4]. Notons néanmoins que la plupart des projets restent à l'état de développement, et les *biobrick* soumises ne sont donc pas toutes fonctionnelles *in-vitro*, même quant elles ont été caractérisées.

Ajoutons à cela le fait que le concept de *biobrick* n'est pas parfait, comme l'explique J.Romm, dans un article intitulé « Five hards truths for the Synthetic Biology » en identifiant cinq grands problèmes [91] :

— La majorité du génome n'est pas définie. De ce fait nous ne connaissons pas le rôle ni l'influence d'une grande partie de celui-ci et donc son influence sur les *biobricks* qui y sont insérées. Mais aussi nous ne connaissons pas l'influence ni le rôle de certaines parties du programme génétique de certaines *biobricks*.
— Parmi les fonctions connues, nous ne savons pas comment fonctionnent celles-ci, ni les mécanismes impliqués.
— La régulation est encore mal connue, notamment concernant l'influence extérieure, par exemple les enzymes régulatrices.
— De nombreux gènes sont incompatibles entre eux malgré une apparente compatibilité.
— Le châssis cellulaire rend le système instable. En effet, le génome de l'organisme hôte influence le système de synthèse qui y est inséré et peut en modifier le comportement.

3. http ://parts.mit.edu/wiki/index.php/UTAustin200
4. http ://2010.igem.org/Team :ULB-Brussels

Un assemblage de *biobricks* ne doit donc pas seulement viser à produire un comportement donné, il est aussi nécessaire lors de leur assemblage de prendre en compte les possibles interactions du châssis et de l'environnement sur l'assemblage. De plus il est nécessaire d'assembler des *biobricks* compatibles biologiquement entre elle, ce qui nécessite une connaissance plus précise de celles-ci et des éléments les composants. Une telle approche nécessite donc de rechercher parmi plusieurs assemblages permettant un comportement donné, celui dont les composants sont biologiquement compatibles entre eux et avec le châssis, tout en prenant en compte les interactions possibles de l'environnement.

Châssis cellulaires

La notion de châssis est un terme récent, servant à désigner le support cellulaire dans lequel est introduit un génome de synthèse. Il peut autant s'agir d'un organisme vivant que d'un environnement sans cellule. Le châssis cellulaire étant une des contraintes de la biologie synthétique, plusieurs supports cellulaires sont utilisés pour répondre à ce problème :

Les vecteurs bactériens

Cette méthode consiste à utiliser un organisme généralement procaryote afin de multiplier l'*ADN* introduit en même temps que le génome classique. Grâce à cette méthode la cellule va obtenir les fonctionnalités de l'*ADN* introduit en plus des siennes. Parmi ces vecteurs, certains ont été étudiés en détail en biologie et sont donc utilisés de façon récurrente en biologie synthétique :

— *Escherichia coli* est un organisme *modèle* pour les procaryotes étudiés par une large communauté. De ce fait, la connaissance biologique sur cet organisme est très poussée, son génome étant parfaitement séquencé [22]. Ce qui en fait une cible privilégiée pour la biologie synthétique.

— *Bacillus subtilis* est aussi très simple à mettre en culture notamment grâce à sa capacité à vivre dans des conditions extrêmes. De plus certaines de ces propriétés comme la production de chitine en font un organisme fonctionnellement intéressant.

— Cas particulier, la *levure* est une famille eucaryote, cousine des champignons. Néanmoins, cette unicellulaire se multiplie de la même façon que les procaryotes

ce qui en simplifie l'utilisation. Les levures font partie des premiers organismes génétiquement modifiés.

La cellule minimale

La *cellule minimale* [79][54] est une cellule à laquelle on a laissé uniquement les portions de son génome lui permettant de survivre, de ce fait on limite les risques d'interférence de ce génome sur le génome de synthèse qui lui est injecté. Le développement d'un tel châssis a deux objectifs : d'une part, mieux comprendre le fonctionnement cellulaire, et définir les éléments minimaux nécessaires à la survie de la cellule, et d'autre part, réduire les risque d'interaction de la cellule sur le système de synthèse qui lui est intégré.

La culture sans cellule

La *culture sans cellule* ("free cell") [70] est une technique qui permet en mettant le brin d'ADN dans un milieu de culture adapté d'obtenir une multiplication de l'ADN. Cette méthode a pour avantage de ne pas dépendre d'un organisme vivant et donc de ne pas être soumise à des variations biologiques mais uniquement à des variables chimiques facilement contrôlables.

Finalement, la biologie de synthèse s'oriente vers une structuration des systèmes vivants en couches d'abstractions successives, partant de la séquence comme base, couvertes par le concept de *biobrick*, lui-même inclus dans un châssis correspondant à une cellule, l'ensemble de ces cellules interagissant potentiellement entre elles pour former un système biologique plus complexe. Une telle approche est similaire à celle mise en place pour structurer l'informatique comme le remarque R. Weiss (figure 4).

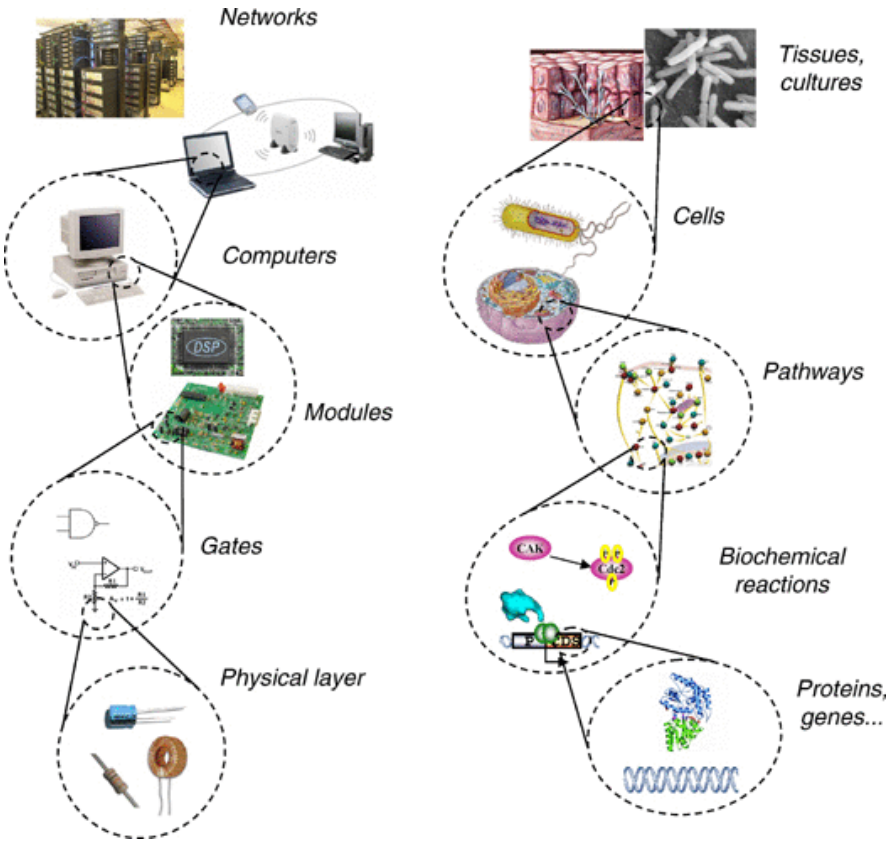


Figure 4 – Une possible structuration du vivant en couches d'abstractions telle que proposée par l'équipe de R. Weiss[4], faisant le parallèle avec les couches d'abstraction propres aux systèmes informatiques.

Afin de couvrir cette structure en couches, l'informatique a mis en place une tour de langages permettant de programmer chaque niveau d'abstraction. Une telle approche serait donc une réponse adéquate en biologie de synthèse, et est reprise par les différents outils disponibles en biologie de synthèse, chacun se plaçant à un niveau d'abstraction donné. Dans le chapitre suivant, nous introduirons donc les travaux relatifs (Chapitre 1) aux langages dédiés à la biologie systémique et synthétique afin de placer nos travaux et leur apports dans le contexte actuel. Nous introduirons dans un second chapitre le langage **Gubs** (Chapitre 2), dont la sémantique est fondée sur la logique hybride. Dans le troisième chapitre nous détaillerons les principes de compilation fondés sur des règles de réécriture ainsi que l'outil **Ggc** (Chapitre 3), en les illustrant par un ensemble d'exemples (Chapitre 4).

Première partie

Gubs, un langage de description comportementale

Chapitre 1

Langages et outils de conception de la biologie de synthèse

État de l'art : La biologie de synthèse, bien qu'émergente, possède un ensemble d'outils d'ingénierie à sa disposition. Afin de proposer un langage répondant aux problématiques actuelles, nous proposons ici une présentation par l'exemple des outils disponibles pour la biologie de synthèse en les divisant en deux grandes catégories complémentaires : d'une part les *CAD* et d'autre part les langages dédiés. Les CAD servent à décrire l'organisation des systèmes et les langages décrivent les interactions entre composants de ces systèmes. En effet, une connaissance de ces outils est nécessaire pour appréhender le positionnement de **Gubs** dans cet environnement ainsi que les problématiques auxquelles nous cherchons à répondre. Nous introduirons de plus l'idée d'un changement de paradigme afin de permettre un passage à l'échelle dans la conception de systèmes biologiques. Ce passage à l'échelle reposant sur l'idée d'une abstraction des composants du vivant comme proposée par Ron Weiss [4].

1.1 Introduction

L'informatique est depuis de nombreuses années liée étroitement à la biologie, apportant des outils afin d'automatiser, de spécifier et d'analyser les systèmes biologiques, qu'il s'agisse par exemple d'outils de modélisation des réseaux, de simulation du comportement (par calcul d'un ensemble d'équations différentielles par exemple), de comparer des séquences et de trouver leur emplacement sur un chromosome par alignement (en calculant le pourcentage de différence), de visualiser graphiquement l'agencement de composants biologiques dans une séquence (par modélisation des chromosomes ou plasmides) ou encore de modéliser l'appontage de protéines sur une séquence. Nous allons ici nous intéresser plus particulièrement aux outils de modélisation et de simulation des systèmes biologiques permettant de faciliter la sélection de composants biologiques qu'il s'agisse de langages dédiés au domaine de la biologie des systèmes ou d'environnements de développement (CAD) reposant sur de tel langages.

Une partie de ces langages sont à l'origine utilisés pour modéliser les interactions entre processus. Néanmoins, l'analogie entre processus et système biologique permet de les appliquer à la modélisation des interactions entre composants biologiques à l'échelle moléculaire comme par exemple les interactions entre protéines [85, 45, 38]. En effet, chaque composant biologique peut être vu comme un processus ayant un fonctionnement autonome et une interaction avec un ensemble d'autres processus correspondant aux autres composants, cette notion peut être passée à l'échelle afin de définir un système biologique comme étant lui-même un processus ayant une interaction avec d'autres systèmes, et ce jusqu'à l'échelle d'un organisme, voire d'une colonie d'organismes.

Une autre approche est fondée sur la logique afin de formaliser les propriétés temporelles d'un système biologique [28] par une formalisation dans une logique adaptée à partir des réseaux protéiques de ces systèmes. Afin de compléter ces outils, nous proposons de placer **Gubs** en amont de ceux-ci en proposant un langage permettant de représenter le comportement d'un système biologique et de trouver automatiquement les composants biologiques correspondant à un tel comportement. Une fois les composants correspondants trouvés, ceux-ci pourront alors être utilisés par les outils existants, qu'il s'agisse de les organiser en séquences, de simuler le

système biologique inféré, ou de le modéliser.

Afin de positionner **Gubs** dans le paysage des outils pour la biologie de synthèse, nous proposons de décrire plusieurs outils représentatifs de la biologie de synthèse. Pour une meilleur lisibilité, nous identifions deux grandes catégories d'outils de conception (voir figure 1.7) : les outils de simulation et de modélisation, et les langages [81][102] de programmation biologique pour la biologie des systèmes ou la biologie de synthèse. Dans une première section, nous décrirons la variété des outils CAD pour la biologie de synthèse, permettant la simulation et la modélisation de systèmes biologiques, mais aussi leur structure, permettant ainsi un assemblage en laboratoire plus aisé, nous nous intéressons ici aux langages de description des systèmes biologiques, permettant d'automatiser certains processus facilités par ces CAD. Ces CAD reposent sur des grammaires de description des systèmes biologiques, correspondant pour la plupart à deux grands langages considérés comme des standards en biologie de synthèse : SBOL et SBML. Dans la seconde section nous présenterons plusieurs langages dédiés à la biologie des systèmes et à la biologie de synthèse. GEC [84] est le premier langage développé pour la biologie synthétique, là où Proto [15][14] est le plus récent proposant des principes de compilations. Nous finirons en introduisant Kappa [47], un langage pour la biologie systémique utilisé en biologie de synthèse. Les langages pour la biologie synthétique se fondent sur les langages développés en électronique tels que VHDL [101], reprenant l'idée de modules servant d'abstraction à une composition de briques élémentaires définies pour être composables et réutilisables entre elles. Il s'agit de descriptions structurelles des systèmes biologiques décrivant structurellement chaque brique élémentaire, voir chaque module.

1.2 Environnements CAD

Dans cette section nous étudierons les environnements de développement CAO (CAD) [1] [2] [3] [4], en se fondant sur un ensemble d'exemples allant de la simulation et la modélisation d'un système entier à son assemblage en laboratoire : TinkerCell [34] qui

1. http ://www.sbolstandard.org
2. http ://www.celldesigner.org
3. http ://www.tinkercell.com
4. http ://www.genocad.org

inclut des outils de simulation et utilise une approche d'assemblage orientée réseau, GenoCAD [26], un environnement plus orienté description de séquence, Geneious [5] permettant l'analyse de séquences d'*ADN*, et GeneDesign [6] permettant de définir automatiquement les manipulations à effectuer en laboratoire pour l'assemblage de séquences.

1.2.1 TinkerCell

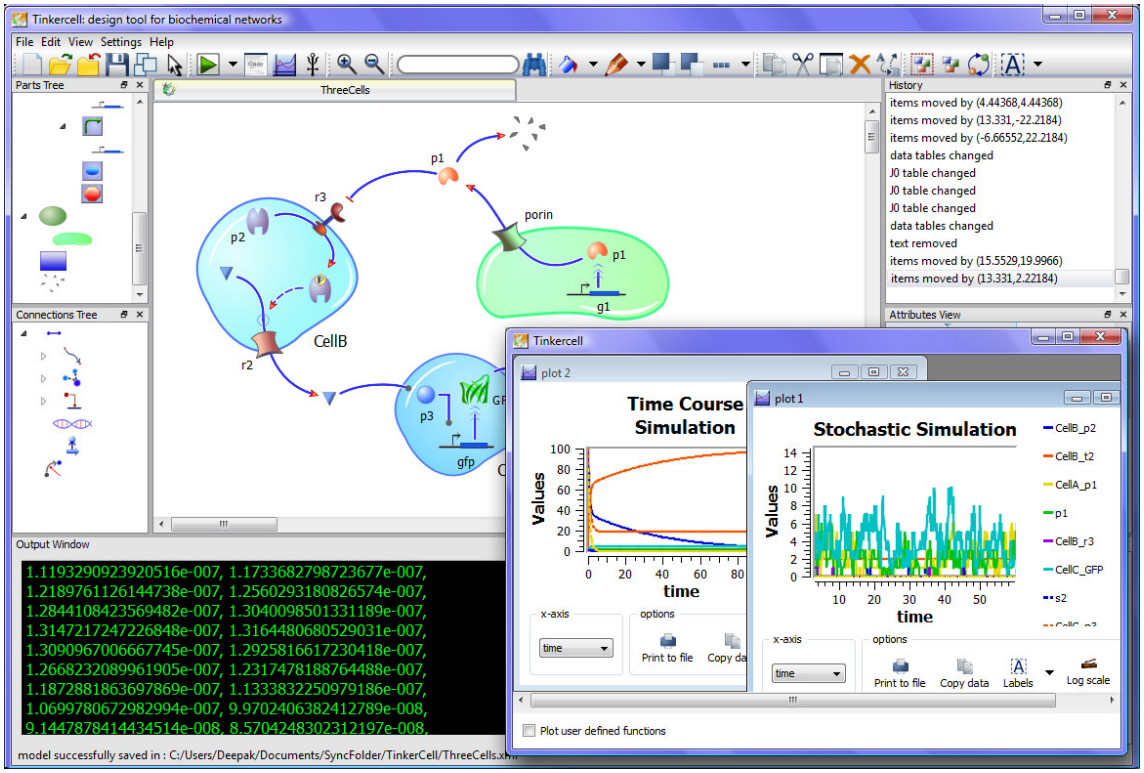


Figure 1.1 – Interface de TinkerCell

TinkerCell (Figure 1.1) est une plate-forme d'ingénierie pour la construction et le test de circuits cellulaires. TinkerCell est une plate-forme extensible pour l'édition et la simulation de réseaux cellulaires. Les utilisateurs peuvent utiliser le logiciel à différents niveaux, graphiquement ou via une console interactive. Bien que l'interface principale soit visuelle, les programmeurs peuvent ajouter de nouvelles fonctionnalités en écrivant des programmes personnalisés en C ou Python. TinkerCell est conçu

5. http ://www.geneious.com/
6. http ://www.genedesign.org

pour intégrer les informations de base de données tel que les biobricks, ainsi les informations des modèles contiennent des constantes de vitesse, des séquences de gènes et les forces de promoteur. Les réseaux peuvent être « modulaires » et reliés entre eux. TinkerCell est multiplateforme et écrit en C ++. Une console de Python est prévue pour le contrôle interactif.

1.2.2 GenoCAD

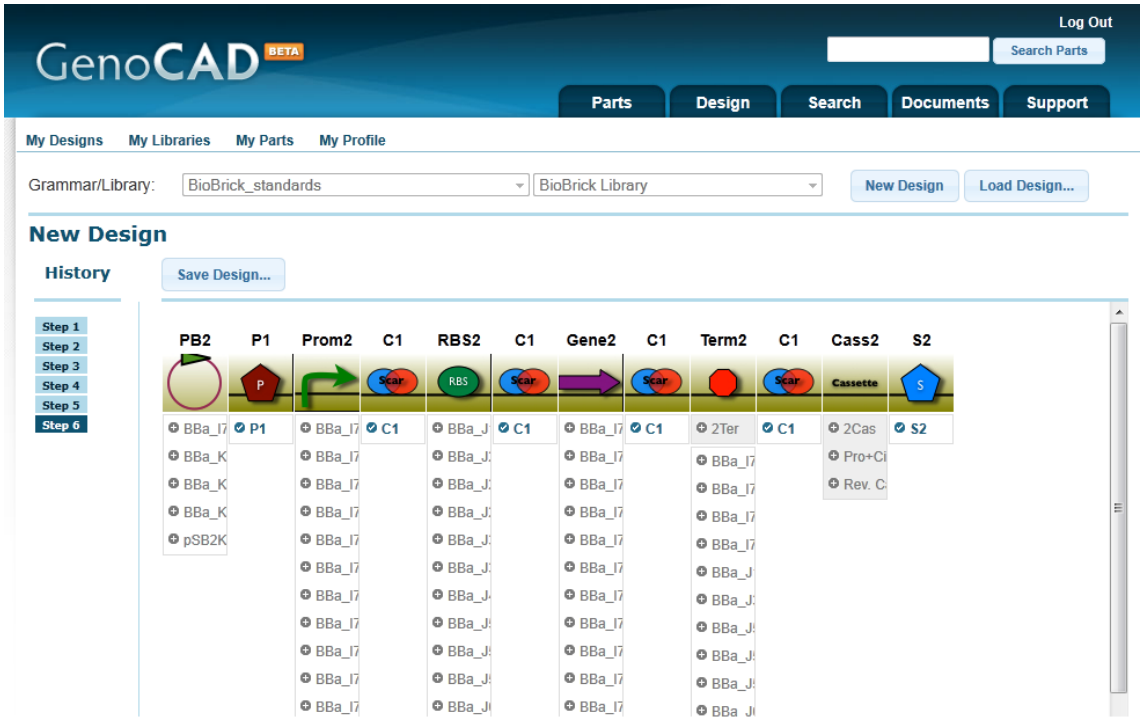


Figure 1.2 – Interface d'utilisation en ligne de GenoCAD

GenoCAD(Figure 1.2) est une application Web guidant les utilisateurs à travers la conception par un assemblage de « parts » de systèmes génétiques. GenoCAD utilise des grammaires afin de formaliser les stratégies de conception pour les systèmes génétiques synthétiques. Cette approche fournit une voie à l'organisation des bibliothèques de pièces génétiques en fonction de leurs fonctions biologiques. Il fournit également un cadre pour la conception systématique de nouvelles constructions génétiques compatibles avec les principes de conception exprimés dans la grammaire. En utilisant des algorithmes d'analyse, GenoCAD permet la vérification des constructions existantes. A l'heure actuelle, GenoCAD a proposé une preuve de concept montrant

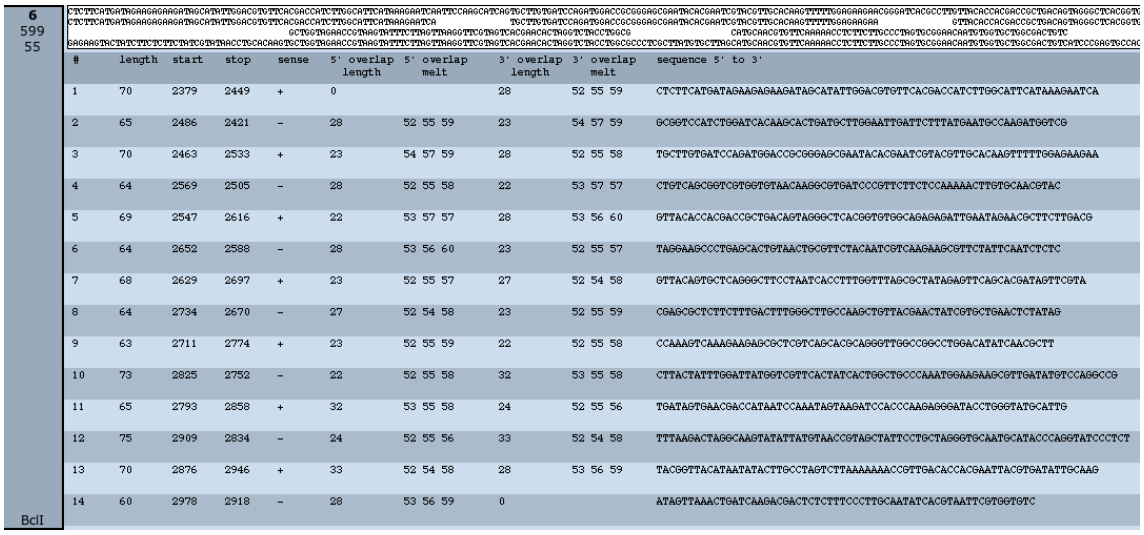
la possibilité de simuler les systèmes générés par celui-ci. Néanmoins, comme toute simulation, elle nécessite des données biologiques précises que la base de données de GenoCAD ne possède pas actuellement.

1.2.3 Geneious

Figure 1.3 – Interface d'analyse des séquences de Geneious

Geneious(Figure 1.3) est un outil propriétaire multifonction permettant de travailler au niveau de la séquence d'un système biologique. Il permet l'analyse et la comparaison de séquences entre elles, leur assemblage et leur modélisation. De plus, Geneious permet de faciliter les manipulations de clonage par modélisation des « primers » et simulation de PCR (La PCR est une technique qui permet de repérer un fragment d'ADN ou de gènes précis, puis de le multiplier rapidement.). Il se fonde sur un ensemble de bases de données extraites du web.

1.2.4 GeneDesign



#	length	start	stop	sense	5' overlap length	5' overlap melt	3' overlap length	3' overlap melt	sequence 5' to 3'
1	70	2379	2449	+	0		28	52 55 59	CTCTTCATGATAGAAGAGAAGATAGCATATTGGACGTGTTCACGACCATCTTGGCATTCATAAAGAATCA
2	65	2486	2421	-	28	52 55 59	23	54 57 59	GCGGTCCATCTGGATCACAAGCACTGATGCTTGGAATTGATTCTTTATGAATGCCAAGATGGTCG
3	70	2463	2533	+	23	54 57 59	28	52 55 58	TGCTTGTGATCCAGATGGACCGCGGGAGCGAATACACGAATCGTACGTTGCACAAGTTTTTGGAGAAGAA
4	64	2569	2505	-	28	52 55 58	22	53 57 57	CTGTCAGCGGTCGTGGTGTAACAAGGCGTGATCCCGTTCTTCTCCAAAAACTTGTGCAACGTAC
5	69	2547	2616	+	22	53 57 57	28	53 56 60	GTTACACCACGACCGCTGACAGTAGGGCTCACGGTGTGGCAGAGAGATTGAATAGAACGCTTCTTGACG
6	64	2652	2588	-	28	53 56 60	23	52 55 57	TAGGAAGCCCTGAGCACTGTAACTGCGTTCTACAATCGTCAAGAAGCGTTCTATTCAATCTCTC
7	68	2629	2697	+	23	52 55 57	27	52 54 58	GTTACAGTGCTCAGGGCTTCCTAATCACCTTTGGTTTAGCGCTATAGAGTTCAGCACGATAGTTCGTA
8	64	2734	2670	-	27	52 54 58	23	52 55 59	CGAGCGCTCTTCTTTGACTTTGGGCTTGCCAAGCTGTTACGAACTATCGTGCTGAACTCTATAG
9	63	2711	2774	+	23	52 55 59	22	52 55 58	CCAAAGTCAAAGAAGAGCGCTCGTCAGCACGCAGGGTTGGCCGGCCTGGACATATCAACGCTT
10	73	2825	2752	-	22	52 55 58	32	53 55 58	CTTACTATTTGGATTATGGTCGTTCACTATCACTGGCTGCCCAAATGGAAGAAGCGTTGATATGTCCAGGCCG
11	65	2793	2858	+	32	53 55 58	24	52 55 56	TGATAGTGAACGACCATAATCCAAATAGTAAGATCCACCCAAGAGGGATACCTGGGTATGCATTG
12	75	2909	2834	-	24	52 55 56	33	52 54 58	TTTAAGACTAGGCAAGTATATTATGTAACCGTAGCTATTCCTGCTAGGGTGCAATGCATACCCAGGTATCCCTCT
13	70	2876	2946	+	33	52 54 58	28	53 56 59	TACGGTTACATAATATACTTGCCTAGTCTTAAAAAAACCGTTGACACCACGAATTACGTGATATTGCAAG
14	60	2978	2918	-	28	53 56 59	0		ATAGTTAAACTGATCAAGACGACTCTCTTTCCCTTGCAATATCACGTAATTCGTGGTGTC

BclI

Figure 1.4 – Interface de sélection des oligos de GeneDesign.

GeneDesign (Figure 1.4) est une suite d'algorithmes permettant aux utilisateurs de modifier plusieurs caractéristiques de séquences codantes de protéines, y compris l'usage des codons et de RBS. Il va alors générer une liste d'oligonucléotides (Courts segments d'ARN ou ADN, longs de quelques dizaines de nucléotides obtenus par synthèse chimique, sous forme de simple brin.) et une feuille de route pour l'assemblage de la séquence par PCR. Il est écrit en Perl et disponible sur internet ou en installation locale.

1.2.5 Biocham

Biocham [28](Figure 1.5) est un CAD permettant de modéliser les systèmes biochimiques. Il se fonde sur un ensemble de langages qui formalisent les propriétés temporelles d'un système biologique en permettant une modélisation cinétique et stochastique des systèmes ainsi que leur simulation par une formalisation en logique temporelle des réseaux protéiques. De plus Biocham permet d'inférer automatiquement les valeurs de certains paramètres biologiques du système à partir de la description de comportements expérimentaux. En résumé, l'objectif de tels langages est de définir des modèles permettant une prédiction du comportement du système en fonction d'un ensemble de paramètres d'expériences, et ainsi, valider des résultats expérimentaux.

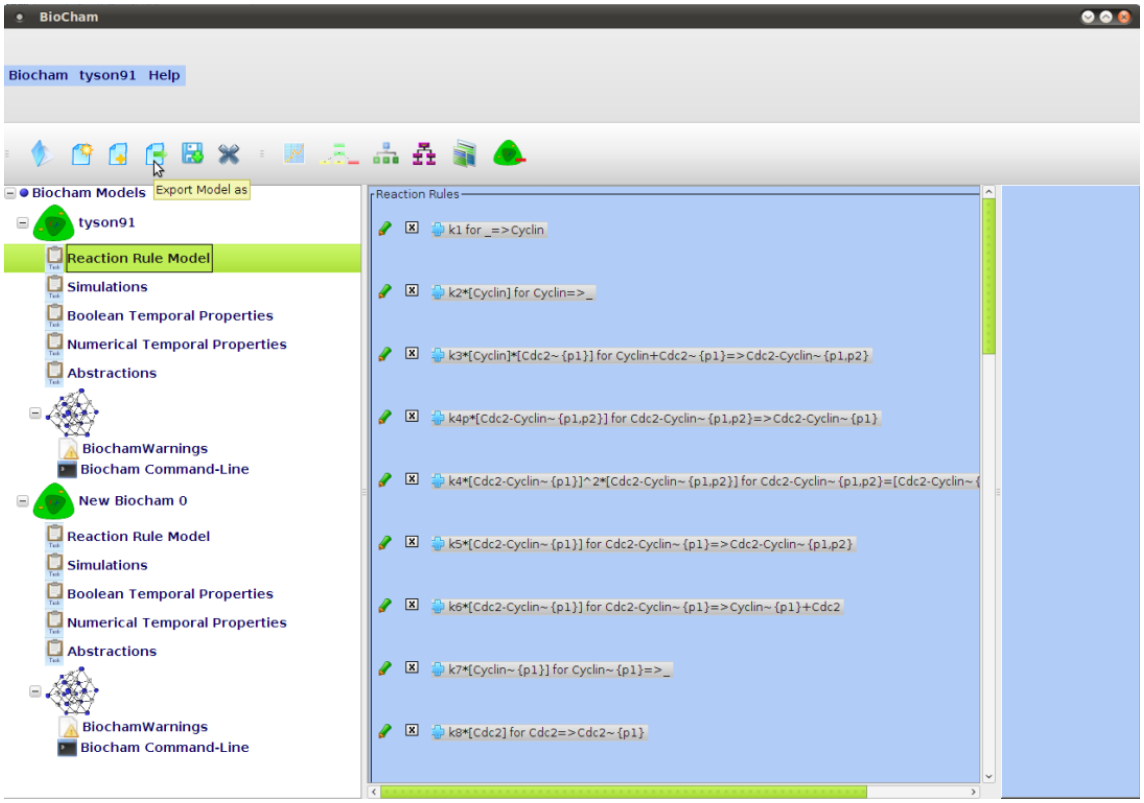


Figure 1.5 – Interface de description des règles de réactions de Biocham.

1.3 Langages

En biologie de synthèse, les langages de description structurelle [41, 83, 19] permettent de définir des séquences du génome comme "bien-formées" grâce à des grammaires ; et ceci de façon modulaire et hiérarchique. Bien que la description de la séquence soit nécessaire, le programmeur doit déjà anticiper le comportement du dispositif qu'il souhaite concevoir. En outre, le concept de comportement ne se retrouve pas dans le programme alors qu'il en a initialement motivé la conception. Dans **Gubs**, la conception est motivée par une description de comportement et la sélection des séquences est reportée à la phase de compilation. De plus, la taille de la description structurelle est également soumise à une explosion combinatoire lorsque la complexité des systèmes programmés augmente.

Les langages de programmation amorphes ont également été étudiés pour spécifier les composants biologiques à l'échelle de colonies de cellules, ici considérées comme

support de calcul possible pour un programme. J. Beal [16] a démontré une preuve de concept de cette approche dans Proto, montrant la faisabilité d'une chaîne de compilation automatisée. Dans **Gubs**, la chaîne de compilation est fondée sur un ensemble de règles de réécritures dont la correction a été formellement prouvée en s'appuyant sur une sémantique décrivant les contraintes induites par les systèmes ouverts.

Le développement d'un langage pour les systèmes biologiques implique à l'heure actuelle de considérer plusieurs inconnues dues à leur dimension ouverte : manque de connaissances à propos de toutes les interactions dans les circuits biologiques et définition imprécise des conditions initiales. Nous connaissons seulement le résultat d'une chaîne d'effets. Alors, la contrainte majeure pour la programmation de systèmes ouverts semble être : comment fournir un langage expressif pour décrire la dynamique de ces systèmes, mais assez simple pour saisir l'essence des questions biologiques dans un programme assez court afin de permettre la programmation de grands systèmes biologiques avec un programme humainement réalisable ?

Dans le futur, la conception en biologie de synthèse nécessitera sûrement différentes couches de programmation fondées sur différents paradigmes adressant chaque niveau d'un système biologique (figure 4). Dans une tour de langages, partant d'un langage décrivant les interactions au niveau des colonies cellulaires, utilisant un langage de programmation amorphe tel que Proto [16] ou un langage de description des systèmes dynamiques possédant des structures dynamiques comme MGS [60], et terminant par une description structurelle programmée dans un langage de grammaire tel que GEC[84] ou GenoCAD[41], **Gubs** occupe le niveau intermédiaire, permettant la description du comportement à l'échelle d'une entité cellulaire.

Dans cette section, nous commençons par introduire les langages de description des systèmes de synthèse, puis nous présentons un langage de modélisation dédié à la biologie des systèmes, utilisé à présent aussi en biologie de synthèse pour finir sur deux langages dédiés spécifiquement à la biologie de synthèse.

1.3.1 Langages de description des systèmes

SBOL et SBML

SBOL [56](Synthetic Biology Open language) est un langage développé en 2008 s'inspirant de VHDL afin de représenter de façon abstraite les systèmes de synthèse à tous les niveaux, des « parts » génétiques telles que les biobricks, jusqu'aux systèmes biologiques de synthèse eux mêmes. Une application immédiate est de faciliter l'extraction de ces assemblages à partir de la littérature.

SBML [68](Systems Biology Markup Language) est quant à lui un format de représentation des modèles biologiques basé sur le langage XML représentant leurs interactions et décrivant en détail la structure de tous les éléments du système (sites d'appontages, mutations, détail des séquences...). SBML permet de représenter différentes classes de phénomènes biologiques tels que les réseaux métaboliques, les voies de signalisation cellulaires, les réseaux de régulation, les maladies infectieuses, et bien d'autres.

1.3.2 Biologie des systèmes

Kappa

Le langage Kappa est un langage de règles permettant la modélisation des interactions au sein d'un réseau protéique. Il permet de décrire en détail un ensemble de composants biologiques en définissant pour chacun des règles d'assemblage et de potentialité d'action permettant par la suite une modélisation par simulation stochastique du système décrit par cet ensemble de règles.

Pour ce faire, Kappa définit un ensemble d'agents ayant chacun un ensemble de site d'interactions. Pour chaque site, un ensemble de règles d'interaction avec les sites d'autres agents peuvent être créés. Chacune de ces règles va posséder une probabilité de réaction influant sur le nombre de réactions par instant de temps dans le système. Les interactions peuvent aller de l'assemblage de plusieurs agents à la création de nouveaux agents en réaction à certaines interactions. Chaque agent va ensuite posséder une certaine population lors de la simulation influant sur l'évolution du système. Le langage Kappa a entre autre été utilisé lors de la compétition IGEM par l'équipe d'Édinbourg en 2010 afin de décrire les interactions au sein de leur

système (figure 1.6)

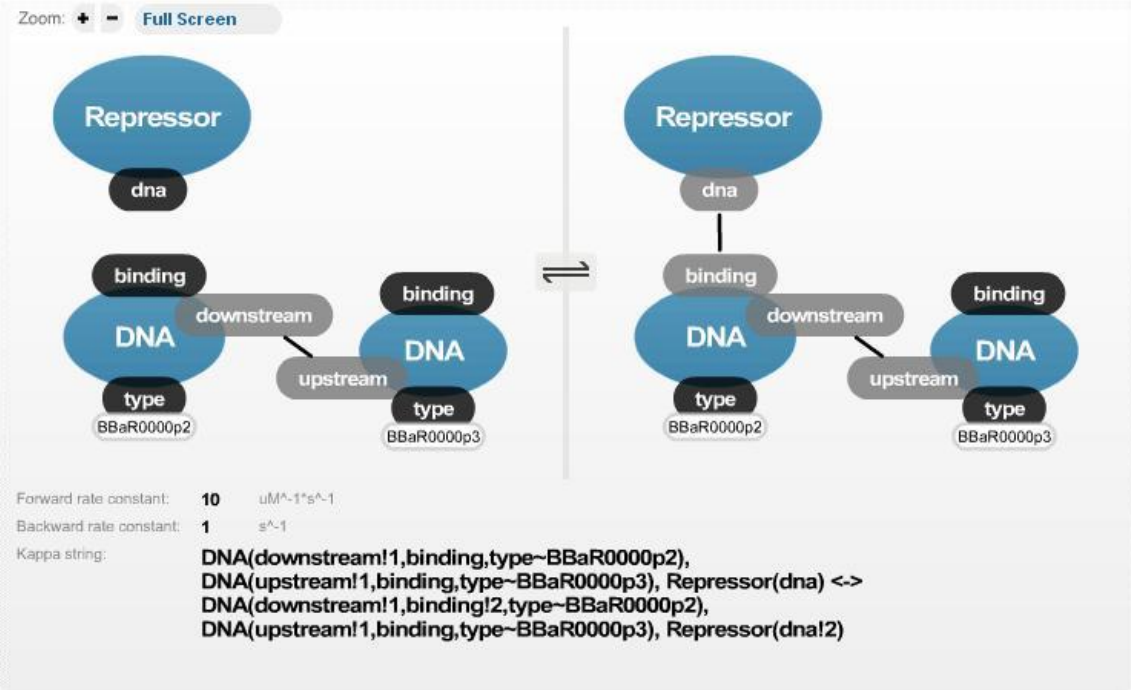


Figure 1.6 – Cet exemple décrit un répresseur se liant à un site de liaison disponible sur un promoteur.

