T.E.R.
Maîtrise d'informatique 2001-2002

Hyperstructure et Modélisation de Chimie Artificielle dans le Langage MGS

Clément Boin Nicolas Thibault

8 août 2003

Encadré par Jean-Louis Giavitto & Olivier Michel

Table des matières

1 Introduction

1.1 Présentation de la chimie artificielle

La chimie artificielle a pour but de réaliser virtuellement des réactions chimiques grâce à l'utilisation de programmes réalisant une expérience numérique. Les modèles de la chimie artificielle relèvent plus de modèles discrets et combinatoires que de modèles continus. On s'intéressera ici à l'implémentation de ces réactions de manière informatique, en utilisant un langage nommé *MGS*. Nous nous pencherons sur les structures à utiliser pour modéliser ces expériences, et sur les interactions qu'il peut y avoir entre les objets en présence dans les réactions.

L'utilisation de la chimie artificielle a de multiples avantages par rapport à la chimie "classique". Une expérience peut être recommencée autant de fois qu'on le désire sans utiliser de matière première. Les résultats peuvent être étudiés et partagés très rapidement du fait qu'ils sont par nature déjà stockés numériquement. On peut aussi signaler que les incertitudes de mesures qui ont nécessairement lieu dans une expérience chimique sont considérablement réduites par l'utilisation d'un ordinateur.

Pour qu'une expérience de chimie artificielle fonctionne bien, il faut bien sûr qu'elle soit convenablement programmée, c'est à dire que la modélisation informatique soit la plus proche possible de ce qui se passe dans la réalité. C'est l'inconvénient majeur de la chimie artificielle, on ne peut pas savoir immédiatement si le résultat escompté par le programme sera exactement le même dans la vie réelle.

On peut définir une expérience de chimie artificielle grâce aux trois structures suivantes :

- *un ensemble d'objets mathématiques* : nombres, chaînes de caractères, lambda-termes ...(Les molécules) ;
- *des règles de transformation*, c'est à dire la manière dont réagissent les objets entre eux. (Les réactions) ;
- *des règles d'interaction dynamique*, pour prendre en compte les aspects dynamiques des molécules, comme les temps de diffusion, l'organisation spatiale, les collisions ...

Cette manière de voir la chimie artificielle n'est pas la seule, il existe des formes plus complexes de modélisation. Nous nous restreindrons ici à cette vision.

De nombreux liens sur la chimie artificielle sont disponibles sur Internet à cette adresse : `http://ls11-www.cs.uni-dortmund.de/achem`

1.2 *MGS* (Modèle Général de Simulation)

1.2.1 Systèmes dynamiques

Un des objectifs du projet *MGS* [1] est de développer un langage dédié à la modélisation et la simulation de processus biologiques à structure dynamique.

Un *système dynamique*[2] correspond à un phénomène qui évolue dans le temps. C'est un phénomène caractérisé par un certain nombre d'états observables, les variables du système. L'ensemble des valeurs des variables à un instant donné t, constitue l'état du système au temps t. Ces variables évoluent dans le temps, et sont reliés entre elles par des relations.

1.2.2 Présentation de *MGS*

Le langage *MGS* a été développé par MM. Jean-Louis Giavitto, Olivier Michel et Julien Cohen. Deux interprètes du langage sont disponibles, l'un programmé en OCAML, et l'autre en C++. Dans toute la suite du rapport, nous nous référerons aux résultats que nous a donné le compilateur réalisé en C++.

Les langages dédiés fournissent au programmeur des abstractions et des notations adaptées à un domaine d'application particulier. La simulation de processus biologiques à structure dynamique est particulièrement difficile car il est ardue de spécifier la structure et la dynamique de ce type de systèmes. On peut cependant remarquer que, bien souvent, les lois d'évolutions sont décrites informellement comme un ensemble de transformations locales agissant sur un ensemble organisé d'entités.

Les mécanismes de *MGS* s'inspirent initialement de la biologie, et des processus chimiques. Il va nous permettre de modéliser simplement des structures de chimie artificielle. Mais il est tout à fait envisageable d'utiliser *MGS* pour d'autres usages que la chimie artificielle.

MGS est un langage fonctionnel enrichit par des nouveaux types de valeurs (les collections topologiques) et par une nouvelle notion (les transformations) :

- *Les collections topologiques* permettent d'utiliser simplement des ensembles mathématiques, des séquences, et des multi-ensembles (un en-

semble qui prend en compte plusieurs occurrences d'un même élément), ainsi que des structures plus complexes.
- *Les transformations* sont des fonctions qui s'appliquent sur les éléments d'une collection. Une transformation fonctionne de la manière suivante : dans une collection A donnée, on remplace une sous-collection B vérifiant certaines propriétés, par un autre collection C. La collection C dépend uniquement de la collection B, et est adjacente en cas d'application de la transformation aux éléments de A. L'inclusion de C dans $A - B$ dépend de la forme des collections en présence.

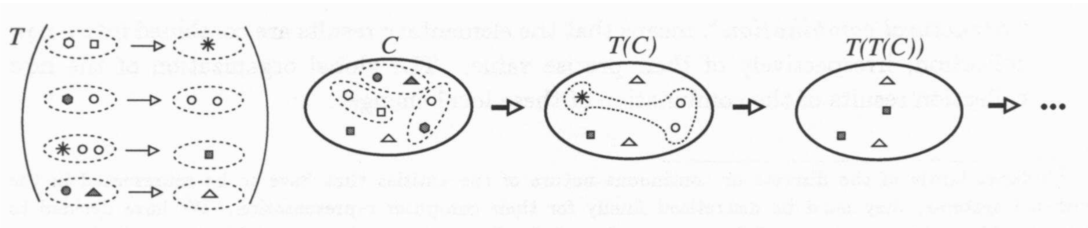


FIG. 1 – Transformation et itération d'une transformation.
Une transformation T est un ensemble de transformations élémentaires appliquées en parallèle et indépendamment. Les transformations élémentaires n'interagissent pas entre elles, c'est-à-dire qu'elles s'appliquent à des sous-collections différentes et sans intersection. Une transformation peut être itérée un nombre quelconque de fois, par exemple pour atteindre un point-fixe.

1.2.3 Les enregistrements

Dans la suite de l'exposé, nous allons rencontrer des données utilisant les enregsitrements.

Un *enregistrement MGS* est un type particulier de collection. Bien qu'étant une collection, « l'aspect collection » des enregistrements reste frustre et on peut tout aussi bien les considérer comme des valeurs atomiques.

Un enregistrement *MGS* est très similaire à un enregistrement Pascal, une *struct C* ou bien un *record CAML*. Un enregistrement *MGS* associe des valeurs avec des noms symboliques appelé *champ*. Les valeurs attachées aux champs peuvent être de n'importe quel type, y compris d'autres enregistrements ou bien d'autres collections.

Construction. Un enregistrement peut se construire à l'aide des accolades :

```
{a = 1,  b = "red"}
```

est une expression qui construit un enregistrement avec deux champs : un champ de nom *a* et de valeur *1* et un champ de nom *b* et de valeur, la chaîne *"red"*. A droite du signe égal qui suit un nom de champ, peut figurer une expression arbitraitrement complexe :

```
{a = 1+2,  b = f(g(3, "red")),  c = {x=0, y='math_PI}}
```

Accès. L'accès à la valeur d'un champ se fait grâce à la notation pointée :

```
{a = (1+2), ; b = "red"}.a
```

est une expression qui retourne la valeur du champ *a*, i.e. : *3*. L'accès à un champ inexistant renvoie la valeur <undef> sans déclencher d'erreur.

1.2.4 Programme, commande, définition et expression

Un programme *MGS* est une suite de demandes se terminant par un double point-virgule ; ; . Il y a trois type de demandes :

1. les expressions,
2. les définitions de types,
3. et les commandes,

Ces demandes peuvent être écrites directement au clavier, une fois qu'on a lancé un interprète (par exemple dans une fenêtre "xterm"). L'entré de l'interprète *MGS* est un éditeur-ligne. Cela veut dire que tant que l'utilisateur n'a pas frappé sur le "retour-chariot", il est possible d'éditer la ligne courante à l'aide de plusieurs commande de type "emacs". L'utilisateur dispose aussi d'un historique des lignes déjà entrées. La ligne courante est prise en compte et examinée par l'interprète uniquement après la frappe du retour-chariot. Cependant, le découpage en ligne est arbitraire et n'impose aucune contrainte à la syntaxe du langage.

Pour en savoir plus sur *MGS*, de nombreux liens sont disponibles à cette adresse : `http://www.lami.univ-evry.fr/mgs`

Nous allons présenter dans cet exposé deux types de modélisation d'expériences de chimie artificielle en MGS. Pour commencer voici un petit exemple pour mieux comprendre le fonctionnement général du langage.

1.3 *Exemple* : Nombres premiers

Cet exemple est tiré d'un document de Peter Dittrich [4]. Il s'agit de filtrer les nombres premiers sur la séquence des entiers de 2 à m. On peut voir ce problème comme une expérience de chimie artificielle :

- Les objets sont des entiers contenus dans un ensemble $S = \{2, 3, .., m\}$
- La transformation est définie de la manière suivante :

$$x + y \rightarrow \mathrm{x + z \ avec \ z} = \left\{ \begin{array}{ll} y/x & \text{si x mod y} = 0 \\ y & \text{sinon} \end{array} \right\}$$

- Un état du système correspond à "l'éprouvette" à un instant donné, et la fonction de transition d'un état à un autre état est l'application d'une règle de transformation.

Voici l'implémentation en *MGS* de ce problème :

```
//Générer la séquence des objets (les entiers de 2 à m)
fun objets(m) = if m==1 then ():set
                        else m::(objets(m-1)) fi;;


//Transformation
trans premier =
{
    r = x,y/(y%x == 0)&(x!=y) => x,(y/x)};;
};;
```

Ligne de commande et résultat donné par le compilateur :

```
mgs> premier['iter='fixpoint](objets(20));;
2, 3, 5, 7, 11, 13, 17, 19, ():set
```

La fonction *objets* génère une collection de type séquence des entiers de 2 à m. La transformation *premier* applique sur cette collection :

- la règle r, si les conditions à gauche de la flèche sont respectées,
- rien, si la transformation n'a pas fonctionné.

La transformation s'arrête lorsqu'il n'y a plus de transformations possibles, c'est à dire lorsqu'on atteint le point fixe. On utilise pour cela l'option *iter* avec la valeur *fixpoint*, qui applique les règles f de la transformation *premier* jusqu'à ce que l'on ait $x_{n+1} = f(x_n)$, où x est l'ensemble des entiers (objets).

Cet exemple nous montre que l'implémentation d'un modèle en *MGS* est très proche de sa définition formelle. Nous allons donc pouvoir traiter des problèmes chimiques plus complexes dans ce langage.

2 Modélisation d'un réseau auto-catalytique

2.1 Exposé chimique du problème

Le problème chimique général est de comprendre les principes permettant d'organiser et de sélectionner des réseaux auto-catalytique. Un réseau auto-catalytique est un ensemble de réactions entre polymères (chaînes de monomères) de la forme suivante :

$$M1 + M2 \underset{\texttt{Kr}}{\overset{\texttt{Kf}}{\rightleftharpoons}} M12$$

Kf et Kr sont les coefficients de probabilité associés à la réussite de chaque sens de la la réaction.Quand une réaction est possible :

- soit elle se fait totalement (conversion de l'intégralité des produits M1 et M2 en M12)
- soit une fraction seulement des produits est convertie (par exemple une fraction proportionnelle à Kf/Kr ou un coefficient α arbitraire)

Ce que l'on voudrait pouvoir étudier expérimentalement avec l'aide d'un simulateur, c'est quels sont les polymères qui apparaissent "en grande quantité" et de façon stable quand on perturbe le réseau, c'est à dire, par exemple, en arrêtant de fournir certains produits, en interdisant certaines réactions, en rajoutant certaines équations ...

2.2 Idée générale de modélisation

La faculté des sciences et techniques de Rouen travaille actuellement avec un simulateur basé sur la transformation des quantités de produits présents[5]. Cette architecture est à opposer avec l'approche que nous allons envisager, fondée sur la représentation individuelle de chaque molécule. Cette stratégie est appelée “individual based modeling” en anglais. Elle est plus souple que la modélisation de la variation de quantité scalaire car elle permet de modéliser des phénomènes ponctuels (par exemple le positionnement dans l'espace) et d'envisager les réactions de manière plus concrète : les molécules se rencontrent et réagissent. Mais cette modélisation a l'inconvénient de ne pas permettre de traiter les immenses quantités de molécules présentes dans une éprouvette.

L'idée générale est de modéliser une réaction entre deux polymères par la réécriture d'un multi-ensemble de sequences d'entiers.

Nous allons donc étudier ce type de réactions en générant un réseau autocatalytique au hasard, avec des polymères dénotés par des entiers (chaque chiffre représente un monomère, un polymère sera donc représenté par un nombre). Notre modélisation, de manière simplifiée, aura par exemple l'aspect suivant :

$$
\begin{aligned}
1+2 &\rightarrow 12 \\
1+2 &\leftarrow 12 \\
\\
12+34 &\rightarrow 1234 \\
12+34 &\leftarrow 1234
\end{aligned}
$$

12 représente le polymère composé des monomères 1 et 2.

Il faut ajouter à cela le fait que certaines réactions ne sont possibles qu'en présence d'un polymère catalyseur. Par exemple :

$$1+2 \quad \stackrel{14}{\rightarrow} \quad 12$$

indique que la réaction $1+2 \rightarrow 12$ n'est possible qu'en présence du polymère 14.

Pour mettre en oeuvre notre modélisation, il nous faudra donc résoudre les problèmes posés par :
- la représentation d'un monomère,
- la représentation d'un polymère,
- la représentation d'une réaction,

- les coefficients de réussite de chaque réaction,
- la présence ou absence d'un catalyseur pour chaque réaction,
- la génération aléatoire de tous les éléments chimiques précités,
- la structure du réseau autocatalytique (de l'ensemble des réactions),
- l'alimentation en nouveaux polymères au cours de l'expérience,
- la représentation de la solution chimique courante (le contenu de l'éprouvette à l'instant t).

2.3 Stratégie d'implémentation en *MGS*

2.3.1 Représentation des polymères

Un polymère est représenté par une chaîne d'entiers de 0 à 9. Mais pour mettre en forme une telle structure, deux options se sont présentées : représenter ladite chaîne par une séquence ou bien par un multi-ensemble. Notre premier choix a été de choisir le multi-ensemble pour la raison suivante : lorsque deux produits se rencontrent, quel que soit la manière dont ils s'approchent l'un de l'autre, ils vont donner le même produit résultat. Or si on implémente les polymères comme des séquences, la réaction $1 + 2 \to 12$ donne la concaténation des séquences 1 et 2 soit la séquence 12. Mais si, lors de l'exécution, on rencontre d'abord la séquence 2, puis la séquence 1, soit nous choisissons dans notre implémentation de refuser la transformation, soit nous choisissons de concaténer malgré tout les deux séquences (2 et 1) pour obtenir la séquence 21. Mais aucune de ces deux solutions n'est chimiquement satisfaisante : la première implique que la réaction ne peut avoir lieu que si 1 rencontre 2, et non l'inverse (alors que l'ordre ne devrait pas avoir d'importance). Quant à la deuxième, elle implique qu'une même réaction chimique peut donner deux produits différents (12 et 21 ne sont pas des séquences équivalentes).

Ces observations nous ont amené à implémenter une première version où un polymère est un multi-ensemble, dans le but de contourner ces premières difficultés. En effet, l'ordre n'ayant pas d'importance dans un multi-ensemble,le produit résultat de (1, 3) :bag et (1, 2) :bag sera le multi-ensemble (1, 1, 2, 3) :bag. Le problème de l'ordre est résolu, mais un autre problème se pose : en chimie, la chaîne 12 n'étant pas la même que la chaîne 21, il est un peu réducteur de les représenter tous deux par le même multi-ensemble. C'est pour cette raison que nous nous sommes à nouveau penchés sur une modélisation avec des séquences, chimiquement plus juste.

Notre deuxième modélisation, avec séquences, résout les problèmes d'ordre de la manière suivante : une réaction, pour être entièrement spécifiée, ne considérera plus uniquement les produits qu'elle prend au départ, mais également l'ordre dans lequel elle devra les concaténer. Si 1 et 2 donnent 21, alors 2 et 1 donnent aussi 21 : l'ordre de rencontre entre polymères n'est donc plus un problème. De plus, la modélisation en séquence tient aussi compte de l'ordre, donc est chimiquement plus satisfaisante ($1 + 2 \rightarrow 12$ représente la même réaction que $2 + 1 \rightarrow 12$, mais ne représente pas la même réaction que $1 + 2 \rightarrow 21$).

2.3.2 Création des produits et réactions de manière aléatoire

Un certains nombre de structures ont des éléments générés aléatoirement suivant certaines règles :

- Un polymère est une chaîne de caractères dont les caractères sont des entiers de 0 à n, où n appartient à 0..9, générés aléatoirement avec la fonction *random* de *MGS*, répartis de manière équiprobable entre 0 et n.

- La séquence des polymère de départs (*elt_ init*) est générée à partir de polymères créées aléatoirement comme expliqué ci-dessus.

- La séquence des polymères qu'il est possible de créer par réaction de concaténation, ou sans concaténation à partir des polymères de départs (*elt_ init*) est générée à partir de *elt_ init* . Par exemple, si elt_init = (12, 34), alors on aura : elt_poss=(12, 34, 1234, 3412).

- La séquence des réactions (*reaction*) est générée à partir de réactions créées aléatoirement comme expliqué ci-dessous.

- Une réaction est représentée par le type enregistrement suivant :
  `state Reaction = {poly1, poly2, poly_res, kf, kr, cata};;`

  où :
  - *poly1* et*poly2* sont les deux produits de départ, tirés aléatoirement dans *elt_ init.*

  - *poly_ res* le produit résultat, concaténation de*poly1* et *poly2*.

  - *kf* le coefficient correspondant au taux de réussite de la réaction dans le sens $poly1 + poly2 \rightarrow poly\_res$, *kr* au taux de réussite la réaction dans le sens inverse $poly\_res \rightarrow poly1 + poly2$. Tout deux sont des pourcentages aléatoires tirées entre 0 et 100 de manière équiprobable au moment de l'exécution.

- *cata* qui correspond au catalyseur de la réaction (dans les deux sens), tiré aléatoirement dans *elt_poss*.

Il est important de remarquer que stocker *poly_res* dans un élément de type réaction n'est pas une redondance d'information mais permet de fixer un fois pour toute l'ordre dans lequel se fera la concaténation entre produits lors de l'exécution.

2.3.3 Exécution des réactions

Nous travaillons avec une solution de produits représentée par un multi-ensemble et initialisé à sol := "" :: () : *solution* (la chaîne de caractère nulle représente le catalyseur par défaut, c'est à dire l'absence de catalyseur. Cette chaîne de caractère doit toujours être présente dans la solution car lorsqu'une réaction n'a pas besoin de catalyseur pour être effectuée, son champs *cata* contient la chaîne de caractère nulle dont la présence va donc être nécessaire au moment de l'exécution pour que la réaction puisse se faire).