1.3.3 Biologie de synthèse

Gec

Gec est l'un des premiers langages développés pour la Biologie synthétique. Il a été développé par Michael Pedersen et permet de créer des systèmes biologiques avec un langage proche du C. L'intérêt de ce langage est de permettre, une fois les composants à utiliser définis, de coder le système comme n'importe quel logiciel informatique.

Le principe de Gec repose sur deux choses : d'un côté une base de données de composants biologiques que l'utilisateur va définir, décrivant l'ensemble des propriétés de chaque composants biologiques qu'il souhaite utiliser jusqu'à leur séquence d'ADN. Cette base de données est donc rapidement sujette à erreur si les « parts » ainsi définies sont nombreuses et complexes. D'un autre coté, Gec propose de décrire la structure d'un système biologique comme l'assemblage d'un ensemble des éléments

décrits dans la base de données, décrivant leurs interactions et leur agencement.

Proto

L'idée de la compilation en biologie [15, 16] a déjà fait l'objet de plusieurs travaux proposant d'utiliser les portes logiques biologiques [48, 113] comme composants atomiques.

Proto est un langage de programmation spatiale, dont l'utilisation a été détournée par J. Beal afin de décrire les interactions au sein de populations cellulaires de façon abstraite. De plus, Proto propose actuellement des méthodes de compilation semi-automatiques ainsi qu'une optimisation et une simulation des circuits génétiques créés. Dans un premier article [14] J. Beal à en effet introduit la compilation de Proto par une application de règles de réécriture, introduisant cette preuve de concept par l'exemple de la compilation par l'application des règles à la main du *Band-detector* [13] proposé par Ron Weiss que nous reprenons dans le chapitre 4. Dans un article plus récent [15] il suggère la possibilité d'appliquer de manière automatique ces règles et d'appliquer des optimisations au système biologique inféré afin d'obtenir un système biologique similaire à celui d'origine.

1.4 Vers une description comportementale

Ce tour d'horizon des outils présents en biologie systémique et de synthèse nous permet de faire plusieurs constats :

- De nombreux outils permettent une description structurelle des systèmes biologiques ;
- ces outils permettent une modélisation de ces circuits ;
- une simulation de ces circuits nécessite que ceux-ci soient bien définis et se fait sur des circuits dont les éléments sont connus ;
- l'assemblage se fait au niveau de l'ADN et au maximum au niveau des biobricks ;
- l'assemblage n'est pas automatique, mais semi-automatisé.

Pour conclure cette analyse, la biologie de synthèse possède de nombreux outils permettant une description structurelle de bas niveau de ses composants « atomiques » (ADN, biobrick) ainsi que leur simulation et une facilitation de leur assemblage. Néanmoins, dans l'idée d'une augmentation en taille et en complexité des systèmes

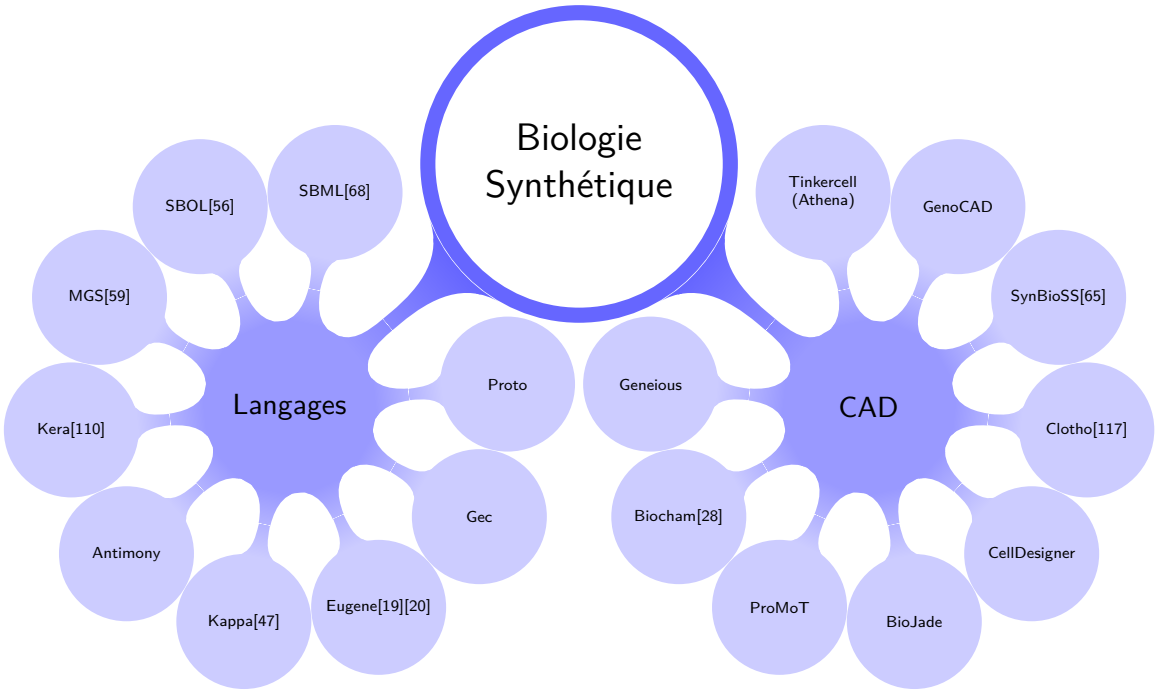


Figure 1.7 – Vue d'ensemble non exhaustive d'une collection d'outils utilisés en biologie synthétique et en biologie systémique.

biologiques de synthèse, une description structurelle d'un système à l'échelle du génome, au vue du nombre d'éléments à prendre en compte, sera sujette à erreur et donc non sûre. Or, la sureté et la sécurité des systèmes biologiques est un des points cruciaux pour la biologie de synthèse [99]. Ce constat nous permet de définir les besoins actuels en biologie de synthèse comme étant un langage de haut niveau abstrayant ceux existant de bas niveau. De plus ce langage doit permettre de décrire de grands systèmes biologiques et permettre un assemblage automatique de composants complexes (assemblage de systèmes biologiques simples) et veiller à la sureté et la sécurité de ces assemblages.

Ce passage à l'échelle est une démarche qui a déjà vue son application dans la synthèse de circuits électroniques (VHDL[101], Verilog[108]) en proposant une approche de description comportementale permettant de spécifier à l'outil de création de circuit le comportement attendu du système. Ce comportement s'exprime au travers de contraintes que doit respecter celui-ci face à des stimuli ou signaux.

Le passage à de grands systèmes biologiques nécessite donc un changement de paradigme : proposer un langage de description comportementale des grands systèmes biologiques de synthèse. Nous allons donc proposer un langage de description comportementale. On va décrire le comportement de l'ensemble et/ou d'une partie du système. La description du comportement fait appel à la nécessité d'identifier un ensemble de primitives se rapportant à un modèle de calcul. Ce modèle, en lui-même, permet d'exprimer le comportement attendu des différentes unités.

Dans notre cadre, il faut insister sur la difficulté de définir un modèle de calcul biologique garantissant la fiabilité du résultat. De ce fait, la relation entre modèle et vivant est inversée, l'idée ici est de définir un modèle de comportement puis de créer un système biologique répondant aux contraintes du modèle, au lieu d'un modèle définissant le vivant.

L'approche comportementale permet la synthèse de différents systèmes équivalents sur le plan comportemental, bien que différents sur celui structurel.

Elle constitue aussi un cadre privilégié pour répondre aux problèmes de « perturbation » entre composants, d'adaptation d'une fonction à un organisme et pour renforcer la sureté de fonctionnement dans la mesure où l'analyse peut s'effectuer sur la description du comportement en lui-même.

Pour illustrer cette approche, prenons l'exemple d'une inhibition, si le comportement recherché est une inhibition d'un gène A, plusieurs approches permettent d'obtenir ce comportement :

— Une inhibition par un autre gène.
— Par dégradation de l'ARN par un enzyme.
— Par riboswitch, une inhibition par un repli du brin d'ARN sur lequel se trouve la séquence codante du gène, empêchant le ribosome de traduire l'ARN.

Dans le cas d'un langage de description comportementale, ce sera le compilateur qui choisira automatiquement dans une base de données de composants aux comportements prédéfinis lequel a le comportement le plus optimal pour répondre au comportement défini par le modèle en suivant des conditions pouvant ou non être choisies par l'utilisateur. A cette fin, nous introduisons **Gubs**, un langage de description comportementale. A l'étape de compilation, un programme **Gubs** sera alors substitué par un ensemble de composants extraits d'une base de données. Cette

base de données sera elle-même composée de composants biologiques auxquels est associée une description en **Gubs** de leur comportement.

Chapitre 2

Gubs : un langage de description comportementale pour les systèmes ouverts dédiés à la biologie de synthèse.

Résumé : Dans ce chapitre nous introduisons le langage **Gubs** dédié à la biologie de synthèse. Ce langage permet la spécification comportementale de systèmes biologiques en proposant une syntaxe répondant aux contraintes biologiques sous forme de relations causales avec une prise en compte des interactions environnementales. La sémantique de **Gubs** se fonde sur la logique multimodale hybride permettant d'exprimer un comportement qui doit être unifié sur une trace. Nous commencerons par décrire informellement le langage puis nous introduirons sa syntaxe. Dans une troisième partie nous décrirons la sémantique sous-jacente. Nous finirons enfin par quelques exemples de programme **Gubs**.

2.1 Présentation

Le langage **Gubs** permet la description des comportements d'un système biologique validé par un ensemble d'observations correspondant à des expériences. A cette fin, le comportement est décrit par un ensemble de *relations causales* (aussi appelées plus simplement *causes* dans ce manuscrit) formant une chaîne d'évènements pertinents à observer le définissant. Finalement, Un programme **Gubs** correspond donc à l'observation des comportements de l'expérience considérés comme clés par l'utilisateur, et nous permet d'ignorer ceux inconnus ou secondaires.

À partir de cette définition, l'observation se formalise par la notion de *point d'observation*. Elle s'inscrit en ce sens dans un protocole d'expérience où l'on évalue la réalisation d'un comportement par sa présence. L'utilisateur définit alors un ensemble d'éléments du programme dont il souhaite suivre en particulier le comportement, tel un ensemble de marqueurs de trace d'expérience.

De plus, **Gubs** devant permettre la description de systèmes ouverts, nous avons introduit la notion d'interaction avec l'environnement. Nous avons fait le choix de différencier les interactions de l'environnement de celles propres au système afin de clairement représenter leur influence sur tout ou partie des relations causales composant le système. Un programme **Gubs** se traduit finalement comme un ensemble de relations causales représentant le système biologique, ces relations causales étant contraintes par un ensemble d'interactions avec l'environnement. Les éléments interagissant au sein des relations causales vont correspondre aux éléments biologiques du système dont nous souhaitons observer la présence ou l'absence. Ce modèle nous permet de faire abstraction de l'ensemble des éléments du système pour ne se focaliser que sur ceux ayant un rôle à jouer dans le comportement à observer. Ces éléments biologiques peuvent être raffinés par un ensemble d'attributs quantitatifs ou qualitatifs spécifiant leur état avec plus de précision.

Afin d'organiser topologiquement les comportements du système biologique, ces relations causales sont compartimentées afin de décrire plusieurs ensembles de comportements indépendants. Une telle représentation permet de se rapprocher de la structure compartimentée propre à un système biologique ; qu'il s'agisse de cellules indépendantes ou de compartiments cellulaires.

Pour finir, un ensemble de métadonnées peuvent être ajoutées afin de diriger et de faciliter la compilation. Ces métadonnées permettent de spécifier un ensemble de contraintes sur certains éléments biologiques ou sur le système. Elles permettent aussi de spécifier des propriétés pour les compartiments, ou d'informer le compilateur d'un ensemble d'éléments déjà présents dans le système et à prendre en compte lors de la compilation du comportement spécifié, ou encore de spécifier au compilateur une partie de base de données à utiliser en priorité ou à laquelle se restreindre.

Pour résumer, **Gubs** est un langage de spécification de comportements observables dans un contexte de système ouvert biologique.

Dans la suite de ce chapitre, nous fonderons nos exemples sur le programme **Gubs** de la figure 2.1 extrait des travaux de René Thomas [106]. Le détail de ce programme sera expliqué par la suite.

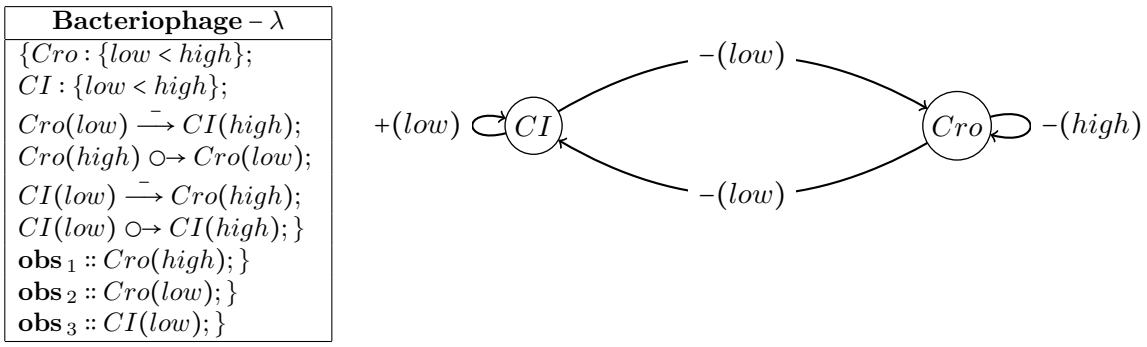


Figure 2.1 – Modèle du réseau de régulation du λ phage à deux états.

2.2 Syntaxe

Dans cette section nous introduirons plus en détail et formellement l'ensemble des propriétés syntaxiques de **Gubs**. Notre présentation partira des éléments atomiques du langage : les agents, puis les relations causales les unissant pour finir par une vue globale d'un programme **Gubs**. La grammaire de **Gubs** est récapitulée à la fin de cette section.

2.2.1 Agents et attributs

Les *agents* représentent les objets biologiques du système que nous décrivons, il s'agit des objets atomiques du langage. Dans la figure 2.1, les agents sont les éléments

biologiques tels que Cro ou CI.

Constantes et variables

Dans **Gubs**, nous distinguons deux types d'agents : *les constantes* et les *variables*, cette différentiation est classique en informatique et nous permet dans un cadre biologique de faire la différence entre les constantes désignant un ensemble d'éléments réels prédéfinis dans un corpus de connaissances ; et les variables représentant une abstraction de ces objets prédéfinis, pouvant potentiellement être substituées par n'importe quelle constante.

En biologie, les constantes correspondent par exemple à une protéine, un promoteur ou un gène d'intérêt. Par exemple, l'agent *PLacI* correspond au promoteur du gène *LacI*. Une variable peut quand à elle qualifier une classe d'agents biologiques comme par exemple un gène ou une protéine, par exemple *var1* pourra être substitué lors de la compilation par *LacI*.

Par convention, une constante commence par une majuscule (*i.e.*, *LacI*), et une variable commence par une minuscule (*i.e.*, *var1*).

État d'un agent

Les différents états observables des agents vont définir leur différents comportements possibles. A ces états s'associeront des possibilités d'actions sur l'état d'autres agents. Par défaut, un agent possède deux états abstraits : présent ou absent, pouvant correspondre à l'état actif ou inactif d'un gène par exemple. Un composant biologique n'étant pas à proprement parler « présent » ou « absent », afin de définir ces deux états, nous définirons un agent comme actif à partir du moment où il entre en interaction avec le système décrit, et inactif quand il n'a aucune influence sur ce système. On peut voir cette interprétation comme une normalisation du seuil de concentration à partir duquel une protéine va activer un promoteur par exemple. Dans la figure 2.1, les agents sont tous définis comme positifs (les relations causales abstrayant ceux en forme négative).

Par convention, un agent présent sera décrit par son nom seul (*i.e.*, *LacI*), et un agent n'étant pas au seuil d'activation sera décrit par la négation de son nom (*i.e.*, $\overline{LacI}$).

Attributs

Néanmoins, un agent biologique ne possède pas simplement deux états, il peut voir sa concentration varier, et son interaction avec le système évoluer en fonction de ce paramètre. Afin de prendre en compte des agents pouvant interagir à différents seuils de leur activité, nous étendons la notion d'état par celle d'*attributs* permettant de caractériser symboliquement l'ensemble des interactions possibles d'un agent. Plus précisément, un attribut qualifie une « capacité » d'activation, définie comme l'ensemble des causes qu'il peut déclencher (ou au déclenchement desquelles il peut participer) dans l'état indiqué par cet attribut.

Dans la figure 2.1, les attributs correspondent aux variables entre parenthèses, par exemple low dans $Cro(low)$, on notera que les attributs ont été définis dans les premières lignes du programme : $Cro : \{low < high\}$

La signification réelle des attributs est une question de convention dépendant de la construction cible (*e.g.*, réaction protéique, réseau de gènes), que nous illustrons par la suite. Par exemple, l'activité de régulation d'un gène peut correspondre à l'observation de différents seuils dans la concentration de l'ARN induisant son activité. À un seuil donné, un gène va réguler un certain ensemble de gènes, et à un autre seuil, sa régulation va s'appliquer à un autre ensemble de gènes (voir figure 2.2). Les différents seuils vont alors correspondre aux différents niveaux d'activité d'un gène, correspondant à différentes activités de régulation [18]. Par exemple, si nous identifions trois différentes activités de régulation pour un gène G, l'état du gène correspondra à trois attributs différents $\{Low, Mid, High\}$ définissant symboliquement trois comportements possibles. Par exemple, $G(Low)$ décrit le fait que l'agent G est dans l'état Low, et prêt à l'activité correspondant à cet attribut. Lorsqu'un seul état est suffisant pour caractériser la capacité d'action d'un agent, l'agent est identifié par sa propre capacité. Ainsi, G seul sans attribut signifie que l'agent G est présent. A l'opposé, $G(\overline{Low})$ signifie que l'état de l'agent diffère de Low ($\overline{G}$ quand un agent n'a qu'une seule propriété). Il est important de noter que ne pas être dans un état défini par un attribut ne signifie pas nécessairement que l'agent est dans un état correspondant à un autre de ses attributs. En effet, dans un système ouvert, l'état d'un agent peut être n'importe lequel et ne pas nécessairement correspondre à un attribut prédéfini.

Pour autre exemple, dans un réseau de signalisation [109], la phosphorylation

d'une protéine *Phos* induit un changement de sa conformation structurelle conduisant a un potentiel de signalisation spécifique ne pouvant arriver dans sa conformation non phosphorylée *UnPhos*.

Afin de décrire les liens existants entre plusieurs attributs d'un même agent, nous définissons deux types de relations entre attributs : Une relation d'ordre, $\prec$, signifiant "possède moins de capacité que" et une inégalité, $\not\approx$, signifiant « possède une capacité différente de ». Ainsi, $Low \prec Mid$ implique que la capacité nécessaire pour l'action au niveau *Mid* inclut la capacité relative au niveau *Low*, en effet, généralement, dans un modèle de régulation génétique [49], l'ensemble des gènes régulés par un certain niveau d'activité le sera aussi par un niveau supérieur. En supposant que *Phos* et *UnPhos* représente respectivement la configuration phosphorylée et non phosphorylée d'une protéine P, nous avons la propriété $Phos \not\approx UnPhos$. De plus, $P(Phos)$ implique implicitement $P(\overline{UnPhos})$. Notons que dans ce cas, utiliser les attributs *Phos* et *UnPhos* ne sert qu'à une meilleure lecture du code, en effet cette définition est équivalente à P et $\overline{P}$.

Formellement, les attributs et les relations entre attributs pour un agent donné seront déclarés en préambule d'un programme comme :

$$G : \{Low \prec Mid \prec High\}, P : \{Phos \not\approx UnPhos\}.$$

Ici, G est défini avec les attributs $Low, Mid, High$ organisés selon la relation d'ordre $\prec$. La relation d'ordre donnée par $\prec$ qualifie un ordre partiel (réflexif, transitif, antisymétrique). Cependant, en pratique elle est souvent utilisée comme un ordre total.

De même, P est défini avec les attributs *Phos* et *UnPhos* décrits comme indépendant par la relation $\not\approx$.

Notons aussi que si aucune relation n'est définie entre attributs, nous définissons les attributs comme un ensemble avec la relation $\not\approx$.

Attributs numériques

Ces attributs peuvent être soit des constantes soit des variables, ou encore des valeurs numériques sur un intervalle donné. Dans le cadre de valeurs numériques,

un agent peut alors être défini pour une valeur précise ou sur un intervalle précis permettant ainsi une approche quantitative en plus de celle qualitative fournie par les attributs variables ou constants. Cette approche quantitative permet de définir une concentration, ce qui nous permet d'écrire $LacI(0.3)$ en définissant $LacI : [0, 1]$, mais aussi de définir une activation selon un certain intervalle de valeurs, par exemple : $Tet[0.1, 0.4]$ qui signifie que l'agent est actif dans cet intervalle de concentration.

Notons qu'une telle représentation numérique peut être aisément discrétisée en un ensemble d'attributs qualitatifs. Par exemple, définir $LacI : [0, 1]$ équivaut à $LacI : \{val_0 \prec val_1\}$, et écrire $LacI(0.3)$ équivaut alors à introduire un nouvel attribut tel que $LacI : \{val_0 \prec val_{0.3} \prec val_1\}$. Une telle substitution est faite à l'étape de compilation, afin de donner plus de latitude à **Ggc** lors de la sélection des substitutions possibles, comme expliqué plus en détail dans le chapitre 3.

Ensembles d'agents

Finalement, la description d'un état d'un agent peut être étendu à une collection d'état d'agents comme suit : $g_1 + \ldots + g_n$, signifiant que tous les agents g_i sont observables à leurs états respectifs simultanément. Ces ensembles d'agents vont ensuite pouvoir interagir par le biais de relations causales les reliant entre eux.

2.2.2 Dépendances comportementales et points d'observation

Causalité

Une dépendance comportementale identifie une relation entre les comportements comme une relation de cause à effet sur les évènements. Fondamentalement, les dépendances doivent définir le contrôle des agents les uns sur les autres. Toutefois, la définition de la causalité doit également aborder l'ouverture d'un système en l'adaptant à ce contexte.

Dans la figure 2.1, les relations causales entre agents sont de plusieurs formes : $\xrightarrow{-}$ dans $Cro(low) \xrightarrow{-} CI(high)$, ou $\circ\!\!\rightarrow$ dans $Cro(high) \circ\!\!\rightarrow Cro(low)$

Une définition historique de la causalité, proposée par Hume [69], est formulée en termes de régularité sur les évènements :« [Nous devons définir] une cause comme étant un objet, suivi par un autre, et où tous les objets similaires au premier sont suivis par des objets similaires au second ». Bien que cette définition caractérise de

manière appropriée la notion de contrôle, l'ouverture du système implique de tenir compte des actions de l'environnement qui modifient éventuellement la chaine de dépendance causale. Par exemple, une activation programmée $G_1 \xrightarrow{+} G_2$ peut être contredite par une inhibition existante $G_3 \xrightarrow{-} G_2$ traitant le même gène cible G_2. D'où le fait que, quand G_1 est actif, il est possible que G_2 ne soit pas actif à cause de la force de régulation de G_3 qui est supérieure à la force de régulation de G_1, ce qui contredit l'activation attendue par une inhibition cachée.

Par conséquent, poussée à la limite, cette considération empêche la possibilité de décrire un comportement causal car toute action programmée peut être interrompue de façon inattendue par un évènement externe. En adaptant la définition de Hume, nous identifions l'action d'une cause par l'apparition de son effet. Si l'effet est observé, la relation de cause à effet est effective, ce qui est différent de l'approche de base envisagée : si la cause est observée, la relation de cause à effet est effective. Cette définition poursuit aussi un objectif pratique consistant à garantir la correction de la compilation même si la réalisation ne produit pas l'effet attendu. Elle permet ainsi de distinguer les principes de la compilation conduisant à un assemblage de composants devant être correct vis-à-vis de la spécification du programme, de son action réelle qui nécessite une sélection appropriée des composants relevant de l'optimisation.

L'observation des effets devient le seul évènement indiquant le déclenchement de dépendances de causalité. En effet, l'observation d'une cause ne peut être considérée comme un indicateur parce que son action peut être interrompue par des évènements extérieurs, c'est le problème de *préemption*. En d'autres termes, la définition proposée de la causalité reflète le fait que le dispositif peut être non fonctionnel en raison d'une intervention externe. Toutefois, la fonctionnalité est toujours spécifiée correctement parce que cette suite est prise en compte dans la définition de la causalité validée par l'observation des effets. De plus, comme aucune cause externe à la description est supposée déclencher les effets des dépendances pour la nouvelle fonctionnalité, la *surdétermination* par des causes inconnues est censée être empêchée, assurant que le programme est le seul dispositif entrainant les effets attendus du système biologique.

Par conséquent, la définition de la dépendance causale sera régie par l'effet, ce qui nous conduit à la définition suivante de la dépendance : *"si l'effet 'e' vient à*

apparaître, alors $'c'$ serait apparu". En outre, la portée du futur (resp. passé) est réduit à une *période de futur proche (resp. passé)*, représentant le fait qu'une réponse est toujours attendue avec un certain délai. Notons que, la définition proposée contourne le problème de préemption car si l'effet ne se produit pas la question de l'existence d'une cause est dénuée de sens. Cette définition est proche de la définition de la causalité proposée par Lewis [77] sous la forme de condition contrefactuelle, dont nous proposons une adaptation : "Si $'c'$ n'était pas apparu, alors $'e'$ ne serait pas apparu".

Dépendances comportementales primitives

La définition de la causalité proposée, nous permet de définir une relation de causalité dite *normale* représentée par le symbole $\circ\!\!\rightarrow$. Comme décrit dans le diagramme suivant, cette cause correspond à un élément c déclenchant à sa résolution l'activation d'un effet e.

Néanmoins, une telle relation causale, bien que modélisant la notion de causalité, reste un modèle abstrait des réactions biologiques. En effet, par exemple, une régulation génétique est un processus graduel où un gène va s'exprimer, et ainsi augmenter la concentration en ARN correspondant jusqu'à ce que cette concentration soit suffisante pour activer le promoteur d'un autre gène dont l'expression commencera alors à augmenter aussi tandis que le premier gène va stopper sa production d'ARN qui vont lentement se dégrader tout en continuant à activer le second gène. Une telle description se traduit sous **Gubs** en un second type de causalité nommée *persistante* et notée $\odot\!\!\rightarrow$. En se référant au graphe correspondant 2.2 on observe que le premier gène noté c et le second noté e possède un comportement cohérent avec celui décrit.

En génie génétique, une recombinaison permet l'émergence d'un gène régulé ou d'un caractère héréditaire de façon permanente. Un tel mécanisme est caractéristique de la dépendance nommée *résiduelle* et notée $\oplus\!\!\rightarrow$, dont le comportement correspond au troisième graphe de la figure 2.2. Ces dépendances considérées comme des primitives dans le sens où elles ne peuvent pas être exprimées par les autres sans affaiblir leurs propriétés (voir la table 3.3). Pour la dépendance normale la cause précède l'effet permettant à l'effet d'être observé ; pour la dépendance persistante la cause précède toujours l'effet, mais elle est maintenue tandis que l'effet est observé ; et pour la dépendance résiduelle, l'effet est maintenu malgré la disparition de la cause. Plus formellement, les dépendances comportementales sont définies comme suit (voir la

section 2.3 pour leur description formelle) :

— $c \bigcirc\!\!\rightarrow e$: si e apparaît alors c est apparu dans un passé proche.
— $c \odot\!\!\rightarrow e$: si e apparaît alors c est apparu dans un passé proche et est toujours présent en cet instant.
— $c \oplus\!\!\rightarrow e$: si e apparaît alors, soit e est apparu dans un passé proche ou sinon e n'est pas apparu dans un passé proche et c est alors forcément apparu.

La figure 2.2 illustre la correspondance entre les traces expérimentales, traces symboliques et l'histoire pour les dépendances causales. Comme les agents peuvent être étendus à des ensembles d'agents, toutes les dépendances peuvent être étendues à un ensemble de causes déclenchant un ensemble de conséquences, *i.e.*, $c_1 + \ldots + c_n \bigcirc\!\!\rightarrow e_1 + \ldots + e_m$. Par exemple, nous pouvons définir l'activation et l'inhibition comme suit :
$g_1 \xrightarrow{+} g_2 \equiv g_1 \odot\!\!\rightarrow g_2, \overline{g}_1 \bigcirc\!\!\rightarrow \overline{g}_2$ et $g_1 \xrightarrow{-} g_2 \equiv \overline{g}_1 \odot\!\!\rightarrow g_2, g_1 \bigcirc\!\!\rightarrow \overline{g}_2$. Ensuite, le programme représentant un circuit de régulation négative avec deux gènes, *i.e.*, $g_1 \xrightarrow{+} g_2, g_2 \xrightarrow{-} g_1$, correspond à : $\{g_1 \odot\!\!\rightarrow g_2, \overline{g}_1 \bigcirc\!\!\rightarrow \overline{g}_2, \overline{g}_2 \odot\!\!\rightarrow g_1, g_2 \bigcirc\!\!\rightarrow \overline{g}_1\}$.

Définition 1 (Inhibition forte)**.** *Par la suite nous décrirons comme inhibition forte notée* $g_1 \xrightarrow{-} g_2$ *une inhibition exprimée par le programme* $\overline{g}_1 \odot\!\!\rightarrow g_2, g_1 \bigcirc\!\!\rightarrow \overline{g}_2$

Cette définition peut être affaiblie en exprimant une inhibition comme étant $g_1 \bigcirc\!\!\rightarrow \overline{g}_2$, on parlera alors d'inhibition faible.

Définition 2 (Activation forte)**.** *Par la suite nous décrirons comme activation forte notée* $g_1 \xrightarrow{+} g_2$ *une activation exprimée par le programme* $g_1 \odot\!\!\rightarrow g_2, \overline{g}_1 \bigcirc\!\!\rightarrow \overline{g}_2$

Cette définition peut être affaiblie en exprimant une activation comme étant $g_1 \odot\!\!\rightarrow g_2$, on parlera alors d'activation faible.

Points d'observation

Afin de définir les comportements que l'on souhaite observer en particulier dans un programme, nous définissons la notion de *points d'observation*. Les points d'observation décrivent l'ensemble des observations attendues le long d'une trace d'exécution expérimentale. Tout comme un scientifique observerait l'état d'un système

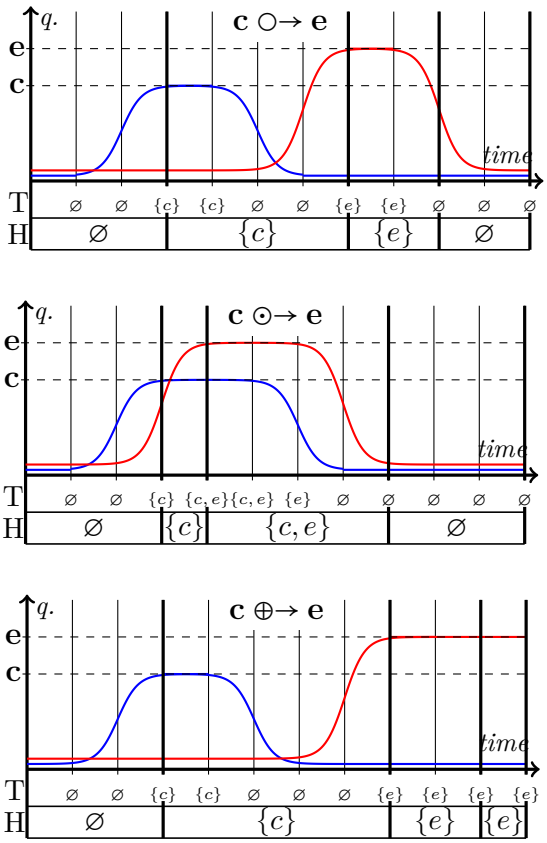


Figure 2.2 – Les courbes représentent les comportements typiques des dépendances causales en fonction de l'évolution au cours du temps d'une quantité (q) associée aux agents c et e (*e.g.*, taux de transcription d'ARN pour une régulation génétique par exemple). Les états présent ou absent des agents symboliques c et e sont ici tous les deux associés au seuil maximal de leur concentration. La trace symbolique (T) est issue d'une découpe périodique de l'évolution de ces quantités en identifiant les instants ou c ou e apparaissent. Une histoire consistante (H) correspondant à la définition de chaque relation causale est représentée en dessous de chaque trace. Le premier graphique représente la causalité normale : $c \bigcirc\!\!\rightarrow e$, le second la persistante : $c \odot\!\!\rightarrow e$ et le troisième la résiduelle : $c \oplus\!\!\rightarrow e$.

et l'émergence d'un comportement donné, dans **Gubs**, il suffira de définir le moment de cette analyse, comme un point d'observation.

Dans la figure 2.1, les points d'observation sont définis pour $Cro(high)$ et $Cro(low)$ ainsi que $CI(low)$ par : $\mathbf{obs}_1 :: Cro(high)$, $\mathbf{obs}_2 :: Cro(low)$ et $\mathbf{obs}_3 :: CI(low)$.

Par exemple, souhaiter observer le fait qu'un gène G est au niveau $High$ s'écrira Obs **: :**$G(High)$. Comme l'activation d'une fonction correspond à l'observation de l'effet, les points d'observation sont utilisés pour déterminer les effets qui doivent être nécessairement respectés. Par extension, ces points d'observation peuvent être assimilés à des exigences expérimentales. Par exemple, dans les circuits de régulation négative, les points d'observation caractéristiques sont les suivants : obs_1 **: :**$(g_1 + \overline{g_2})$, obs_2 **: :**$(\overline{g_1} + g_2)$.

2.2.3 Compartiments et environnements

Contexte

Afin de décrire les interactions de l'environnement sur notre programme, nous introduisons ici la notion de *contexte*. Cette notion nous permet de différencier clairement les interactions propres au programme **Gubs** de celles déclenchées par l'environnement et donc externes au système décrit par le programme. Plus précisément, un contexte réfère à un stimulus agissant sur le système, qu'il s'agisse de conditions environnementales ou d'une signalisation externe. L'application d'un contexte k à un ensemble de dépendances d s'écrit $[k]\{d\}$ ou k est un agent (*i.e.*, variable ou constante) et d un ensemble de relations causales non vide (ceci pour des raisons de sémantiques, voir section suivante). Une telle écriture signifie que les dépendances comportementales d ne se déclenchent que si le contexte k est présent.

Dans la figure 2.1, aucun contexte n'est défini, les interactions de l'environnement n'étant pas définies. on pourrait néanmoins imaginer une variable externe g venant interagir sur le première causalité : $[g]\{Cro(low) \xrightarrow{-} CI(high)\}$;

Par exemple, récemment, Ye et al. [118] ont exploré l'utilisation de signalisation opti-génétique afin de contrôler l'expression de transgènes d'intérêt. La présence d'ultra-violet induit l'expression de transgènes (tg) via une cascade de signaux menant à l'appontage du facteur de transcription Nfat à un promoteur spécifique(Pnfat).

Le programme suivant utilise la notion de contexte pour résumer ce processus : [BlueLight]{NFAT $\odot\!\!\rightarrow$ tg}. Notons que, comme un agent peut être étendu à un ensemble d'agents, un contexte peut être défini comme un ensemble de contextes, $[k_1, \ldots, k_n]d$, signifiant que l'ensemble des conditions k_i doivent être présentes pour que les relations causales de d se déclenchent. Cette interprétation est de plus équivalente à une cascade de contextes, $[k_1][k_2]\ldots[k_n]d$. En outre, les points d'observation et les définitions d'attribut ne sont pas dépendants des contextes.

Compartiments

Finalement un programme **Gubs** est un ensemble de définitions d'attribut, de points d'observation et de dépendances comportementales encapsulés dans des contextes.

Afin de décrire l'organisation spatiale du système biologique nous introduisons un dernier élément dans le langage, les *compartiments*. Un compartiment encapsule un ensemble de dépendances, les rendant locales à celui-ci.

Dans la figure 2.1, le comportement décrit correspondant au réseau de régulation du λ-phage. Un compartiment *Phage* encapsulant l'ensemble du programme pourrait être défini.

Par exemple $C\{g_1 \circ\!\!\rightarrow g_2\}$ décrit une relation de dépendance normale apparaissant au sein du compartiment C.