Le principe général d'exécution pour un pas d'évolution est d'itérer une transformation générique plusieurs fois où les différents paramètres non fixés prendrons successivement comme valeurs les différents champs des éléments de type *Reaction* stockés dans la séquence*reaction* . Ainsi, toutes les réactions générées aléatoirement seront exécutées sur la base de la transformation matricielle suivante :

```
fun nieme(n,liste) = if (n==1) then hd(liste)
                               else nieme(n-1,tl(liste))fi;;

fun condition(x) = let r=random(101) in (r<x);;

trans Biochimie =

{reaction_a = a, b, c /((a==nieme(i,reaction).poly1)
 &(b==nieme(i,reaction).poly2)&(c==nieme(i,reaction).cata))
              => if condition(nieme(i,reaction).kf)
                    then (nieme(i,reaction).poly_res)::c::():solution
                    else a::b::c::():solution fi;
```

```
reaction_b = a, c /((a==nieme(i,reaction).poly_res)
&(c==nieme(i,reaction).cata))
           => if condition(nieme(i,reaction).kr)
                then (nieme(i,reaction).poly1)::(nieme(i,reaction).poly2)
                     ::c::():solution
                else  a::c::():solution  fi;};;
```

Nous ne détaillons pas à nouveau le principe d'exécution d'une transformation MGS car cela a déjà été fait dans le paragraphe 3.2.2, mais nous nous contenterons d'expliquer les éléments ajoutées dans cette nouvelle transformation. La fonction nieme(i,reaction) renvoie le i^{me} élément de type Reaction stocké dans la liste des réactions et qui spécifie la i^{me} réaction. Il s'agit donc en fait de tester a, b, et c pour voir s'ils correspondent aux produits de la i^{me} réaction, et cela en incrémentant i a chaque nouvel appel de la transformation, permettant donc un parcourt de toutes les réactions stockées. La fonction iter() en question est la suivante :

```
fun iter() = if (i>NBREACTION) then sol
                               else sol:=Biochimie(sol);i:=i+1;iter() fi;;
```

L'utilisateur du programme peut ensuite appliquer le nombre d'évolution qu'il désire à la solution grâce à la fonction *evol* suivante :

```
fun evol (n) = if (n==0) then sol
                         else i:=1;
                              sol:=alea(elt_poss)::sol;
                              iter();
                              evol(n-1) fi;;
```

La fonction evol itère l'évolution de la solution n fois en circuit ouvert, c'est à dire qu'à chaque itération, un élément extérieur est ajouté à partir de la "réserve" de polymère *elt_poss*. Nous pouvons donc suivre l'état de la réaction au bout de n étapes d'évolution, qui dépendra de :

- l'état de la réaction à l'étape n-1
- l'arrivée aléatoire d'un nouveau produit en provenance de *elt_poss*
- l'application de l'étape n proprement dite, c'est à dire l'exécution sur la solution courante de l'ensemble des réactions.

2.3.4 Index des fonctions visibles à l'utilisateur

init() ne prend aucun argument et initialise les différents éléments, c'est à dire met la chaîne de caractère vide dans la solution courante, et génère les ensembles "éléments de départ", "éléments possibles" et "réactions" de manière aléatoire en suivant les règles expliquées au paragraphe 2.3.2.

evol(n) fait évoluer de n pas la solution courante à partir de l'état dans laquelle elle se trouve, sans initialiser celle-ci préalablement.

evol_init(n) fait évoluer de n pas la solution courante en initialisant préalablement la solution courante. Afin de mieux comprendre la différence d'utilisation entre *evol(n)* et *evol_init(n)*, voici deux exemples :

```
// exemple n°1 :              est équivalent à :

   init();;                   init();;
   evol(10);;                 evol(100);;
                              evol_init(10);;

// exemple n°2 :              est équivalent à :

   init();;                   init();;
   evol(10);;                 evol_init(20);;
   evol(10);;
```

compte() compte pour chaque élément son nombre d'occurrences dans la solution courante en renvoyant la séquence des couples (poly, nombre) où poly est le polynôme concerné et nombre l'entier qui représente sont nombre d'apparitions dans la solution courante.

dominant() rend l'élément qui apparaît le plus fréquemment dans la solution courante (si plusieurs éléments sont dans ce cas, la fonction renvoie de manière arbitraire le premier qu'elle rencontre)

rm() supprime l'élément dominant de la liste des "éléments possibles" (ici, dominant est utilisé dans le même sens que celui de la fonction *dominant()*). Nous avons choisi de ne pas spécifier l'élément à supprimer car se qui intéresse l'utilisateur chimiste, c'est de supprimer l'élément dominant des éléments d'entrée pour voir dans qu'elle mesure la stabilité de la solution s'en trouve affectée.

trace() génère un fichier *trace.tab* correspondant à la représentation graphique des quantités des différents éléments de la solution et un fichier *trace.leg* correspondant à la légende nécessaire pour lire le graphe tracé à partir du fichier *trace.tab*.

2.4 Exemple d'utilisation et interprétation des résultats

2.4.1 Exemple n°1

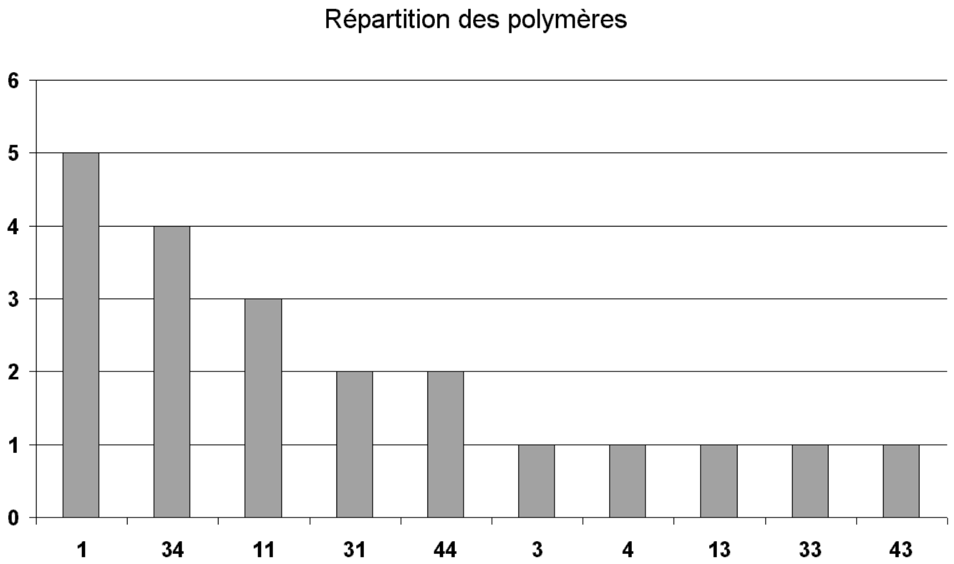
Après une première initialisation, avec la paramètres globaux suivants :

```
NBMONOMERE:=4;;   // les monomères sont {1,2,3,4}
TMAX:=2;;         // taille maximum d'un polymère
NBPOLY:=3;;       // taille de la liste des polymères
NBREACTION:=3;;   // taille de la liste des réactions
TOCATA:=10;;      // probabilité (en pourcentage)
                     de générer un catalyseur
```

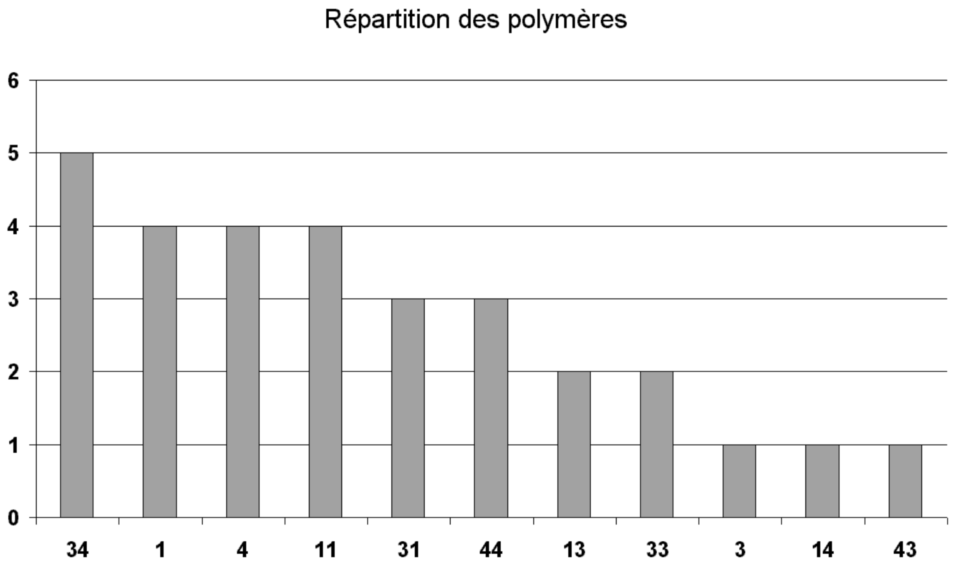
on obtient, de manière aléatoire à l'aide de la fonction *init()* :

- polymères initiaux : 1,3,4
- liste des réactions :
  - {cata=, kf=47, kr=7, poly1=3, poly2=3, poly_res=33}
  - {cata=, kf=4, kr=16, poly1=1, poly2=3, poly_res=13}
  - {cata=, kf=14, kr=89, poly1=4, poly2=1, poly_res=41}

Nous appliquons 20 pas d'évolution (par exemple) à la solution initialement vide, pour obtenir la solution suivante :



Le polymère dominant étant 1, avec 5 occurrences, nous le supprimons de la liste des polymères susceptibles d'alimenter la solution (avec la fonction *rm()*. Puis nous faisons à nouveau évoluer la solution (10 pas supplémentaires) pour voir dans quelle mesure cette modification affecte l'équilibre du système :



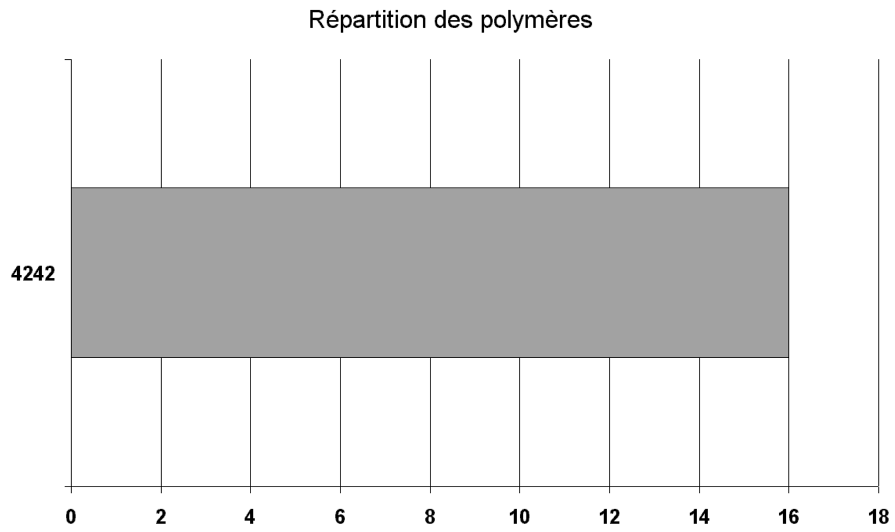
Le polymère dominant n'est plus 1 mais 34 (avec 5 occurrences pour 1 contre 4 pour 34). La hiérarchie entre concentrations des produits en présence a donc été modifiée, mais de manière peu significative. Cet exemple nous montre le bon fonctionnement de notre programme mais n'est pas suffisamment significatif pour être intéressant chimiquement. Nous allons donc étudier un second exemple plus particulier qui lui nous permettra de tirer des conclusions chimiques.

2.4.2 Exemple n°2

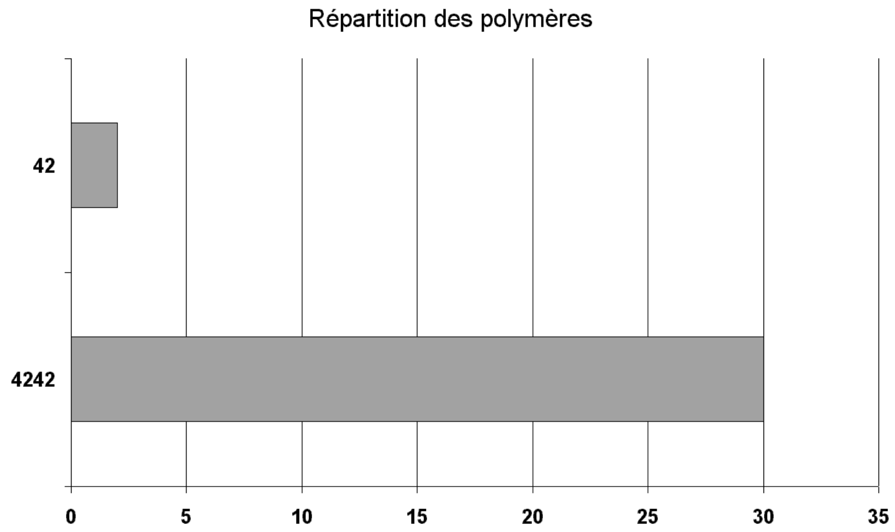
Les paramètres globaux sont les mêmes que pour l'exemple précédent sauf le nombre des polynômes différents qui est désormais de 2 ainsi que le nombre de réactions, également de 2. On génère alors de manière aléatoire les ensembles suivants :

- polymères initiaux : 42, 42
- liste des réactions :
  - {cata=, kf=42, kr=67, poly1=42, poly2=42, poly_res=4242}
  - {cata=, kf=62, kr=78, poly1=42, poly2=42, poly_res=4242}

Nous appliquons 20 pas d'évolution à la solution initialement vide, pour obtenir la solution suivante :



Nous obtenons alors un résultat très marqué : la solution n'est composée qu'avec le polymère 4242. Nous supprimons donc 4242 de la liste des polymères susceptibles d'alimenter la solution. Puis nous faisons à nouveau évoluer la solution (30 pas supplémentaires) pour voir dans quelle mesure cette modification affecte l'équilibre du système :



Nous pouvons observer ce résultat à priori étonnant : la solution est composée de trente polymères 4242 et uniquement de deux 42, alors que nous n'alimentons plus la solution avec 4242, mais uniquement avec 42 (les deux éléments possibles étant auparavant 42 et 4242). Cela nous renseigne donc de manière très précise sur ce réseau de réactions de réactions chimiques : le fort taux de 4242 ne s'explique pas par un ajout du polymère mais par un fort taux d'application de la réaction constructive $42 + 42 \rightarrow 4242$ et/ou un faible taux d'application de la réaction destructive $4242 \rightarrow 42 + 42$. Voilà donc un exemple (Certes, élémentaire) où l'utilisation du programme nous renseigne sur certaines propriétés chimiques du réseau autocatalytique (l'origine de la domination du polymère 4242).

2.4.3 Limites de notre modélisation

Notre modélisation à un défaut très prononcé au niveau de la masse d'informations que nous pouvons traiter : celle-ci doit être faible si on veut pouvoir exécuter en un temps raisonnable le programme (le deuxième exemple a nécessité une trentaine de secondes pour 50 pas d'évolution avec seulement 2 sortes de polymères possibles et 2 réactions à exécuter à chaque pas). Il y a deux raisons à cette lenteur :

- Le choix délibéré de modéliser les molécules individuellement, et non par des concentrations (cf paragraphe 2.2), augmente considérablement le nombre des données à traiter.
- Le principe d'évaluation des différentes réactions en MGS sur un multi-ensemble est de tester un nombre trouver couples pouvant satisfaire les conditions d'applications de la réaction (cf paragraphe 3.2.2). Étant

donné que nous travaillons en circuit chimique ouvert, nous ajoutons à chaque pas d'évolution un polymère, donc au bout de n pas d'évolution, l'interprète *MGS* aura à prendre en compte tous les couples possibles parmi n éléments, ce qui devient vite très lourd à gérer.

3 Modélisation d'un réseau biochimique

À présent, nous allons traiter une expérience différente de la chimie artificielle, toujours avec *MGS*. Il s'agit de modéliser un réseau biochimique.

3.1 Expérience de *A. E. Bugrim*

Pour étudier cette expérience nous nous sommes référés à un article de *A. E. Bugrim*[6]. Nous n'avons pas traité l'expérience entièrement, nous nous sommes focalisés sur des points bien précis pour tenter d'obtenir en fin de compte un comportement informatique proche de ce que pourrait être celui d'une cellule vivante telle qu'elle est décrite dans cet article.

On cherche à réaliser une modélisation informatique d'un processus de signalisation. Il s'agit d'un ensemble de réactions qui s'effectuent à l'intérieur d'une cellule vivante. Le but est d'activer depuis l'extérieure de la cellule une protéine à l'intérieure de celle-ci. La réaction débute avec l'activation de deux stimuli qui déclenchent une chaîne de réactions à travers la cellule. La réaction est détaillée sur les figures 2 et 3.

3.1.1 Les molécules

Les objets de la réaction sont ici des molécules. Voici les différents types de molécules qui interviennent dans la réaction :

- Les stimuli : Agonist et Stimulus. Ce sont des agents extérieurs à la cellule qui activent les récepteurs membranaires.
- Les récepteurs membranaires : Un récepteur adrégénique (Rec) et un canal (Channel). Un récepteur s'active à la réception du signal Agonist. Le canal s'ouvre par l'action du Stimulus.
- Les protéines : G-Protein, AC, Ca.PhK. Ces protéines ont deux états possibles : actif et inactif.
- Les corps chimiques transformables : ils réagissent selon les réactions suivantes :

$$\begin{aligned} ATP &\rightarrow cAMP \\ cAMP + R_2C_2 &\rightarrow C + (cAMP)_2.R \\ C + I &\rightarrow CI \\ Ca^{2+} + PhK &\rightarrow Ca.PhK \end{aligned}$$

3.1.2 Les réactions

Voir Figures 2 et 3.

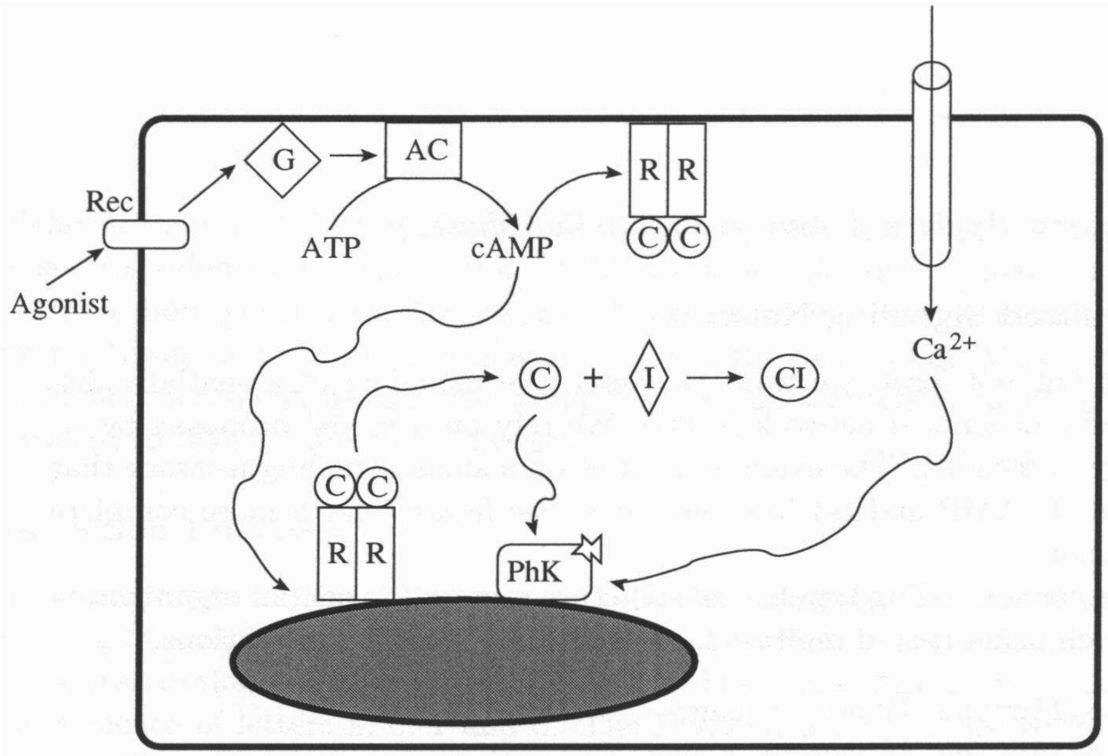


FIG. 2 – Schéma logique pour l'analyse d'un réseau chimique dans l'espace (Expérience de A.E. Bugrim)