L'ensemble des compartiments est organisé hiérarchiquement et tous les compartiments sont donc inclus dans un autre, excepté pour le compartiment le plus extérieur. Bien que les compartiments se réfèrent directement à l'organisation cellulaire compartimentée (*e.g.*, noyau, mitochondrie), ils sont également utilisés pour mettre l'accent sur l'isolement de certaines interactions en enfermant syntaxiquement les dépendances dans un compartiment. De ce fait, par défaut les agents et les relations causales définis dans un environnement sont strictement indépendant de ceux dans d'autres compartiments, même le compartiment parent.

Notons qu'afin de représenter une interaction possible entre deux compartiments, nous pouvons faire référence depuis un compartiment à un agent défini dans un autre. Ainsi $C.s$ correspond à l'état d'un agent s dans le compartiment C. Notons

aussi que la notion de compartiment, est considéré en terme de sémantique comme une réécriture des variables sous forme d'une chaine de caractères concaténant la hiérarchie de compartiments les contenant. Ceci est défini plus en détail dans la section sémantique (section 2.3).

(1)	program	::= {behaviour}
(2)	program	::= $\{\epsilon\}$
(3)	behaviour	::= behaviour; behaviour | behaviour
(3′)	behaviour	::= compartment | dependence | context | observation | defattributes
(4, 5, 6)	dependence	::= states $\bigcirc\!\!\rightarrow$ states | states $\odot\!\!\rightarrow$ states | states $\oplus\!\!\rightarrow$ states
(7)	defattributes	::= agents : {attrels}
(7′)	attrels	::= attrels, attrel | attrel
(8)	observation	::= *string* **: :**states
(9)	compartment	::= compname {behaviour}
(10)	context	::= [agents] {behaviour}
(11)	states	::= states + state | state
(12)	state	::= compname.state
(13, 14)	state	::= agent(attribute) | agent($\overline{\text{attribute}}$)
(15, 16)	state	::= agent | $\overline{\text{agent}}$
(17, 18)	attrel	::= attribute ≺ attribute | attribute $\not\approx$ attribute
(19)	attrel	::= attribute
	compname	::= *string* | *String*
	agent	::= *string* | *String*
	attribute	::= *string* | *String*

Table 2.1 – Syntaxe d'un programme **Gubs**. *string* et *String* correspondent respectivement à une chaîne de caractère commençant par une minuscule pour une variable et par une majuscule pour une constante. La numérotation est associée à celle de la table de sémantique (Table 2.3)

Exemple de programme

Dans la figure suivante, nous introduisons un programme **Gubs** représentant le réseau de régulation à deux états du bactériophage λ proposé par René Thomas[106]. Cet exemple est repris par la suite dans la section 2.4. Notons qu'afin de simplifier le programme nous utilisons la régulation forte définie précédemment (Définition 1 et 2). En termes de programme, l'inhibition et l'activation se définissent comme des macros.

Notons qu'écrire la rétro inhibition de Cro sous la forme $Cro(high) \overset{-}{\longrightarrow} Cro(high)$

Bactériophage – λ
$\{Cro : \{low < high\};$
$CI : \{low < high\};$
$Cro(low) \xrightarrow{-} CI(high);$
$Cro(high) \circ\!\!\rightarrow Cro(low);$
$CI(low) \xrightarrow{-} Cro(high);$
$CI(low) \circ\!\!\rightarrow CI(high);\}$
$\mathbf{obs}_1 :: Cro(high);\}$
$\mathbf{obs}_2 :: Cro(low);\}$
$\mathbf{obs}_3 :: CI(low);\}$

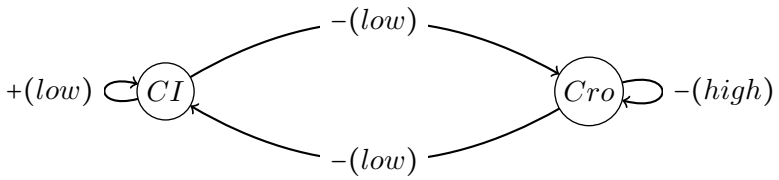


Figure 2.3 – modèle du réseau de régulation du λ phage à deux états

entraîne un comportement qualifié d'inobservable s'interprétant par une formule insatisfiable (Voir section 2.3.3), obligeant à écrire

$$Cro(high) \circ\!\!\rightarrow Cro(low)$$

ce qui est plus cohérent biologiquement parlant, s'agissant d'une baisse de concentration.

Notons aussi que les inhibitions sont exprimées pour l'attribut le plus « élevé » de chaque agent afin que tous les plus « faibles » soient inhibés aussi.

2.3 Sémantique

Dans cette section, après un bref récapitulatif sur la logique hybride, nous introduirons en détail la sémantique de **Gubs**. La sémantique dénotationnelle de **Gubs** est récapitulée en table 2.3 et sera illustrée par des exemples relatifs à des relations causales élémentaires ainsi que sur le réseau de régulation du bactériophage.

2.3.1 Une sémantique fondée sur la logique

Gubs se définit comme un langage décrivant l'observation d'un comportement dont les objets résultant de la compilation ne sont pas générés comme un code binaire, mais sont le résultat d'un assemblage de composants. Le comportement du système biologique engendré doit se conformer aux propriétés comportementales décrites dans le programme. **Gubs** est donc un langage de spécification dont les programmes décrivent des propriétés comportementales que l'on souhaite reproduire

par assemblage de composants. Dans ce cadre, on peut l'assimiler à un langage évolué de requêtes portant sur une sélection de composants biologiques capables de réaliser le comportement programmé. De ce fait, nous utilisons la logique pour donner une sémantique au langage **Gubs** car elle constitue un moyen d'exprimer ces requêtes. Les motivations pour la logique modale hybride seront expliquées dans la sous section 2.3.1.

Le protocole de validation de l'assemblage constitue un élément central d'explication des règles données par la sémantique. En effet, il établit le lien entre les formules qui sont le résultat de l'interprétation des programmes et une validation du comportement qui serait faite à partir d'expériences biologiques. Aussi, nous débuterons cette section en expliquant ce protocole.

Protocole de validation d'une fonction biologique décrite en Gubs

La validation constitue un des éléments importants à la compréhension de la sémantique proposée car elle donne une réponse à la question : Comment passer des expériences à un modèle de kripke ? Dans cette section nous définissons le scénario d'un protocole visant à décrire les différentes étapes concourant à ce rapprochement. Il détermine une manière concrète de comprendre les règles de la sémantique.

La validation des propriétés s'appuie d'abord sur un ensemble d'expériences mesurant l'évolution de grandeurs physiques relatives aux agents constituant le système biologique résultant de la compilation. Ces grandeurs déterminent l'état des agents. Un programme **Gubs** décrivant une abstraction symbolique de ces grandeurs, nous supposerons qu'il soit possible d'en extraire une *trace* qui caractérise symboliquement cette évolution. Ceci amène notamment à établir une correspondance entre grandeurs physiques et attributs qui peut être arbitrairement définie. Formellement, une trace, $(T_t)_{1\leq t\leq m}$, est une séquence finie décrivant une succession *d'évènements* où chaque évènement correspond à l'état des agents à un instant donné. Les différents états d'un agent correspondent à ses attributs. Par exemple, l'évolution de la concentration de *Low* vers *High* pour G peut se décrire par la trace suivante composée de 6 instants [1] :

$$(\{G(\underset{1}{Low})\},\{G(\underset{2}{Low})\},\{G(\underset{3}{Mid})\},\{G(\underset{4}{Mid})\},\{G(\underset{5}{Mid})\},\{G(\underset{6}{High})\}),\quad_{7}.$$

1. L'étape 7 est insérée comme une étape supplémentaire pour se conformer à la définition de la division chronologique.

Cependant, tous les évènements dans une trace ne sont pas nécessairement pertinents pour la validation des propriétés décrites par le programme. Par exemple, si nous nous concentrons sur l'évolution d'une concentration de *Low* à *High* pour G, seuls trois évènements sont pertinents pour cette description : $G(Low)$ puis $G(Mid)$ et finalement $G(High)$, sans tenir compte des étapes d'évolution intermédiaires qui se produisent entre les deux, ni de la répétition d'évènements identiques. Nous allons donc adopter une autre représentation contractant la trace pour ne retenir que des évènements jugés nécessaires à la validation de la fonction biologique synthétisée. Elle se nomme *histoire*. Elle correspond à une division chronologique d'une trace en plusieurs *périodes*. Cette division se définit par les dates de départ de chaque période.

Étant donnée une trace $(T_t)_{1\leq t\leq m}$, et une division chronologique, $(d_i)_{1\leq i\leq n}$ tel que $d_i < d_{i+1}$, l'histoire se définit donc comme une séquence d'ensembles d'évènements se produisant lors de chaque période (*i.e.*, $(H_i)_{1\leq i<n}$, tel que chaque $H_i = \bigcup_{d_i\leq t<d_{i+1}} T_t$). En résumé, une histoire est une division intentionnelle d'une trace pour souligner les évènements caractéristiques et pertinents correspondant aux propriétés recherchées du comportement du système biologique produit.

Dans l'exemple précédent, une division de la trace chronologique conduisant à une histoire conforme à l'évolution attendue de *Low* à *High* pour G est $(1, 3, 6, 7)$ ce qui définit la suite d'intervalles de temps discrets $([1, 2], [3, 5], [6, 6])$. L'histoire qui en résulte est : $(\{G(Low)\}, \{G(Mid)\}, \{G(High)\})$. Notons que $(1, 2, 4, 7)$ convient aussi mais donnera l'histoire : $(\{G(Low)\}, \{G(Mid)\}, \{G(Mid), G(High)\})$. Remarquons q'une autre division chronologique $(1, 3, 7)$ peut conduire à une autre histoire où les niveaux *Mid* et *High* ne sont pas explicitement distingués en deux périodes distinctes. Ainsi l'histoire ne suit pas la progression attendue d'une concentration faible à élevée.

La validation d'une fonction biologique de synthèse à partir d'un programme **Gubs** opère sur les histoires. Formellement, *une histoire correspond à un modèle de Kripke (Définition 3) possédant la propriété topologique de linéarité* (*i.e.*, un graphe réduit à un unique chemin). Les variables présentes dans chaque monde (nœud) correspondent aux évènements de l'histoire (état des agents), c.f. Table 2.3. Plus précisément, un agent dans un certain état se symbolise par une variable particulière dont le nom est celui de l'agent concaténé au nom de son attribut représentant cet état. La relation d'accessibilité entre deux mondes décrit la progression temporelle de l'histoire entre deux périodes.

Une histoire est dite *cohérente* par rapport à un programme si elle satisfait la formule résultant de son interprétation. Cependant, il est possible de déduire de plusieurs expériences, plusieurs histoires qui ne sont pas nécessairement cohérentes prises indépendamment (Par exemple si deux causes peuvent mener au même effet comme décrit dans le paragraphe 2.3.2). Elles représenteront alors des résultats partiels d'analyse de la fonction. Afin que ces histoires soient ensemble cohérentes avec le programme, leur réunion devra former un modèle de Kripke satisfiant la formule résultant de son interprétation.

Dans ce scénario, le passage d'un ensemble d'expériences à un modèle de Kripke par l'intermédiaire d'une trace sera supposé être réalisé par l'expérimentateur. Son automatisation dépasse le cadre de notre sujet. En résumé, le scénario proposé pour valider une fonction biologique de synthèse à partir d'expériences, en se rapportant au cadre formel proposé, repose sur des étapes déterminant une abstraction symbolique des résultats d'expériences pour aboutir à un modèle de Kripke. Ce modèle est vu comme un ensemble d'histoires correspondant à ses chemins où chaque histoire/chemin représente une description partielle du comportement du système biologique analysé.

Comme nous le verrons dans les exemples, ce scénario guide l'interprétation informelle mais concrète que l'on peut donner des règles de traduction de la sémantique. Ainsi, du point de vue du programme et de sa compilation, l'élément de validation considéré sera un modèle de Kripke.

Il faut remarquer aussi qu'il est possible de construire un modèle de Kripke satisfaisant une formule par les méthodes à tableaux [33]. Cette possibilité nous permet de définir les histoires qu'il est nécessaire de valider expérimentalement. Les méthodes à tableaux nous permettent donc d'obtenir pour un programme donné, *une* histoire cohérente.

Notons que la notion d'histoire est aussi présente dans le langage Kappa [76, 46]. Il apparaît intéressant d'établir une comparaison avec celle que nous proposons pour la préciser.

Dans le cas de Kappa, une histoire est un scénario menant à l'apparition d'un agent. Plus précisément, pour un programme Kappa donné, il s'agit de toutes les étapes successives menant à l'apparition d'un agent donné dans un état particulier correspondant aux étapes structurelles d'évolution des agents [76]. L'évolution est déduite statiquement à partir de l'ensemble des règles du programme et correspond

aux relations causales définies entre règles sans prendre en compte des répétitions. une répétition est caractérisée comme étant deux configurations d'agents menant au même état observable.

Kappa utilise donc des agents mais qui représentent des états de composants biologiques pour décrire une histoire représentant une évolution structurelle jusqu'à la formation d'un assemblage que l'on souhaite obtenir.

La notion d'histoire développée dans le langage Kappa est donc similaire dans son principe à celle que nous proposons dans la mesure où toutes deux décrivent les chaînes causales entre règles du programme menant à l'apparition d'un état du système jugé comme essentiel. Néanmoins, elles diffèrent sur certains points.

Dans Kappa, une histoire décrit des modifications structurelles « minimales » pour aboutir à une transformation finale. Dans notre cas, histoire se rapporte à l'observation expérimentale du système pour établir une analogie de comportements entre le programme décrit et celui résultant de la compilation conduisant à un assemblage de composants. Dans Kappa l'histoire explique la formation d'un agent alors que dans notre cas, elle est résulte d'observations de la transformation des états des agents. On pourrait schématiquement conclure que Kappa décrit par une histoire une vision intérieure au système donnant les étapes de transformation structurelle alors que notre notion d'histoire en décrit une vision extérieure s'appuyant sur la notion de trace expérimentale.

Enfin, dans la construction des histoires, là où Kappa fonctionne par construction des traces possibles puis par réduction de celle-ci pour garantir l'absence de répétition, notre approche pour les construire s'appuie sur les méthodes à tableaux définissant des modèles de Kripke « minimaux » représentant les séquences d'observations minimales à avoir pour valider un système biologique. Nous avons aussi introduit la notion de contexte formalisée par une relation d'accessibilité particulière qui permet d'associer une histoire à un contexte particulier. Le contexte ne définit pas les relations causales mais détermine les conditions de leur activité correspondant à des conditions environnementales extérieures au système.

Logique Hybride

Gubs fonde sa sémantique sur la logique hybride. Nous avons fait ce choix pour différentes raisons : Premièrement, afin de modéliser l'idée d'évènements ne se produisant pas simultanément, il est nécessaire d'introduire une notion de temporalité.

La logique propositionnelle classique ne nous suffit plus. Le choix qui s'offre alors est la logique modale. En effet elle nous permet grâce au modèle de Kripke [75] de représenter les relations causales au sein d'un réseau de régulation génétique. Afin de pouvoir modéliser l'interaction de l'environnement sur un système biologique, nous utilisons une approche multimodale nous permettant de libeller les arcs des modèles de Kripke et ainsi de garder une indépendance forte entre les actions dues à l'environnement et celles dues au système. Finalement, la logique hybride introduit entre autres la notion de mondes nommés, ce qui dans notre cas correspond exactement à la notion de point d'observation. L'ensemble de ces prérequis nous a orienté vers ce choix.

Nous rappelons ici la syntaxe et la sémantique de la logique hybride $\mathcal{H}(\mathbf{A}, @)$. La logique hybride [21, 24] offre la possibilité de libeller des mondes par de nouveaux symboles appelés *nominaux*. Ils seront utilisés dans la satisfaction des opérateurs modaux $@_a$; la formule $@_a\phi$ affirme que ϕ est satisfaite à l'unique point nommé par le nominal a identifiant une valeur de vérité particulière d'une formule en ce point.

Étant donné un ensemble de symboles propositionnels, PROP, un ensemble de symboles relationnels REL, et un ensemble de nominaux NOM disjoints de PROP, un ensemble de formules bien formées de signature $\langle \mathsf{PROP}, \mathsf{NOM}, \mathsf{REL} \rangle$ est défini comme suit :

$$\phi ::= \top \mid p \mid a \mid \neg\phi \mid \phi \wedge \phi \mid @_a\phi \mid \langle k \rangle \phi \mid \langle k \rangle^{-}\phi \mid \mathbf{A}\phi.$$

avec $p \in \mathsf{PROP}, a \in \mathsf{NOM}$ et $k \in \mathsf{REL}$. En outre, la syntaxe est étendue à d'autres opérateurs logiques [2] : $\bot, \vee, \rightarrow, [k], \mathbf{E}$, de manière habituelle.
La sémantique de $\mathcal{H}(\mathbf{A}, @)$ est fondée sur la satisfaction de modèles de Kripke (Table 2.2). L'expression $\mathcal{M}, w \Vdash \phi$ est interprétée comme la satisfaction de la formule ϕ par le modèle $\mathcal{M}$ dans le monde w où $\Vdash$ correspond à la *relation de réalisabilité* (*i.e.*, "est un modèle de"). De façon générale, un modèle *satisfait* une formule, noté $\mathcal{M} \Vdash \phi$, si et seulement si elle est satisfaite en un monde de ce modèle (*i.e.*, $\exists w \in \mathrm{Dom}\, \mathcal{M} : \mathcal{M}, w \Vdash \phi$). De plus, dans le cas de la logique hybride, chaque sous-formule associée à l'opérateur $@_a$ doit être satisfaite à partir du monde nommé a Dans notre cas, la présence de l'opérateur $\mathbf{A}$, impose que la formule soit satisfaite en tout monde du modèle (*i.e.*, $\forall w \in \mathrm{Dom}\, \mathcal{M} : \mathcal{M}, w \Vdash \phi$).

2. $\bot = \neg\top, \psi \vee \phi = \neg(\neg\psi \wedge \neg\phi), \psi \rightarrow \phi = \neg(\psi \wedge \neg\phi), [k]\phi = \neg\langle k \rangle \neg\phi, \mathbf{E}\phi = \neg\mathbf{A}\neg\phi.$

Définition 3 (Modèle de Kripke)**.** *Un modèle de Kripke est défini comme une structure :*

$$\mathcal{M} = \langle W, (R_k)_{k \in \tau}, V \rangle$$

où $W = Dom\,\mathcal{M}$ *est un ensemble non vide de* mondes, $\tau \subseteq \mathsf{REL}$ *un sous ensemble de symboles relationnels décrivant les* modalités *(i.e., label sur les arcs),* $R_k \subseteq W \times W, k \in \tau$ *une relation d'accessibilité,* $V : (\mathsf{PROP} \cup \mathsf{NOM}) \to 2^W$ *une interprétation attribuant à chaque nominal et variable propositionnelle un ensemble de mondes tel que chaque nominal ne corresponde qu'à un monde au plus (i.e.,* $\forall a \in \mathsf{NOM} : |V(a)| \leq 1$*). Par convention,* R *correspond à l'union des relations d'accessibilité,* $R = (\bigcup_{k \in \tau} R_k)$.

La *théorie modale* d'un modèle $\mathcal{M}$ par rapport à un ensemble de formules F, $\mathrm{TH}_F(\mathcal{M})$, est l'ensemble des formules de F satisfaites par $\mathcal{M}$, *i.e.*, $\mathrm{TH}_F(\mathcal{M}) = \{\phi \in F \mid \mathcal{M} \Vdash \phi\}$.

$\mathsf{KS}(\phi)$ désigne l'ensemble de tous les modèles satisfaisant ϕ, *i.e.*, $\mathsf{KS}(\phi) = \{\mathcal{M} \mid \mathcal{M} \Vdash \phi\}$.

$\mathcal{M}, w \Vdash \top$	iff $true$
$\mathcal{M}, w \Vdash a$	iff $w \in V(a)$, $a \in \mathsf{NOM} \cup \mathsf{PROP}$
$\mathcal{M}, w \Vdash \neg\phi$	iff $\mathcal{M}, w \nVdash \phi$
$\mathcal{M}, w \Vdash \phi_1 \wedge \phi_2$	iff $\mathcal{M}, w \Vdash \phi_1$ and $\mathcal{M}, w \Vdash \phi_2$
$\mathcal{M}, w \Vdash @_a\phi$	iff $\exists w' \in W : \mathcal{M}, w' \Vdash \phi$ and $\{w'\} = V(a)$
$\mathcal{M}, w \Vdash \langle k \rangle \phi$	iff $\exists w' \in W : \mathcal{M}, w' \Vdash \phi$ and wR_kw'
$\mathcal{M}, w \Vdash \langle k \rangle^- \phi$	iff $\exists w' \in W : \mathcal{M}, w' \Vdash \phi$ and $w'R_kw$
$\mathcal{M}, w \Vdash \mathbf{A}\phi$	iff $\forall w' \in W : \mathcal{M}, w' \Vdash \phi$

Table 2.2 – Interprétation de la logique hybride.

2.3.2 Sémantique de Gubs

L'interprétation d'un programme **Gubs** est une formule en logique multimodale hybride possédant l'opérateur "*always*", $\mathcal{H}(\mathbf{A}, @)$. Formellement, en termes de logique, un programme **Gubs** se traduit comme un ensemble de relations causales et de points d'observation. Notons que ces ensembles peuvent être vides. Par conséquent, il est considéré comme observable si et seulement si sa formule correspondante est satisfiable. La validité / satisfiabilité est définie à partir d'un modèle de Kripke (Définition 3) rassemblant un ensemble d'histoires possibles.

Rappelons qu'un monde dans un modèle de Kripke représente un évènement défini par un ensemble d'états des agents à un moment donné d'une histoire (voir figure : 2.2). L'opérateur [ ] se traduit ici par *« s'observe dans tous les futurs proches possibles »* et $\langle\ \rangle$ par l'expression *« s'observe au moins dans un des futurs proches possibles »* (resp. $\langle\ \rangle^-, [\ ]^-$ pour les passés proches). En outre, les relations d'accessibilité, $(R_k)_{k\in\tau}$, représentent une « évolution temporelle » en fonction de certains contextes. Soit K_P l'ensemble des contextes du programme, une modalité $k \in \tau$ est une partie non vide de K_P (*i.e.*, $\tau = \mathbf{2}^{K_P} \setminus \{\varnothing\}$). Les états des agents sont des variables dans les formules et les points d'observations sont quant à eux définis comme des nominaux pour identifier des mondes spécifiques.

Soit $\langle \mathsf{W}, \bullet, \Lambda \rangle$ l'ensemble des mondes W possédant l'opération de concaténation et l'élément neutre, le monde vide Λ et soit $\mathsf{F}_{\mathcal{H}}$ l'ensemble des formules bien formées de $\mathcal{H}(\mathbf{A}, @)$. La sémantique dénotationnelle est définie par quatre fonctions :

— $[\![.]\!] : \mathsf{P} \to \mathsf{F}_{\mathcal{H}}$,
— $[\![.]\!]_P : \mathsf{P} \to \mathsf{W} \to \mathbf{2}^{\mathsf{W}} \to \mathsf{F}_{\mathcal{H}}$,
— $[\![.]\!]_B : \mathsf{B} \to \mathsf{W} \to \mathsf{F}_{\mathcal{H}}$,
— $[\![.]\!]_R : \mathsf{R} \to \mathsf{W} \to \mathsf{F}_{\mathcal{H}}$

où $\mathsf{P}, \mathsf{B}, \mathsf{R}$ correspondent respectivement à l'ensemble des programmes **Gubs**, l'ensemble des états de l'ensemble des agents et l'ensemble des relations sur les attributs.

— $[\![.]\!]$ est la fonction d'initialisation de l'interprétation.
— $[\![.]\!]_P$ fournit une interprétation des comportements : les relations causales, les compartiments, les contextes et les points d'observation.
— $[\![.]\!]_B$ définit l'interprétation d'un agent et d'un ensemble d'agents.
— Finalement, $[\![.]\!]_R$ correspond à l'interprétation des relations entre attributs.

(1)	$[\![\{b\}]\!]$	$= \mathbf{A}\,([\![b]\!]_P\,(\Lambda)(\varnothing))$
(2)	$[\![\epsilon]\!]_P\,(C)(K)$	$= \top$
(3)	$[\![b_1, b_2]\!]_P\,(C)(K)$	$= [\![b_1]\!]_P\,(C)(K) \wedge [\![b_2]\!]_P\,(C)(K)$
(4)	$[\![s_1 \circ\!\!\rightarrow s_2]\!]_P\,(C)(K)$	$= [\![s_2]\!]_B\,(C) \rightarrow \langle K\rangle^-\,([\![s_1]\!]_B\,(C))$
(5)	$[\![s_1 \odot\!\!\rightarrow s_2]\!]_P\,(C)(K)$	$= [\![s_2]\!]_B\,(C) \rightarrow ([\![s_1]\!]_B\,(C) \wedge \langle K\rangle^-\,([\![s_1]\!]_B\,(C)))$
(6)	$[\![s_1 \oplus\!\!\rightarrow s_2]\!]_P\,(C)(K)$	$= [\![s_2]\!]_B\,(C) \rightarrow ((\langle\ \rangle^-\,[\![s_2]\!]_B\,(C)) \vee (\langle K\rangle^-\,[\![s_1]\!]_B\,(C)))$
(7)	$[\![g_1, \cdots, g_n : \{r_1, \cdots, r_m\}]\!]_P\,(C)(K)$	$= \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m} [\![r_j]\!]_R\,(C \bullet g_i)$
(8)	$[\![l :: s]\!]_P\,(C)(K)$	$= @_l\,[\![s]\!]_B\,(C)$
(9)	$[\![C'\{b\}]\!]_P\,(C)(K)$	$= [\![b]\!]_P\,(C \bullet C')(K)$
(10)	$[\![[K]\{b\}]\!]_P\,(C)(K')$	$= [\![b]\!]_P\,(C)(K \cup K')$
(11)	$[\![s_1 + \ldots + s_n]\!]_B\,(C)$	$= \bigwedge_{i=1}^{n} [\![s_i]\!]_B\,(C)$
(12)	$[\![C'.s]\!]_B\,(C)$	$= [\![s]\!]_B\,(C \bullet C')$
(13)	$[\![g(a)]\!]_B\,(C)$	$= C.g_a$
(14)	$[\![g(\overline{a})]\!]_B\,(C)$	$= \neg C.g_a$
(15)	$[\![g]\!]_B\,(C)$	$= C.g$
(16)	$[\![\overline{g}]\!]_B\,(C)$	$= \neg C.g$
(17)	$[\![a_1 \prec a_2]\!]_R\,(g)$	$= g_{a_2} \rightarrow g_{a_1}$
(18)	$[\![a_1 \not\approx a_2]\!]_R\,(g)$	$= g_{a_1} \rightarrow \neg g_{a_2} \wedge \neg g_{a_2} \rightarrow g_{a_1}$
(19)	$[\![a]\!]_R\,(g)$	$= \top$

Table 2.3 – Sémantique de **Gubs**. Dans cette définition, a représente un attribut, b un comportement, g un agent, s un ensemble d'états d'agents ou l'état d'un agent, r une relation entre attributs, C un compartiment, K un ensemble de contextes, et b un ensemble de comportements (*i.e.*, contextes, compartiments, dépendances, attributs, points d'observation). Dans cette sémantique, les agents et leur concaténation à leurs attributs et compartiments forment des variables propositionnelles. Les contextes définissent des symboles relationnels. Enfin, les points d'observation sont des nominaux.

Exemples

De cette table, nous en déduisons des modèles de Kripke satisfaisant les relations causales en Figure 2.4 et des modèles satisfaisant la notion de contexte, d'attribut et de point d'observation en Figure 2.5. Dans ces modèles nous considèrerons que l'effet est présent (*i.e.*, qu'un point d'observation a été défini pour l'effet). Notons que les modèles présentés ne sont pas minimaux pour un parallèle plus explicite entre la sémantique et le modèle.

Causes. La Figure 2.4 décrit des modèles de Kripke satisfaisant les différentes relations causales. Ces modèles correspondent à des histoires cohérentes et montrent

les conditions différentes les interprétant. Il faut remarquer que le modèle (1) est un sous-modèle (limité aux nœuds en blanc) du modèle (3). De même, le modèle (2) est un cas particulier du modèle (1) en imposant la présence de la cause dans le monde de l'effet. Nous pouvons en déduire qu'un modèle satisfaisant une relation causale persistante satisfait aussi une relation causale normale, elle-même satisfaisant une relation causale résiduelle. Ce constat se généralise et sera utilisé lors de la compilation (Table 3.3).

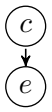
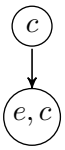
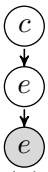
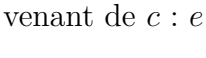
$c \bigcirc\!\!\rightarrow e$	$c \odot\!\!\rightarrow e$	$c \oplus\!\!\rightarrow e$
$e \rightarrow \langle\ \rangle^{-}(c)$	$e \rightarrow c \wedge \langle\ \rangle^{-}(c)$	$e \rightarrow ((\langle\ \rangle^{-}e) \vee (\langle\ \rangle^{-}c))$
 (1)	 (2)	 (3)

Figure 2.4 – Modèles de Kripke correspondant aux trois types de cause possible. Nous noterons c la cause et e l'effet. Dans le premier cas, le modèle (1) impose simplement à c de se trouver dans un monde précédent e. Dans le deuxième cas (modèle (2)), c doit aussi apparaître dans le monde où apparaît e. Enfin dans le troisième cas (modèle (3)), la présence de e impose que soit soit c apparaît dans le monde précédent, soit e apparaît dans le monde précédent (monde en gris).

Contextes, attributs et points d'observation. Dans la figure 2.5 le modèle (1) présente le cas d'un effet ne se déclenchant qu'en présence d'un certain contexte k défini comme label de l'arc. Le modèle (2) présente le cas d'une cause possédant deux attributs définis selon une relation de précédence, la présence du plus grand(a_2) déclenche alors le même effet que celle de a_1. Enfin le troisième modèle défini simplement un point d'observation *obs* sur l'effet.

Notons que dans le modèle (1), dans le cas d'une cause rémanente, le contexte n'est nécessaire qu'au déclenchement de c, et n'est donc présent dans la formule que dans le $\langle\ \rangle$ définissant l'arc venant de c : $e \rightarrow ((\langle\ \rangle^{-}e) \vee (\langle k \rangle^{-}c))$: 

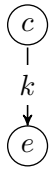
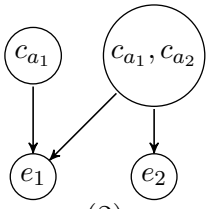
$[k]\{c \bigcirc\!\!\rightarrow e\}$	$c : \{a_1 \prec a_2\}; c(a_1) \bigcirc\!\!\rightarrow e_1; c(a_2) \bigcirc\!\!\rightarrow e_2$	$c \oplus\!\!\rightarrow e; obs :: e$
$e \rightarrow \langle k \rangle^{-}(c)$	$c_{a_2} \rightarrow c_{a_1} \wedge e_1 \rightarrow \langle\ \rangle^{-}(c_{a_1}) \wedge e_2 \rightarrow \langle\ \rangle^{-}(c_{a_2})$	$e \rightarrow \langle\ \rangle^{-}(c) \wedge @_{obs} e$
 (1)	 (2)	 (3)

Figure 2.5 – Modèles de Kripke correspondant à la notion de contexte, d'attributs et de point d'observation.

Attributs incompatibles. Certaines définitions d'attribut peuvent aboutir à un programme non valide. Par exemple, le programme suivant :

$$\begin{array}{ll} \{ & C : \{a_1 \not\approx a_2\}; \\ & C(a_1) + C(a_2) \bigcirc\!\!\rightarrow E; \\ & obs :: E; \} \end{array}$$

se traduit par la formule et le modèle de Kripke de la figure 2.6, ce qui est impossible en raison de la présence forcée à la fois de C_{a_2} et $\overline{C_{a_2}}$ dans le même monde.

$$\begin{array}{ll} \mathbf{A}(& C_{a_1} \leftrightarrow \overline{C_{a_2}} \quad \wedge \\ & E \rightarrow \langle\ \rangle^{-} C_{a_1} \wedge C_{a_2} \quad \wedge \\ & @_{obs} E) \end{array}$$

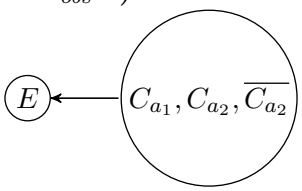


Figure 2.6 – formule et modèle de Kripke faux dans le cas d'attributs contradictoires.

Pour illustrer certaines constructions élémentaires de relations causales que nous retrouvons fréquemment en modélisation des systèmes biologiques, nous allons décrire un ensemble d'exemples dont nous donnerons les conséquences concrètes relative à leur satisfaction. Pour ces exemples, nous supposerons que l'effet est réalisé, c'est-à-dire observé. Cette hypothèse conduit à expliciter les points d'observation.

Cause alternative. Considérons le programme suivant :

$$\begin{array}{ll} \{ & G_1 \circ\!\!\rightarrow G; \\ & G_2 \circ\!\!\rightarrow G; \\ & obs :: G; \} \end{array}$$

La formule correspondant à ce programme est :

$$\begin{array}{lll} \mathbf{A}(& G \rightarrow \langle\ \rangle^{-}\ G_1 & \wedge \\ & G \rightarrow \langle\ \rangle^{-}\ G_2 & \wedge \\ & @_{obs} G) & \end{array}$$

Les modèles de Kripke impliquant uniquement les agents du programme et satisfaisant cette formule sont les suivants :

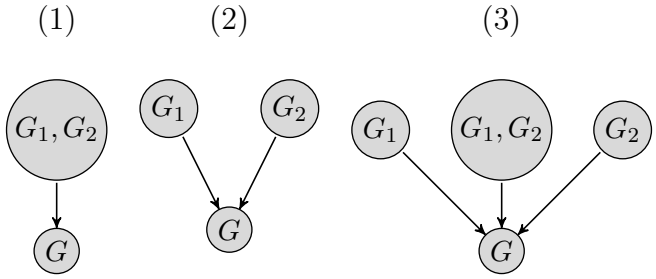


Figure 2.7 – Modèles de Kripke interprétant l'exemple de causes alternatives.

Nous rapportons l'interprétation de ces modèles aux histoires qu'ils incluent. Le modèle (1) correspond à l'apparition conjointe de deux causes lorsque l'effet a été observé. Ceci conduirait à en déduire la présence nécessaire de G_1 et de G_2 comme cause de G. Toutefois, le modèle (2) met en évidence deux histoires distinctes qui ne sont pas cohérentes indépendamment mais dont l'union est cohérente (*i.e.*, satisfait la formule). En se focalisant uniquement sur les histoires, ce modèle amène à l'hypothèse qu'une des causes suffit à déclencher G. Ce programme exprimerait donc le fait que G_1 *ou* G_2 *cause* G. Cependant, pour satisfaire la formule du programme, il est nécessaire d'observer ces deux histoires car aucune des deux n'est cohérente seule. Le modèle (3) correspond à l'union des histoires des modèles (1) et (2). De manière informelle, on pourrait donc interpréter ce programme comme l'expression de causes alternatives au déclenchement de G avec cependant la nécessité d'observer les deux alternatives pour le satisfaire.

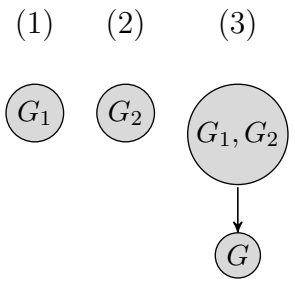


Figure 2.8 – modèle de Kripke satisfaisant le programme $G \circ\!\!\rightarrow G_1 + G_2$ sans observateur.

Cause conjointe.

$$\begin{array}{ll}\{ & G_1 + G_2 \circ\!\!\rightarrow G; \\ & obs :: G; \}\end{array}$$

La formule correspondant à ce programme est :

$$\begin{array}{ll}\mathbf{A}(& G \rightarrow \langle\ \rangle^{-} G_1 \wedge G_2 \quad \wedge \\ & @_{obs} G)\end{array}$$

Le modèle de Kripke minimal correspond à celui de la figure 2.7 (1). Il indique la nécessité d'avoir la présence des deux causes au sein d'un même monde. En effet, le modèle (2) ne satisfait pas la formule. En se rapportant à l'histoire qu'il décrit, il s'interprètera comme la présence nécessaire des deux causes lorsque l'effet est déclenché et que l'on exprimera comme G_1 *et* G_2 *cause* G.

Préemption de l'effet. Considérons le programme suivant :

$$\{G \circ\!\!\rightarrow G_1 + G_2; \}$$

La formule correspondant à ce programme est :

$$\mathbf{A}(G_1 \wedge G_2 \rightarrow \langle\ \rangle^{-} G)$$

Dans cet exemple, l'observateur n'a pas été inséré intentionnellement afin de montrer différents cas de satisfiabilité de cette formule par un modèle. En considérant que les agents effets doivent toutefois apparaître, trois modèles élémentaires satisfont cette formule.