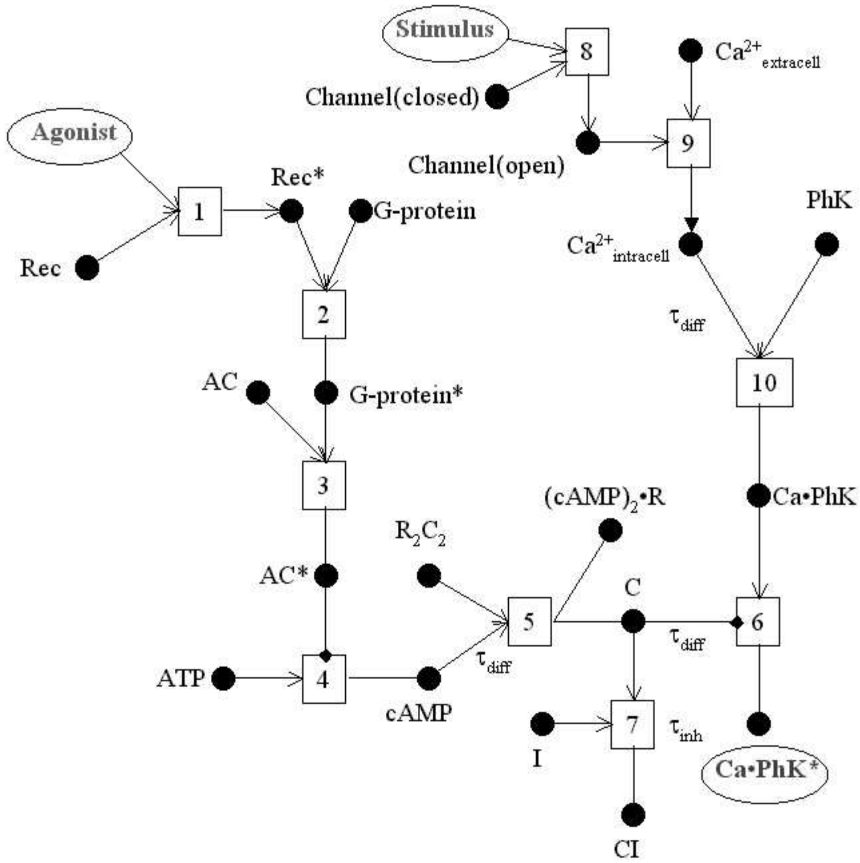


FIG. 3 – Réseau d'intéractions de l'expérience de Bugrim (les actions ne sont pas localisées dans l'espace)

3.1.3 Structure générale de la cellule

Une cellule est un ensemble de 4 parties imbriquées. Le reticulum endoplasmique (membrane du noyau) est contenu dans le cytosol (volume interne de la cellule). Celui-ci est entouré par le plasma (membrane extérieure de la cellule). L'ensemble est inclus dans un univers chimique. Voir représentation schématique sur figure 4.

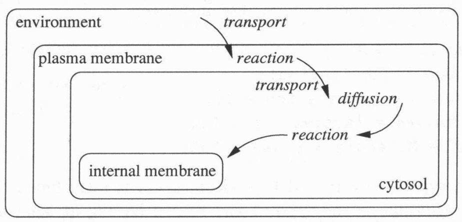


FIG. 4 – Représentation des imbrications de la cellule. *environnement* correspond à l'univers, et *internal membrane* au reticulum.

3.1.4 Les aspects dynamiques de la réaction

Certaines réactions ne s'effectuent pas immédiatement,car des facteurs biologiques complexes ralentissent le processus (par exemple : forme, taille de la cellule, température ...). *Bugrim* décrit son expérience en simplifiant ces facteurs, à l'aide d'un temps de diffusion. Il correspond à la durée du trajet de la molécule pour atteindre sa cible. Pour la réaction d'inhibition $C + I \to CI$, *Bugrim* utilise un taux d'inhibition, correspondant au pourcentage de C inhibé. Il s'agit à nouveau d'une simplification.

3.2 Modélisation de l'expérience en *MGS*

3.2.1 Les molécules

Chaque molécule est représentée par un enregistrement. Celui-ci est caractérisé par un champs *nom* commun à toutes les molécules. C'est une chaîne de caractère. Chaque élément créé représente une unique molécule chimique (pour représenter une quantité plus forte de molécule, on utilisera plusieurs occurrences de celle-ci, et non et champs concentration supplémentaire). Cette modélisation permet de travailler directement sur les molécules elles-mêmes plutôt que sur des taux de concentration qui leurs seraient associées. Ensuite, en fonction du comportement de celles-ci on ajoute un des deux champs suivant :

actif : Au cours de la réaction, les molécule s'activent ou se désactivent en fonction de leur environnement. Pour modéliser ce comportement, on utilise un entier, qui prendra les valeurs 0 (pour une molécule inactive) ou 1 (pour une molécule active). *Exemple :*

```
state Molecule = {nom};;
state Activite = {actif};;
```

```
state Recepteur = Molecule + Activite + {nom = "recepteur"};;
```

diffuse : C'est un entier positif, initialisé à 0, et incrémenté de 1 à chaque pas d'évolution de la réaction. C'est une manière simplifiée de modéliser la trajet de la molécule dans la cellule. Elle devient opérante lorsqu'elle a atteint sa cible, c'est à dire lorsque son champs diffuse est supérieur ou égal à un taux fixé à l'avance dans une variable globale. Chaque molécule concernée par ce champs aura un taux de diffusion particulier correspondant à l'estimation de son temps de parcours.

Exemple :

```
state Molecule = {nom};;
state Diffuse = {diffuse};;

state Ca2p = Molecule + {nom = "ca++"} + Diffuse;;
```

3.2.2 Les réactions

Les réactions de l'expérience sont placées dans une transformation appelée *Biochimie*. On l'applique sur une ensemble de molécules. L'expérience de *Bugrim* nécessite 11 réactions.

La stratégie d'évaluation d'une transformation de *MGS* est la suivante : Les réactions s'exécutent dans l'ordre, sur toutes les molécules de l'ensemble considéré.

Détails d'une réaction sur un exemple :

```
state Actif = {actif = 1};;
state InActif = {actif = 0};;

state Recepteur1 = Recepteur + Actif;;
state Gprot0 = Gprot+InActif;;

Reaction2 = r:Recepteur1, g:Gprot0 =>
    (r+{actif=0})::(g+{actif=1})::():Plasma;
```

Pour que la réaction s'applique :

- Les éléments à gauche de la flèche doivent être contenus dans l'ensemble courant. Il faut donc que deux éléments soient présents : un de type *recepteur*, l'autre de type *gprot*.
- Les conditions à gauche de la flèche doivent être respectées. Le récepteur r doit ici être dans l'état actif (Recepteur1 correspond à un élément de

type *recepteur* dont l'état est actif) et la G-protéine g doit être dans l'état inactif (Gprot0 correspond à un élément de type *gprot* dont l'état est inactif).

Si la réaction s'applique, la partie gauche de la flèche est remplacée par la partie droite. Ici, r et g sont remplacés respectivement par un récepteur inactif et une g-protéine active, tous deux insérés dans un multi-ensemble de type *Plasma*. Chimiquement, cette opération correspond à l'activation d'une g-protéine par un récepteur membranaire actif. Une fois l'opération effectuée, le récepteur devient à nouveau inactif.

3.2.3 Structure générale de la cellule

Une cellule est un ensemble de sous-ensembles imbriqués les uns dans les autres. Chaque sous-ensemble doit être modélisé par une structure qui prend en compte la quantité de chaque élément ainsi que la libre circulation de celui-ci. On utilise donc des multi-ensembles *MGS* qui réponde totalement aux contraintes requises. Un multi-ensemble *MGS* est l'équivalent d'un ensemble mathématique où les multiples occurrences des éléments sont prises en compte. La libre circulation dans la cellule est assurée par l'aspect ensemble (pas de classement entre les éléments) et la quantité des éléments par leurs occurrences multiples.

L'un des problèmes posé par la modélisation de l'expérience de *Bugrim* est la représentation informatique des imbrications entre sous-ensemble cellulaires. *MGS* permet de répondre à cette question car les multi-ensembles sont des structures très souples qui permettent de mêler des éléments de différents types.

```
collection Volume = bag;;
collection Membrane = bag;;\emph{elt\_poss}

collection Univ = Volume;;
collection Plasma = Membrane;;
collection Cytosol = Volume;;
collection Retic_Endo = Membrane;;

RETICULUM := {nom="r2c2"}::
             ():Retic_Endo;;

CYTOSOL := {nom="i"}::
       RETICULUM::():Cytosol;;
```

```
PLASMA := {nom="r2c2"}::
      CYTOSOL::():Plasma;;

U :=    {nom= "agonist",actif=1}::
    PLASMA::():Univ;;
```

L'imbrication désirée est rendue par l'utilisation d'un multi-ensemble comme simple élément du multi-ensemble immédiatement supérieur.

3.2.4 Les aspects dynamiques de la réaction

Les temps de diffusion : On utilise pour modéliser un temps de diffusion d'une molécule le champs *diffuse* décrit précédemment. Dans la transformation Biochimie, à chaque pas d'évolution on appelle *IncrCyt* qui s'occupe d'incrémenter chaque molécule concernées par un temps de diffusion :

```
trans IncrCyt =
{
 I1 = ca:Ca2p => ca+{diffuse=ca.diffuse+1};
 I2 = a:cAMP => a+{diffuse=a.diffuse+1};
 I3 = c:Cenzime => c + {diffuse=c.diffuse+1};
};;
```

Une réaction utilisant une molécule ayant un champs *diffuse* s'effectue lorsque celui-ci atteint la valeur de sa variable globale associée. *Exemple :*

```
DIFFC:=10;;     //constante de diffusion pour C
DIFFCA:=20;;    //pour CA
DIFFCAMP:=20;;  //pour cAMP

state cAMP = Molecule + Diffuse + {nom = "camp"};;
fun cAMP1(x) = cAMP(x) & x.diffuse >= DIFFCAMP;;
```

Les réactions non-déterministes : Pour la réaction $C + I \rightarrow CI$,le problème est de modéliser le taux d'inhibition décrit par *Bugrim*, qui représente le pourcentage de C transformé en CI. étant donné que nous travaillons avec des molécules individualisées et non avec des concentrations, il nous est impossible de déclarer que $x\%$ de C sera transformé et que $100 - x\%$ ne le sera pas. En effet, si par exemple nous étions en présence d'une seule molécule C, inhiber 50% de la quantité totale des molécules C n'aurait aucun sens. C'est pour cette raison que nous avons choisi de représenter cette transformation par une réaction aléatoire :

```
fun Aleatoire(t,c,i) = let r=random(101) in
               if (t>r) then {nom="ci"}::():Cytosol
                else c::i::():Cytosol fi;;
```

La fonction *Aleatoire* renvoie, suivant le résultat du test $t > r$, la molécule CI ou bien les molécules C et I inchangées. r est calculé aléatoirement de la manière suivante : il peut prendre de manière équiprobable n'importe quelle valeur entière entre 0 et 100 (inclus).Or t étant le taux d'inhibition (en pourcentage),la probabilité que C se transforme en CI est de $t\%$ (car la probabilité d'avoir $t > r$ évalué à vrai , et donc de renvoyer CI est égale à $t\%$).

Grâce à cette méthode, l'état final de la réaction ne sera pas forcément le même. En effet, au moment de l'exécution de la fonction *Aleatoire*, en fonction d'un taux d'inhibition fixé par l'utilisateur en variable globale, la molécule C sera inhibée ou non. Or suivant le taux de diffusion que l'on affecte à la molécule C, elle sera actif plus ou moins tôt, donc plus son taux de diffusion est grand, plus il tardera à être actif, donc plus ses chances d'être inhibé avant d'être actif sont grandes, et donc plus les chances qu'il ne puisse pas activer la molécule $Ca.PhK$ sont grandes (car il faut pour cela que C soit actif et non inhibé).

Pour résumer, on peut simplement dire que l'enjeu biologique de l'expérience de Bugrim (c'est à dire activer $Ca.PhK$) dépent du taux d'inhibition de C ainsi que de son taux de diffusion.

3.2.5 Application récursive de la transformation

Pour simuler une exécution au cours du temps, on applique une transformation un certain nombre de fois jusqu'à atteindre un état chimiquement stable. On utilise pour cela une transformation *Recur* qui distribue l'évolution sur les membranes et les volumes. Et puis une transformation *Evolution* qui en utilisant le transformation *Recur*, applique la transformation biochimie.

```
trans Recur =
{
  nest1 = p:Membrane => (i:=i+1;Evolution[1](p));
```

```
  nest2 = p:Volume => (j:=j+1;Evolution[1](p));
};;

fun Evolution(x) = Recur(Biochimie(x));;
```

Pour utiliser cette transformation, il suffira de l'appliquer plusieurs fois à une solution de base. *MGS* peut le faire facilement, l'exemple suivant applique n fois la transformation Évolution sur une collection x :

```
Evolution[n](x);;
```

MGS permet normalement de faire évoluer un ensemble jusqu'à son point fixe par une transformation, sans avoir à spécifier le nombre d'itérations voulues, mais l'implémentation du passage du temps sur certaines molécules empêche cela. En effet, sur certaines molécule, un champs *diffuse* est incrémenté à chaque pas d'évolution, il n'existe donc pas par définition de point fixe à notre système (puisqu'il prends en compte de manière simplifiée le concept d'évolution du temps).

3.2.6 Utilisation

Le programme se trouve dans un fichier unique nommé *bugrim.mgs*. Pour voir la réaction, il suffit d'utiliser l'interprète *MGS*, par exemple avec la commande

```
./mgs-c++ bugrim.mgs
```

qui évalue le fichier et effectue une première évolution sur la solution initiale nommée *U*. On voit le résultat de cette évolution apparaître à l'écran sous forme de collections *MGS*. Pour une meilleure lisibilité nous avons utilisé un programme de visualisation en 3D dont le fonctionnement est détaillé dans le paragraphe 3.3.

Après une première évolution, il est possible de recommencer la manoeuvre. Dans un premier temps il faut initialiser la solution avec la fonction *init()*, puis utiliser la fonction *iter(n)*, où n correspond au nombre d'itération désirées.

On peut signaler une limite au langage *MGS* lors de l'utilisation de la fonction *random(n)* qui génère un entier aléatoire. Les valeurs rendues par cette fonction sont toujours les mêmes après chaque réinitialisation de *MGS*. Si on désire obtenir des résultats différents, il faut effectuer plusieurs fois le programme pour faire évoluer les valeurs rendues par la fonction *random*.

3.2.7 Conclusion

Dans cette partie nous avons suggéré une méthode de représentation d'un réseau spatiallement distribué de réactions chimiques interne à une cellule. La caractéristique principale de la méthode est que la géométrie de l'espace inter-cellulaire est simplifié pour s'adapter à une langage informatique tel que *MGS*.

3.3 Visualisation de l'expérience avec ImoView

3.3.1 Présentation d'ImoView

Pour obtenir une représentation plus agréable de l'expérience de *A. E. Bugrim*, tout en restant proche du langage *MGS*, nous avons utilisé un outil de visualisation nommé Imoview. Il a été développé par François Letierce dans le cadre de son TER en maîtrise d'informatique en 2000-2001 à l'université d'Evry[3]. Il utilise un langage nommé TEOM qui permet de représenter des objets dans un espace en trois dimensions (par exemple : un cube, une sphère..). Il permet aussi de représenter des imbrications d'objets entre eux, ce qui correspond très bien à la notion de collection rencontrée en *MGS*. Pour représenter un cube avec Imoview, on utilise la syntaxe suivante :

```
Box { Size < 2, 1, 3 >
      Color < 1, 1, 0 > }
```

On place ces lignes dans un fichier que l'on compile grâce au programme *Imoview*. Ensuite la représentation se dessine dans une fenêtre. On peut ensuite faire bouger les objets, faire apparaître les axes, les plans, les points d'éclairage, et plein d'autres choses encore...

Pour avoir plus de renseignements à propos d'imoview, une présentation détaillée du projet est disponible à l'adresse suivante :
http ://www.lami.univ-evry.fr/∼teropengl/

3.3.2 Création de fichiers ImoView avec *MGS*

Pour réaliser automatiquement la création de la représentation graphique de l'expérience de *Bugrim* nous imprimons dans un fichier (ou plusieurs) ce que l'on veut représenter. Par exemple, si l'on reprend l'élément *Box* définit dans le paragraphe précédent :

```
"tmp.imo" << "Box {Size < 2, 1, 3 >\n Color < 1, 1, 0 >}"
```

La commande « imprime dans le fichier *tmp.imo* la chaîne de caractères donnée à gauche.

On utilisera aussi la commande *print_coll* de *MGS* qui imprime une collection complète dans un fichier donné.

3.3.3 Choix de représentation

Dans le cadre de l'expérience traitée ici, on utilise pour chaque molécule une figure géométrique différente. On fait varier également les couleurs en fonction de l'état de celles-là. Par exemple, une molécule qui s'active sera colorée en rouge.

Pour représenter les imbrications de collection, il existe en Imoview des figures nommées *Container* qui permettent de contenir plusieurs éléments. Ces éléments sont organisés automatiquement dans l'espace, en 1,2,ou 3 dimensions. Pour notre problème nous utiliserons imbrications de grilles à 2 dimensions. Chacune représente une partie de la cellule (Plasma, Cytosol, Reticulum endoplasmique), et dans chacune d'elles se trouvent des molécules représenter par des figures géométriques.

Nous avons programmé deux manières de visualiser l'expérience. L'une qui crée un fichier par pas d'évolution de la réaction, et l'autre qui écrit toutes les étapes dans un même fichier en décalant les représentation dans l'espace. Dans celle-ci, les représentations apparaissent les unes après les autres grâce à une commande de pause. A la fin, toutes les étapes apparaissent dans la même fenêtre de visualisation.

3.3.4 Évolution de l'expérience de *A. E. Bugrim*

Après avoir exécuté le programme *bugrim.mgs*, pour obtenir un fichier *tmp.imo* qui contient la représentation de l'évolution de l'expérience en trois dimensions, il suffit d'utiliser la fonction *iterimo(n)* plutôt que *iter(n)*.

Pour la visualisation des images en 3D de l'expérience de *Bugrim*, il faut se reporter aux pages annexes en fin de rapport.

4 Conclusion

L'un des enjeux de ce TER était de montrer en quoi MGS est un langage qui s'adapte bien aux demandes multiples de la chimie artificielle et plus généralement de la simulation biochimique. L'objectif est atteint car nous avons réussi à modéliser sans difficultés insurmontables un exemple de chimie artificielle (une solution de polymères alimentée en permanence de l'extérieur pour les réseaux autocatalytiques) et une expérience biochimique (un réseau d'imbrications cellulaires pour l'expérience de Bugrim). Cette facilité vient des structures déjà implémentées dans le language même (séquences, ensembles et multi-ensembles), mais surtout des notions de collections et de transformations. En effet, celles-ci sont définies en *MGS* de manière très larges et donc peuvent englober de nombreux concepts, notamment ceux rencontrés en chimie artificielle. Néanmoins, notre travail possède plusieurs limites, à commencer par l'efficacité. En effet, notre modélisation de réseau auto-catalytique est plus théorique qu'effective : le temps de calcul de l'interprète *MGS* est tel qu'il est impossible d'utiliser notre programme sur des quantités réalistes. De plus, nous avons été gêné par la fonction random de *MGS*, que nous utilisons à plusieurs reprises pour faire intervenir la notion de hasard, mais qui nous renvoit toujours la même séquence d'entiers, même après une nouvelle initialisation. Nous avons contourné ce problème en utilisant plusieurs fois la fonction random avant utilisation du programme à proprement parlé, pour obtenir des résultats différents. De plus, alors que *MGS* permet un niveau d'abstraction assez élevé dans le code, nous n'avons pas pu, surtout pour l'implémentation de l'expérience de *Bugrim* pu programmer sans nous abstraire du problème concret posé par le problème, ce qui implique un code peu réutilisable. Cela est du au caractère peut généralisable des propriétés d'une molécule, qui implique une individualisation de chaque corps biologique et donc abaisse le niveau d'abstraction du programme.