Dans les deux premiers exemples de la figure 2.8 l'absence d'un des effets n'impose pas la présence de la cause dans un monde le précédant par la relation d'accessibilité. Toutefois remarquons que ces exemples satisfont la formule. Par contre, lorsque que les deux effets sont présents la cause doit les précéder nécessairement. Ils illustrent la prise en compte du phénomène de préemption d'un effet qui est un élément central pour la définition des règles sémantiques (Règles 4,5,6 Table 2.3). Il faut aussi remarquer que les observateurs sont déterminants dans le processus de validation pour identifier les propriétés recherchées du comportement. Sans eux, un modèle de Kripke vide, c'est-à-dire sans histoire, satisfait n'importe quel programme. Dans une certaine mesure, la définition des observateurs formalise le fait de bon sens que sans observation/expérience tout peut être considéré comme vrai c'est-à-dire dans notre contexte satisfiable.

L'absence d'observateur dans un programme peut aussi signifier la volonté de ne pas figer ses propriétés en se rapportant à une fonctionnalité précise produisant ces observations. Autrement dit, un programme sans observation peut avoir différentes propriétés fonctionnelles qui seront spécialisées dans différentes applications. Cette considération motive le fait que les composants décrits en **Gubs** et destinés à être assemblés pour former des fonctions biologiques de synthèse ne possèdent pas d'observateur.

De ces exemples, on peut conclure que l'interprétation informelle déduite des modèles de Kripke repose sur les histoires/chemins contenus dans ces modèles. Ces histoires peuvent être considérées comme différentes facettes du comportement du système biologique généré répondant aux propriétés décrites par le programme. Elle met aussi en avant le rôle des observateurs pour la spécialisation d'une fonctionnalité d'un système biologique.

2.3.3 Observabilité forte

L'observabilité est une propriété du programme qualifiant le fait que les actions engendrées peuvent potentiellement être toutes observées dans au moins une expérience, c'est-à-dire une histoire. En effet, il peut arriver qu'aucune histoire ne soit cohérente avec un comportement programmé. Par exemple, le programme $\{Obs :: g, \overline{g} \odot\!\!\rightarrow g\}$ n'est pas observable dans une trace. En effet, son interprétation donne la formule suivante : $\mathbf{A}((@_{Obs}g) \wedge (g \rightarrow ((\langle\ \rangle^{-}\neg g) \wedge \neg g)))$, qui est fausse dans tous les modèles. Ainsi *Obs* doit à la fois satisfaire g et $\neg g$ par définition de la dépendance persistante.

Un programme **Gubs** sera dit *observable* si et seulement si la formule résultant de son interprétation est valide dans un modèle au moins (*i.e.*, ce modèle satisfait cette formule). Par conséquent, l'interprétation d'un programme inobservable est une contradiction. Un programme non observable peut être assimilé à une erreur de programmation.

L'observation du comportement est essentielle pour valider un programme. Cette validation peut en particulier permettre de vérifier la présence d'incohérences comportementales, telles que des comportements contradictoires. En effet, deux comportements contradictoires mèneront à une formule logique insatisfiable. La détection de ces erreurs peut être effectuée au moment de la compilation en utilisant la méthode à tableaux [33] qui détermine automatiquement si une formule est satisfiable dans un modèle. En effet, **Gubs** utilise un fragment de la logique $\mathcal{H}(\mathbf{A}, @)$ nommé *HL(@)* qui est décidable. Ceci nous permet d'appliquer les méthodes à tableaux en utilisant les outils automatisés tels que Spartacus, Htab ou Herod [63, 67, 36].

L'observabilité étant fondée sur l'interprétation d'un programme traduit en une formule de logique hybride, un programme observable correspond à une formule satisfiable. Par conséquent, nous utilisons les méthodes à tableaux pour la logique hybride qui sont prouvées décidables pour les fragments logiques hybrides sans Le *binder* [71]. Pour ce fragment, les méthodes à tableaux sont prouvées exp-time avec un minimum logarithmique [43]. En se référant à la sémantique de **Gubs** (Table 2.3), les formules résultantes sont en forme conjonctive avec au maximum 3 clauses disjonctives dans le cas de causes persistantes. Chaque application des règles de disjonction va créer une nouvelle branche dans l'arbre formé par la résolution à tableaux, la complexité résultant des formules de **Gubs** sera donc en $\mathcal{O}(3^n)$ où n est le nombre de ligne du programme *normalisé* (Section 3.2) nous pouvons en déduire la proposition suivante :

Proposition 1. *Déterminer l'observabilité d'un programme* **Gubs** *se calcule en un temps de complexité* $\mathcal{O}(3^n)$ *où* n *est le nombre de lignes du programme.*

Le temps d'exécution des méthodes à tableaux dans notre cas les rend rapidement inexécutable pour des programmes de taille raisonnable. Afin d'assurer la correction du code, nous introduirons dans le chapitre compilation (Chapitre : 3) la notion d'observabilité faible permettant de tester la non-observabilité d'un programme en

temps polynomial.

Pour conclure cette section, nous proposons dans la figure 2.4 un fragment du programme à deux agents présenté dans la section 2.4. Ce fragment est interprété en logique hybride selon la sémantique. Ainsi les relations d'ordre < entre attributs $Cro : \{low < high\}$ se traduisent par une relation de cause logique et un renommage des constantes pour représenter chaque attribut : $Cro\_high \rightarrow Cro\_low$, les causes persistantes, rémanentes et normales sont quant à elles traduites simplement selon la sémantique : $\overline{Cro(low)} \odot\!\!\rightarrow CI(high)$ devient alors $CI\_high \rightarrow ((\langle\ \rangle^{-}\neg Cro\_low) \wedge \neg Cro\_low)$, de même les points d'observation sont définis comme des nominaux tel que : $\mathbf{obs}_1 :: Cro(high)$ devient $@_{obs_1}(Cro\_high)$.

Programme **Gubs**		Interprétation en logique hybride	
{		**A**(	
$Cro : \{low < high\}$	;	$Cro\_high \rightarrow Cro\_low$	$\wedge$
$CI : \{low < high\}$	;	$CI\_high \rightarrow CI\_low$	$\wedge$
...		...	
$\overline{Cro(low)} \odot\!\!\rightarrow CI(high)$	;	$CI\_high \rightarrow ((\langle\ \rangle^{-}\neg Cro\_low) \wedge \neg Cro\_low)$	$\wedge$
$Cro(low) \circ\!\!\rightarrow \overline{CI(high)}$	;	$\neg CI\_high \rightarrow (\langle\ \rangle^{-} Cro\_low)$	$\wedge$
$\mathbf{obs}_1 :: Cro(high)$	;	$@_{obs_1}(Cro\_high)$	$\wedge$
...		...	
}		)	

Table 2.4 – Fragment de l'interprétation du programme **Gubs** correspondant au modèle de régulation du λ-phage à deux agents en logique hybride.

2.4 Exemples

Dans cette section, afin d'illustrer la syntaxe et la sémantique de **Gubs** nous reprenons deux exemples correspondant aux réseaux de régulation immunitaire du λ-phage tel que décrit par René Thomas[106]. Cette régulation possède deux modèles, un à deux gènes, décrit précédemment dans la figure 2.3 et un à quatre que nous reprenons dans la figure 2.9.

Cette figure propose une description en **Gubs** décrivant en détail le réseau à quatre états. Notons que ce modèle peut être simplifié vers celui à deux états en

Bactériophage – λ
$\{Cro : \{low < mid < high\};$
$\{CI : \{low < mid < high\};$
$\{N : \{low < mid < high\};$
$\{CII : \{low < mid < high\};$
$Cro(low) \xrightarrow{-} CI(high);$
$Cro(mid) \xrightarrow{-} N(high);$
$Cro(high) \circ\!\!\rightarrow Cro(mid);$
$Cro(high) \xrightarrow{-} CII(high);$
$N(low) \xrightarrow{+} CII(high);$
$CII(low) \xrightarrow{+} CI(high);$
$CI(mid) \xrightarrow{-} Cro(high);$
$CI(mid) \circ\!\!\rightarrow CI(high);\}$
$CI(mid) \xrightarrow{-} CII(high);$
$CI(low) \xrightarrow{-} N(high);$

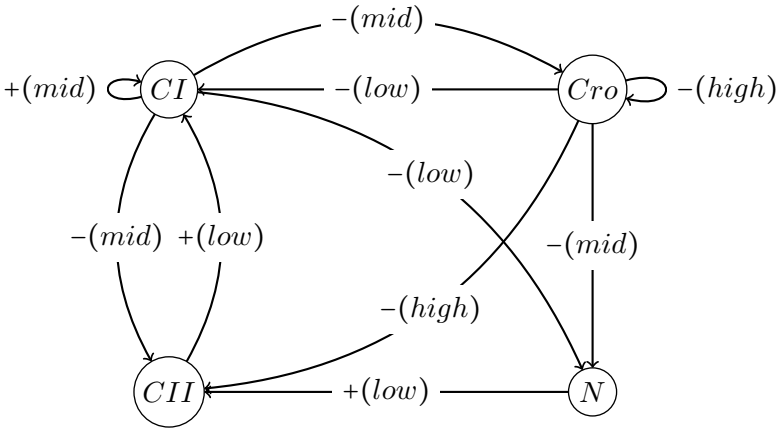


Figure 2.9 – modèle du réseau de régulation du λ-phage à quatre états

considérant CII et N comme non clés lors de l'expérience. La complexité relative du programme résultant est donc ici principalement due à la description détaillée de chaque transition du modèle. Dans un second temps, nous proposons de traduire le modèle à deux agents en logique hybride (Table 2.5), puis enfin en un modèle de Krikpe (Figure 2.10).

Programme **Gubs**		Interprétation en logique hybride	
{		**A**(	
$Cro : \{low < high\}$	;	$Cro\_high \rightarrow Cro\_low$	$\wedge$
$CI : \{low < high\}$	;	$CI\_high \rightarrow CI\_low$	$\wedge$
$\overline{Cro(low)} \odot\!\!\rightarrow CI(high)$	;	$CI\_high \rightarrow ((\langle\,\rangle^{-}\neg Cro\_low) \wedge \neg Cro\_low)$	$\wedge$
$Cro(low) \circ\!\!\rightarrow \overline{CI(high)}$	;	$\neg CI\_high \rightarrow (\langle\,\rangle^{-} Cro\_low)$	$\wedge$
$\overline{Cro(high)} \circ\!\!\rightarrow Cro(low)$	;	$Cro\_high \rightarrow (\langle\,\rangle^{-} Cro\_low)$	$\wedge$
$\overline{CI(low)} \odot\!\!\rightarrow Cro(high)$	;	$Cro\_high \rightarrow ((\langle\,\rangle^{-}\neg CI\_low) \wedge \neg CI\_low)$	$\wedge$
$CI(low) \circ\!\!\rightarrow \overline{Cro(high)}$	;	$\neg Cro\_high \rightarrow (\langle\,\rangle^{-} CI\_low)$	$\wedge$
$CI(low) \circ\!\!\rightarrow CI(high)$	;	$CI\_high \rightarrow (\langle\,\rangle^{-} CI\_low)$	$\wedge$
$\mathbf{obs}_1 :: Cro(high)$	;	$@_{obs_1}(Cro\_high)$	$\wedge$
$\mathbf{obs}_2 :: Cro(low)$	;	$@_{obs_2}(Cro\_low)$	$\wedge$
$\mathbf{obs}_3 :: CI(low)$	;	$@_{obs_3}(CI\_low)$	$\wedge$
}		)	

Table 2.5 – Interprétation du programme **Gubs** correspondant au modèle de régulation du λ-phage à deux agents en logique hybride.

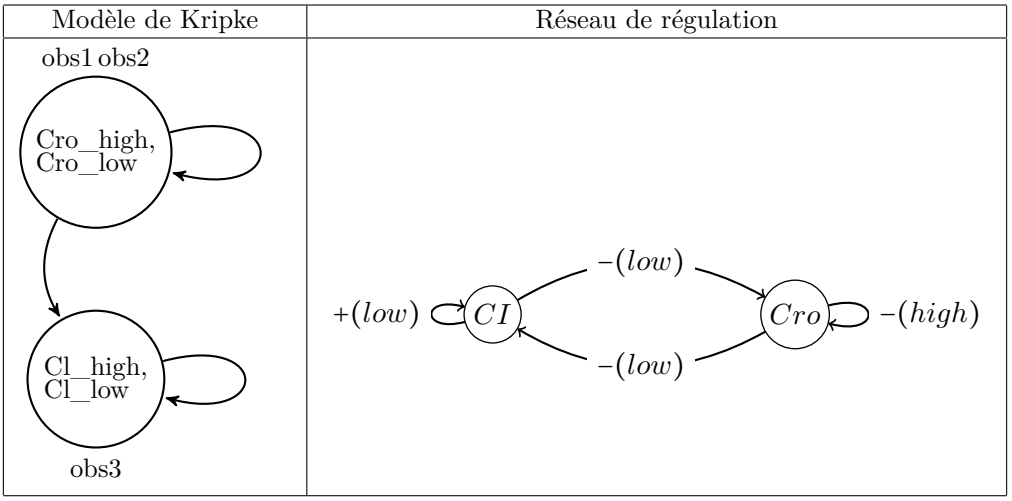


Figure 2.10 – Modèle de Kripke correspondant à l'interprétation du programme **Gubs** représentant le réseau de régulation du λ-phage à deux états. Dans ce modèle, les variables propositionnelles sont Cro_high, Cro_low, Cl_high et Cl_low. Les nominaux sont $obs1$, $obs2$ et $obs3$. L'ensemble des mondes correspond aux deux mondes du modèle, libellés par les nominaux $(obs1, obs2)$ pour le premier et $obs3$ pour le deuxième. Il n'y a pas de symbole relationelles, aucun contexte n'existant.

On notera notamment la ressemblance entre le modèle de Kripke et le réseau de régulation original.

Dans ce chapitre, nous avons introduit le formalisme de **Gubs** afin de spécifier le comportement de systèmes biologiques. Pour cela nous avons défini un langage adapté aux systèmes ouverts prenant en compte leurs spécificités. S'agissant d'un langage de spécification, nous l'avons fondé sur une sémantique en logique modale. Ceci nous permet de plus de définir la notion d'observabilité permettant d'assurer la correction du programme. L'objectif étant d'obtenir à partir de cette spécification un assemblage de composants biologiques, nous introduisons dans le chapitre suivant la compilation d'un programme **Gubs** vers un assemblage d'éléments biologiques.

Deuxième partie

Ggc, un compilateur de spécifications comportementales

Chapitre 3

Compilation

Résumé : Dans ce chapitre, nous abordons les principes de compilation inhérents à la synthèse de fonctions biologiques à partir d'un langage. En effet, là où classiquement en informatique la compilation repose sur une traduction d'un langage vers un autre, et donc un changement de syntaxe, la compilation d'une fonction biologique de synthèse repose sur une autre approche consistant à trouver des composants dont l'assemblage possède un comportement analogue à celui spécifié. La compilation est le résultat d'un assemblage de composants dont le comportement collectif réalise la fonction du programme. Dans un premier temps nous introduirons informellement les grandes lignes de la compilation, et sa problématique biologique. Puis nous formaliserons la notion d'*inclusion comportementale* formalisant l'analogie de synthèse fonctionnelle qui repose techniquement sur l'ACI unification. Nous introduirons enfin les algorithmes de compilation pour conclure sur les raffinements induits par la dimension biologique et une présentation de l'outil **Ggc** ainsi que des résultats obtenus.

La compilation définit l'ensemble des méthodes permettant de traduire un programme dans un langage de plus bas niveau. Dans notre contexte, la compilation consiste à définir à partir de la spécification d'une fonction biologique un assemblage de composants dont le comportement collectif est analogue comportementalement à cette spécification. Cette approche se distingue de celle des langages informatiques.

En effet, usuellement, la compilation [1, 2] se définit comme la traduction d'un langage en un autre de plus bas niveau, qu'il s'agisse d'un langage de haut niveau vers du binaire exécutable ou bien un bite-code (exécutable par une machine virtuelle). La compilation nous introduit la nécessité de l'outil compilateur qui permet de spécifier dans un langage de haut niveau des programmes potentiellement extrêmement complexes, et ainsi de réduire les risques d'erreurs si ces programmes avaient à être codés directement dans un langage de bas niveau.

Un premier changement évident est le médium utilisé, là où l'informatique utilise un processeur de silicium dont l'ensemble des fonctions est bien défini, la biologie utilise un vecteur cellulaire. Ce vecteur évolue [92] et possède des fonctionnalités qui lui sont propres, nécessaires à sa survie et potentiellement inconnues qui peuvent interagir avec le programme et produire un comportement différent de celui spécifié à l'origine. Nous sommes donc confrontés au problème de trouver un vecteur adéquat, et un système résistant aux modifications introduites par ce vecteur.

Le deuxième changement concerne la cible de compilation. Si le support final peut être la séquence d'ADN, les composants utilisés peuvent être de différentes natures et de différents niveaux. Un composant peut être un promoteur, une cassette ou bien un système biologique tel qu'un virus par exemple. Ainsi un composant est un objet abstrait ayant différentes échelles. De plus ceci limite les possibilités de traduction classique par décoration d'arbre syntaxique car les objets composés sont de taille et de complexités différentes. Il est donc nécessaire de trouver un cadre méthodologique unifiant leur traitement à la compilation.

La troisième différence réside dans la dimension ouverte du système : l'environnement peut interagir avec notre spécification, et ces interactions peuvent, dans le programme compilé, produire des comportements non spécifiés, voir contraires à ceux spécifiés. Bien que l'environnement soit modélisable lors de la programmation sous **Gubs**, notons qu'il s'agit ici d'interactions non spécifiées ou inconnues. En effet, un composant biologique peut reproduire un comportement donné ou un sous-ensemble des comportements de la spécification, l'assemblage de ces composants ne

reproduira pas forcément le comportement global de la spécification, et possèdera potentiellement des comportements supplémentaires, dus soit à l'assemblage, soit à la connaissance partielle des propriétés du composant biologique.

Nous devons donc trouver une méthode permettant de coupler des composants biologiques à une spécification formelle. La problématique de la synthèse fonctionnelle se traduit par les questions suivantes :

— Etant donné un assemblage de composants Q, cet assemblage est-il observable ?
— Étant donné un ensemble de composants Q formant un programme observable, cet ensemble inclut-il comportementalement P ?
— Étant donné un ensemble de composants Q, est-ce le plus petit assemblage couvrant P ?

Dans ce chapitre nous introduisons les étapes de la compilation d'un programme **Gubs** en un système biologique. Dans un premier temps, nous devons vérifier l'observabilité du programme, puis nous allons assembler des composants biologiques afin de simuler le comportement décrit par le programme. Comme le design correspond ici à une description comportementale/fonctionnelle, nous avons besoin de combler le fossé entre la description structurelle et fonctionnelle.

Cette étape est appelée la *synthèse fonctionnelle*. L'objectif est de sélectionner un ensemble de composants dont l'assemblage conserve le comportement du programme. Ensuite, plusieurs assemblages de composants pouvant répondre à la spécification du programme, nous devons définir celui étant le plus optimal.

L'ensemble de ces étapes implique de définir une forme normale afin de faciliter la comparaison entre programme et leur optimisation. Nous allons aussi devoir définir l'observabilité d'un assemblage afin de sélectionner les composants compatibles. Enfin, la sélection de programmes s'appuiera sur des contraintes biologiques. Finalement nous définirons une chaîne de compilation décrite à la figure 3.3

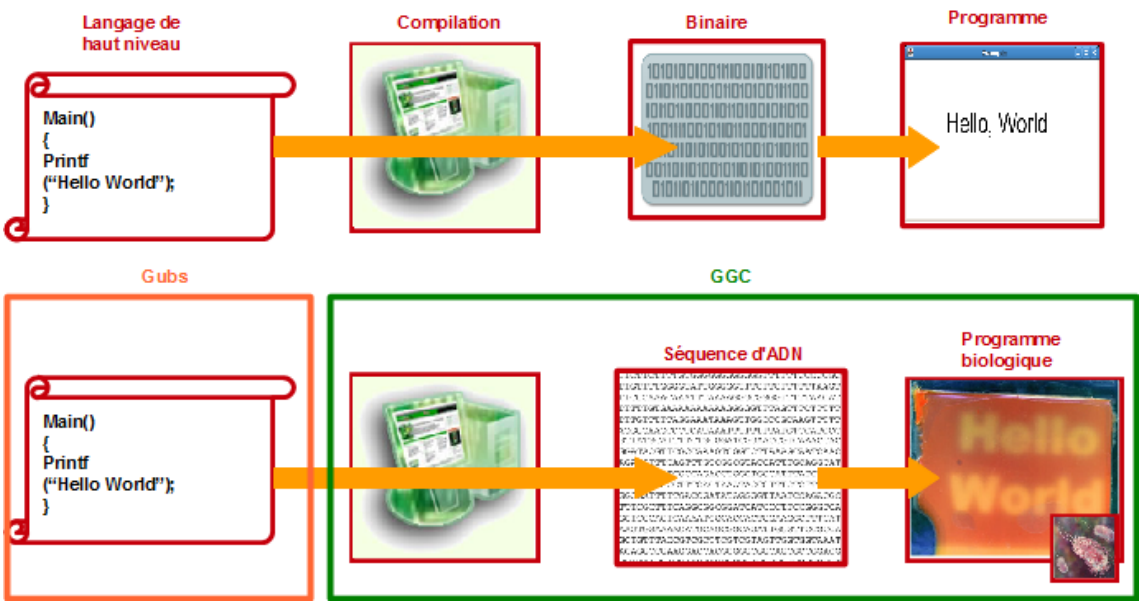


Figure 3.1 – Compilateur

3.1 Présentation

Gubs étant un langage de description comportementale, nous devons trouver un ensemble d'éléments couvrant le comportement du programme décrit, et non simplement sa syntaxe. La *synthèse fonctionnelle* repose sur un ensemble de règles et de propriétés permettant de répondre à l'idée de compilation d'un système biologique, et donc d'une spécification partielle. La synthèse fonctionnelle se traduit en plusieurs étapes :

Avant toute chose, nous devons vérifier que le programme est observable, ceci se fait grâce à la sémantique de **Gubs**, un programme observable sera défini comme un programme dont la traduction en logique hybride est satisfiable, en utilisant la propriété d'*observabilité* comme défini dans la section sémantique (section 2.3). La première étape de compilation consiste ensuite à réorganiser le programme sous forme normale afin d'avoir une spécification homogène des relations facilitant la compilation et permettant de caractériser la fonctionnalité des relations causales.

Ensuite la compilation consiste en la définition d'un assemblage de composants dont le comportement collectif « simule » le comportement spécifié.

Néanmoins, trouver un composant simulant chaque comportement du programme ne répond pas entièrement à la problématique, encore faut-il que l'ensemble de

ces composants forme un système biologique fonctionnel. La première étape pour arriver à un tel système consiste à vérifier que cet ensemble de composants est lui-même observable, à cette fin nous définissons la notion d'*observabilité forte*. La seconde étape consiste à vérifier que pour chaque variable du programme d'origine, celle-ci n'est unifiée que par une unique constante dans la réalisation finale. Nous définissons pour cela un système formel caractérisant les opérations d'assemblage. Ce système est prouvé correct (voir en annexe) par rapport à la propriété « d'inclusion comportementale » et correspond techniquement à l'*ACI-Unification*. En effet, un programme peut être vu comme un ensemble de comportements, de même pour la base de données, le problème se réduit donc à trouver un sous-ensemble de la base de données couvrant l'ensemble des composants du programme. Néanmoins, le problème de l'ACI-Unification étant NP-difficile, nous utilisons un ensemble de propriétés sur la syntaxe du langage afin de simplifier sa complexité. Et dans le cadre de grandes bases de données, nous utilisons un algorithme d'approximation utilisant les méthodes propres aux algorithmes génétiques.

Afin de prendre en compte la dimension biologique des systèmes, nous avons introduit dans **Gubs** un ensemble de contraintes sous forme de méta-informations à la compilation (pouvant être ajoutées par l'utilisateur) permettant de raffiner la sélection des composants, telles que l'*orthogonalité* des éléments sélectionnés, mais aussi l'utilisation en priorité de *promoteurs* forts, et la sélection de composants simples, avec le moins d'effets non voulus. De plus nous devons minimiser le risque que les propriétés du système autres que celles décrites dans la spécification (par exemple ce qui permet au système de survivre ou de se reproduire) aient un impact sur le comportement spécifié.

3.2 Normalisation d'un programme

Dans cette première section nous introduisons les mécanismes permettant de mettre en forme normale un programme **Gubs**. L'objet de la mise en forme normale est de définir d'une part une forme homogène aux relations causales et d'autre part d'identifier une structure interne codant les relations causales. Elle nous permet d'exprimer les relations sous forme de termes, ce qui nous permet de traduire la problématique de la compilation en celle de l'ACI-unification afin d'appliquer un algorithme d'unification sur l'ensemble des composants.

À cette fin nous introduisons un système formel définissant les propriétés des agents et de leurs états [1]. Les règles (SCom.), (SAss.) et (SCont.) décrivent respectivement la commutativité, l'associativité et la non-redondance des agents. Finalement, (Incl.) définit le fait qu'un comportement peut être étendu avec un autre à partir du moment où le comportement original reste présent.

- États des agents -

$$\frac{S_1 + S_2}{S_2 + S_1}\ (\text{SCom.}) \qquad \frac{S_1 + (S_2 + S_3)}{(S_1 + S_2) + S_3}\ (\text{SAss.}) \qquad \frac{S + s}{S + s + s}\ (\text{SCont.}) \qquad \frac{S + s}{S}\ (\text{Incl.})$$

Table 3.1 – Règles pour les états des agents. S_i décrit un ensemble, $s_1 + \ldots + s_n$, d'états d'agent, incluant des négations, et Δ décrit le reste du programme.

Ces règles permettent de définir les collections d'agents comme des ensembles.

3.2.1 Forme normale

Nous définissons la forme normale des relations causales définies dans **Gubs**, $\aleph(P)$, avec les propriétés suivantes :

1. Unicité : Pour un programme P donné, sa forme normale est unique.
2. Idempotence : $\aleph \circ \aleph(P) = \aleph(P)$
3. Conservation de la sémantique : $\forall P : \mathcal{M} \Vdash [\![P]\!] \Leftrightarrow \mathcal{M} \Vdash [\![\aleph(P)]\!]$
4. Normalisation : $\forall P_1, P_2 : \aleph(P_1) = \aleph(P_2) \Leftrightarrow (\mathcal{M} \Vdash [\![P_1]\!] \Leftrightarrow \mathcal{M} \Vdash [\![P_2]\!]$

La transformation d'un programme P en sa forme normale repose sur des règles classiques de commutativité et de distributivité. En effet, l'ensemble des causes sont commutatives 3.3. On peut donc introduire un ordre total sur les agents correspondant à l'ordre lexicographique, et l'étendre aux relations causales en les ordonnant dans un premier temps par catégorie (normale, persistante, rémanente) puis par leur agent le plus à gauche de la cause en premier, puis ainsi de suite jusqu'à celui le plus à droite de l'effet. En forme normale, le contexte est distribué à chaque cause et les agents sont renommés en les étendant par l'ensemble des compartiments les encapsulant. De même les définitions d'attribut sont distribuées à l'ensemble des causes. Ainsi à chaque cause est associé l'ensemble des définitions d'attribut des

1. Les règles sont de la forme : $\frac{\text{hypothèse}}{\text{conclusion}}$.

agents y apparaissant.

$$[k_1, ...k_n](\{A_C \cup A_E\}, \sum_{i=1}^{m} C_i \circledast\!\!\rightarrow \sum_{j=1}^{l} E_j)$$

où $k_1, ...k_n$ correspond à l'ensemble des agents des contextes incluant la relation causale, A_C et A_E correspondent respectivement à l'ensemble des définitions d'attributs des agents présents dans la cause et dans l'effet, C l'ensemble des agents présents dans la cause, et E ceux dans l'effet.

Par exemple, la forme normale de $[k]C\{a \bigcirc\!\!\rightarrow b\}D\{c \bigcirc\!\!\rightarrow d\}$ est

$$[k]C.a \bigcirc\!\!\rightarrow C.b, [k]D.c \bigcirc\!\!\rightarrow D.d$$

La forme normale $\varphi = [k_1, ...k_n]\{A_C \cup A_E\}, \sum_{i=1}^{m} C_i \circledast\!\!\rightarrow \sum_{j=1}^{l} E_j$ d'une relation causale p peut alors être interprétée comme un terme $\theta(p) = (\circledast\!\!\rightarrow)(\{K\}, \{A\}, \{C\}, \{E\})$, où $A = \{a_{c_1}, ...a_{e_l}\}$ est l'ensemble des définitions d'attributs de toutes les variables et/ou constantes apparaissant dans {C} et {E} et où $K = \{k_1, ...k_n\}$, $C = \{c_1, ...c_m\}$ et $E = \{e_1, ...e_l\}$ sont les ensembles d'agents constitués de variables et de constantes dans le cadre de P et uniquement de constantes pour les composants de la base de données de composant Q.

Finalement, un programme **Gubs** se traduit en un ensemble de termes : $\Theta = \{\theta(p); p \in P\}$.

3.3 Synthèse fonctionnelle

Dans le cadre de la biologie de synthèse, à l'étape de compilation, un programme est transformé en un assemblage de composants biologiques insérés dans un vecteur cellulaire (par exemple une bactérie), qui doit se comporter conformément à la spécification programmée. Cet assemblage comporte plusieurs composants extraits d'une bibliothèque (*e.g.*, "parts registry"). Cette phase de synthèse fonctionnelle a pour objectif de sélectionner un ensemble de composants dont le comportement collectif reproduit le comportement spécifié par le programme. A cette fin, à chaque composant on associe un programme **Gubs** décrivant son comportement. De ce fait, l'assemblage de composants correspond à un ensemble de programmes préservant le comportement du programme compilé.

La préservation du comportement se formule par l'*inclusion comportementale* qui formalise le fait que les traits observationnels d'une fonction spécifique doivent être reconnus dans une trace correspondant aux résultats expérimentaux de la fonction d'un assemblage de composants. Un assemblage est fonctionnellement correct si dans son comportement, celui du programme original est retrouvé. En d'autres termes, nous pouvons construire des histoires consistantes avec le comportement programmé à partir des histoires consistantes avec la description comportementale du système. L'inclusion comportementale est définie à partir de l'interprétation du programme comme une conséquence logique (Définition 4).

Définition 4 (Inclusion comportementale)**.** *Un programme Q* inclut comportementalement *un autre programme P, si et seulement si l'interprétation de ce dernier (P) est une conséquence logique de celle du premier (Q) :*

$$P \sqsubseteq Q \triangleq \forall \mathcal{M} : \mathcal{M} \Vdash [\![Q]\!] \implies \mathcal{M} \Vdash [\![P]\!] .$$

où $\mathcal{M}$ est un modèle (une structure de Kripke)

L'inclusion comportementale est un pré-ordre [2] tel que le programme vide noté ϵ est un minimum ; ce qui signifie qu'un programme avec aucun comportement attendu peut être observé dans toutes les traces. Un programme dont l'interprétation est $\bot$, est quant à lui un maximum. La figure 3.2 illustre l'inclusion comportementale pour un programme P.

Cette définition de l'inclusion comportementale nous permet de déduire la proposition 2. En effet, si un programme P est observable, alors il est nécessaire que l'assemblage de composants biologiques pour couvrir le comportement de P soit observable aussi afin de préserver l'observabilité lors de l'inclusion comportementale.

Proposition 2. *Un programme inclus comportementalement dans un programme observable,* ***obs P****, est observable :* $\forall P, Q \in \mathsf{P} : (\boldsymbol{obs}\, Q) \wedge (P \sqsubseteq Q) \implies \boldsymbol{obs}\, P$.

Nous allons à présent nous concentrer sur l'observabilité afin de répondre à la question : Étant donné un assemblage de composants Q, cet assemblage est-il observable ?

2. c'est une relation réflexive et transitive

P	Q
$\{g_1 \bigcirc\!\!\rightarrow g_3,$	$\{[k_1]\{g_0 \bigcirc\!\!\rightarrow g_1\},$
$[k_3]\{g_3 \bigcirc\!\!\rightarrow g_4\},$	$[k_2]\{g_0 \bigcirc\!\!\rightarrow g_2\},$
$[k_4]\{g_3 \bigcirc\!\!\rightarrow g_5\},$	$g_1 \bigcirc\!\!\rightarrow g_3,$
$[k_6]\{g_8 \bigcirc\!\!\rightarrow g_9\},$	$g_2 \bigcirc\!\!\rightarrow g_5,$
$[k_7]\{g_8 \bigcirc\!\!\rightarrow g_{10}\},$	$[k_3]\{g_3 \bigcirc\!\!\rightarrow g_4\},$
$g_9 \bigcirc\!\!\rightarrow g_{11},$	$[k_4]\{g_3 \bigcirc\!\!\rightarrow g_5\},$
$g_{10} \bigcirc\!\!\rightarrow g_{11},$	$g_6 \bigcirc\!\!\rightarrow g_8,$
$a :: g_4\}$	$[k_5]\{g_6 \bigcirc\!\!\rightarrow g_7\},$
	$[k_6]\{g_8 \bigcirc\!\!\rightarrow g_9\},$
	$[k_7]\{g_8 \bigcirc\!\!\rightarrow g_{10}\},$
	$g_9 \bigcirc\!\!\rightarrow g_{11},$
	$g_{10} \bigcirc\!\!\rightarrow g_{11},$
	$a :: g_4, b :: g_5, c : g_{11}\}$

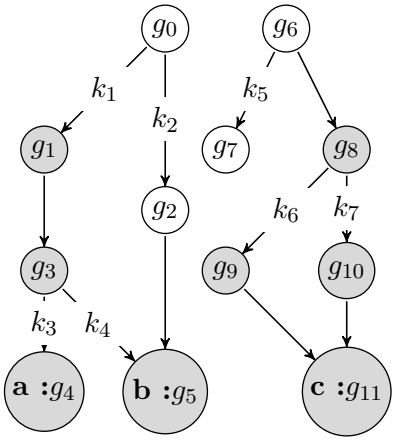


Figure 3.2 – Exemple d'inclusion comportementale. Si nous définissons le modèle de P comme l'ensemble des mondes colorés en gris, alors les histoires consistantes pour P contiennent nécessairement les mondes colorés en gris. A partir des points d'observation a, b, c, le modèle correspondant aux mondes en gris valide le modèle d'origine. De plus on notera que le comportement de P est inclus dans le modèle de Q représenté par le graphe en entier.

En effet, la proposition 2 impose que l'assemblage soit observable. Nous vérifions donc au préalable de la synthèse que l'assemblage proposé le soit. Néanmoins, le temps de vérification par les méthodes à tableaux de l'observabilité réduit son application à des assemblages de faible taille. Pour contourner ce problème de temps d'exécution, nous allons introduire une condition suffisante appelée *observabilité forte* qui permet de définir un algorithme en temps polynomial. Si l'observabilité forte est satisfaite, alors le programme est observable, c'est à dire que son interprétation est satisfiable.

3.3.1 Observabilité forte

L'observabilité notée *obs*, consiste à vérifier que les éléments assemblés soient compatibles. C'est-à-dire que l'assemblage est valide. Cette vérification s'applique au niveau de la base de données. Comme la compilation procède par assemblage de composants, nous allons définir une relation de compatibilité permettant de définir si deux composants peuvent être assemblés.

$$Q_1 \psi Q_2 \triangleq \mathbf{obs}(Q_1 \cup Q_2)$$

Cependant, nous souhaitons définir des ensembles de Q_i tous compatibles ensemble

et non simplement deux à deux. En effet, nous voulons limiter le fait que la sélection de plusieurs composants interdise celle d'un autre à cause de son incompatibilité avec l'assemblage de ces composants, malgré sa compatibilité avec chacun d'eux.

$$Q_1 \psi Q_2, Q_2 \psi Q_3, Q_3 \psi Q_1$$

n'implique pas nécessairement

$$\mathbf{obs}(Q_1 \cup Q_2 \cup Q_3)$$

car

$$\mathbf{obs}(Q_1 \cup Q_2), \mathbf{obs}(Q_2 \cup Q_3), \mathbf{obs}(Q_3 \cup Q_1)$$

n'implique pas

$$\mathbf{obs}(Q_1 \cup Q_2 \cup Q_3).$$

Or, dans ce cadre, ψ apparaît ne pas avoir les propriétés suffisantes pour satisfaire à cette condition, comme le montre l'exemple suivant, soit :

1. $Q_1 = a \odot\!\!\rightarrow b$,
2. $Q_2 = b \odot\!\!\rightarrow c$,
3. $Q_3 = a \odot\!\!\rightarrow \bar{c}$.

Ces composants, bien que compatibles deux à deux, ne sont pas compatibles tous les trois ensemble. En effet, la présence d'une causalité persistante entraîne le fait que pour Q_1, la présence de a entraîne la présence de a, b par la suite. Or, la présence de b implique la présence de b, c par Q_2 et la présence de a implique aussi la présence de $\bar{c}, a$. Si on associe Q_1 et Q_2 on doit nécessairement avoir un monde avec $\{a, b, c\}$ et en associant Q_3 aux deux premiers on devrait donc avoir un monde avec $\{a, b, c, \bar{c}\}$, ce qui est impossible. L'observabilité forte se qualifie donc par une relation entre composants et est définie comme suit :

Définition 5 (Observabilité forte)**.** *Soit* Γ *un ensemble de composants, une relation* $\Xi \subseteq \Gamma \times \Gamma$ *est dite fortement observable si et seulement si elle possède les propriétés suivantes :*

1. *Equivalence :* Ξ *est une relation d'équivalence (réflexive, symétrique et transitive).*
2. *Observabilité :* $\forall Q_1 \in \Gamma, \forall Q_2 \in \Gamma, Q_1 \,\Xi\, Q_2 \Rightarrow \boldsymbol{obs}(Q_1 \cup Q_2)$

3. *Agrégation :* $\forall Q_1, Q_2, Q_3 \in \Gamma, Q_1 \,\Xi\, Q_2 \wedge Q_1 \,\Xi\, Q_3 \Leftrightarrow Q_1 \,\Xi\, Q_2 \cup Q_3$

L'une des propriétés importantes de cette propriété est le fait que les composants reliés par celle-ci forment eux-mêmes un assemblage observable.

Théorème 1. *Si les couples* (Q_1, Q_2) *et* (Q_2, Q_3) *sont en relation par* Ξ *et donc observables, alors le triplet* (Q_1, Q_2, Q_3) *est observable.*

$$Q_1 \Xi Q_2, Q_2 \Xi Q_3 \Rightarrow \boldsymbol{obs}(Q_1 \cup Q_2 \cup Q_3)$$

De façon plus générale, soit $Q_1 \dots Q_n$ *un ensemble de composants,*

$$Q_1 \Xi Q_2 \Xi \dots Q_n \Rightarrow \boldsymbol{obs}(\bigcup_{i=1}^{n} Q_i)$$

Nous allons définir la relation d'observabilité forte afin de tester en temps polynomial si un programme est observable. Ce test se fonde sur une condition suffisante, permettant d'assurer en cas de réussite que le programme est observable.

Théorème 2. *Deux règles sont en relation d'observabilité forte si et seulement si il n'existe pas de chaîne causale reliant un état à son opposé. Un état est défini comme l'opposé d'un autre si et seulement si :*

1. *Il s'agit de sa négation ou,*
2. *il existe une relation d'opposition (*$\approx$*) entre ces deux états.*

$$C_1 \circledast\!\!\rightarrow E_1 \overset{A}{\underset{\Xi}{}} C_2 \circledast\!\!\rightarrow E_2$$
$$\triangleq$$
$$\forall x_1 \in C_1 \cup E_1 \cup C_1 \cup E_2 et$$
$$\forall x_2 \in C_1 \cup E_1 \cup C_1 \cup E_2 et$$
$$x_1 \neq opp(x_2)$$

De ce fait, le test peut s'effectuer par assemblages successifs de composants afin d'assurer que l'assemblage final est observable. Il s'effectue par clôture transitive des relations persistantes en vérifiant que pour un agent a nous n'avons pas $\odot \overset{*}{\rightarrow} \bar{a}$.

Afin de répondre à la deuxième question nous définissons le principe de *synthèse fonctionnelle*, décrivant les règles de réécriture utilisées pendant le processus de compilation.

3.3.2 Synthèse fonctionnelle

La synthèse fonctionnelle est l'opération consistant à sélectionner des composants dans une bibliothèque biologique et à les assembler afin de générer un système incluant comportementalement le programme spécifié. Le comportement de chaque composant est décrit comme un programme **Gubs**. Dans son cas le plus simple, la synthèse fonctionnelle peut être considérée comme la substitution de variables par des constantes. Par exemple, dans l'activation suivante $\{G_1 \xrightarrow{+} g_2\}$, g_2 sera substituée par le gène G_2, car le composant Q décrit l'activation $\{G_1 \xrightarrow{+} G_2\}$. Néanmoins, des situations plus complexes peuvent arriver lors de la sélection des composants. Par exemple, si le composant décrivant l'activation $G_1 \xrightarrow{+} G_2$ ne se produit qu'en présence d'une autre régulation *i.e.*, $Q = \{G_1 \xrightarrow{+} G_2, G_3 \xrightarrow{+} G_4\}$ alors la sélection du composant Q ajoute une nouvelle régulation.

3.3.3 Règles de synthèse fonctionnelle

La synthèse fonctionnelle est définie par un ensemble de règles (Table 3.2) gouvernant l'assemblage de composants. Seuls les relations causales et les attributs seront synthétisés fonctionnellement. Les points d'observation sont considérés comme des annotations servant au processus de compilation. Pour assurer la correction, chaque transformation doit préserver le comportement d'origine. De plus, chaque programme résultant de l'application d'une règle doit inclure comportementalement le programme d'origine.

Formellement, la synthèse fonctionnelle est modélisée par une relation sur les programmes notée $\leftarrow$, *i.e.*, $Q \leftarrow_\sigma P$ où P est le programme initial et Q l'assemblage de composants assurant sa synthèse, tels que chaque règle assure que :

$$Q \leftarrow_\sigma P \text{ est correcte par rapport à la substitution } \sigma, \text{signifiant que :}$$
$$P[\sigma] \Subset Q[\sigma] \text{ et } Q[\sigma] \text{ est observable.}$$

Une substitution complète est une application, $\sigma = \{v_i \mapsto b_i\}_i$, sur les variables et les constantes de telle sorte qu'une variable peut être substituée par une autre variable ou une constante, et une constante ne peut être substituée que par elle-même [3]. Par exemple nous avons :

3. $P\sigma$ ou $P[\sigma]$ représente son application à un programme P et les substitutions identité sont omises.

$$\{Obs :: G(l) + b_2, b_1 \circ\!\!\rightarrow G(l)\}[\{b_1 \mapsto B_1, b_2 \mapsto B_2, l \mapsto Low\}] = \{Obs :: G(Low) + B_2, B_1 \circ\!\!\rightarrow G(Low)\}$$

Notons aussi que l'inclusion comportementale est préservée par la substitution (Proposition 3).

Proposition 3. *Pour toute substitution σ, nous avons : $P \Subset Q \implies P[\sigma] \Subset Q[\sigma]$.*

Les règles de synthèse fonctionnelle se décrivent par un système formel décrivant les opérations à effectuer pour l'assemblage (Table 3.2). $P \subseteq_{\text{Asm}} Q$ indique le fait que le programme Q corresponds à un assemblage incluant P *i.e.*, $Q = (Q_1, P, Q_2)$ où Q_1 où Q_2 peuvent être des programmes vides.

- Instanciation -

$$\frac{P[\sigma] \subseteq_{\text{Asm}} Q[\sigma] \qquad \mathbf{obs}(Q[\sigma]) \qquad Q \in \Gamma}{Q \leftarrow_\sigma P} \text{ (Inst.)}$$

- Commutativité, Contraction -

$$\frac{Q \leftarrow_\sigma P, P'}{Q \leftarrow_\sigma P', P} \text{ (Com.)} \qquad \frac{Q \leftarrow_\sigma P}{Q \leftarrow_\sigma P, P} \text{ (Cont.)}$$

- Assemblage -

$$\frac{Q \leftarrow_\sigma P \qquad Q' \leftarrow_{\sigma'} P' \qquad \sigma|_{\text{VA}(P)\cap\text{VA}(P')} = \sigma'|_{\text{VA}(P)\cap\text{VA}(P')} \qquad \mathbf{obs}(Q[\sigma], Q'[\sigma'])}{Q, Q' \leftarrow_{\sigma\cup\sigma'} P, P'} \text{ (Asm.)}$$

Table 3.2 – Règles de synthèse fonctionnelle

La règle (Inst.) décrit le fait qu'une instanciation observable d'une partie d'un composant dans la bibliothèque est synthétisée fonctionnellement. La règle (Com.) exprime la commutativité de l'assemblage. La règle (Cont.) réduit les formules redondantes du programme. Finalement, la règle (Asm.) détaille les conditions pour l'assemblage de deux composants, chacun représentant la synthèse fonctionnelle d'une partie de la fonction spécifiée. Le chapitre 4 consacré à des applications concrètes contient un ensemble de cas d'application de ces règles.

Théorème 3. *Les règles de synthèse fonctionnelle (Table 3.2) définissent les principes de compilation et sont correctes par rapport à la propriété d'inclusion comportementale.*

Fondamentalement, compiler un programme **Gubs** revient à appliquer de façon itérative les règles de réécriture jusqu'à ce que le comportement du composant assemblé couvre celui du programme original.

Techniquement, un tel algorithme correspond à une unification [73] ou chaque relation causale est associé à un terme. Plus précisément il s'agit d'une *ACI-unification* [8] : Associative (règle Asm.), Commutative (règles Com. and SCom.) et idempotente (règles Cont. and SCont.) couplée à un algorithme de satisfiabilité [33] pour valider l'observabilité. Néanmoins, l'ACI-unification est un problème NP-complet[72]. Pour accélérer son traitement, nous allons définir un algorithme qui tire bénéfice des particularités du langage **Gubs** tel que si cet algorithme échoue, alors l'unification n'est pas faisable. Notons que nous pouvons raffiner ce problème en le définissant comme un problème de set-unification [103, 53, 105].

L'ACI-Unification utilise un algorithme qui explore toutes les possibilités jusqu'à trouver une unification ou échouer [9]. Afin d'améliorer cet algorithme, nous allons prendre en compte les spécificités du langage afin de réduire l'espace des possibilités de celui-ci en définissant un ensemble de contraintes préalables à l'unification. L'algorithme de compilation sélectionnera un ensemble de composants dans une base de données qui valide pour un programme P ces contraintes et qui sera défini comme un candidat potentiel à l'unification. Néanmoins cette approche ne réduit pas suffisamment le temps d'exécution de l'algorithme, en effet, sur une grande base de données, le nombre de candidats potentiels peut rapidement exploser. De ce fait, pour une grande base de données, une sous partie de celle-ci sera traitée. Cette sélection est effectuée par un algorithme évolutionnaire *dirigé* où chaque individu représente un sous ensemble possible de composants couvrant P. Par « dirigé », nous exprimons le fait que nous effectuons une pré-sélection de chaque individu (sous-ensemble de composants) afin que ceux-ci indépendamment ne puissent pas causer l'échec de l'unification (Subsection 3.4).

A cette fin, l'algorithme de synthèse fonctionnelle est structuré en deux niveaux : l'algorithme d'ACI-unification et l'algorithme évolutionnaire dirigé. Dans les deux cas, un individu représente un sous-ensemble de composants. La fonction de *fitness* prend en compte trois caractéristiques : le nombre de composants utilisés pour l'unification dans un individu et le nombre de règles unifiées dans le programme d'origine. Ces deux premières caractéristiques sont motivées par la nécessité de réduire la complexité du système résultant. La troisième caractéristique est un ensemble de propriétés

biologiques que nous traiterons dans la section suivante. Finalement les meilleurs individus possèdent le minimum de composants unifiés, le maximum de règles unifiées dans le programme et les meilleures propriétés biologiques.

Notons que pour chaque individu (sous-ensemble de composants), l'unification peut n'utiliser qu'une sous-partie de ses composants. Dans ce cas, les composants non utilisés ne sont pas inclus dans le système final, seuls les composants associés au comportement d'un programme sont sélectionnés. Néanmoins nous parlons ici de composants, si seule une partie des règles d'un composant d'un individu sont utilisées, alors, l'ensemble du composant est utilisé. En effet, un composant est un objet biologique considéré comme indivisible.

L'échec de la compilation est dû à l'impossibilité d'unifier toutes les relations causales du programme. Nous noterons qu'une telle situation est fréquemment due à la représentation abstraite d'un *pathway* biologique dans un programme. Par exemple, si la relation causale suivante est attendue $G_1 \odot\!\!\rightarrow G_2$, là où le *pathway* $G_1 \odot\!\!\rightarrow G_2; G_2 \odot\!\!\rightarrow G_3$ apparaît dans la description du composant biologique. Un autre cas d'échec vient de l'approximation du type de causalité en considérant qu'utiliser une spécification plus contraignante serait inutile dans le programme. Par exemple, la relation causale suivante $G_1 \circ\!\!\rightarrow G_2$ suffit pour définir le comportement attendu et seulement $G_1 \odot\!\!\rightarrow G_2$ est présent dans la description des composants. Afin d'éviter ces deux cas d'erreur, des variations des individus sont générés (mutations d'optimisation dirigée) possédant une spécification modifiée afin de répondre à ces problèmes en utilisant les règles d'optimisation formalisées par les règles Trans., N2P. et R2N. (Table 3.3). Ces variations sont ensuite incluses comme de nouveaux individus dans la population.

- Dépendances -

$$\frac{Q \leftarrow_\sigma S_1 \odot\!\!\rightarrow S_2, S_2 \odot\!\!\rightarrow S_3, \Delta}{Q \leftarrow_\sigma S_1 \odot\!\!\rightarrow S_3, \Delta}\ \text{(Trans.)} \qquad \frac{Q \leftarrow_\sigma S_1 \odot\!\!\rightarrow S_2, \Delta}{Q \leftarrow_\sigma S_1 \circ\!\!\rightarrow S_2, \Delta}\ \text{(N2P.)}$$

$$\frac{Q \leftarrow_\sigma S_1 \circ\!\!\rightarrow S_2, \Delta}{Q \leftarrow_\sigma S_1 \oplus\!\!\rightarrow S_2, \Delta}\ \text{(R2N.)}$$

Table 3.3 – Règles pour les relations causales. S_i décrit un ensemble, $s_1 + \ldots + s_n$, d'états d'agent, incluant des négations, et Δ décrit le reste du programme.

Ces règles définissent les solutions alternatives pour exprimer des comportements similaires. La règle (trans.) étend la chaîne des dépendances persistantes ($S_1 \odot\!\!\rightarrow S_3$) en ajoutant une dépendance intermédiaire (S_2) pour raffiner un *pathway*. La règle

(N2P.) affaiblit la relation de dépendance normale ($S_1 \circ\!\!\rightarrow S_2$) vers une version persistante ($S_1 \odot\!\!\rightarrow S_2$) vu que cette dernière correspond à une dépendance normale avec une propriété en plus. Finalement, la règle (R2N.) affaiblit une dépendance résiduelle ($S_1 \oplus\!\!\rightarrow S_2$) vers une normale ($S_1 \circ\!\!\rightarrow S_2$), celle-ci pouvant être considérée comme une dépendance résiduelle ne possédant une répétition de l'effet que sur une seule étape. En se fondant sur ces règles, toute chaîne causale peut être écrite sous la forme de dépendances persistantes.

Théorème 4. *Les règles de dépendance sont correctes en accord avec la notion d'inclusion comportementale (Table 3.3).*

3.4 Algorithme d'assemblage

Le résultat de la compilation est composé d'une liste des composants et d'une liste de substitutions attribuant une constante pour chaque variable du programme compilé. La résolution est orientée vers un algorithme heuristique visant à trouver un ensemble minimal de composants couvrant le comportement d'un programme. L'algorithme de synthèse fonctionnelle se découpe en plusieurs étapes clés. Dans le cas d'une base de données de petite taille, il se définit comme suit :

1. Définition des relations causales locales ;
2. Initialisation d'un compteur n correspondant au nombre d'exécutions de l'algorithme avant time-out ;
3. Générer une population d'individus où chaque individu couvre localement P ;
4. Sélectionner le *meilleur* individu *observable* Q couvrant P *localement* satisfaisant les contraintes sur les agents ;
5. Appliquer l'ACI-Unification de P avec Q ;
6. Si l'unification est complète, stocker la substitution et la liste des composants utilisés lors de l'unification comme solution, supprimer l'individu unifié, $n = n - 1$;
7. sinon, si la solution est meilleure que la dernière stockée, stocker l'unification partielle et la substitution comme solution temporaire, supprimer l'individu unifié, $n = n - 1$;
8. Répéter les étapes à partir de 4 jusqu'à $n = 0$;

9. Sélectionner la (ou les) meilleure solution stockée (substitution et liste de composants) ;

Nous allons décrire à présent plus en détail chaque étape de cet algorithme.

3.4.1 Contraintes sur les relations causales

L'heuristique implique de définir une relation entre les relations causales de P et les composants Q_j de la base de données. Cette relation prend en compte la structure des relations causales et la nature des agents. P est considéré comme un ensemble de relations causales dites *causes atomiques* $P = \{p_i\}_i$. Cette relation qualifie une *association locale des relations causales* de P où chaque cause atomique de P peut être associée à une cause atomique d'un sous-ensemble de composants $Q = \bigcup_j Q_j$. Cette association définit les composants candidats à l'unification d'un programme : si cette association est impossible, alors l'unification échouera. Par contre, elle ne garantit pas le succès de l'unification car elle ne définit que des conditions locales à chaque règle et ne prend pas en compte les implications des règles causales précédemment unifiées.

Une règle causale p_i est associée à un composant Q_j si et seulement si il existe une règle causale q_j de Q_j telle que :

1. Le type de la cause de q_j est le même que celui de p_i.
2. Toutes les constantes de p_i sont présentes dans q_j de leur côté respectif de la relation causale (cause ou effet).
3. La cardinalité de chaque ensemble d'agents représentant la cause et l'effet de q_j est supérieure ou égale à la cardinalité des ensembles respectifs de p_i.
4. Chaque agent de p_i possède un équivalent dans q_j avec la même relation d'ordre sur ses attributs.
5. Pour chaque agent de p_i, le nombre de ses attributs est inférieur ou égal à son agent équivalent dans q_j.

Une fois que la relation entre les règles causales de P et les composants est établie, l'algorithme sélectionne un sous-ensemble de composants tel qu'il existe au moins une relation causale q_j validant les contraintes précédentes pour chaque p_i de P. Notons que différents composants Q_j de Q peuvent couvrir la même règle causale p_i de P.

3.4.2 Génération d'une population d'individus

Une fois définis les ensembles de composants couvrant chaque cause du programme, nous générons un ensemble d'individus par combinaison de composants couvrant localement P. Par exemple, si $P = A \circ\!\!\rightarrow v_1; v_1 \circ\!\!\rightarrow v_2$; et $Q_1 = A \circ\!\!\rightarrow B;$, $Q_2 = A \circ\!\!\rightarrow C;$, $Q_3 = B, D \circ\!\!\rightarrow C;$,$Q_4 = A \circ\!\!\rightarrow B; B \circ\!\!\rightarrow C$, alors nous pouvons définir d'après les contraintes sur les causes la population suivante : $I_1 = Q_1, Q_1$,$I_2 = Q_1, Q_2$,$I_3 = Q_1, Q_3$,$I_4 = Q_1, Q_4$,$I_5 = Q_2, Q_1$,$I_6 = Q_2, Q_2$,$I_7 = Q_2, Q_3$,$I_8 = Q_2, Q_4$,$I_9 = Q_4, Q_1$,$I_10 = Q_4, Q_2$,$I_11 = Q_4, Q_3$,$I_12 = Q_4, Q_4$. En effet, $Q3$ ne peut se substituer localement à la première cause de P, ne possédant pas la bonne constante.

3.4.3 Contraintes sur les agents

Avant d'appliquer l'algorithme d'ACI-Unification d'un individu Q et de P, nous validons un ensemble de conditions nécessaires sur les agents et vérifions l'observabilité de l'assemblage via la propriété d'observabilité forte. Les contraintes sur les agents sont les suivantes :

1. Le nombre d'occurrences d'une constante dans P doit être inférieur ou égal à son nombre d'occurrences dans Q.
2. Pour chaque variable v de P, il existe au moins une constante de Q qui n'est pas une constante de P et dont le nombre d'occurrences est supérieur ou égal au nombre d'occurrences de v dans P. De plus, la relation d'ordre sur ses attributs doit être la même et le nombre d'attributs de la constante doit être supérieur ou égal au nombre d'attributs de la variable v.

Les contraintes sur les agents sont ensuite raffinées en les appliquant à chaque côté de l'apparition de l'agent dans une relation causale, cause ou effet. Dans l'exemple précédent, l'ensemble des individus est observable. Néanmoins, seuls les individus I_3, I_4, I_11 et I_12 valident les contraintes sur les agents.

3.4.4 Sélection du meilleur assemblage

A présent, ayant défini un ensemble de règles de compilation, plusieurs questions se posent.

— Comment adapter cette compilation à la réalité biologique des systèmes ? À cette fin, nous proposons ici une approche utilisant une définition formelle des

contraintes biologiques, puis une application de ces règles lors de la sélection et de l'assemblage des composants.

— De plus la description étant fonctionnelle, comment sélectionner parmi plusieurs assemblages possibles, le plus optimal ?

— Et que définir comme optimum ?

Afin d'affiner la sélection des composants et de générer des assemblages possédant une viabilité *in-vitro* plus grande, nous prenons en compte plusieurs propriétés connues des systèmes biologiques et nous les intégrons comme contraintes de "*fitness*" de l'algorithme de compilation.

Taille de l'assemblage

Une première contrainte est la taille du système généré. En effet la taille du système augmente les risques d'interactions non voulues entre composants, et la présence de « bruit » (perturbation des signaux d'intérêt par un ensemble de signaux générés par le système). Réduire le nombre au contraire amène une stabilité du système. À cette fin nous allons chercher à minimiser le nombre de composants biologiques permettant de réaliser une fonction spécifiée en **Gubs**. Les composants de la base de données pouvant avoir des propriétés en plus de celles nécessaires, mais étant indivisibles, le simple fait de sélectionner l'assemblage comprenant le moins de composants n'est pas suffisant. En effet, un assemblage peut contenir peu de composants, mais ceux-ci peuvent être de grande taille et avoir la majorité de leurs agents inutilisés pour réaliser la fonction spécifiée. Ainsi, afin de minimiser le nombre d'agents biologiques utilisés pour construire le système, nous prendrons en compte ces deux propriétés :

— Le nombre de composants utilisés pour l'unification d'un individu.

— Le nombre de règles unifiées dans chacun des composants.

Un assemblage sera donc minimal si à la fois le nombre d'éléments utilisés est minimal ET le nombre de règles unifiées est maximal. Numériquement, la règle utilisée est appelée rapport d'unification ρ_σ, où Q_i est un des m composants unifiés et $|Q_{i\setminus u}|_a$ correspond au nombre d'agents de Q_i qui n'ont pas été unifiés.

Définition 6 (Rapport d'unification)**.**

$$\rho_\sigma = \sum_{i=1}^{m} \frac{|Q_{i\setminus u}|_a}{|Q_i|_a}$$

Le rapport d'unification doit être minimal pour maximiser la qualité de l'assemblage.

Orthogonalité et promoteur

Une des propriétés biologiques [91] à prendre en compte est l'orthogonalité des composants utilisés : deux systèmes biologiques sont dit orthogonaux s'ils n'ont aucune influence l'un sur l'autre et de même pour chacun de leurs composants [52]. Afin d'améliorer cette propriété, on cherchera à minimiser dans un assemblage de composants le nombre d'agents pouvant être activés à la fois par des agents unifiés et par d'autres non unifiés. A cette fin nous introduisons une deuxième règle de "*fitness*" nommée rapport d'orthogonalité $\rho_{\perp}$.

Définition 7 (Rapport d'orthogonalité)**.** *Soit Q un assemblage de composants, Q_i le ieme composant de Q comportant un ensemble de relations causales, $\mathfrak{R}$ une relation causale telle que $\mathfrak{R} \in Q_{i\setminus u}$ et soit a un agent unifié, il est dit orthogonal $a_{\perp}$ si $\forall Q_i \in Q, \forall \mathfrak{R} \in Q_{i\setminus u}, \forall x \in \mathfrak{R} : x \neq a$.*

$$\rho_{\perp} = \sum_{i=1}^{m} \frac{1}{|Q_i|_{a_{\perp}}}$$

Le rapport d'orthogonalité doit être minimal pour maximiser la qualité de l'assemblage.

Une seconde propriété biologique pouvant permettre de minimiser l'impact des composants les uns sur les autres est la qualité des promoteurs[98]. En effet, un promoteur fort, ayant un fort rendement de production d'ARN sera moins sensible au bruit produit par son environnement en introduisant une forte redondance de son signal, et de ce fait moins sujet à des inhibitions non voulues par des causes externes. De plus cette forte production permet de différencier plus efficacement son activité du bruit environnant. A cette fin, nous proposons de noter les promoteurs de la base de données de composants en fonction de leur force. Ainsi lors de la sélection, les composants comprenant des promoteurs forts seront sélectionnés en priorité.

Châssis

Comme expliqué dans le chapitre d'introduction , le châssis a une forte influence sur les composants [44]. En effet, une bactérie possède déjà un ensemble d'éléments

servant à sa survie et à sa reproduction et ces caractéristiques peuvent interférer avec celles introduites et vice versa. Afin de palier ce problème nous proposons deux approches. La première consiste à utiliser des éléments biologiques appartenant à des espèces proches, voire à l'espèce en question servant de châssis, ainsi les risques d'interaction dus au châssis sont minimisés. Néanmoins, cette solution réduit le champ de recherche et d'assemblage possible afin de créer de nouvelles fonctionnalités. Une seconde approche serait alors d'utiliser un châssis "*free-cell*" [66], mais un tel châssis est encore au stade de développement, et est limité dans les propriétés implémentables. Dans le cadre de **Ggc**, nous privilégions, par ordre de priorité, la sélection de composants appartenant à l'espèce du châssis en priorité, puis à des familles proches, puis enfin à d'autres châssis afin de maximiser les chances de synthétiser un système viable "*in-vitro*". À cette fin, les composants de la base de données comportent parmi leurs champs de métadonnées les espèces dans lesquelles ils sont présents.

Exemple

Parmi les assemblages définis précédemment, nous avions retenu I_3, I_4, $I_1 1$ et $I_1 2$ comme étant ceux validant les propriétés sur les agents ainsi que l'observabilité forte. Parmi ces individus, $I_1 2 = Q_4, Q_4$ possède le meilleur rapport d'unification et le meilleur rapport d'orthogonalité. En effet, il suffit à lui seul à couvrir le programme P initial. L'assemblage $I_1 1 = Q_4, Q_3$ possède lui les plus mauvais rapports.

3.4.5 ACI-Unification

À cette étape, l'individu est unifié avec le programme en utilisant l'ACI-unification. L'algorithme utilisé est alors le suivant :

1. Si P ne contient aucune variable, renvoyer la substitution de ses agents ;
2. Sélectionner la cause P_i de P possédant le plus de variables ;
3. Unifier chaque variable de P_i par une constante valide de son homologue Q_i dans Q ;
4. Si c'est impossible, stocker l'unification partielle et reprendre 2 avec une autre constante valide ;
5. Sinon substituer dans tout P les variables unifiées ; goto 1 ;

On définira comme valide, une constante c qui valide les propriétés sur les agents pour une variable v. Dans le cadre de notre exemple, $I_1 2$ s'unifie en substituant v_1 par B et v_2 par C, le programme P s'unifie donc avec Q_4.

3.4.6 Algorithme évolutionnaire

La génération des individus conduisant à une explosion combinatoire, dans le cadre de grandes bases de données (plus de 200 composants), nous utilisons un algorithme évolutionnaire en association avec l'algorithme d'assemblage. Ici, nous décrivons plus en détail le fonctionnement de cet algorithme évolutionnaire permettant la synthèse fonctionnelle. Cet algorithme évolutionnaire prend place au sein de l'algorithme de compilation à l'étape de génération des individus candidats. En effet, au lieu de générer l'ensemble des individus possibles par combinaison de chaque individu couvrant localement les causes de P, nous générons aléatoirement un sous ensemble de ces individus. La population ainsi obtenue est ensuite brassée par l'algorithme à chaque passe de l'algorithme d'assemblage afin d'en extraire après mutation et cross-over des individus couvrant P.

Les algorithmes évolutionnaires [40, 58] sont une classe d'algorithmes d'optimisation méta-heuristiques inspirés du principe d'évolution Darwinienne, copiant le processus d'évolution biologique : l'algorithme évolutionnaire sélectionne des solutions candidates et les fait évoluer par recombinaisons et mutations afin d'améliorer leur qualité quantifiée par une fonction de fitness stochastique. D'une manière générale, un algorithme évolutionnaire résout un problème d'optimisation multi-objectif spécifié comme suit [119] :

$$\text{minimize}\, F(x) = (f_1(x), \ldots, f_n(x)) \text{ tel que } x \in X,$$

où X est un ensemble viable de solutions/individus choisis dans un domaine X' et validé par un prédicat p (*i.e.*, $X = \{x \in X' \mid p(x)\}$) et F est une séquence de fonctions objectifs/*fitness*, $f_i : X \to \mathbb{R}$.

En conséquence, l'application de l'algorithme évolutif nécessite de préciser les trois éléments (l'encodage d'un individu $x \in X$, la contrainte de viabilité p et les fonctions de *fitness* f_i) en accord avec le problème à résoudre, ici, le problème de la synthèse fonctionnelle.

Individus

Un individu correspond à une proposition de solution pour résoudre la synthèse fonctionnelle. Il représente un sous-ensemble de composants $C = \{Q_i\}_i$ choisis dans une base de données Γ. Ensuite, comme les individus correspondent à un sous ensemble fini d'un ensemble de référence (base de données), ils sont implémentés par un vecteur booléen de taille $|\Gamma|$ tel que 1 identifie un élément sélectionné et 0 pour les autres.

Fonctions de Fitness

Les fonctions de *fitness* permettent de guider la sélection d'individus viables afin d'améliorer la qualité de la synthèse. Par définition du problème de synthèse fonctionnelle, le nombre de composants (*i.e.*, le nombre d'éléments égal à 1 dans un vecteur) se définit comme une fonction de fitness, puisque nous cherchons à le minimiser ; de même pour le nombre de causes utilisées dans chaque individu pour couvrir le programme P. Nous classons les contraintes sur les agents comme des contraintes de fitness. En effet, celles-ci peuvent conduire à une unification partielle du programme.

Contraintes de viabilités

Les contraintes de viabilité sont liées à l'observabilité d'un individu, d'une part, et la capacité de déterminer si un individu inclut comportementalement le programme à compiler, d'autre part. Pour cela, nous utilisons donc l'*Observabilité forte* et les contraintes sur les causes.

Notons que l'ACI-unification d'un individu n'est pas une contrainte de viabilité. En effet une unification partielle est considérée comme une solution viable, pouvant suggérer l'absence d'éléments nécessaires dans la base de données, ou le manque de précision du programmeur lors de sa spécification. Notons aussi qu'ici les mutations et les cross-overs ne se produisent qu'entre Q_i couvrant une même cause de P afin de conserver les propriétés de couverture locale.

Parmi les premiers travaux sur la biologie de synthèse, on notera ceux décrits par P.Francois [55] pour la conception d'un circuit oscillant bistable, à l'aide de l'algorithme génétique. Dans [96], les auteurs appliquent un algorithme de Monte-Carlo afin de synthétiser *de-novo* un RBS (*ribosome binding site*). Dans [90, 89], les

auteurs utilisent un environnement fondé sur un algorithme évolutionnaire afin de produire automatiquement de petits réseaux de régulation à partir d'une base de données de composants biologiques qui correspondent à un comportement représenté par l'évolution de la concentration en *ARN* de certains gènes d'intérêts. Ces travaux mettent en évidence l'applicabilité de l'algorithme évolutionnaire pour la conception automatique en biologie synthétique. Bien que l'utilisation de l'algorithme évolutionnaire dans **Gubs** diffère, il sous-tend le même objectif et ouvre la possibilité d'une intégration fondée sur les mêmes principes d'optimisation. En outre, des capacités d'extension sont également envisagées pour faciliter l'évolution des logiciels. Principalement, ces développements pourraient améliorer la sélection des composants, notamment en intégrant la compatibilité biologique entre les composants sans qu'elle soit nécessairement mentionnée explicitement dans le programme afin de faciliter la tâche du programmeur. Ainsi, la conception du compilateur doit prendre en compte ces deux prérequis : la synthèse fonctionnelle et la durabilité de logiciel. Ces exigences permettent d'orienter le développement vers l'utilisation d'une optimisation méta-heuristique, et plus précisément un algorithme évolutionnaire fournissant un cadre approprié pour la résolution de la synthèse fonctionnelle tout en facilitant les développements ultérieurs. De même, d'autres fonctions de fitness peuvent être ajoutées pour un meilleur guidage de la sélection des composants, notamment en prenant en compte les aspects biologiques. Comme un algorithme évolutionnaire traite de multiples objectifs, leur ajout est techniquement facile. L'accent est alors mis plutôt sur la capacité à modéliser correctement quantitativement des contraintes biologiques.

Afin de récapituler le fonctionnement de l'algorithme de synthèse fonctionnelle, nous proposons de décrire celui-ci dans le schéma de la figure 3.3.

Après avoir décrit formellement le fonctionnement de l'algorithme de synthèse fonctionnelle, nous nous penchons dans la section suivante sur son implémentation sous la forme du compilateur **Ggc**.

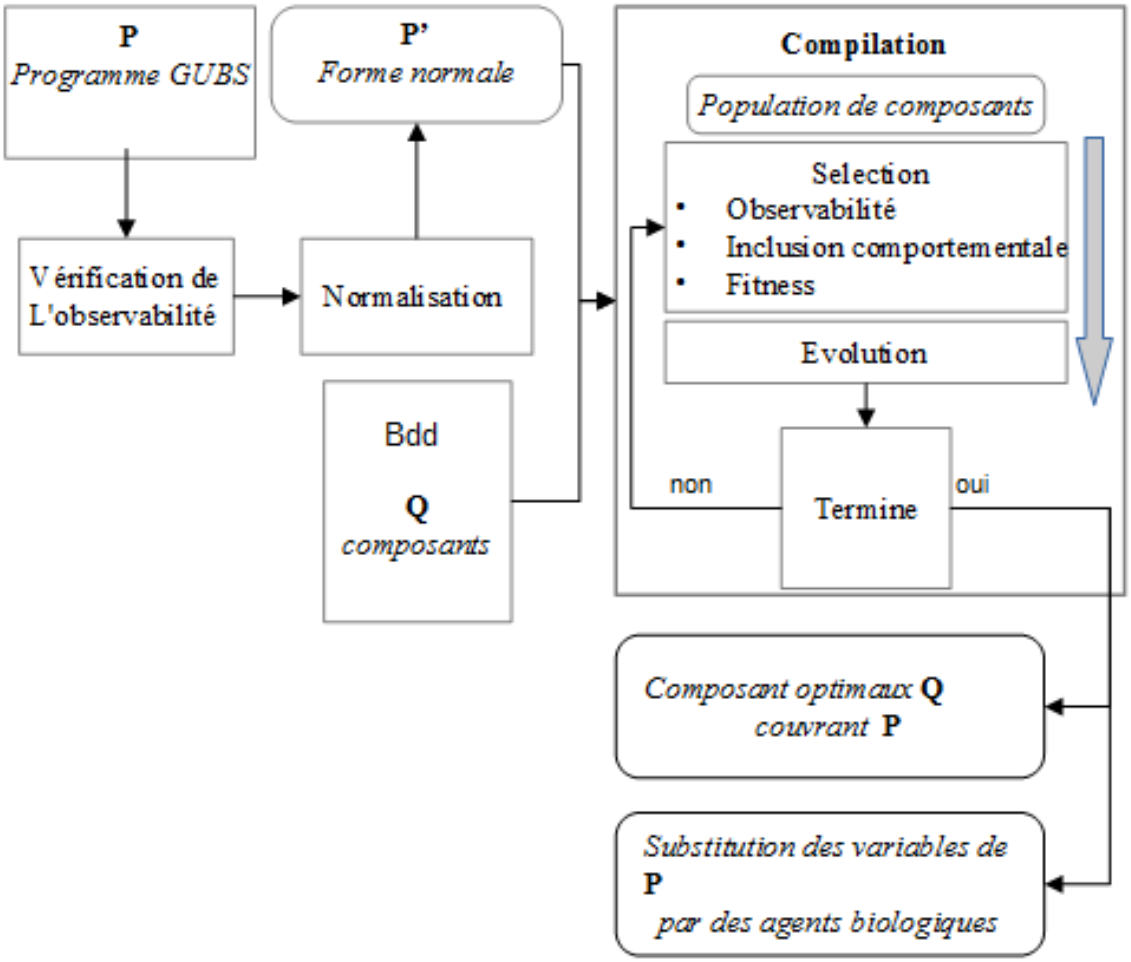


Figure 3.3 – Schéma de compilation de **Gubs**.

3.5 Ggc

Cette section introduit les détails de l'outil **Ggc**. Dans un premier temps nous présentons les fonctionnalités de l'outil ainsi que la répartition en modules du compilateur. Ensuite nous introduisons un ensemble de résultats de compilation dans le cas de base de données de petite taille (200 composants) afin de montrer l'efficacité des algorithmes de sélection des causes et des agents. Le détail de chaque fichier du compilateur se trouve lui en annexe (Section 4.5).

3.5.1 Structure de l'outil

Ggc est un compilateur de 2000 lignes rédigé en Ocaml instanciant la compilation telle que décrite précédemment. Il permet de trouver automatiquement pour un programme donné en entrée et une base de données donnée, un ensemble de composants de cette base de données couvrant le comportement du programme. **Ggc** renvoie alors la ou les meilleures solutions sous forme d'une liste de couples (substitution, liste de composants).

Ggc se compose de plusieurs modules en interaction :

Traduction : Il s'agit du module de traduction permettant la traduction de fichiers vers les structures de données adéquates.

Normalisation : Le module de normalisation permet de transformer un programme **Gubs** en sa forme normale, permettant sa compilation.

Observe : Observe permet d'appeler un programme appliquant les méthodes à tableaux sur la formule correspondant à un programme.