5 Annexes

5.1 Visalisatision de l'expérience de *A. E. Bugrim*

5.1.1 A-I

5.1.2 A-II

5.1.3 A-III

5.2 Code source : Réseau auto-catalytique

```
// ***********
// autocat.mgs
// ***********

NBMONOMERE:=4;;         // nombre de monomères différents
                        // (si NBMONOMERE=3 => {1,2,3})

TMAX:=2;;               // taille maximum d'un polymère
NBPOLY:=2;;             // taille de la liste des polymères
NBREACTION:=2;;         // taille de la liste des reactions
TOCATA:=10;;


// *****
// types
// *****

collection solution = bag;;
collection sol_compte = bag;;


// ****************************************
// fonction de condition aléatoire générique
// ****************************************

fun condition(x) = let r=random(101) in (r<x);;




// ************************************************
// fonction qui rend le nième élément d'une séquence
// ************************************************

fun nieme(n,liste) = if (n==1) then hd(liste)
                               else nieme(n-1,tl(liste))fi;;
```

```
// *****************************************
// création d'un polynôme de manière aléatoire
// *****************************************

fun genere2(r) = if r==0 then ""
                         else let m=(random(NBMONOMERE)+1)
                                in ""+m+genere2(r-1)
                 fi;;

fun genere_poly(taille) = let r=(random(taille)+1) in genere2(r);;



// *****************************************************
// création de la liste des poly nômes de manière aléatoire
// *****************************************************

fun genere_seq_init(taille) = if (taille==0) then ():seq
    else genere_poly(TMAX)::genere_seq_init(taille-1)fi;;

fun genere_seq2(elt,liste) = if (liste == ():seq)
    then ():seq
    else (elt+hd(liste))::genere_seq2(elt,tl(liste))fi;;

fun genere_seq3(liste1,liste2) = if (liste1 == ():seq)
    then liste2
    else (genere_seq2(hd(liste1),liste2)@genere_seq3(tl(liste1),liste2))fi;;

fun genere_seq (liste) = genere_seq3(liste,liste);;


// ****************************************************
// création de la liste des réactions de manière aléatoire
// ****************************************************

state Reaction = {poly1, poly2, ordre, kf, kr, cata};;

fun alea2(r,liste) = if r==1 then hd(liste)
                             else alea2(r-1,tl(liste))fi;;

fun alea(liste) = let r=(random(size(liste))+1) in alea2(r,liste);;
```

```
fun aleacata(liste) = let coeff=random(101) in
    if coeff<TOCATA then alea(liste) else ""fi;;


fun genere_seq_reaction2(n,liste_elt,liste_cata) =
                        if (n == 0)
                        then ():seq
                        else  let p1 = alea(liste_elt)
                              and  p2 = alea(liste_elt) in
                                        {poly1 = p1,
                                        poly2 = p2,
                                        poly_res = p1+p2,
                                        kf = random(101),
                                        kr = random(101),
                                        cata = aleacata(liste_cata)}
                  ::genere_seq_reaction2(n-1,liste_elt,liste_cata)fi;;

fun genere_seq_reaction(n,liste_elt) =
    genere_seq_reaction2(n,liste_elt,genere_seq(liste_elt));;



// **************
// Initialisation
// **************

elt_init := ():seq;;
elt_poss := ():seq;;
reaction := ():seq;;
sol := ():solution;;
i:=1;;

fun affiche_init() =
("\n\nRéactions: \n",reaction,"\n\nSolution de depart:\n",sol,
 "\n\nPolymères de depart:\n ",elt_init,"\n");;



fun init() =
(
```

```
 elt_init := genere_seq_init(NBPOLY);
 elt_poss := genere_seq(elt_init);
 reaction := genere_seq_reaction(NBREACTION,elt_init);
 sol := ""::():solution;
 i:=1;
 affiche_init());;



// *****************
// Réaction générique
// *****************

fun condition(x) = let r=random(101) in (r<x);;

trans Biochimie =
{

 reaction_a = a, b, c /((a==nieme(i,reaction).poly1)
 &(b==nieme(i,reaction).poly2)&(c==nieme(i,reaction).cata))

          => if condition(nieme(i,reaction).kf)
             then (nieme(i,reaction).poly_res)::c::():solution
             else a::b::c::():solution fi;

 reaction_b = a, c /((a==nieme(i,reaction).poly_res)
 &(c==nieme(i,reaction).cata))

          => if condition(nieme(i,reaction).kr)
             then (nieme(i,reaction).poly1)
                ::(nieme(i,reaction).poly2)::c::():solution
             else  a::c::():solution  fi;

};;




// ****************************************************
// spécification à chaque itération de la réaction générique
```

```
// *******************************************************

fun iter() = if (i>NBREACTION) then sol
                else sol:=Biochimie(sol);i:=i+1;iter() fi;;



// ***********************
// évolution de la solution
// ***********************

fun evol (n) = if (n==0) then sol
                              else i:=1;
                                   sol:=alea(elt_poss)::sol;
                                   iter();
                                   evol(n-1) fi;;

fun evol_init (n) = (sol := ""::():solution; evol(n));;



// *****************************************************
// La fonction compte() compte les éléments de la solution
// *****************************************************

fun fst (x) = hd(x);;

fun snd (x) = hd(tl(x));;

fun incr(a,liste) = if liste == ():sol_compte
        then (a,1)::():sol_compte
        else if fst(hd(liste)) == a
             then (a,snd(hd(liste))+1)::tl(liste)
             else hd(liste)::incr(a,tl(liste))fi fi;;

fun compte2 (sol,res) = if sol == ():solution
                then res
                else compte2(tl(sol),incr(hd(sol),res))fi;;

fun compte () = tl(compte2 (sol,():sol_compte));;
```

```
fun concentration_max2 (sol_c,x) = if (sol_c == ():sol_compte)
          then x
          else  if (snd(x) >= snd(hd(sol_c)))
                then concentration_max2(tl(sol_c),x)
                else concentration_max2(tl(sol_c),
                     (fst(hd(sol_c)),snd(hd(sol_c))))fi fi;;


// ******************************************************************
// Suppression de l'élément dominant de la liste des elts possibles
// ******************************************************************

fun dominant2 (sol_c) = concentration_max2(sol_c,(<undef>,0));;


fun dominant () = dominant2(compte());;


fun suppr (elt,liste) = if liste==():seq
                   then ():seq
                   else if (hd(liste)==elt)
                        then suppr (elt,tl(liste))
                        else (hd(liste)::suppr(elt,tl(liste)))fi fi;;



fun rm () = elt_poss:= suppr(fst(dominant()),elt_poss);;


// ******************************************************
// Histogramme des concentrations finales des polymères
// ******************************************************

fun trace3(c,l) =  (if c==():sol_compte
        then ""
        else l +" "+snd(hd(c))+"\n"+ trace3(tl(c),l+1) fi);;

fun trace2(c,n)= "trace"+n+".tab" << trace3(c,1);;
```

```
// ********************************
// Faire la légende de l'histogramme
// ********************************

fun leg2(c,l) =  (if c==():sol_compte
        then ""
        else l +" -> "+fst(hd(c))+"\n"+ leg2(tl(c),l+1) fi);;

fun leg(c,n)= "trace"+n+".leg" << leg2(c,1);;


// ***********************************************************
// fonction qui génère le fichier à utiliser pour l'histogramme
// ***********************************************************

fun trace(n)= let c=compte() in trace2(c,n);leg(c,n);;


// ************************
// appel de l'initialisation
// ************************

init();;
```

5.3 Code source : expérience de *A. E. Bugrim*

5.3.1 Modélisation en *MGS*

```
// == Description des strutures dans l'espace==

collection Volume = bag;;
collection Membrane = bag;;

collection Univ = Volume;;
collection Plasma = Membrane;;
collection Cytosol = Volume;;
collection Retic_Endo = Membrane;;



// == Définitions génériques ==

state Molecule = {nom};;
state Diffuse = {diffuse};;
state Activite = {actif};;
state Actif = {actif = 1};;
state InActif = {actif = 0};;



//Variables globales

DIFFC:=3;;
DIFFCA:=2;;
DIFFCAMP:=2;;
TAUXINIB:=50;;


// Définitions

state Agonist = Molecule + Activite + {nom = "agonist"};;
state Agonist1 = Agonist + Actif;;

state Recepteur = Molecule + Activite + {nom = "recepteur"};;
state Recepteur1 = Recepteur + Actif;;
```

```
state Gprot = Molecule + Activite + {nom = "gprot"};;
state Gprot1 = Gprot+Actif;;
state Gprot0 = Gprot+InActif;;

state AC = Molecule + Activite + {nom="ac"};;
state AC1 = AC + Actif;;
state AC0 = AC + InActif;;

state ATP = Molecule + {nom = "atp"};;

state cAMP = Molecule + Diffuse + {nom = "camp"};;
fun cAMP1(x) = cAMP(x) & x.diffuse >= DIFFCAMP;;

state I = Molecule + {nom = "i"};;

state CaPhK = Molecule + Activite + {nom = "caphk"};;
state CaPhK1 = CaPhK + Actif;;
state CaPhK0 = CaPhK + InActif;;

state Ca2p = Molecule + {nom = "ca++"} + Diffuse;;
fun Ca2p1(x) = Ca2p(x) & (x.diffuse >= DIFFCA);;

state cAMP2R = Molecule + {nom = "(camp)2.r"};;

state Cenzime = Molecule + {nom ="c"} + Diffuse;;
fun Cenzime1(x) = Cenzime(x) & (x.diffuse >=DIFFC);;

state CI = Molecule + {nom = "ci"};;

state R2C2 = Molecule + {nom = "r2c2"};;

state PhK = Molecule + { nom ="phk"};;

state Stimulus = Molecule + Activite + {nom = "stimulus"};;
state Stimulus1 = Stimulus + Actif;;

state Channel = Molecule + Activite + {nom = "channel"};;
state ChannelOpen = Channel + Actif;;
state ChannelClosed = Channel + InActif;;
```

```
// == Transformations auxilaires ==

trans ActiveRecepteur = {r:Recepteur => (r + {actif =1})};;

trans ActiveChannel = { c:Channel => c+ {actif=1}};;

trans TransAtp[i=0] =
       { a:ATP/i<1 => (i:=i+1; {nom="camp"}+{diffuse=0})};;

trans AjoutPlasmaCa =
       { c:Cytosol  => ({nom="ca++",diffuse = 0})::c};;

trans DesChannel = {c:ChannelOpen => (c+{actif=0})};;

fun Aleatoire(t,c,i) = let r=(random(101)) in if (t>r)
                           then {nom="ci"}::():Cytosol
                           else c::i::():Cytosol fi;;

fun AleaR2C2(a,re) = let r=(random(2)) in
                (if (r==1) then
       (delete({nom="r2c2"},re):: {nom="c",diffuse=0}::
              {nom="(camp)2.r"}::():Cytosol)
                           else a::re::():Cytosol fi);;


trans IncrCyt =
{
 I1 = ca:Ca2p => ca+{diffuse=ca.diffuse+1};
 I2 = a:cAMP => a+{diffuse=a.diffuse+1};
 I3 = c:Cenzime => c + {diffuse=c.diffuse+1};
};;




// Définition de réactions biochimiques
i:=0;;
j:=0;;
k:=0;;
```

```
trans Biochimie =
{
 Incr = c:Cytosol => IncrCyt(c)::():Cytosol;
          //Incrémenter les tps de diffusion

 Reaction1 = a:Agonist1, p:Plasma =>
           (a+{actif=0})::ActiveRecepteur(p)::():Univ;

 Reaction2 = r:Recepteur1, g:Gprot0 =>
           (g+{actif=1})::(r+{actif=0})::():Plasma;

 Reaction3 = g:Gprot1, a: AC0 =>
           (g+{actif=0})::(a+{actif=1})::():Plasma;

 Reaction4 = a: AC1, c:Cytosol =>
           (a+{actif=0}):: TransAtp(c)::():Plasma;

 Reaction5 = a:cAMP1, re:Retic_Endo/member({nom="r2c2"},re) =>
           AleaR2C2(a,re);

 Reaction5b = r:R2C2, c:Cytosol/member({nom="camp",diffuse=DIFFCAMP},c) =>
            delete({nom="camp",diffuse=DIFFCAMP}, ({nom="c",diffuse=0}::
            {nom="(camp)2.r"}::c))::():Plasma;

 Reaction6 = c:Cenzime1, x:CaPhK0 => c::(x+{actif=1})::():Cytosol;

 Reaction7 = c:Cenzime, i:I => Aleatoire(TAUXINIB,c,i);

 Reaction8 = s:Stimulus1, p:Plasma =>
             (s+{actif=0})::ActiveChannel(p)::():Univ;

 Reaction9 = c:Ca2p, p:Plasma/member({nom="channel",actif=1},p) =>
             AjoutPlasmaCa(DesChannel(p))::():Univ;

 Reaction10 = x:Ca2p1, y:PhK => {nom="caphk", actif =0}::():Cytosol;
};;
```

```
//== Initialisation

RETICULUM := {nom="r2c2"}::
              ():Retic_Endo
;;

CYTOSOL := {nom="i"}::
           {nom="phk"}::
           {nom="atp"}::
            RETICULUM::
            ():Cytosol
;;

PLASMA := {nom= "recepteur",actif=0}::
          {nom="r2c2"}::
          {nom= "gprot",actif=0}::
          {nom= "ac",actif=0}::
          {nom= "channel",actif=0}::
          CYTOSOL::
          ():Plasma
;;

U :=    {nom= "agonist",actif=1}::
        {nom= "stimulus",actif=1}::
        {nom= "ca++",diffuse=0}::
        PLASMA::
        ():Univ
;;
```