Contraintes : Ce module définit l'ensemble des contraintes à valider pour qu'un assemblage de composants **Gubs** soit considéré comme viable.

Fitness : Ce module définit les contraintes de fitness permettant de sélectionner le meilleur assemblage de composants.

Core : Ce module est le cœur du compilateur, exécutant l'algorithme de synthèse fonctionnelle sur un programme en appliquant les contraintes de viabilité et de fitness.

UI : Ce module sert à gérer les entrées utilisateur et à coordonner l'appel de tous les autres modules.

3.6 Modules d'application

Chacun de ces modules est composé de plusieurs sous modules ayant des fonctionnalités particulières, celle-ci sont décrites en détail en annexe (Section 4.5).

3.6.1 Utilisation de Ggc

Nous décrivons ici l'ensemble des options de compilation de **Ggc** permettant son utilisation. Par défaut, l'appel de **Ggc** est :

./ggc [nom du programme]

A cette ligne de commande s'ajoute un ensemble d'arguments supplémentaires :

[-pl :] ./ggc -pl [nom du programme$_1$]...[nom du programme$_n$] : Le nom du programme peut être défini comme une liste de programmes, correspondant à un unique programme.

[-Pl :] ./ggc -pl [nom du programme$_1$]...[nom du programme$_n$] : Le nom du programme peut être défini comme une liste de programmes, dans ce cas chaque programme sera compilé indépendamment.

[-db :] ./ggc [nom du programme] -db [nom de la bdd] : Une base de données autre que celle de **Gubs** de base peut être définie.

[-dbl :] ./ggc [nom du programme] -dbl [nom de la bdd$_1$]...[nom de la bdd$_n$] : Il peut s'agir d'une liste de base de données.

[-x :] ./ggc [nom du programme] -x [nom de la DTD] : Une nouvelle DTD xml peut être définie.

[-xdbl :] ./ggc [nom du programme] -dbl [bdd$_1$][DTD$_1$]...[bdd$_n$][DTD$_n$] : Une liste de base de données avec chacune une DTD peut être définie.

[-o(1-2) :] ./ggc -o1 [nom du programme] : Définit un niveau d'optimisation correspondant à l'affaiblissement des règles : -o1 affaiblit les persistantes, -o2 affaiblit les persistantes et les normales.

[-c :] ./ggc -c [nom du programme] : Applique la règle d'extension par clôture transitive des composants de la base de données.

[-pp :] ./ggc -pp [nom du programme] : Effectue le pretty printing du programme.

[-CT :] ./ggc -CT [nom du programme] : Applique les contraintes définies par l'utilisateur. Attention, ces contraintes peuvent potentiellement mener à une Erreur de compilation si elles font référence à des éléments inexistants.

[-tb :] ./ggc -t [nom du programme] : Applique les méthodes à tableaux au programme avant sa compilation. Attention cette option peut ralentir considérablement le programme.

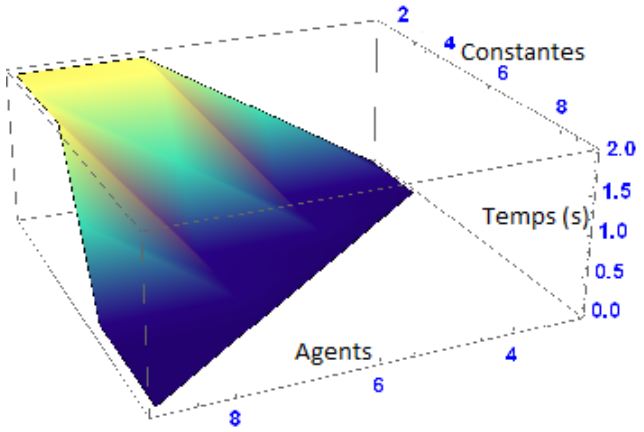
[-o :] ./ggc [nom du programme] -o [nom du fichier de sortie] : Définit le fichier d'output de la compilation. Par défaut le résultat s'affiche en ligne de commande.

[-t :] ./ggc [nom du programme] -t [temps] : Définit le temps de time-out du compilateur en seconde. Par défaut, 2 heures.

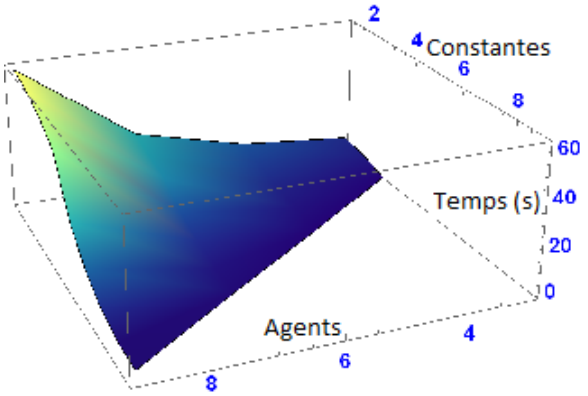
3.7 Résultats numériques

Afin de valider empiriquement l'algorithme de synthèse fonctionnelle, nous avons effectué plusieurs tests d'exécution de **Ggc** sur plusieurs jeux de données. Les jeux de tests se fondent sur un ensemble de bases de données de 100 composants générés aléatoirement. Afin d'assurer la présence d'une solution aux programmes de test dans la base de données et ainsi assurer qu'un time-out du compilateur est seulement dû au fait qu'il n'a pas pu la trouver et non qu'il n'existe aucune solution, nous avons généré chaque programme en sélectionnant un ensemble de composants de la base de données. Les temps d'exécution ainsi obtenus ont été représentés sous forme de deux courbes respectivement pour des programmes de 10 relations causales et de 25 relations causales. Afin de montrer l'impact du nombre d'agents et du nombre de constantes sur le temps de compilation, nous avons fait varier sur une échelle de 1 à 10 ces deux valeurs dans le cas de programmes de taille fixe (10 et 25 relations causales).

Les courbes ainsi obtenues confirment les calculs théoriques. En effet, on observe que le temps évolue exponentiellement en fonction du nombre de variables et non du nombre d'agents car les constantes s'unifient avec elles-mêmes uniquement. Ainsi le nombre de constantes constitue un facteur de réduction du temps de l'algorithme. En conclusion, un programme efficacement traité par **Gubs** nécessite une connaissance au moins partielle des composants à utiliser qui seront décrits sous forme de constantes. On remarquera aussi que sur un programme court (figure 3.4 (A)) le temps maximal d'exécution est extrêmement court (moins de deux secondes). Ce temps augmente drastiquement dans la deuxième courbe (figure 3.4 (B)), ce qui est en accord avec l'explosion combinatoire prévue lors de la définition de l'algorithme.



(*A*)



(*B*)

Figure 3.4 – La première courbes représente l'évolution du temps (en s) de compilation pour un programme de 10 relations causales et la seconde pour un programme de 25 relations causales.

Conclusion

Pour conclure, dans ce chapitre, nous avons défini un système de règles de compilation adapté au vivant. Nous avons ainsi introduit la notion de synthèse fonctionnelle ainsi qu'un algorithme permettant de l'implémenter. Cet algorithme repose sur le principe d'ACI-Unification et propose dans le cas de grandes bases de données ou de programmes de grande taille, l'utilisation d'un algorithme d'approximation afin de trouver une solution en temps polynomial. Pour finir nous avons implémenté cet algorithme en Ocaml sous la forme d'un compilateur nommé **Ggc** et effectué plusieurs tests d'exécution sur celui-ci, confirmant la complexité de l'algorithme calculé, tout en proposant un temps de calcul raisonnable pour des programmes de petite taille (possédant peu de causes).

Chapitre 4

Exemples

Résumé Dans ce chapitre nous proposons deux exemples, reprenant des constructions biologiques servant de référence en biologie de synthèse. Chacun de ces exemples a pour but de montrer à la fois la spécification d'un système en **Gubs** et un aspect de la compilation, qu'il s'agisse des règles théoriques ou de leur implémentation afin d'automatiser la compilation. La première est le *Band detector* proposé par l'équipe de Ron Weiss. Cet exemple présente un système *senseur-actuateur* reposant sur un principe de *corrum-sensing*. Dans le cadre de cet exemple, nous proposons de montrer l'application des règles de réécriture. Le deuxième exemple est le *Repressilator*, un oscillateur bistable composé de trois gènes. Cet exemple nous permet de décrire en détail le fonctionnement des algorithmes utilisés par **Ggc**.

4.1 Band detector

Dans cette section nous introduisons la compilation du Band-detector proposé par l'équipe de Ron Weiss [13] en 2005. Le Band-detector se divise en deux populations bactériennes : la première sert à diffuser la molécule AHL dans le milieu via un phénomène de quorum sensing. Ainsi plus les cellules réceptrices sont éloignées de l'émetteur, plus la concentration sera faible. La diffusion est contrôlée par un événement externe. La seconde est un récepteur sensible à la concentration d'AHL, réagissant différemment suivant que cette concentration soit forte, moyenne ou faible, en activant ou en inhibant la protéine de fluorescence GFP. La construction de l'exemple vise à former des motifs de couleurs en fonction de cette concentration.

4.1.1 Modèle original

Le comportement du Band-detector en fonction de la concentration d'AHL est détaillé dans la figure 4.1. La figure 4.2 présente quant à elle plusieurs exemples de coloration changeant suivant la position et la quantité de cellules émettrices, l'activation ou l'inhibition de la GFP se caractérisant par des bandes colorées ou non autour de la source émettrice : dans la figure, la bande du milieu fluoresce, en raison de la concentration moyenne. Cet exemple décrit comment, à partir d'une simple définition abstraite de son comportement, un système complexe peut être synthétisé. En conséquence, **Gubs** peut être utilisé pour décrire un comportement avec un haut niveau d'abstraction tout autant qu'avec un bas niveau d'abstraction, détaillant les composants impliqués dans l'assemblage. Nous nous penchons ici sur la compilation théorique, en utilisant l'ensemble des règles de réécriture définies pour la compilation. Plus particulièrement, nous introduisons chaque étape de réécriture permettant de passer d'un programme dans un code de haut niveau vers celui du plus bas niveau correspondant aux composants biologiques. Chaque étape correspond à l'application d'une règle de réécriture définie dans les Tables 3.2, 3.3, ce qui assure sa correction et donc sa cohérence fonctionnelle dans le contexte d'un système ouvert.

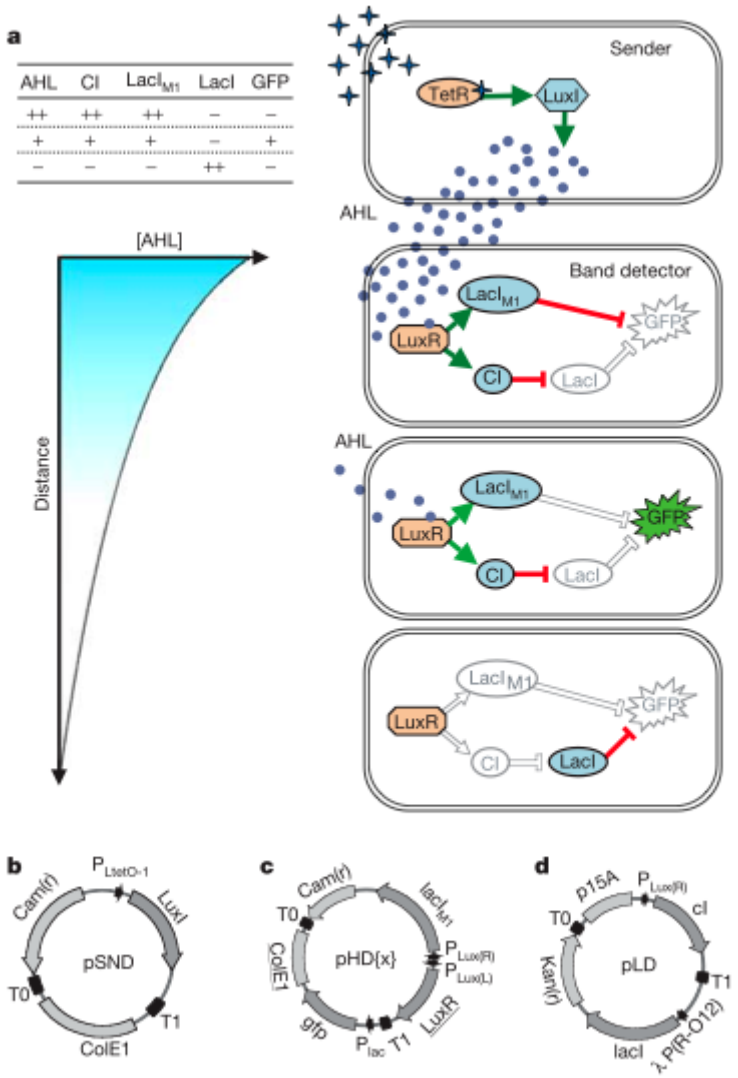


Figure 4.1 – Évolution de l'activation de la GFP en fonction de la concentration en AHL et modèle biologique du Band-detector. La concentration en AHL à partir des cellules émettrices (en haut à droite) sert à activer le circuit de régulation génétique des cellules réceptrices. Quand AHL est à un niveau élevé ou faible, la GFP est inhibée, et quand la concentration est moyenne, elle est exprimée. Le plasmide émetteur est montré dans (b), et les plasmides récepteurs dans (c) et (d)

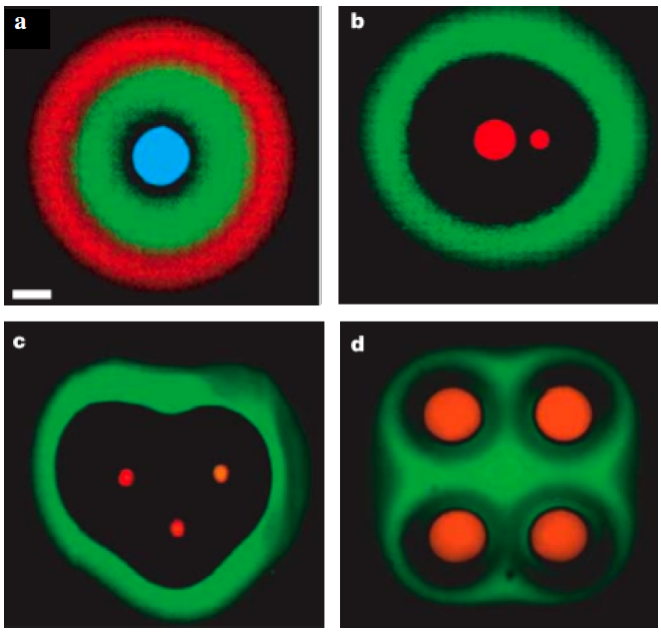


Figure 4.2 – Exemple du Band-detector en laboratoire. Les régions circulaires au centre sont des cellules émettrices actives, et les régions colorées sont quant à elles des cellules réceptrices exprimant la protéine de fluorescence GFP.

4.1.2 spécification en Gubs

Dans un premier temps nous proposons un modèle abstrait représentant le comportement du Band-Detector, en abstrayant les composants biologiques pour ne se concentrer que sur ceux permettant de décrire le comportement d'activation et d'inhibition de la GFP. Ce modèle plus simple est présenté dans la figure 4.3 et correspond au réseau de régulation du Band-detector. Les composants biologiques seront eux automatiquement retrouvés à l'étape de compilation à partir d'une base de données de composants.

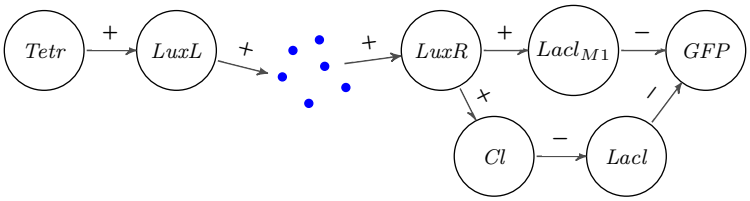


Figure 4.3 – Réseau de régulation du Band-detector. Les molécules en bleu correspondent à AHL.

D'un point de vue informatique, nous pouvons associer ce design à la transmission d'un message associé à un senseur/activateur activant la fluorescence, qui correspond à un court programme **Gubs** présenté ci-après : la molécule de diffusion *AHL* est contrôlée par un contexte, et les points d'observation permettent de surveiller la *GFP*. Nous définissons deux catégories de cellules : *Sender* correspondant à l'émetteur et le *Receiver* correspondant aux cellules réceptrices. Nous définissons donc deux programmes **Gubs** identifiant chaque type de cellules.

$$
\begin{aligned}
Sender = \quad & \{\ AHL : \{low \circledast mid \circledast high\}, \\
& [Light]\{detect \bigcirc\!\!\!\!\rightarrow AHL(low), detect \bigcirc\!\!\!\!\rightarrow AHL(mid), detect \bigcirc\!\!\!\!\rightarrow AHL(high)\}\} \\
Receiver = \quad & \{\ AHL(low) \bigcirc\!\!\!\!\rightarrow \overline{GFP}, AHL(mid) \bigcirc\!\!\!\!\rightarrow GFP, AHL(high) \bigcirc\!\!\!\!\rightarrow \overline{GFP}, obs_1 :: GFP, obs_2 :: \overline{GFP}\}
\end{aligned}
$$

La molécule de diffusion *AHL* est définie comme une constante. *detect* est une variable utilisée pour représenter l'activation de la diffusion d'*AHL* par un élément externe que nous décrirons comme étant la lumière*Light* (afin de mettre l'accent sur l'extériorité de ce composant). Le gène *LuxR* a trois niveaux d'activation : au niveau 2, il active à la fois *LaclM1* et *Cl*, au niveau 1, la quantité d'*AHL* ne permet l'activation que de *Cl*, et finalement, au niveau 0, aucun n'est activé.

4.1.3 Principe de compilation par réécriture

Afin de simplifier le suivi de la compilation, nous libellons les causes du programme Sender-Receiver (Table 4.1). Nous montrons ici qu'à partir du programme sender-receiver nous obtenons par réécriture la construction originale, en utilisant une sélection appropriée de composants au sein d'une base de données comprenant ceux nécessaires à l'assemblage (Table 4.2).

Nous nous concentrons ici sur les étapes clés de la compilation du Sender. La synthèse complète est détaillée par la suite. La compilation consiste à trouver les composants appropriés tels que leur assemblage inclut comportementalement le comportement du programme Sender-receiver, avec la particularité que la molécule de diffusion doit être la même dans les deux programmes.

Lors du découpage du programme en forme normale, P_{ij} correspond à la j^{ieme} relation causale normalisée du programme P_i où P_1 est l'émetteur (Sender) et P_2 le récepteur (Receiver). Nous commençons ici par présenter de façon générale la réécriture en utilisant P_{11}, dont la compilation est proche de P_{12} etP_{13}. Notons

Sender	*Receiver*
$P_{11} = \{[Light]\{detect \bigcirc\!\!\rightarrow \mathsf{AHL}(low)\}\}$	$P_{21} = \{\mathsf{AHL}(low) \bigcirc\!\!\rightarrow \overline{\mathsf{GFP}}\}$
$P_{12} = \{[Light]\{detect \bigcirc\!\!\rightarrow \mathsf{AHL}(mid)\}\}$	$P_{22} = \{\mathsf{AHL}(mid) \bigcirc\!\!\rightarrow \mathsf{GFP}\}$
$P_{13} = \{[Light]\{detect \bigcirc\!\!\rightarrow \mathsf{AHL}(high)\}\}$	$P_{23} = \{\mathsf{AHL}(high) \bigcirc\!\!\rightarrow \overline{\mathsf{GFP}}\}$

Avec $\{\mathsf{AHL} : \{low \circledast mid \circledast high\}\}$ comme attributs de AHL.

Table 4.1 – Séparation des causes.

que P_{11} ne peut être directement instanciée avec un composant car, d'un coté, le composant Q_1 contient un contexte similaire à celui de P_{11} mais appliquée au gène *Tetr* au lieu de AHL, et d'un autre coté, Q_3 possède la molécule AHL mais aucun contexte n'est défini. Donc afin d'associer P_{11} avec les composants Q_1, Q_2 et Q_3, dans un premier temps la dépendance normale est convertie en une dépendance persistante via la règle (N2P.).

$$\frac{Q_1, Q_2, Q_3 \leftarrow_\sigma \{[light]\{detect \odot\!\!\rightarrow AHL(low)\}\}}{Q_1, Q_2, Q_3 \leftarrow_\sigma P_{11}} \text{ (N2P.)}$$

Ensuite, la dépendance résultante peut aussi être étendue afin de pouvoir être associée avec Q_1, Q_2, Q_3 par l'application de la règle (Trans.) deux fois. On va donc générer v_1 et v_2 comme variables fraiches.

$$\frac{\dfrac{Q_1, Q_2, Q_3 \leftarrow_\sigma P'_{11} = \{[light]\{detect \odot\!\!\rightarrow v_2, v_2 \odot\!\!\rightarrow v_1, v_1 \odot\!\!\rightarrow AHL(low)\}}{Q_1, Q_2, Q_3 \leftarrow_\sigma [light]\{detect \odot\!\!\rightarrow v_1, v_1 \odot\!\!\rightarrow AHL(low)\}} \text{ (Trans.)}}{Q_1, Q_2, Q_3 \leftarrow_\sigma [light]\{detect \odot\!\!\rightarrow AHL(low)\}} \text{ (Trans.)}$$

Finalement, on obtient un nouveau programme P'_{11} compatible avec Q_1, Q_2, Q_3, dans lequel chaque variable est substituée par une constante correspondant à un élément

$Q_1 =$ $\{[\text{Light}]\{detect \bigcirc\!\!\rightarrow \mathsf{Tetr}\}\}$
$Q_2 =$ $\{\mathsf{Tetr} \xrightarrow{+} \mathsf{LuxI}\}$
$Q_3 =$ $\{\mathsf{AHL} : \{low \circledast mid \circledast high\}, \mathsf{LuxI} \xrightarrow{+} \mathsf{AHL}(low), \mathsf{LuxI} \xrightarrow{+} \mathsf{AHL}(mid), \mathsf{LuxI} \xrightarrow{+} \mathsf{AHL}(high)\}$
$Q_4 =$ $\{\mathsf{AHL} : \{low \circledast mid \circledast high\}, \mathsf{LuxR} : \{low \circledast \{mid \prec high\}\},$
$\mathsf{AHL}(mid) \bigcirc\!\!\rightarrow \mathsf{LuxR}(mid), \mathsf{AHL}(high) \bigcirc\!\!\rightarrow \mathsf{LuxR}(high)\}$
$Q_5 =$ $\{\mathsf{LuxR} : \{low \circledast \{mid \prec high\}\}, \mathsf{LuxR}(mid) \xrightarrow{+} \mathsf{CI}, \mathsf{LuxR}(high) \xrightarrow{+} \mathsf{CI} + \mathsf{LacIM1}\}$
$Q_6 =$ $\{\mathsf{CI} \xrightarrow{-} \mathsf{LacI}\}$
$Q_7 =$ $\{\mathsf{LacIM1} \xrightarrow{-} \mathsf{GFP}\}$
$Q_8 =$ $\{\mathsf{LacI} \xrightarrow{-} \mathsf{GFP}\}$

Table 4.2 – Fragment de la base de données dédiée au Band-detector.

biologique par application de la règle (Inst.). Pour P'_{11} on obtient :

$$\frac{Q_1, Q_2, Q_3[\sigma = \{light/Light, v_1/Tetr, v_2/Luxl\}] \subseteq_{Asm} P'_{11}[\sigma] \qquad obs(Q_1, Q_2, Q_3[\sigma])}{Q_1, Q_2, Q_3 \leftarrow_{\sigma} [light]\{detect \odot\!\!\rightarrow v_1, v_1 \odot\!\!\rightarrow v_2, v_2 \odot\!\!\rightarrow AHL(low)\}} \text{(Inst.)}$$

En suivant ce processus pour P_{12} et P_{13}, on obtient respectivement P'_{12} et P'_{13}. L'assemblage final correspond à la synthèse fonctionnelle du programme *Sender*.

$$\frac{\begin{array}{ccc} Q_1, Q_2, Q_3 \leftarrow_{\sigma} P'_{11} & Q_1, Q_2, Q_3 \leftarrow_{\sigma} P'_{12} & Q_1, Q_2, Q_3 \leftarrow_{\sigma} P'_{13} \\ \vdots & \vdots & \vdots \\ Q_1, Q_2, Q_3 \leftarrow_{\sigma} P_{11} & Q_1, Q_2, Q_3 \leftarrow_{\sigma'} P_{12} & Q_1, Q_2, Q_3 \leftarrow_{\sigma''} P_{13} \end{array}}{Q_1, Q_2, Q_3 \leftarrow_{\sigma \cup \sigma' \cup \sigma''} P_{11}, P_{12}, P_{13}} \text{(Asm.)}$$

On obtient finalement le programme substitué suivant.

$$\begin{aligned} Sender \quad = \quad & \{\mathsf{AHL} : \{low \circledast mid \circledast high\}, [\mathrm{Light}]\{detect \circ\!\!\rightarrow \mathsf{Tetr}\}, \\ & \mathsf{Tetr} \xrightarrow{+} \mathsf{Luxl}, \mathsf{Luxl} \xrightarrow{+} \mathsf{AHL}(low), \mathsf{Luxl} \xrightarrow{+} \mathsf{AHL}(mid), \mathsf{Luxl} \xrightarrow{+} \mathsf{AHL}(high)\} \end{aligned}$$

4.1.4 Compilation détaillée du Band-detector

Après avoir décrit les grandes lignes de la compilation du Band-detector dans la section précédente, nous nous penchons ici sur la compilation complète de celui-ci. Cette compilation est détaillée dans les trois tables suivantes (4.3,4.4 et 4.5).

- Sender -

$$\frac{Q_1, Q_2, Q_3[\sigma = \{detect/Detect, light/Light, v_1/Tetr, v_2/Luxl\}] \subseteq_{Asm} P'_{11}[\sigma] \qquad obs(Q_1, Q_2, Q_3[\sigma])}{Q_1, Q_2, Q_3 \vdash_{\sigma} P'_{11}\}} \text{ (Inst.)} \quad \mathbf{3}$$

$$\frac{Q_1, Q_2, Q_3[\sigma' = \{detect/Detect, light/Light, v_3/Tetr, v_4/Luxl\}] \subseteq_{Asm} P'_{12}[\sigma'] \qquad obs(Q_1, Q_2, Q_3[\sigma'])}{Q_1, Q_2, Q_3 \vdash_{\sigma'} P'_{12}} \text{ (Inst.)} \quad \mathbf{3}$$

$$\frac{Q_1, Q_2, Q_3[\sigma'' = \{detect/Detect, light/Light, v_5/Tetr, v_6/Luxl\}] \subseteq_{Asm} P'_{13}[\sigma''] \qquad obs(Q_1, Q_2, Q_3[\sigma''])}{Q_1, Q_2, Q_3 \vdash_{\sigma''} P'_{13}} \text{ (Inst.)} \quad \mathbf{3}$$

$$\frac{\dfrac{\dfrac{P'_{11} = [light]\{detect \odot\!\!\rightarrow v_1, v_1 \odot\!\!\rightarrow v_2, v_2 \odot\!\!\rightarrow AHL(low)\}}{[light]\{detect \odot\!\!\rightarrow v_1, v_1 \odot\!\!\rightarrow AHL(low)\}} \text{ (Trans.)}}{[light]\{detect \odot\!\!\rightarrow AHL(low)\}} \text{ (Trans.)}}{P_{11}} \text{ (N2P.)} \quad \mathbf{2}$$

$$\frac{\dfrac{\dfrac{P'_{12} = [light]\{detect \odot\!\!\rightarrow v_3, v_3 \odot\!\!\rightarrow v_4, v_4 \odot\!\!\rightarrow AHL(mid)\}}{[light]\{detect \odot\!\!\rightarrow v_3, v_3 \odot\!\!\rightarrow AHL(mid)\}} \text{ (Trans.)}}{[light]\{detect \odot\!\!\rightarrow AHL(mid)\}} \text{ (Trans.)}}{P_{12}} \text{ (N2P.)} \quad \mathbf{2}$$

$$\frac{\dfrac{\dfrac{P'_{13} = [light]\{detect \odot\!\!\rightarrow v_5, v_5 \odot\!\!\rightarrow v_6, v_6 \odot\!\!\rightarrow AHL(high)\}}{[light]\{detect \odot\!\!\rightarrow v_5, v_5 \odot\!\!\rightarrow AHL(high)\}} \text{ (Trans.)}}{[light]\{detect \odot\!\!\rightarrow AHL(high)\}} \text{ (Trans.)}}{P_{13}} \text{ (N2P.)} \quad \mathbf{2}$$

$$\frac{Q_1, Q_2, Q_3 \vdash_{\sigma} P_{11} \qquad Q_1, Q_2, Q_3 \vdash_{\sigma'} P_{12} \qquad Q_1, Q_2, Q_3 \vdash_{\sigma''} P_{13}}{Q_1, Q_2, Q_3 \vdash_{\sigma \cup \sigma' \cup \sigma''} P_{11}, P_{12}, P_{13}} \text{ (Asm.)} \quad \mathbf{1}$$

1. Premièrement nous allons découper le programme Sender en trois sous-programmes P_{11}, P_{12}, et P_{13}, chacun correspondant à une relation causale comme décrit dans la section précédente.
2. Initialement, P_{11}, P_{12} et P_{13} ne sont pas compatibles avec des composants de la base de données, nous devons donc appliquer les règles d'optimisation. Ici nous allons appliquer l'extension pour obtenir (P'_{11}, P'_{12} et P'_{13}) afin de trouver une association possible.
3. Finalement on peut associer P'_{11}, P'_{12} et P'_{13} avec les composants Q_1, Q_2, Q_3.

Table 4.3 – Compilation du programme Sender.

- Receiver -

$$\dfrac{Q_4, Q_5, Q_6, Q_8[\sigma = \{v_1/LuxR, v_2/Cl, v_3/Lacl\}] \subseteq_{Asm} P'_{21}[\sigma] \qquad obs(Q_4, Q_5, Q_6, Q_8[\sigma])}{Q_4, Q_5, Q_6, Q_8 \vdash_{\sigma} P'_{21}} \text{(Inst.)} \quad \mathbf{3}$$

$$\dfrac{Q_4, Q_5, Q_6, Q_8[\sigma' = \{v_4/LuxR, v_5/Cl, v_6/Lacl\}] \subseteq_{Asm} P'_{22}[\sigma'] \qquad obs(Q_4, Q_5, Q_6, Q_8[\sigma'])}{Q_4, Q_5, Q_6, Q_8 \vdash'_{\sigma} P'_{22}} \text{(Inst.)} \quad \mathbf{3}$$

$$\dfrac{Q_4, Q_5, Q_7[\sigma'' = \{v_7/LuxR, v_8/LacM1\}] \subseteq_{Asm} P'_{23}[\sigma''] \qquad obs(Q_4, Q_5, Q_7[\sigma''])}{Q_4, Q_5, Q_7 \vdash''_{\sigma} P'_{23}} \text{(Inst.)} \quad \mathbf{3}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{P'_{21} = AHL(low) \odot\!\!\rightarrow v_1, v_1 \odot\!\!\rightarrow v_2, v_2 \odot\!\!\rightarrow v_3, v_3 \odot\!\!\rightarrow \overline{GFP}}{AHL(low) \odot\!\!\rightarrow v_1, v_1 \odot\!\!\rightarrow v_2, v_2 \odot\!\!\rightarrow \overline{GFP}} \text{(Trans.)}}{AHL(low) \odot\!\!\rightarrow v_1, v_1 \odot\!\!\rightarrow \overline{GFP}} \text{(Trans.)}}{AHL(low) \odot\!\!\rightarrow \overline{GFP}} \text{(Trans.)}}{P_{21}} \text{(N2P.)} \quad \mathbf{2}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{P'_{22} = AHL(mid) \odot\!\!\rightarrow v_4, v_4 \odot\!\!\rightarrow v_5, v_5 \odot\!\!\rightarrow v_6, v_6 \odot\!\!\rightarrow GFP}{AHL(mid) \odot\!\!\rightarrow v_4, v_4 \odot\!\!\rightarrow v_5, v_5 \odot\!\!\rightarrow GFP} \text{(Trans.)}}{AHL(mid) \odot\!\!\rightarrow v_4, v_4 \odot\!\!\rightarrow GFP} \text{(Trans.)}}{AHL(mid) \odot\!\!\rightarrow GFP} \text{(Trans.)}}{P_{22}} \text{(N2P.)} \quad \mathbf{2}$$

$$\dfrac{\dfrac{\dfrac{P'_{23} = AHL(high) \odot\!\!\rightarrow v_7, v_7 \odot\!\!\rightarrow v_8, v_8 \odot\!\!\rightarrow \overline{GFP}}{AHL(high) \odot\!\!\rightarrow v_7, v_7 \odot\!\!\rightarrow \overline{GFP}} \text{(Trans.)}}{AHL(high) \odot\!\!\rightarrow \overline{GFP}} \text{(Trans.)}}{P_{23}} \text{(N2P.)} \quad \mathbf{2}$$

$$\dfrac{Q_4, Q_5, Q_6, Q_8 \vdash_{\sigma} P_{21} \qquad Q_4, Q_5, Q_6, Q_8 \vdash_{\sigma'} P_{22} \qquad Q_4, Q_5, Q_7 \vdash_{\sigma''} P_{23}}{Q_4, Q_5, Q_6, Q_7, Q_8 \vdash_{\sigma \cup \sigma' \cup \sigma''} P_{21}, P_{22}, P_{23}} \text{(Asm.)} \quad \mathbf{1}$$

1. Comme pour le programme Sender, le programme Receiver est découpé en trois sous-programmes P_{21}, P_{22} et P_{23}, chacun correspondant à une cause différant par la concentration en AHL.
2. P_{21}, P_{22} et P_{23} initialement ne s'associent à aucun composant de la base de données. Donc nous les étendons vers P'_{21}, P'_{22} et P'_{23} par application de la règle (Ext.).
3. P'_{21} et P'_{23} décrivent le même comportement pour deux concentrations différentes d'AHL. Ensuite, leur variables respectives (v_1 et v_7) sont substituées par la même constante LuxR. P'_{22} décrit la présence de GFP et s'associe avec les composants $\{Q_4, Q_5, Q_6, Q_8\}$. Finalement, P'_{21}, P'_{22} et P'_{23} s'associent avec les composants $\{Q_4, Q_5, Q_6, Q_7, Q_8\}$.

Table 4.4 – Compilation du programme Receiver.

- Final design -

Sender
$\{\mathsf{AHL} : \{low \lessdot mid \lessdot high\},$ $[\mathrm{Light}]\{detect \circ\!\!\rightarrow \mathsf{Tetr}\},$
$\mathsf{Tetr} \xrightarrow{+} \mathsf{LuxL},$ $\mathsf{LuxI} \xrightarrow{+} \mathsf{AHL}(low),$
$\mathsf{LuxI} \xrightarrow{+} \mathsf{AHL}(mid),$ $\mathsf{LuxI} \xrightarrow{+} \mathsf{AHL}(high)\}$
Receiver
$\{\mathsf{AHL} : \{low \lessdot mid \lessdot high\},$ $\mathsf{LuxR} : \{low \lessdot \{mid \prec high\}\},$
$\mathsf{AHL}(mid) \circ\!\!\rightarrow \mathsf{LuxR}(mid),$ $\mathsf{AHL}(high) \circ\!\!\rightarrow \mathsf{LuxR}(high),$
$\mathsf{LuxR}(mid) \xrightarrow{+} \mathsf{CI},$ $\mathsf{LuxR}(high) \xrightarrow{+} \mathsf{LacIM1},$
$\mathsf{CI} \xrightarrow{-} \mathsf{LacI},$ $\mathsf{LacIM1} \xrightarrow{-} \mathsf{GFP},$ $\mathsf{LacI} \xrightarrow{-} \mathsf{GFP}\}$

Table 4.5 – La compilation complète nous permet d'obtenir les deux programmes substitués présentés ci-dessus.

4.2 Repressilator

Dans cette section nous introduisons la compilation du Repressilator [51] en utilisant **Ggc**. Pour cet exemple nous utilisons la base de données décrite dans la figure 4.10. Le Repressilator est un assemblage biologique conçu afin d'exhiber une oscillation stable. Cette oscillation est rapportée grâce à une protéine de fluorescence verte (*GFP*). Il agit comme un oscillateur électrique avec une période fixe. Ce réseau est implémenté dans *Escherichia coli* par des méthodes standard de biologie moléculaire. Ce système a ensuite été observé afin de vérifier que les colonies créées possédaient le comportement d'oscillation attendu.

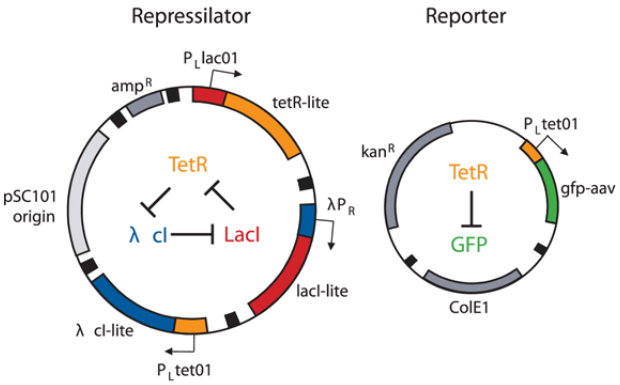


Figure 4.4 – modèle original du Repressilator tel que décrit dans l'article "A synthetic oscillatory network of transcriptional regulators" proposé par M.B.Elowitz en 2000

4.2.1 Spécification du Repressilator

Dans un premier temps nous détaillons la traduction de ce système (Figure 4.4) en une spécification **Gubs** (Figure 4.6). A cette fin, nous définissons un modèle abstrait représentant le comportement du Repressilator (Figure 4.5). En effet la

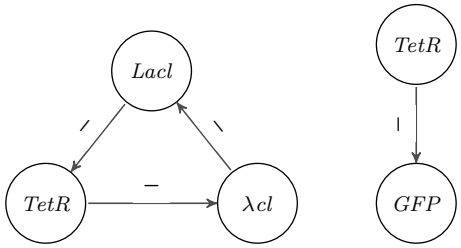


Figure 4.5 – Modèle représentant le comportement du Repressilator sous forme d'un réseau génétique.

spécification en **Gubs** cherche à représenter fidèlement le comportement du système biologique sous une forme permettant d'abstraire le détail biologique des éléments utilisés, ceux-ci étant redéfinis lors de la compilation. Dans la suite de cette exemple, nous utiliserons la syntaxe de **Gubs** telle que définie en pratique dans **Ggc**. En effet, il s'agit ici de décrire la compilation telle qu'elle est implémentée par le compilateur et non plus d'appliquer le système de règles de réécriture.

```
Repressilator{
@Observer{obs1:TetR_lite; obs2:!TetR_lite;}
@Behavior{g1 -> !g2; g2 -> !TetR_lite; TetR_lite -> !g1;}
};
Reporter{
@Observer{obs1:GFP; obs2:!GFP;}
@Behavior{TetR_lite -> !GFP;}
};
```

Figure 4.6 – programme **Gubs** du Repressilator

Afin de représenter le comportement du Repressilator, nous définissons deux compartiments décrivant chacun un des plasmides de la construction (`Repressilator` et `Reporter`). Le premier décrit le comportement de la boucle d'inhibition avec trois relations causales et deux points d'observations représentant chacun la présence ou l'absence de $TetR_{lite}$. Le second décrit simplement l'activation du rapporteur GFP.

4.2.2 Compilation du Repressilator

L'utilisation de **Ggc** nous donne la solution décrite dans la figure suivante (Figure 4.7).

```
Parts list:                          Substitution:
  Q4, Q5, Q6, Q15, Q16                  g1::> LambdaCL_lite
                                        g2::> Lacl_lite
Details:
  Q4: Lambda Cl                      Added genes:
  Q5: Lacl_lite                        add1::> LambdaPr
  Q6: Tetr_lite                        add2::> PLlac01
  Q15: PLtet01                         add3::> PLtet01
  Q16: activation tetr                 add4::> PLtet01
```

Figure 4.7 – Résultat de la compilation par Ggc du Repressilator. Notons qu'afin de trouver une solution en utilisant la base de données de la figure 4.10 le compilateur utilise la règle d'extension (Trans.) afin d'ajouter 4 gènes.

4.2.3 Application des algorithmes de compilation

Nous allons à présent détailler le fonctionnement de l'algorithme utilisé par **Ggc** lors de la compilation de la partie boucle d'inhibition Repressilator (Figure 4.6). Afin de clarifier l'exemple, en raison de la combinatoire induite par l'algorithme, nous l'appliquerons sur une partie de la base de données (cette partie est donnée dans la figure 4.8).

Pour simplifier la lisibilité, nous libellons les relations causales du programme du Repressilator sous la forme $p_\star$ comme suit :

$$p_1 : g1 \odot\!\!\rightarrow !g2, p_2 : g2 \odot\!\!\rightarrow !TetR\_lite, p_3 : TetR\_lite \odot\!\!\rightarrow !g1.$$

Ce découpage en relation causale correspond ici à la mise en forme normale du programme de Repressilator.

1. Par application de la première étape de sélection de l'algorithme, les contraintes sur les causes, nous définissons les relations causales $p_i, i \in \{1, 2, 3\}$ qui sont localement associées avec chaque Q_j.
   — p_1 est localement associée à tous les composants : en effet, aucune contrainte ne permet d'éliminer des composants.

Database
<id>Q1</id><name>TetR_lacIP</name> <meta>TetR cassette with LacI promoter</meta> <prog>@B{LacI⊙→!TetR_lite;}</prog>
<id>Q2</id><name>LacI_CIP</name> <meta>LacI-lite cassette with lamda cI promoter</meta> <prog>@B{CI⊙→!LacI;}</prog>
<id>Q3</id><name>CI_TetP</name> <meta>Lambda cI cassette with Tet promoter</meta> <prog>@B{!TetR_lite⊙→!CI;}</prog>
<id>Q4</id><name>Gfp_TetP</name> <meta>GFP cassette with Tet promoter</meta> <prog>@B{!TetR_lite⊙→!Gfp;}</prog>
<id>Q5</id><name>LaclM1_LuxRP</name> <meta>LaclM1 cassette with LuxR promoter</meta> <prog>@B{LuxR⊙→LaclM1; LaclM1⊙→!Gfp}</prog>

Figure 4.8 – Fragment de la base de données utilisée pour l'application de l'algorithme de compilation

— p_2 est localement associée à Q_1 en raison de la constante *TetR_lite* dans la partie effet de la relation.
— Et finalement, p_3 est localement associée à Q_3 et Q_4 en raison de la présence de $TetR_lite$ dans la partie cause de la relation.

2. Ainsi nous pouvons réduire le nombre de combinaisons à tester pour l'unification afin de passer de $3^5 = 243$ à 8 :

$$\{(Q_1, Q_1, Q_3); (Q_2, Q_1, Q_3); (Q_3, Q_1, Q_3); (Q_4, Q_1, Q_3); (Q_5, Q_1, Q_3);$$

$$(Q_1, Q_1, Q_4); (Q_2, Q_1, Q_4); (Q_3, Q_1, Q_4); (Q_4, Q_1, Q_4); (Q_5, Q_1, Q_4)\}$$

3. Par application des contraintes sur les agents on obtient ensuite :
— Par application de la deuxième règle à $g2$, (Q_1, Q_1, Q_3) et (Q_1, Q_1, Q_4) ne peuvent pas couvrir P car *TetR_lite* est déjà présente dans P et ne peut donc pas être une substitution de la variable $g2$.
— Comme pour $g2$, (Q_3, Q_1, Q_3), (Q_4, Q_1, Q_3), (Q_3, Q_1, Q_4), (Q_4, Q_1, Q_4) ne peuvent pas couvrir P car *TetR_lite* est utilisée dans P donc elle ne peut pas substituer $g1$.

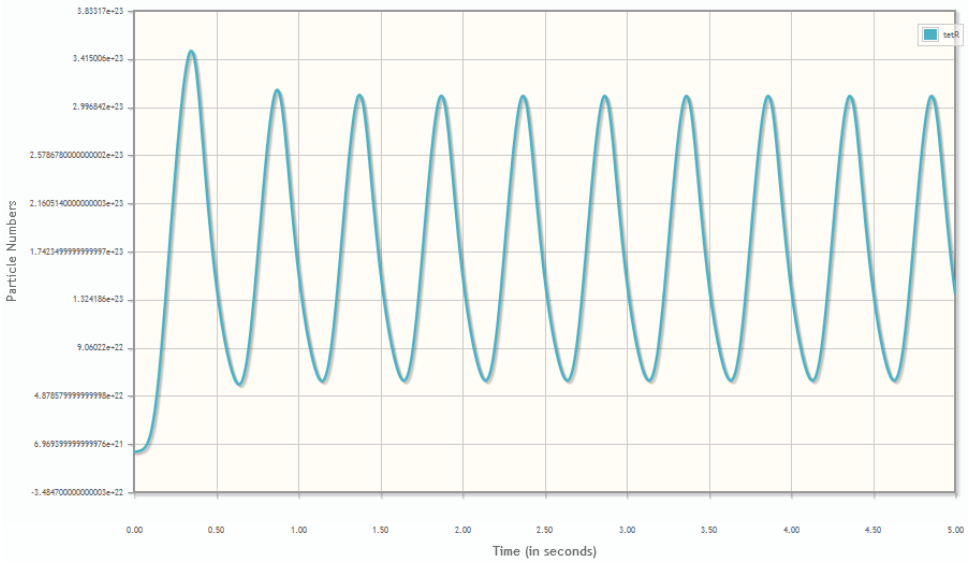
— L'unification de (Q_5, Q_1, Q_3) échoue car aucune constante autre que *TetR_lite* n'apparaît plus d'une fois.
— De façon similaire, l'unification de (Q_5, Q_1, Q_4) échoue car aucune constante autre que *TetR_lite* n'apparaît au moins une fois dans les causes et une fois dans les effets.

4. De ce fait, (Q_2, Q_1, Q_3) et (Q_2, Q_1, Q_4) constituent les choix restants.
5. L'étape suivante de l'algorithme consiste en l'unification de chaque variable avec une constante :
   — Avec cette étape, en utilisant (Q_2, Q_1, Q_4), $g2$ peut être substituée par $LacI$ mais $g1$ ne peut être unifiée.
   — La seule solution restante est (Q_2, Q_1, Q_3) où $g2$ est substituée avec $LacI$ et $g1$ avec CI.
6. Cette dernière solution (Q_2, Q_1, Q_3) correspond bien biologiquement à la description du Repressilator.

4.2.4 Modélisation de la spécification

Nous proposons à présent d'effectuer une simulation de l'assemblage obtenue par compilation. Pour ce faire, nous utilisons les informations biologiques présentes dans la base de données. Ici les informations biologiques sont prévues pour être utilisées dans COPASI via GenoCAD. Dans GenoCAD, chaque composant de la base de données est vu comme une cassette caractérisée par une séquence : Promoteur, RBS, séquence codante, terminateur. La cassette est composée d'un complexe promoteur/gène représentant la relation causale. Le RBS et le terminateur sont quant à eux ajoutés de façon standard. Cette simulation, en plus de valider notre exemple, permet de montrer la possibilité d'une chaîne connectant un langage fondé sur une approche comportementale vers une approche structurelle, et ce de façon automatique. Ici, nous associons un programme **Gubs** à chaque cassette, et une fois celles-ci assemblées la simulation obtenue correspond aux observations de l'expérience originale, comme le montre la figure 4.9.

- Simulation via COPASI de l'évolution de la concentration de la molécule TetR -



- Base de données étendue avec la description des cassettes utilisées dans GenoCAD -

Base de données
<id>Q1</id><name>TetR_lacIP</name> <meta>TetR cassette with LacI promoter</meta> < *BData* >**a069g,a069m,a069l,a069n**< /*BData* > <prog>@B{LacI⊙→!TetR_lite;}</prog>
<id>Q2</id><name>LacI_CIP</name> <meta>lacI cassette with cI promoter</meta> $<BData>$a069h,a069m,a069k,a069n$</BData>$ <prog>@B{CI⊙→!LacI;}</prog>
<id>Q3</id><name>CI_TetP</name> <meta>cIts cassette with TetR promoter</meta> < *BData* >**a069i,a069m,a069j,a069n**< /*BData* > <prog>@B{!TetR_lite⊙→!CI;}</prog>

Figure 4.9 – Simulation et base de données des composants utilisés étendue avec les informations biologiques pour la simulation

```
<name>Q4</name><meta>Lambda Cl</meta>
<prog>@B{LambdaCL_lite -> LambdaPr; LambdaPr -> !Lacl_lite;}</prog>
//=====================================================================
<name>Q5</name><meta>Lacl_lite</meta>
<prog>@B{Lacl_lite -> PLlac01; PLlac01 -> !TetR_lite;}</prog>
//=====================================================================
<name>Q6</name><meta>TetR_lite</meta>
<prog>@B{TetR_lite -> PLtet01; PLtet01 -> !LambdaCL_lite;}</prog>
//=====================================================================
<name>Q7</name><meta>sensor</meta>
<prog>@B{[Light]{Detect -> Tetr};}</prog>
//=====================================================================
<name>Q8</name><meta>Tetr</meta>
<prog>@B{Tetr -> Luxl;}</prog>
//=====================================================================
<name>Q9</name><meta>Luxl</meta>
<prog>@A{AHL:[][low != mid != high];}
@B{Luxl -> AHL(low); !Luxl -> !AHL(low); Luxl -> AHL(mid);
!Luxl -> !AHL(mid); Luxl -> AHL(high); !Luxl -> !AHL(high);}</prog>
//=====================================================================
<name>Q10</name><meta>Ahl</meta>
<prog>@A{AHL:[][low != mid != high]; LuxR:[mid < high][low != mid];}
@B{AHL(mid) -> LuxR(mid); AHL(high) -> LuxR(high);}</prog>
//=====================================================================
<name>Q11</name><meta>luxr</meta>
<prog>@A{LuxR:[mid < high][low != mid];}
@B{LuxR(mid) -> Cl,!LaclM1; LuxR(low) -> !Cl; LuxR(high) -> Cl,LaclM1;}</prog>
//=====================================================================
<name>Q12</name><meta>Cl</meta>
<prog>@B{Cl -> !Lacl; !Cl -> Lacl;}</prog>
//=====================================================================
<name>Q13</name><meta>laclm1</meta>
<prog>@B{LaclM1 -> !GFP; !LaclM1 -> GFP;}</prog>
//=====================================================================
<name>Q14</name><meta>gfp inhibitor lacl</meta>
<prog>@B{Lacl -> !GFP; !Lacl -> GFP;}</prog>
//=====================================================================
<name>Q15</name><meta>PLtet01</meta>
<prog>@B{PLtet01 -> !GFP;}</prog>
//=====================================================================
<name>Q16</name><meta>activation tetr</meta>
<prog>@B{TetR_lite -> PLtet01;}</prog>
```

Figure 4.10 – Fragment de la base de données des composants. Le code XML correspond à la description en **Gubs** de l'ensemble des composants biologiques utilisés pour la base de données. Pour plus de clarté, les informations biologiques ont été supprimées dans cette figure.

Conclusion

Dans le langage **Gubs**, nous proposons de caractériser un paradigme de programmation abstrayant les interactions moléculaires dans le contexte de systèmes ouverts, qui diffère des approches dédiées à la modélisation des systèmes biologiques. Les interactions sont symbolisées par l'observation de chaîne de dépendances causales dont l'interprétation est guidée par l'effet. Nous avons démontré la possibilité d'une compilation des programmes **Gubs** fondée sur des règles de réécriture, ainsi que l'automatisation de ces règles de réécriture via le compilateur **Ggc** permettant de trouver automatiquement un assemblage de composants biologiques effectuant la synthèse fonctionnelle d'une spécification en **Gubs**. L'algorithme de compilation est actuellement implémenté en Ocaml et se fonde sur un algorithme d'ACI-unification adapté aux propriétés de **Gubs**. Afin de répondre à la complexité de la compilation pour de grands systèmes biologiques, l'algorithme d'ACI-Unification est associé avec un algorithme. Nous avons appliqué cet algorithme sur plusieurs exemples de taille variable, dont deux de biologie de synthèse qui sont déroulés dans le chapitre Exemple 4.

Afin de conclure cette thèse, nous proposons ici de résumer les problématiques introduites et les approches que nous avons proposées. Une première question consiste à formaliser le comportement d'un système ouvert et plus particulièrement un système biologique afin de trouver la meilleure façon de définir et de prendre en compte l'ensemble des contraintes existantes dans de tels systèmes. En effet, la dimension ouverte implique de possibles interactions de l'environnement sur le système, qu'il s'agisse d'interactions directes comme une inhibition ou une surdétermination, ou de bruit. De plus, la biologie nous apporte des contraintes sur le châssis et la sélection même des composants, ceux-ci pouvant être dépendants d'une espèce, ou de leur localisation.

A cette fin, nous avons proposé une approche par un langage abstrait permettant de répondre à la problématique de l'ouverture en définissant une notion de causalité afin de traiter la préemption. Dans le cas de la surdétermination, nous avons fait la proposition que : notre système étant un système biologique de synthèse, la fonctionnalité est spécifiquement inexistante dans l'organisme ; proposition raisonnable car le système n'aurait pas de raison d'être développé si cette fonctionnalité est déjà présente. Notre deuxième approche, complémentaire à celle-ci, consiste à réduire les interactions non voulues au sein du système de synthèse en sélectionnant des composants orthogonaux entre eux et en sélectionnant prioritairement les plus résistants aux interactions non voulues (promoteurs forts). Les contraintes biologiques restent néanmoins présentes, en effet, elles sont à l'heure actuelle résolues par des expériences en laboratoire permettant de déterminer la viabilité d'un assemblage, et il est donc difficile actuellement de prédire la viabilité in-vivo d'un assemblage défini in-silico. Néanmoins, pour faciliter cette démarche, **Ggc** permet de stocker dans la base de données de composants des informations biologiques relatives à chaque composant biologique en vue d'effectuer une simulation du système généré. De plus, nous avons défini un algorithme de compilation facilement adaptable afin d'y ajouter de nouvelles contraintes de viabilité et de fitness pour la sélection des composants. Et donc d'ajouter des contraintes biologiques. Nous avons déjà implémenté un ensemble de règles permettant de définir les organismes dans lesquels un composant peut être inséré, ainsi que sa localisation sur un chromosome ou dans un plasmide.

Afin d'assurer la sûreté et la correction du code **Gubs**, nous avons introduit un ensemble de règles telles que l'observabilité permettant de valider le fait qu'un programme ne possède pas des comportements incompatibles entre eux, et ceci

automatiquement en utilisant la sémantique du langage et des outils de preuves existants. Nous avons étendu cette notion à l'observabilité forte permettant d'assurer qu'un assemblage obtenu à partir d'une spécification produit une trace d'expérience observable.

Une autre problématique majeure est le passage à l'échelle afin de décrire de grands systèmes biologiques. À cette fin nous avons proposé un langage de spécification du comportement des systèmes biologiques permettant de décrire un système biologique comme l'ensemble de ses comportements, et d'abstraire ses séquences jusqu'à l'étape de compilation. **Gubs** est un langage de haut niveau permettant de décrire un système biologique à de multiples échelles : de l'interaction protéine/promoteur au comportement global d'un système défini comme, par exemple, une oscillation. En effet, un ensemble de réactions peut être abstrait en un ensemble d'entrées et de sorties d'un unique comportement, le compilateur se chargeant de trouver la chaîne causale adéquate.

Une dernière problématique majeure est la complexité de la compilation d'un tel langage : Comment passer d'une spécification abstraite d'un système ouvert à un assemblage de composants biologiques concret ? Et donc comment faire le lien entre la spécification et la séquence lui correspondant ? À cette fin nous avons défini la base de données de composants comme une table d'association entre un ensemble de composants biologiques réels (possédant une séquence d'*ADN* connue), et leur traduction en un programme abstrait représentant leur comportement. Par la suite la compilation va donc chercher à associer un ensemble de composants, considéré comme le meilleur assemblage selon un ensemble de critères biologiques et théoriques tels que sa taille, l'orthogonalité de ses composants, et la force de ses promoteurs dont le comportement couvrira celui du programme **Gubs**. Nous avons donc défini la notion d'inclusion comportementale et de synthèse fonctionnelle permettant d'assurer qu'un assemblage de composants extrait d'une base de données possède bien un comportement global correspondant au moins au comportement défini dans le programme **Gubs**. Nous utilisons un algorithme d'ACI-unification, substituant les variables d'un programme par les constantes leur correspondant dans les composants biologiques correspondants. Cet algorithme étant NP, nous l'avons associé dans le cadre de base de données de grandes tailles à un algorithme d'évolution dirigée.

Une perspective de ce travail à court terme est de faciliter l'intégration de nouveaux

composants en interfaçant **Ggc** directement avec des bases de données biologiques existantes, vues comme des collections de composants. La problématique est alors de traduire les informations stockées dans la base de données en des programmes **Gubs**. Une telle traduction peut être fondée sur la caractérisation de certains motifs de comportement associé à certaines classes de composants biologiques comme par exemple dans l'exemple du Repressilator (Section 4.2), où les composants biologiques sont associés à des biobricks.

Ici nous nous sommes concentrés sur deux propriétés, l'orthogonalité [52], et la qualité des promoteurs [98]. Une autre perspective à court terme est d'améliorer encore la sélection en prenant en compte les performances des composants [91] de façon plus précise, et non seulement des promoteurs en se fondant sur un ensemble de critères biologiques.

A long terme, **Gubs** pourra être intégré à une tour de langages de description des systèmes biologiques telle que celle proposé par R.Weiss, et associé à des outils de vérification de la sûreté des systèmes engendrés, avec comme objectif la conception de systèmes biologiques de grandes tailles (plusieurs milliers de gènes). Dans une telle tour de langage, **Gubs** se place juste au-dessus des langages de description structurelle, permettant de décrire de façon abstraite les systèmes biologiques à l'échelle cellulaire. Cette perspective permettra alors la description du comportement non pas d'un système moléculaire, mais d'une population entière.

Appendice

Contenu : Ce chapitre comprend le détail des preuves de l'ensemble des théorèmes et propositions de ce manuscrit. Une grande partie de ces preuves concerne la correction des règles de réécritures. Pour plus de clarté, nous les avons catégorisé suivant les chapitres dans lesquels elles apparaissent. Et pour chaque preuve, le théorème ou la proposition correspondant est redonné en préambule.

4.3 Preuves : Chapitre 2

Proposition 4. *Le fragment de logique de* **Gubs** *est décidable.*

Démonstration. **Gubs** utilise uniquement les opérateurs @, $\wedge$, $\vee$, $[\ ]^-$, $\langle\ \rangle^-$ et $\forall$, or les opérateurs $[\ ]^-$ et $\langle\ \rangle^-$ sont substituables par $[\ ]$ et $\langle\ \rangle$ et les outils actuels permettent la résolution automatique de formules ne comprenant que ces opérateurs [71]. Un programme **Gubs** est donc traductible dans un fragment de la logique hybride décidable. □

Théorème 5. *Déterminer l'observabilité d'un programme* **Gubs** *se calcule en un temps de complexité* $\mathcal{O}(3^n)$ *où* n *est le nombre de lignes du programme.*

Démonstration. Un programme **Gubs** se traduit en un ensemble de causes C reliées par des conjonctions ($\wedge$), tel que chaque cause s'écrit suivant son type :

- $S_2 \rightarrow \langle K\rangle^- S_1$ pour une cause normale $S_1 \circ\!\!\rightarrow S_2$ avec un contexte K soit $\neg S_2 \vee \langle K\rangle^- S_1$.
- $S_2 \rightarrow (S_1 \wedge \langle K\rangle^- S_1)$ pour une cause persistante $S_1 \odot\!\!\rightarrow S_2$ avec un contexte K soit $\neg S_2 \vee (S_1 \wedge \langle K\rangle^- S_1)$
- $S_2 \rightarrow (\langle\ \rangle^- S_2 \vee \langle K\rangle^- S_1)$ pour une cause rémanente $S_1 \oplus\!\!\rightarrow S_2$ avec un contexte K, soit $\neg S_2 \vee (\langle\ \rangle^- S_2 \vee \langle K\rangle^- S_1)$

La méthode à tableaux créée une nouvelle branche dans le cas de l'application de la règle ($\vee$.) à une formule $F = A, \Delta$ avec $A = k \vee \delta$, tel que les deux branches résultantes sont $F_1 = k, \Delta$ et $F_2 = \delta, \Delta$. Une cause rémanente comprenant deux disjonctions, nous créons 3 nouvelles branches. Par récursivité, il est trivial de montrer alors que dans le cas d'un programme ne contenant que des causes rémanentes, la complexité d'application de la méthode à tableau est de 3^n ou n est le nombre de causes du programme. □

4.4 Preuves : Chapitre 3

Théorème 6. *Tout programme* **Gubs** *peut être mis en forme normale en gardant l'équivalence au niveau des formules et du comportement engendré.*

Démonstration. Soit un programme P tel que $P = \bigcup_1^n (C_i)$ où C est un ensemble de compartiments comprenant m définitions d'attribut A_j et l contextes K_k. Chaque contexte contenant lui-même f relations causales R_t.

Compartiment La notion de compartiment étant définie dans la sémantique comme un renommage des agents, on peut trivialement les distribuer au niveau des agents.

Contexte De même, les contextes correspondant aux labels des opérateurs $\langle\ \rangle$ et $[\ ]$, ils sont automatiquement distribués par la sémantique au niveau de chaque relation causale.

Attributs Les attributs sont définis comme un renommage des agents, ainsi que par une formule définissant leur relation. Soit S l'ensemble des formules définissant leur relation, et donc $S \wedge \Delta$ l'ensemble de ces relations associé aux formules correspondant au reste du programme par définition des opérateurs logiques on peut écrire la formule $S \wedge S \wedge \Delta$. Si Δ contient plus d'une relation causale, on peut la réécrire comme $\Delta_1 \wedge \Delta_2$ et donc définir la formule : $S \wedge \Delta_1 \wedge S \wedge \Delta_2$ et ainsi distribuer. De plus S se définissant comme un ensemble de relations, on peut diviser cet ensemble en $S_1 \wedge S_2$ où chacun correspond à un sous-ensemble de relations. Puis similairement distribuer ces sous-ensembles aux relations causales auxquelles ils correspondent par les règles de commutativité de l'opérateur $\wedge$.

À partir de cette étape, la mise en forme normale d'un programme sans contexte, attribut ou compartiment est triviale par les règles sur les opérateurs logiques. □

Proposition 5. *Un programme inclus comportementalement dans un programme observable,* $\boldsymbol{obs\ P}$*, est observable (selon la proposition 2) :* $\forall P, Q \in \mathsf{P} : (\boldsymbol{obs}\, Q) \wedge (P \trianglelefteq Q) \implies \boldsymbol{obs}\, P$.

Démonstration. Par contradiction, supposons que P n'est pas observable, alors il n'existe pas de modèle qui satisfasse la formule. Comme Q est observable, nous en déduisons qu'il existe des modèles satisfaisants Q, mais aucun sous-modèle satisfaisant P, ce qui contredit la définition de l'inclusion comportementale. □

Proposition 6. *Soit* $\psi \in \mathsf{F}_{\mathcal{H}}$ *une formule, soit* $\sigma : (\mathsf{NOM} \cup \mathsf{PROP} \cup \mathsf{REL}) \to (\mathsf{NOM} \cup \mathsf{PROP} \cup \mathsf{REL})$ *une substitution de nominaux, variables et symboles relationnels, soit* $\mathcal{M} = \langle W, (R_k)_{k \in \tau}, V \rangle$ *un modèle, nous définissons le modèle* $\tilde{\mathcal{M}} = \langle W, (\tilde{R}_k)_{k \in \tilde{\tau}}, \tilde{V} \rangle$ *de* $\mathcal{M}$ *comme suit :*

1. $\forall a \in \mathsf{NOM} \cup \mathsf{PROP}, \forall w \in W : w \in V(a\sigma) \iff w \in \tilde{V}(a)$
2. $\forall k \in \tilde{\tau} : w R_{k\sigma} w' \iff w \tilde{R}_k w'$;

Nous avons : $\mathcal{M}, w \Vdash \psi\sigma \iff \tilde{\mathcal{M}}, w \Vdash \psi$.

Démonstration. La preuve est définie par induction sur la formule :

Sans perte de généralité, nous assumons que ψ est en forme normale de négation, avec la négation n'apparaissant qu'immédiatement devant les variables. Rappelons que toute formule peut être mise en forme normale de négation.

— $\mathcal{M}, w \Vdash a \iff \tilde{\mathcal{M}}, w \Vdash a, a \in \mathsf{PROP} \cup \mathsf{NOM}$. Par (1), nous avons $w \in V(a\sigma) \iff w \in \tilde{V}(a)$ correspondant à l'équivalence.

— $\mathcal{M}, w \Vdash \neg a \iff \tilde{\mathcal{M}}, w \Vdash \neg a$. Par définition de la relation de réalisabilité, ceci est équivalent à : $\tilde{\mathcal{M}}, w \nVdash a \iff \tilde{\mathcal{M}}, w \nVdash a$. Par (1), cette équivalence est maintenue.

— $\mathcal{M}, w \Vdash (\psi_1 \wedge \psi_2)\sigma \iff \tilde{\mathcal{M}}, w \Vdash (\psi_1 \wedge \psi_2)$. Par définition de la substitution, nous avons à prouver que : $\mathcal{M}, w \Vdash (\psi_1\sigma) \wedge (\psi_2\sigma) \iff \tilde{\mathcal{M}}, w \Vdash (\psi_1 \wedge \psi_2)$. Par définition de la relation de réalisabilité nous pouvons formuler la propriété de façon équivalente comme suit :

$$\mathcal{M}, w \Vdash (\psi_1\sigma) \wedge \tilde{\mathcal{M}}, w \Vdash (\psi_2\sigma) \iff \tilde{\mathcal{M}}, w \Vdash \psi_1 \wedge \tilde{\mathcal{M}}, w \Vdash \psi_2.$$

Par hypothèse d'induction, nous avons : $\tilde{\mathcal{M}}, w \Vdash (\psi_1\sigma) \iff \tilde{\mathcal{M}}, w \Vdash \psi_1$ et $\tilde{\mathcal{M}}, w \Vdash (\psi_2\sigma) \iff \tilde{\mathcal{M}}, w \Vdash \psi_2$, impliquant les conditions précédentes.

— $\mathcal{M}, w \Vdash (\psi_1 \vee \psi_2)\sigma \iff \tilde{\mathcal{M}}, w \Vdash (\psi_1 \vee \psi_2)$. La preuve est similaire à la preuve de l'item précédent ($\wedge$).

— $\mathcal{M}, w \Vdash (@_a\psi)\sigma \iff \tilde{\mathcal{M}}, w \Vdash @_a\psi$. Par définition de la substitution, nous avons à prouver que : $\mathcal{M}, w \Vdash (@_{a\sigma}\psi\sigma) \iff \tilde{\mathcal{M}}, w \Vdash @_a\psi$ Par définition de la relation de réalisabilité, ceci est équivalent à :

$$\exists w' \in W : w \in V(a\sigma) \wedge \mathcal{M}, w' \Vdash \psi\sigma \iff \exists w'' \in W : w'' \in \tilde{V}(a)\sigma \wedge \tilde{\mathcal{M}}, w'' \Vdash \psi.$$

En définissant $w' = w''$, à partir de (1) nous avons : $w' \in V(a\sigma) \iff w' \in V(a)$. Par hypothèse d'induction, nous avons : $\mathcal{M}, w' \Vdash \psi\sigma \iff \tilde{\mathcal{M}}, w' \Vdash \psi$. Les deux dernières propriétés impliquent que :

$$\exists w' \in W : w \in V(a\sigma) \wedge \mathcal{M}, w' \Vdash \psi\sigma \iff \exists w' \in W : w' \in \tilde{V}(a)\sigma \wedge \tilde{\mathcal{M}}, w'' \Vdash \psi,$$

ce qui implique la propriété initiale.

— $\mathcal{M}, w \Vdash (\langle k \rangle \psi)\sigma \iff \tilde{\mathcal{M}}, w \Vdash \langle k \rangle \psi$. Par définition de la substitution nous

prouvons que : $\mathcal{M}, w \Vdash \langle k\sigma\rangle\psi\sigma \iff \tilde{\mathcal{M}}, w \Vdash \langle k\rangle\psi$.
Par définition de la relation de réalisabilité, la condition est équivalente à :

$$\exists w' \in W : \mathcal{M}, w' \Vdash \psi\sigma \wedge wR_{k\sigma}w' \iff \exists w'' \in W : \tilde{\mathcal{M}}, w'' \Vdash \psi \wedge w\tilde{R}_k w''.$$

En définissant $w' = w''$, l'équivalence suivante tirée de (2) : $wR_{k\sigma}w' \iff w\tilde{R}_k w'$. Par hypothèse d'induction, nous avons : $\mathcal{M}, w' \Vdash \psi\sigma \iff \tilde{\mathcal{M}}, w' \Vdash \psi$. Les deux dernières propriétés impliquent que :

$$\exists w' \in W : \mathcal{M}, w' \Vdash \psi\sigma \wedge wR_{k\sigma}w' \iff \tilde{\mathcal{M}}, w' \Vdash \psi \wedge w\tilde{R}_k w'$$

ce qui implique la propriété initiale.

— $\mathcal{M}, w \Vdash ([k]\psi)\sigma \iff \tilde{\mathcal{M}}, w \Vdash [k]\psi$. La preuve est similaire à l'item précédent.

— $\mathcal{M} \Vdash (\mathbf{E}\psi)\sigma \iff \tilde{\mathcal{M}} \Vdash \mathbf{E}\psi$. Par définition de la substitution nous prouvons que : $\mathcal{M}, w \Vdash \mathbf{E}(\psi\sigma) \iff \tilde{\mathcal{M}}, w \Vdash \mathbf{E}\psi$.
Par définition de la relation de réalisibilité nous avons :

$$\exists w \in W : \mathcal{M}, w \Vdash (\psi\sigma) \iff \tilde{\mathcal{M}}, w \Vdash \psi,$$

qui est directement vérifiée par l'hypothèse d'induction.

— $\mathcal{M} \Vdash (\mathbf{A}\psi)\sigma \iff \tilde{\mathcal{M}} \Vdash \mathbf{A}\psi$. La preuve est similaire à celle de l'item précédent.

□

Proposition 7. *Pour toute substitution σ, nous avons : $P \Subset Q \implies P[\sigma] \Subset Q[\sigma]$.3*

Démonstration. Premièrement, remarquons que quand $P \Subset Q$, la propriété est validée trivialement. De plus, en supposant que $P \Subset Q$, si $Q[\sigma]$ n'est pas observable, la propriété est aussi vérifiée car un programme non observable inclut comportementalement tous les programmes. (Définition 4).

Dans le reste de la preuve nous supposons que P est inclus comportementalement dans Q et $Q[\sigma]$ est observable (*i.e.*, $P \Subset Q$ et $\mathbf{obs}\, Q[\sigma]$). De plus, par définition de l'observabilité, il existe un modèle $\mathcal{M}$ tel que $\mathcal{M} \Vdash [\![Q[\sigma]]\!]$. Par la proposition 6, nous déduisons qu'il existe un modèle $\tilde{\mathcal{M}}$ tel que : $\tilde{\mathcal{M}} \Vdash [\![Q]\!]$. De plus, comme $P \Subset Q$ par hypothèse, il existe $\tilde{S} \subseteq \mathrm{Dom}\, \tilde{\mathcal{M}}$ tel que : $\tilde{\mathcal{M}}_{\tilde{S}} \Vdash [\![P]\!]$. Par construction de $\tilde{\mathcal{M}}$ nous déduisons qu'il existe un sous-modèle de $\mathcal{M}$, noté $\mathcal{M}'$, correspondant aux propriétés

(1) et (2) de la Proposition 6 qui correspond à $\tilde{\mathcal{M}}_{\tilde{S}}$. De plus, nous avons $\mathcal{M}' \Vdash P[\sigma]$ par la Proposition 6. Pour finir nous concluons que : $P[\sigma] \Subset Q[\sigma]$. □

Théorème 7. *Les règles de synthèse fonctionnelle (Inst.), (Com.), (Cont.) et (Asm.) (Table 3.2) sont correctes.*

Démonstration. Premièrement, remarquons que $P \Subset Q$ est vrai quand $\mathcal{M} \nVdash Q$ par définition de l'inclusion comportementale (Définition 4). Ici, la preuve ne considère pas le cas trivialement vérifié, mais le cas où $\mathcal{M} \Vdash Q$.

Inst. Directement à partir de la définition de l'inclusion comportementale (Définition 4).

Com. Par définition de la sémantique $[\![P, P']\!] = \mathbf{A}(\phi \wedge \phi') = \mathbf{A}(\phi' \wedge \phi) = [\![P', P]\!]$ avec $[\![P]\!]_P = \phi$ et $[\![P']\!]_P = \phi'$. Ensuite, pour tout $\mathcal{M}$ nous avons : $\mathcal{M} \Vdash [\![P, P']\!] \iff \mathcal{M} \Vdash [\![P', P]\!]$. Pour finir, si $Q \Subset P, P'$ nous concluons que : $Q \Subset P', P$.

Cont. Similaire à la preuve de (Com.).