5.3.2 Création du fichier pour l'interface graphique

```
//Création du fichier Imoview

rdeuxcdeux:=
"Grid2 {\nLineUp1 Left \nAxis <1,1,0>\nSpace1 -0.5\nSpace2 -0.5\n
GridList[\nCylinder {Color < 0, 1, 0.5 > \nRadius 0.7 \nHeight 2} ,
           Cylinder {Color < 0, 1, 0.5 >  \nRadius    0.7\nHeight 2},
           Sphere { Color < 0, 0, 1 >  \nRadius    0.7 },
           Sphere {Color < 0, 0, 1 >  \nRadius  0.7 }\n]";;

campdeuxr:= "Grid1{\nAxis <0,1,0>\n Space1 -0.3\n
        GridList[ \nSphere {Color < 0, 0, 1 >\nRadius 0.5},
                \nCylinder {Color <0,0,0.5>\nRadius 0.5\nHeight 2\n},
                \nSphere {Color <0,0,1>\nRadius 0.5}\n]";;

ci := "Grid1 {  Axis <0,1,0>\n
                Space1  (-1)\n
                GridList [ Sphere { Radius 1\nColor <2,2,0>},
                           Sphere { Radius 0.7\nColor <0,0,1>}]";;


caphk2 := "Grid1 {Axis <0,1,0>\n
        Space1    (-0.2)\n
        GridList [ Sphere { Radius 0.5\n Color <0,0,1>},
        Sphere { Radius 0.7\n Color <2,2,2>}]\n";;


caphk2actif:= "Grid1 {Axis <0,1,0>\n
        Space1    -0.2\n
        GridList [ Sphere { Radius 0.5\nColor <2,0,0>},
        Sphere { Radius 0.7\nColor <2,0,0>}]\n";;


fun couleur(x) = if (x.actif==1)||((x.nom=="camp")&(x.diffuse>=DIFFCAMP))
                               ||((x.nom=="c")&(x.diffuse>=DIFFC))
                               ||((x.nom=="ca++")&(x.diffuse>=DIFFCA))
                        then "Color <2,0,0>\n}"
                        else "Color <0,0,1>\n}" fi;;
```

```
fun nom(x) = (x.nom=="r2c2")||(x.nom=="atp")||(x.nom=="i")||(x.nom=="phk")||
             (x.nom=="(camp)2.r")||(x.nom=="ci")||(x.nom=="caphk");;


fun show(x) = let k2= k*50 in (if Univ(x)
                  then  print_coll("tmp.imo", x, show,
"Translated{Translation <"+k2+",0,0>\n Geometry Container{\n
Color< 1, 0, 0 >\nType Ellipsoide\nOpacity 0.0\nContain [Grid2 {
Space2 -1\n GridList [\n",",\n","]}\n]\n}\n}\nCommand Pause\n\n\n"); ""
                  else
                     (if bag(x) then (if x==():Retic_Endo
          then "tmp.imo"<<"Sphere{\nRadius 2\nColor <0.5,0,0>}";""
          else print_coll("tmp.imo", x, show,
"Container{\nColor<0.5,0,0>\nType Ellipsoide\nDraw Half\n
Contain [Grid2 {Space1 2\n Space2 2\nGridList [\n",",\n","]}\n]\n}"); "" fi)
                                else "tmp.imo"<<
            ((if x.nom=="agonist"   then
"AxisRotated { Axis <0,0,1>\n Angle -90\n Geometry Cone{Radius 1.5 \n
                            Height 5\nSolid True\n"
        else (if x.nom=="stimulus"  then "AxisRotated { Axis <1,0,0>\n
          Angle 90\n Geometry Cone{Radius 2 \nHeight 3\nSolid True\n"
        else (if x.nom=="ca++"      then "Sphere {Radius 0.5\n"
        else (if x.nom=="recepteur" then "AxisRotated { Axis <0,0,1>\n
          Angle 90\n Geometry Cylinder{ Radius 1\nHeight 0.5\nSolid True\n"
        else (if x.nom=="channel"   then "AxisRotated { Axis <0,0,1>\n
          Angle 90\n Geometry Cylinder{ Radius 0.6\nHeight 3\nSolid True\n"
        else (if x.nom=="gprot"     then "Sphere{Radius 0.7\n"
        else (if x.nom=="ac"        then "Sphere{ Radius 1\n Slices 6\n"
        else (if x.nom=="r2c2"      then rdeuxcdeux
        else (if x.nom=="atp"       then "Frustum{Radius 1\nHeight 2\n
                                 Taper 0.5\nSolid True\nColor <0, 1, 0>\n"
        else (if x.nom=="i"         then "Sphere{Radius 1\nColor <2,2,0>\n"
        else (if x.nom=="phk"       then "Sphere{Radius 0.7\nColor <2,2,2>\n"
        else (if x.nom=="c"         then "Sphere { Radius 0.7\n"
        else (if x.nom=="camp"      then "Sphere{ Radius 0.5\n"
        else (if x.nom=="(camp)2.r" then campdeuxr
        else (if x.nom=="ci"        then ci
        else (if x.nom=="caphk"     then (if x.actif==0
                                             then caphk2
                                             else caphk2actif fi)
```

```
        else
"Sphere {Radius 1\n" fi)fi)fi)fi)fi)fi)fi)fi)fi)fi)fi)fi)fi)fi)fi)fi)
                           +(if nom(x)
                                then "}"
                                else couleur(x) fi)
                    +(if (x.nom=="agonist")||(x.nom=="stimulus")||
                         (x.nom=="recepteur")||(x.nom=="channel")
                                     then "}\n"
                                     else "\n" fi));"" fi)fi);;
```

5.3.3 Fonctions finales

```
trans Recur =
{
  nest1 = p:Membrane => (i:=i+1;Evolution[1](p));
  nest2 = p:Volume => (j:=j+1;Evolution[1](p));
};;

fun Evolution(x) = Recur(Biochimie(x));;

sol:=U;;

//Itération de la transformation + Création fichier imoview
fun iterimo(n) = if n==0 then sol
       else show(sol);sol:=Evolution[1](sol);k:=k+1;iterimo(n-1) fi;;

//Itération simple
fun iter(n) = if n==0 then sol else sol:=Evolution[1](sol);iter(n-1) fi;;

//Initialisation de la solution
fun init() = sol:=U;;

iter(8);;
```

Références

[1] Jean-Louis Giavitto, Olivier Michel, Julien Cohen : Une présentation du langage *MGS* (Version 0.1), LaMI (Laboratoire des Méthodes Informatiques), Décembre 2001.

[2] Jean-Louis Giavitto, Olivier Michel : *MGS* : a Programming Language for the Transformations of Topological Collections. Rapport de Recherche n°61-2001, LaMI, Mai 2001.

[3] François Letierce : T.E.R. Serveur Graphique OpenGL, Maîtrise d'Informatique 2000-2001, Université d'Evry Val d'Essonne.

[4] P. Dittrich : Artificial chemistry page. `http://ls11-www.cs.uni-dortmund.de/achem`

[5] Maurice Demarty, Lois Le Sceller, Camille Ripoll, Derek Raine, Bernard Gleyse et Vic Norris : Modelling autocatalytic networks alias knitting by numbers Laboratoire des Processus Integratifs Cellulaires, UPRESA CNRS 6037, Faculte des Sciences et Techniques de Rouen, F76821 Mont Saint Aigan, France

[6] A. E. Bugrim : Logic-based approach to the analysis of spatially distributed biochemical networks. Center for Computationnal Biology, Washington University, St. Louis, MO 63110.