Asm. Premièrement remarquons que $\sigma|_{\mathrm{VA}(P)\cap\mathrm{VA}(P')} = \sigma'|_{\mathrm{VA}(P)\cap\mathrm{VA}(P')}$ signifie que la substitution des variables communes est la même pour σ et σ', ce qui nous donne, $Q[\sigma \cup \sigma'] = Q[\sigma]$ et $Q'[\sigma \cup \sigma'] = Q'[\sigma']$. Soit $\sigma'' = \sigma \cup \sigma'$. Ensuite, nous avons la propriété suivante par définition de la sémantique (Table 2.3) et σ''.

$$\forall \mathcal{M} \in \mathsf{KS}([\![(Q, Q')[\sigma'']]\!]) : \mathcal{M} \Vdash [\![Q[\sigma]]\!] \wedge \mathcal{M} \Vdash [\![Q'[\sigma']]\!] .$$

Notons que l'ensemble de modèles, $\mathsf{KS}([\![(Q, Q')[\sigma'']]\!])$, n'est pas vide à partir du moment où, par hypothèse,
$\mathbf{obs}\,(Q[\sigma], Q'[\sigma'])$ sont présents. Comme $Q \leftarrow_{\sigma} P$ et $Q' \leftarrow_{\sigma'} P'$, tout modèle validant Q (resp. Q') valide aussi P, (resp. P') Par définition de la synthèse fonctionnelle. Ensuite nous déduisons que :

$$\forall \mathcal{M} \in \mathsf{KS}([\![(Q, Q')[\sigma'']]\!]) : \mathcal{M} \Vdash [\![P[\sigma]]\!] \wedge \mathcal{M} \Vdash [\![P'[\sigma']]\!] .$$

Ensuite nous concluons que :

$$\forall \mathcal{M} \in \mathsf{KS}([\![(Q, Q')[\sigma'']]\!]) : \mathcal{M} \Vdash [\![(P, P')[\sigma'']]\!] .$$

□

Théorème 8. *Les règles de dépendance (Trans.),(N2P.),(R2N.) (Table 3.3) sont correctes en accord avec la notion d'inclusion comportementale.*

Démonstration. **Trans.** En se fondant sur la sémantique, cette relation est trivialement transitive, en raison de la transitivité de l'opérateur $\Rightarrow$.

N2P. Passer de la causalité normale à la causalité persistante consiste à ajouter une contrainte sur le modèle, Par définition de l'inclusion comportementale, cette opération est valide.

R2N. La démonstration est la même que pour l'item précédent.

□

Théorème 9. *Le problème de la synthèse fonctionnelle est NP-Complet.*

Démonstration. Par réduction au problème de couverture minimale (SP5 dans [57]).
Le problème est NP. Supposons que nous avons une substitution σ et $Q = \{Q_i\}_i$ un ensemble de composants, la vérification de l'inclusion de $P[\sigma]$ dans $\bigcup_{Q_j \in C} Q_j[\sigma]$ se calcule en temps polynomial.
Le problème est NP-complet. La réduction est effectuée sur le problème de couverture minimale en se fondant sur un encodage des éléments par des dépendances.
Instance : une collection X de sous-ensembles d'un ensemble fini S, et un entier positif $k < |C|$.
Question : X contient-il une couverture de S de taille k ou moins (*i.e.*, un sous ensemble $X' \subseteq X$ avec $|X'| \leq k$ tel que chaque élément de S correspond au moins à un membre de X') ?

Réduction. Chaque élément $A \in S$ est assimilé à une constante, traduite en une relation de dépendance $A \circ\!\!\rightarrow A$. Ainsi, la substitution est trivialement la relation d'identité. La base de données est X(*i.e.*, $X = \Gamma$). Finalement, le résultat de la synthèse fonctionnelle est X' (*i.e.*, $X' = C$). □

4.5 Ggc

Dans cette section, nous décrivons en détail chaque module du compilateur **Ggc**.

4.5.1 Traduction

Ce module se divise en 7 fichiers distincts réalisant deux opérations distinctes :

[Gubs_parse, Gubs_Lexx, Gubs_AST] Ce premier triplet de fichiers se compose d'un AST (Gubs_AST), d'un parser (Gubs_parse) et d'un lexer (Gubs_Lexx) permettant de traduire la syntaxe de **Gubs** sous forme d'un arbre de syntaxe. Le parser prend en compte les macros en utilisant les définitions « à la C » permettant ainsi de définir de nouvelles fonctions pour les programmes, telles que l'inhibition ou l'activation. De plus un programme peut être fragmenté en plusieurs fichiers.

[XML_parse, XML_Lexx, XML_AST] Ce second triplet de fichiers correspond à un AST (XML_AST), un lexer (XML_Lexx) et un parser XML (XML_parse) prenant en entrée une base de données en XML et une DTD (Par défaut celle définie pour la base de données **Gubs**) et traduisant la base de données selon la DTD. Une telle approche permet de définir dans le XML des champs « mots-clés » correspondant à des contraintes à respecter par la suite. Une base de données **Gubs** se compose d'un ensemble de composants chacun devant au moins posséder deux champs : *id* identifiant chaque composants de façon unique, et un champ *programme* pour que le compilateur puisse extraire le programme correspondant au composant ; les autres champs peuvent être définis librement dans le DTD et seront considérés comme un ensemble de « mots-clés » correspondant à des contraintes ou des informations. Ainsi, l'utilisateur peut ajouter des contraintes aux fonctions de fitness ou de viabilité directement dirigées par les champs qu'il ajoute à la DTD. Par exemple, définir un champ `chassis` peut permettre de définir pour chaque composant le châssis cellulaire auquel il correspond et ainsi définir des contraintes sur les châssis. Tout comme pour un programme classique, une base de données peut être fragmentée en plusieurs fichiers, et plusieurs bases de données ayant potentiellement des DTD différentes peuvent être utilisées simultanément. Néanmoins dans ce cas, une DTD virtuelle correspondant à l'intersection de toutes les DTD sera utilisée pour définir les informations et contraintes à utiliser.

[Pretty_Gubs] Le dernier fichier est simplement un pretty printer prenant un programme sous forme d'arbre syntaxique et retournant sa forme en syntaxe **Gubs** en l'affichant soit à l'écran soit dans un fichier.

4.5.2 Normalisation

Le module de normalisation se compose de 2 fichiers transformant un arbre de syntaxe **Gubs** en un autre arbre plus approprié à la compilation. La forme normale étant stable mais la syntaxe de **Gubs** pouvant encore évoluer, nous avons considéré que séparer concrètement les deux formes était une solution optimale. En effet, en cas de changement dans la syntaxe, la mise en forme normale dès le « parsing » nécessiterait un changement permanent de la forme normale et de la compilation en résultant, soit, du compilateur entier.

[Normal_AST] Le premier fichier est donc un AST correspondant à un programme en forme normale : pas de compartiment, des variables et constantes renommées, chaque relation causale possède l'ensemble de ses définitions d'attribut, pas de points d'observation.

[Gubs_to_Normal] Le second fichier est un « wrapper » passant de l'AST de traduction vers celui de forme normale.

4.5.3 Observe

Ce module se compose de 2 fichiers, permettant de valider l'observabilité d'un programme via les méthodes à tableaux.

[Normal_to_Logic] Le premier fichier correspond à un traducteur en logique d'un programme sous forme normale. Ceci se fait par simple application des règles de sémantique sur le programme en forme normale.

[Tableaux_call] Le second fichier permet d'appeler un outil d'utilisation des méthodes à tableaux (pour le moment Htab) et d'effectuer les modifications nécessaires sur la formule pour que celle-ci corresponde à la syntaxe du programme. Ce fichier encapsule tout ceci dans une unique fonction renvoyant un booléen indiquant si la formule est satisfiable ou pas, et donc si le programme est correct ou non. Néanmoins les méthodes à tableaux étant lentes à opérer, cette fonctionnalité peut être coupée.

4.5.4 Contraintes

Ce module se compose de 3 fichiers permettant d'appliquer l'ensemble des contraintes de viabilité à un ensemble de composants donnés.

[Causal_C] Le premier fichier définit pour une base de données et un programme donné, l'ensemble des composants de la base de données associé à chaque relation causale du programme, c'est la contrainte sur les relations causales. Pour cela, il applique l'ensemble des contraintes sur les causes définies précédemment.

[Obs_C] Le second fichier définit pour un ensemble de composants donnés si ceux-ci sont observables en appliquant la propriété d'observabilité forte.

[viability_C] Le troisième fichier permet de définir un ensemble de fonctions de viabilité supplémentaires à appeler selon des mots-clés définis dans la DTD de la base de données. L'utilisation de cette option peut produire un crash à la compilation. En effet une fonction faisant appel à un mot-clé n'existant pas entraîne une erreur de compilation. Ces contraintes sont rédigées sous forme de fonctions Ocaml possédant en entrée un ensemble de composants et en sortie un booléen.

4.5.5 Fitness

Ce module se compose de 3 fichiers permettant d'appliquer l'ensemble des contraintes de fitness sur les composants.

[Agents_C] Le premier fichier définit pour un ensemble de composants donné et pour un programme donné si ces composants valident les contraintes sur les agents, renvoyant un pourcentage de couverture entre 0 et 1.

[Bio_C] Le second fichier définit pour un ensemble de composants donné et pour un programme donné, si ces composants valident les contraintes biologiques, telles que le rapport d'unification ou d'orthogonalité en renvoyant un pourcentage normalisé entre 0 et 1.

[Fitness_C] Le troisième fichier, permet, comme pour les contraintes de viabilité de définir des contraintes fondées sur les mots clé de la DTD pour un programme donné et un assemblage de composants, de donner un pourcentage de couverture de ce programme par l'assemblage.

4.5.6 Core

Ce module permet d'effectuer la compilation d'un programme **Gubs** vers un ensemble de composants biologiques. Pour ce faire il se compose de 6 fichiers :

[Component_selec] Le premier fichier permet de sélectionner un composant parmi un ensemble en fonction de son poids défini dans l'AST et de générer ainsi un ensemble de composants considéré comme optimal pour l'unification.

[Call_gen_algo] Le second fichier permet de créer une population d'individus grâce à un algorithme génétique.

[Gen_algo] Le troisième fichier correspond à l'algorithme génétique.

[Unification] Le quatrième fichier correspond à l'unification, il prend un ensemble de composants et un programme et tente d'appliquer l'ACI-unification. Si celle-ci échoue, il renvoie false, sinon il renvoie la substitution obtenue ainsi que l'ensemble des composants utilisés. On notera que ce fichier se trouve dans le module Core et non dans celui **Contraintes** pour une raison d'uniformité. En effet les contraintes de viabilité renvoient un booléen là où l'unification renvoie soit false, soit un couple (substitution, ensemble de composants).

[Func_synth] Le cinquième fichier permet la compilation : Pour une base de données de petite taille (200 ou moins), il applique la fonction de contrainte sur les causes, puis celles définies par l'utilisateur, puis si une relation causale n'a qu'un seul composant la couvrant, unifie immédiatement la relation avec le composant puis substitue tous les agents de cette relation dans le programme. Sinon, il sélectionne pour chaque relation causale du programme le composant ayant le meilleur poids. Ensuite, cette fonction valide l'observabilité forte de l'assemblage de composants puis les contraintes sur les agents, puis les contraintes biologiques et enfin les contraintes ajoutées par l'utilisateur. Pour une grande base de données, l'application est la même, sauf à l'étape de sélection du meilleur composant pour chaque relation causale où cette sélection est faite via l'algorithme génétique.

[Optimize] Le dernier fichier permet d'appliquer les règles d'optimisation à un ensemble d'agents, qu'il s'agisse de l'extension ou de l'affaiblissement des relations causales.

4.5.7 UI

Ce module permet de lancer le compilateur et de coordonner l'ensemble des fonctions de **Ggc**, il se compose de deux fichiers.

[Errors] Le premier fichier permet de définir toutes les erreurs et messages associés pour tout le compilateur.

[Ggc_Main] Le second fichier définit les arguments de la ligne de commande et, pour un programme et une base de données, effectue la compilation du programme en prenant en compte les informations passées en argument.

Bibliographie

Bibliographie

[1] A.V. Aho, M.S. Lam, J.D. Ullman, and R. Sethi. *Compilers : Principles, Techniques, and Tools*. Pearson Education, 1986.

[2] A.V.A. Aho and J.D.A. Ullman. *Principles of Compiler Design*. Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley Publ., 1977.

[3] Russ B. Altman. Editorial : Building successful biological databases. *Briefings in Bioinformatics*, 5(1) :4–5, 2004.

[4] Ernesto Andrianantoandro, Subhayu Basu, David K Karig, and Ron Weiss. Synthetic biology : new engineering rules for an emerging discipline. *Molecular Systems Biology*, 2(1) :n/a–n/a, 2006.

[5] Narayana Annaluru, Héloïse Muller, Leslie A. Mitchell, Sivaprakash Ramalingam, Giovanni Stracquadanio, Sarah M. Richardson, Jessica S. Dymond, Zheng Kuang, Lisa Z. Scheifele, Eric M. Cooper, Yizhi Cai, Karen Zeller, Neta Agmon, Jeffrey S. Han, Michalis Hadjithomas, Jennifer Tullman, Katrina Caravelli, Kimberly Cirelli, Zheyuan Guo, Viktoriya London, Apurva Yeluru, Sindurathy Murugan, Karthikeyan Kandavelou, Nicolas Agier, Gilles Fischer, Kun Yang, J. Andrew Martin, Murat Bilgel, Pavlo Bohutskyi, Kristin M. Boulier, Brian J. Capaldo, Joy Chang, Kristie Charoen, Woo Jin Choi, Peter Deng, James E. DiCarlo, Judy Doong, Jessilyn Dunn, Jason I. Feinberg, Christopher Fernandez, Charlotte E. Floria, David Gladowski, Pasha Hadidi, Isabel Ishizuka, Javaneh Jabbari, Calvin Y. L. Lau, Pablo A. Lee, Sean Li, Denise Lin, Matthias E. Linder, Jonathan Ling, Jaime Liu, Jonathan Liu, Mariya London, Henry Ma, Jessica Mao, Jessica E. McDade, Alexandra McMillan, Aaron M. Moore, Won Chan Oh, Yu Ouyang, Ruchi Patel, Marina Paul, Laura C. Paulsen, Judy Qiu, Alex Rhee, Matthew G. Rubashkin, Ina Y. Soh, Nathaniel E. Sotuyo, Venkatesh Srinivas, Allison Suarez, Andy Wong, Remus Wong, Wei Rose Xie, Yijie Xu, Allen T. Yu, Romain Koszul, Joel S. Bader, Jef D. Boeke, and Srinivasan Chandrasegaran. Total synthesis of a functional designer eukaryotic chromosome. *Science*, 344(6179) :55–58, 2014.

[6] PJ Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers, 2008.

[7] TK Attwood and A Gisel. Concepts, historical milestones and the central place of bioinformatics in modern biology : a European perspective. *Bioinformatics-Trends ...*, pages 3–39, 2011.

[8] F Baader and W Snyder. Unification Theory. In Andrei Robinson, Alan and Voronkov, editor, *Handbook of automated reasoning*, chapter 8, pages 441—-523. The MIT Press, 2001.

[9] Franz Baader and W Büttner. Unification in commutative idempotent monoids. *Theoretical Computer Science*, 56 :345–352, 1988.

[10] Adrien Basso-Blandin and Franck Delaplace. Gubs, a behavior-based language for open system dedicated to synthetic biology. *CoRR*, abs/1206.6098, 2012.

[11] Adrien Basso-Blandin and Franck Delaplace. Gubs, a behaviour-based language for design in synthetic biology. *Sci. Ann. Comp. Sci.*, 23(1) :1–38, 2013.

[12] Adrien Basso-Blandin and Franck Delaplace. GUBS a language for synthetic biology : Specification and compilation. In *Unconventional Computation and Natural Computation - 13th International Conference, UCNC 2014, London, ON, Canada, July 14-18, 2014, Proceedings*, pages 40–53, 2014.

[13] S. Basu, Y. Gerchman, C. H. Collins, F. H. Arnold, and R. Weiss. A Synthetic Multicellular System for Programmed Pattern Formation. *Nature*, 434(7037) :1130–4, April 2005.

[14] J. Beal and J. Bachrach. Infrastructure for Engineered Emergence on Sensor/Actuator Networks. *IEEE Intelligent Systems*, 21(2) :10–19, March 2006.

[15] J. Beal and J. Bachrach. Cells Are Plausible Targets for High-Level Spatial Languages. *2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, pages 284–291, October 2008.

[16] J. Beal, T. Lu, and R. Weiss. Automatic Compilation from High-Level Biologically-Oriented Programming Language to Genetic Regulatory Networks. *PLoS ONE*, 6(8) :e22490, August 2011.

[17] Steven A. Benner and A. Michael Sismour. Synthetic biology. *Nat Rev Genet*, 6 :533–543, July 2005.

[18] Gilles Bernot, Franck Cassez, Jean-Paul Comet, Franck Delaplace, Céline Müller, and Olivier Roux. Semantics of biological regulatory networks. *Electronic Notes in Theoretical Computer Science*, 180(3) :3 – 14, 2007.

[19] L. Bilitchenko, A. Liu, S. Cheung, E. Weeding, B. Xia, M. Leguia, J C. Anderson, and D. Densmore. Eugene–A Domain Specific Language for Specifying and Constraining Synthetic Biological Parts, Devices, and Systems. *PloS one*, 6(4) :e18882, January 2011.

[20] Lesia Bilitchenko, Adam Liu, and Douglas Densmore. The eugene language for synthetic biology. *Methods in enzymology*, 498 :153–72, January 2011.

[21] P. Blackburn, J. F. A. K. van Benthem, and F. Wolter. *Handbook of Modal Logic, Volume 3 (Studies in Logic and Practical Reasoning)*. Elsevier Science Inc., December 2006.

[22] F R Blattner, G Plunkett, C A Bloch, N T Perna, V Burland, M Riley, J Collado-Vides, J D Glasner, C K Rode, G F Mayhew, J Gregor, N W Davis, H A Kirkpatrick, M A Goeden, D J Rose, B Mau, and Y Shao. The complete genome sequence of Escherichia coli K-12. *Science (New York, N.Y.)*, 277(5331) :1453–62, September 1997.

[23] Philip Bourne. Will a biological database be different from a biological journal ? *PLoS Comput Biol*, 1(3) :e34, 08 2005.

[24] T. Braüner. *Hybrid Logic and Its Proof-Theory*. Springer, 2010.

[25] Y. Cai, M. W Lux, L. Adam, and J. Peccoud. Modeling Structure-function Relationships in Synthetic DNA Sequences Using Attribute Grammars. *PLoS Computational Biology*, 5(10), 2009.

[26] Yizhi Cai, Mandy L Wilson, and Jean Peccoud. GenoCAD for iGEM : a grammatical approach to the design of standard-compliant constructs. *Nucleic acids research*, 38(8) :2637–44, May 2010.

[27] Jane Calvert. Synthetic biology : constructing nature ? *The Sociological Review*, 58 :95–112, 2010.

[28] L. Calzone, F. Fages, and S. Soliman. BIOCHAM : An Environment for Modeling Biological Systems and Formalizing Experimental Knowledge. *Bioinformatics*, 22(14) :1805–7, July 2006.

[29] D. Ewen Cameron, Caleb J. Bashor, and James J. Collins. A brief history of synthetic biology. *Nat Rev Micro*, 10 :381–390, 2014.

[30] Barry Canton, Anna Labno, and Drew Endy. Refinement and standardization of synthetic biological parts and devices. *Nat Biotech*, 26 :787 – 793, 2008.

[31] Robert H. Carlson. *Biology Is Technology : The Promise, Peril, and New Business of Engineering Life.* Cambridge, MA : Harvard UP, 2010.

[32] French CE, de Mora K, Joshi N, and al. Synthetic biology and the art of biosensor design. *Institute of Medicine (US) Forum on Microbial Threats.*, 2011.

[33] S. Cerrito and M. C. Mayer. A tableaux based decision procedure for a broad class of hybrid formulae with binders. In *Proceedings of the 20th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, TABLEAUX'11, pages 104–118. Springer-Verlag, 2011.

[34] Deepak Chandran, Frank T Bergmann, and Herbert M Sauro. TinkerCell : modular CAD tool for synthetic biology. *Journal of biological engineering*, 3(1) :19, January 2009.

[35] Shuobing Chen, Haoqian Zhang, Handuo Shi, Weiyue Ji, Jingchen Feng, Yan Gong, Zhenglin Yang, and Qi Ouyang. Automated design of genetic toggle switches with predetermined bistability. *ACS Synthetic Biology*, 1(7) :284–290, 2012.

[36] Marta Cialdea Mayer and Serenella Cerrito. Herod and pilate : Two tableau provers for basic hybrid logic. In JÃ¼rgen Giesl and Reiner HÃ¤hnle, editors, *Automated Reasoning*, volume 6173 of *Lecture Notes in Computer Science*, pages 255–262. Springer Berlin Heidelberg, 2010.

[37] Olivier Cinquin, Jacques Demongeot, and Faculty Medicine. Positive and negative feedback : striking a balance between necessary antagonists. *Journal of theorical biology*, 2002.

[38] F. Ciocchetta and J. Hillston. Bio-PEPA : A Framework for the Modelling and Analysis of Biological Systems. *Theoretical Computer Science*, 410(33-34) :3065–3084, August 2009.

[39] K. Clancy and C. A Voigt. Programming Cells : Towards an Automated Genetic Compiler. *Current Opinion in Biotechnology*, 21(4) :581–572, August 2010.

[40] C. A. Coello. A comprehensive survey of evolutionary-based multiobjective optimization techniques. *Knowledge and Information systems*, 1(3) :129–156, 1999.

[41] M. J Czar, Y. Cai, and J. Peccoud. Writing DNA with GenoCAD. *Nucleic Acids Research*, 37 :W40–7, July 2009.

[42] Ro Dae-Kyun, Paradise Eric M., Ouellet Mario, Fisher Karl J., Newman Karyn L., Ndungu John M., Ho Kimberly A., Eachus Rachel A., Ham Timothy S., Kirby James, Chang Michelle C. Y., Withers Sydnor T., Shiba Yoichiro, Sarpong Richmond, and Keasling Jay D. Production of the antimalarial drug precursor artemisinic acid in engineered yeast. *Nature*, 440 :940–943, 2006.

[43] M. D'Agostino, D. M. Gabbay, R. Hähnle, and J. Posegga. *Handbook of Tableau Methods*. Springer Netherlands, Dordrecht, 1999.

[44] Antoine Danchin. Scaling up synthetic biology : Do not forget the chassis. *{FEBS} Letters*, 586(15) :2129 – 2137, 2012.

[45] V. Danos, J. Feret, W. Fontana, R. Harmer, and J. Krivine. Rule-based modelling of cellular signalling. In *CONCUR*, pages 17–41, 2007.

[46] Vincent Danos, Jérôme Feret, Walter Fontana, Russell Harmer, and Jean Krivine. Abstracting the differential semantics of rule-based models : exact and automated model reduction. In *Logic in Computer Science (LICS), 2010 25th Annual IEEE Symposium on*, pages 362–381. IEEE, 2010.

[47] Vincent Danos, Jérôme Feret, Walter Fontana, Russell Harmer, and Jean Krivine. Rule-based modelling, symmetries, refinements. In Jasmin Fisher, editor, *Formal Methods in Systems Biology*, volume 5054 of *Lecture Notes in Computer Science*, pages 103–122. Springer Berlin Heidelberg, 2008.

[48] A. Prasanna de Silva and Nathan D. McClenaghan. Molecular-scale logic gates. *Chemistry - A European Journal*, 10(3) :574–586, 2004.

[49] F. Delaplace, H. Klaudel, and A. Cartier-Michaud. Discrete Causal ModelView of Biological Networks. In *Proceedings of the 8th International Conference on Computational Methods in Systems Biology - CMSB '10*, pages 4–13, New York, New York, USA, September 2010. ACM Press.

[50] B. Drogue, P. Thomas, L. Balvay, C. Prigent-Combaret, and C. Dorel. Engineering adherent bacteria by creating a single synthetic curli operon. *Journal of visualized experiments*, 2012.

[51] Michael B Elowitz and Stanislas Leibler. A synthetic oscillatory network of transcriptional regulators. *Nature*, 403(6767) :335–338, 2000.

[52] D Endy. Foundations for engineering biology. *Nature*, 438 :449–453, 2005.

[53] F. Fages. Associative-commutative unification. *J. Symb. Comput.*, 3(3) :257–275, June 1987.

[54] Anthony C Forster and George M Church. Towards synthesis of a minimal cell. *Molecular systems biology*, 2 :45, January 2006.

[55] P. Francois and V. Hakim. Design of genetic networks with specified functions by evolution in silico. *PNAS*, 101(2) :580–585, 2004.

[56] Michal Galdzicki, Mandy Wilson, Cesar A Rodriguez, Matthew R Pocock, Ernst Oberortner, Laura Adam, Aaron Adler, J Christopher Anderson, Jacob Beal, Yizhi Cai, et al. Synthetic biology open language (sbol) version 1.1. 0. 2012.

[57] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[58] A. Ghosh and S. Dehuri. Evolutionary algorithms for multi-criterion optimization : A survey. *International Journal of Computing & Information Sciences*, 2(1) :38–57, 2004.

[59] Jean-Louis Giavitto and Olivier Michel. Mgs : a rule-based programming language for complex objects and collections. In Mark van den Brand and Rakesh Verma, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier Science Publishers, 2001.

[60] J.L. Giavitto, O. Michel, J. Cohen, and A. Spicher. Computations in space and space in computations. In *Unconventional Programming Paradigms*, volume 3566 of *Lecture Notes in Computer Science*, pages 97–97. Springer Berlin / Heidelberg, 2005.

[61] D.G. Gibson, J.I. Glass, C. Lartigue, V.N. Noskov, R.Y. Chuang, M.A. Algire, G.A. Benders, M.G. Montague, L. Ma, M.M. Moodie, and al. Creation of a Bacterial Cell Controlled by a Chemically Synthesized Genome. *Science*, 329(5987) :52, May 2010.

[62] Cinzia Di Giusto, Hanna Klaudel, and Franck Delaplace. Systemic approach for toxicity analysis. *BioPPN*, 1159, 2014.

[63] Daniel Götzmann, Mark Kaminski, and Gert Smolka. Spartacus : A tableau prover for hybrid logic. *Electronic Notes in Theoretical Computer Science*, 262(0) :127 – 139, 2010.

[64] Matthijn C. Hesselman, Dorett I. Odoni, Brendan M. Ryback, Suzette de Groot, Ruben G. A. van Heck, Jaap Keijsers, Pim Kolkman, David Nieuwenhuijse, Youri M. van Nuland, Erik Sebus, Rob Spee, Hugo de Vries, Marten T. Wapenaar, Colin J. Ingham, Karin Schroën, Vítor A. P. Martins dos Santos, Sebastiaan K. Spaans, Floor Hugenholtz, and Mark W. J. van Passel. A multiplatform flow device for microbial (co-) cultivation and microscopic analysis. *PLoS ONE*, 7(5) :e36982, 05 2012.

[65] Anthony D. Hill, Jonathan R. Tomshine, Emma M. B. Weeding, Vassilios Sotiropoulos, and Yiannis N. Kaznessis. Synbioss : the synthetic biology modeling suite. *Bioinformatics/computer Applications in The Biosciences*, 24 :2551–2553, 2008.

[66] C. Eric Hodgman and Michael C. Jewett. Cell-free synthetic biology : Thinking outside the cell. *Metabolic Engineering*, 14 :261–269, 2012.

[67] Guillaume Hoffmann and Carlos Areces. Htab : a terminating tableaux system for hybrid logic. *Electronic Notes in Theoretical Computer Science*, 231(0) :3 – 19, 2009.

[68] M. Hucka. The systems biology markup language (SBML) : a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4) :524–531, March 2003.

[69] D. Hume. *A Treatise of Human Nature, Being an Attempt to Introduce the Experimental Method of Reasoning into Moral Subjects*. 1739.

[70] Michael C Jewett, Kara A Calhoun, Alexei Voloshin, Jessica J Wuu, and James R Swartz. An integrated cell-free metabolic platform for protein production and synthetic biology. *Molecular systems biology*, 4 :220, January 2008.

[71] Mark Kaminski, Sigurd Schneider, and Gert Smolka. Terminating tableaux for graded hybrid logic with global modalities and role hierarchies. In Martin Giese and Arild Waaler, editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 5607 of *Lecture Notes in Computer Science*, pages 235–249. Springer Berlin Heidelberg, 2009.

[72] D. Kapur and P. Narendran. NP-completeness of the set unification and matching problems. In *8th International Conference On Automated Deduction*, 1986.

[73] Kevin Knight. Unification : a multidisciplinary survey. *ACM Computing Surveys*, 21(1) :93–124, March 1989.

[74] T.F. Knight. Idempotent vector design for standard assembly of biobricks. Technical report, MIT Synthetic Biology Working Group Technical Reports, 2003.

[75] Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16(1963) :83–94, 1963.

[76] Jean Krivine, Vincent Danos, and Arndt Benecke. Modelling epigenetic information maintenance : A kappa tutorial. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 17–32. Springer Berlin Heidelberg, 2009.

[77] D. Lewis. Causation as Influence. *The Journal of Philosophy*, 97(4) :182–197, 2000.

[78] T. K Lu, A. S Khalil, and J. J Collins. Next-generation Synthetic Gene Networks. *Nature Biotechnology*, 27(12) :1139—-1150, 2009.

[79] Pier Luigi Luisi, Francesca Ferri, and Pasquale Stano. Approaches to semi-synthetic minimal cells : a review. *Die Naturwissenschaften*, 93(1) :1–13, January 2006.

[80] Frédéric Michard. Looking at transpulmonary thermodilution curves : The cross-talk phenomenon. *Chest*, 126(2) :656–657, 2004.

[81] Sebastian Mirschel, Katrin Steinmetz, Michael Rempel, Martin Ginkel, and Ernst Dieter Gilles. PROMOT : modular modeling for systems biology. *Bioinformatics (Oxford, England)*, 25(5) :687–9, March 2009.

[82] Michael G Montague, Carole Lartigue, and Sanjay Vashee. Synthetic genomics : potential and limitations. *Current Opinion in Biotechnology*, 23(5) :659 – 665, 2012.

[83] M. P. Pedersen. Towards Programming Languages for Genetic Engineering of Living Cells. *Journal of the Royal Society, Interface*, 6 Suppl 4 :S437–450, 2009.

[84] Michael Pedersen. *Modular languages for systems and synthetic biology.* PhD thesis, February 2010.

[85] C. Priami, A. Regev, E. Shapiro, and W. Silverman. Application of a Stochastic Name-passing Calculus to Representation and Simulation of Molecular Processes. *Information Processing Letters*, 80(1) :25–31, October 2001.

[86] P. E M Purnick and R. Weiss. The Second Wave of Synthetic Biology : From Modules to Systems. *Nature Reviews. Molecular Cell Biology*, 10(6) :410–22, 2009.

[87] Erik M. Quandt, Michael J. Hammerling, Ryan M. Summers, Peter B. Otoupal, Ben Slater, Razan N. Alnahhas, Aurko Dasgupta, James L. Bachman, Mani V. Subramanian, and Jeffrey E. Barrick. Decaffeination and measurement of caffeine content by addicted escherichia coli with a refactored n-demethylation operon from pseudomonas putida cbb5. *ACS Synthetic Biology*, 2(6) :301–307, 2013.

[88] S. Regot, J. Macia, N. Conde, K. Furukawa, J. Kjellén, T. Peeters, S. Hohmann, E. de Nadal, F. Posas, and R. Solé. Distributed Biological Computation with Multicellular Engineered Networks. *Nature*, 469(7329) :207–211, January 2011.

[89] G. Rodrigo, J. Carrera, T. E. Landrain, and A. Jaramillo. Perspectives on the automatic design of regulatory systems for synthetic biology. *FEBS letters*, 586(15) :2037–42, July 2012.

[90] G. Rodrigo and A. Jaramillo. AutoBioCAD : full biodesign automation of genetic circuits. *ACS synthetic biology*, 2(5) :230–6, May 2013.

[91] JJ Romm. Five hards truths for the Synthetic Biology. *Nature*, 463(January), 2004.

[92] J.L. Rossignol. *Génétique.* Abrégés (Paris. 1971). Masson, 1992.

[93] Alexander K Rowe, Samantha Y Rowe, Robert W Snow, Eline L Korenromp, Joanna RM Armstrong Schellenberg, Claudia Stein, Bernard L Nahlen, Jennifer Bryce, Robert E Black, and Richard W Steketee. The burden of malaria mortality among african children in the year 2000. *International Journal of Epidemiology*, 35(3) :691–704, 2006.

[94] Elizabeth Royston, Ayan Ghosh, Peter Kofinas, Michael T. Harris, and James N. Culver. Self-assembly of virus-structured high surface area nanomaterials and their application as battery electrodes. *Langmuir*, 24(3) :906–912, 2008.

[95] M. Ryadnov, L. Brunsveld, and H. Suga. *Synthetic Biology :*. SPR - Synthetic Biology. Royal Society of Chemistry, 2014.

[96] H. M. Salis, E. Mirsky, and C. Voigt. Automated design of synthetic ribosome binding sites to control protein expression. *Nature biotechnology*, 27(10) :946–50, 2009.

[97] David F. Savage, Jeffrey Way, and Pamela A. Silver. Defossiling fuel : How synthetic biology can transform biofuel production. *ACS Chemical Biology*, 3(1) :13–16, 2008.

[98] Michael R. Schlabach, Jimmy K. Hu, Mamie Li, and Stephen J. Elledge. Synthetic design of strong promoters. *Proceedings of the National Academy of Sciences*, 107(6) :2538–2543, 2010.

[99] Markus Schmidt. Diffusion of synthetic biology : a challenge to biosafety. *Systems and Synthetic Biology*, 2(1-2) :1–6, 2008.

[100] Reshma Shetty, Drew Endy, and Thomas Knight. Engineering BioBrick vectors from BioBrick parts. *Journal of Biological Engineering*, 2 :5+, 2008.

[101] Stefan Sjoholm and Lennart Lindh. *VHDL for Designers*. January 1997.

[102] Lucian P Smith, Frank T Bergmann, Deepak Chandran, and Herbert M Sauro. Antimony : a modular model definition language. *Bioinformatics (Oxford, England)*, 25(18) :2452–4, September 2009.

[103] M.E. Stickel. A unification algorithm for associative-commutative functions. *Journal of the ACM*, 28(3) :423–434, 1981.

[104] Judith Stocker, Denisa Balluch, Monika Gsell, Hauke Harms, Jessika Feliciano, Sylvia Daunert, Khurseed A. Malik, and Jan Roelof van der Meer. Development of a set of simple bacterial biosensors for quantitative and rapid measurements of arsenite and arsenate in potable water. *Environmental Science & Technology*, 37(20) :4743–4750, 2003.

[105] F. Stolzenburg. An algorithm for general set unification and its complexity. *Journal of Automated Reasoning*, 22(1) :45–63, 1999.

[106] Denis Thieffry and RenÃ© Thomas. Dynamical behaviour of biological regulatory networks–ii. immunity control in bacteriophage lambda. *Bulletin of Mathematical Biology*, 57(2) :277–297, 1995.

[107] D. E. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1998.

[108] Donald E. Thomas and Philip R. Moorby. *The Verilog hardware description language, Volume 1.* Springer, 2002.

[109] René Thomas. Regulation of gene expression in bacteriophage lambda. In W. Arber, W. Braun, R. Haas, W. Henle, P.H. Hofschneider, N.K. Jerne, P. Koldovský, H. Koprowski, O. Maaløe, R. Rott, H.G. Schweiger, M. Sela, L. Svruček, P.K. Vogt, and E. Wecker, editors, *Current Topics in Microbiology and Immunology / Ergebnisse der Mikrobiologie und Immunitätsforschung*, volume 56 of *Current Topics in Microbiology and Immunology / Ergebnisse der Mikrobiologie und Immunitätsforschung*, pages 13–42. Springer Berlin Heidelberg, 1971.

[110] P. Umesh, F. Naveen, C.U.M. Rao, and S.A. Nair. Programming languages for synthetic biology. *Systems and Synthetic Biology*, 4(4) :265–269, 2010.

[111] Marc Valls and Victor de Lorenzo. Exploiting the genetic and biochemical capacities of bacteria for the remediation of heavy metal pollution. *FEMS Microbiology Reviews*, 26(4) :327–338, 2002.

[112] Valda Vinson and Elizabeth Pennisi. The allure of synthetic biology. *Science*, 333(6047) :1235, 2011.

[113] Baojun Wang, Richard I Kitney, Nicolas Joly, and Martin Buck. Engineering modular and orthogonal genetic logic gates for robust digital-like synthetic biology. *Nature Commun*, 2 :508, 2011.

[114] Wilfried Weber and Martin Fussenegger. The impact of synthetic biology on drug discovery. *Drug Discovery Today*, 14(19-20) :956 – 963, 2009.

[115] Na Wei, Josh Quarterman, and Yong-Su Jin. Marine macroalgae : an untapped resource for producing fuels and chemicals. *Trends in Biotechnology*, 31(2) :70 – 77, 2013.

[116] L. L. Wolfenbarger and P. R. Phifer. The ecological risks and benefits of genetically engineered plants. *Science*, 290(5499) :2088–2093, 2000.

[117] Bing Xia, Swapnil Bhatia, Ben Bubenheim, Maisam Dadgar, Douglas Densmore, and J Christopher Anderson. Developer's and user's guide to Clotho v2.0 A software platform for the creation of synthetic biological systems. *Methods in enzymology*, 498 :97–135, January 2011.

[118] H. Ye, M. Daoud-El Baba, R-W. Peng, and M. Fussenegger. A Synthetic Optogenetic Transcription Device Enhances Blood-glucose Homeostasis in Mice. *Science*, 332(6037) :1565–1568, June 2011.

[119] A. Zhou, B.Y. Qu, H. Li, S.Z. Zhao, P. N. Suganthan, and Q. Zhang. Multiobjective evolutionary algorithms : A survey of the state of the art. *Swarm and Evolutionary Computation*, 1(1) :32–49, 2011.

Table des matières