GOAML

Manuel utilisateur et manuel technique

30 août 2016

Sommaire

Table des figures

Liste des tableaux

1 Manuel utilisateur

Ce manuel utilisateur explique comment lancer puis utiliser GOAML afin de réaliser une simulation.

1.1 Exécution de GOAML

Il est possible d'exécuter GOAML de plusieurs manières. Cette section aura donc pour objectif de décrire ces différentes possibilités. Il est important de noter que le programme GOAML a été développé avec **JavaSE-1.8**. Il faut donc avoir installé cette version sur son ordinateur pour pouvoir exécuter GOAML.

1.1.1 Exécution à partir d'un fichier JAR

Il s'agit de la manière la plus simple de lancer GOAML. Il suffit de double-cliquer sur le fichier JAR « goaml.jar ».

1.1.2 Exécution en ligne de commande

Il est possible d'utiliser le dossier contenant le projet GOAML pour lancer une exécution de GOAML. Ce dossier contient notamment le dossier « src » contenant le code source (fichier avec l'extension « java ») ainsi que le dossier « bin » contenant les fichiers sources traduits en bytecode (les fichiers avec l'extension « class »). Pour exécuter GOAML à partir de ce dossier, il faut utiliser un terminal (rechercher « Invite de commandes » sur Windows). Il faut ensuite se déplacer dans le dossier « bin » du dossier. Pour ce faire, il faut utiliser la commande **cd** suivi du chemin pour accéder à ce dossier. Par exemple si ce dossier à cet endroit « `C:\Users\nomdelutilisateur\Documents\GOAML\bin` », il faut taper la commande **cd** `C:\Users\nomdelutilisateur\Documents\GOAML\bin`. Il faut ensuite taper la commande **java goaml.App** pour lancer le programme.

1.2 Utilisation de GOAML

Lorsque GOAML est exécuté, une fenêtre apparaît. Cette fenêtre peut être utilisée pour configurer et lancer une simulation à partir d'un fichier XML, **et/ou** afficher graphiquement les résultats d'une simulation à partir d'un fichier CSV (figure 1).

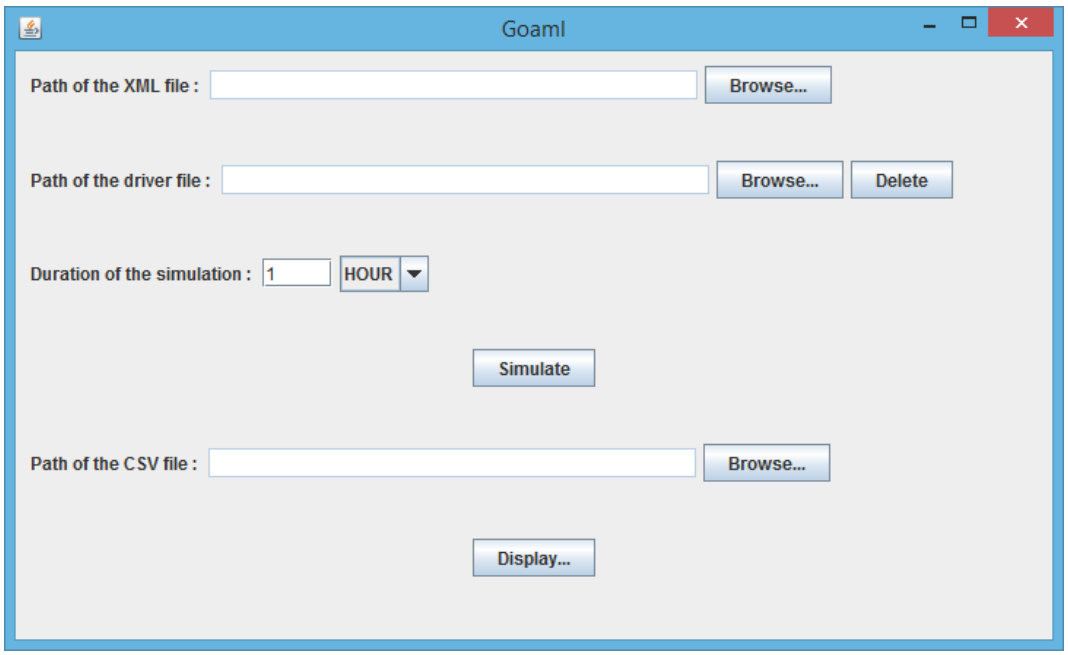

FIGURE 1 – Fenêtre de démarrage de GOAML

1.2.1 Le modèle à simuler

Il faut tout d'abord choisir le fichier XML décrivant le modèle à simuler. Pour ce faire, il faut utiliser le premier bouton « Browse... » comme le montre la figure 2.

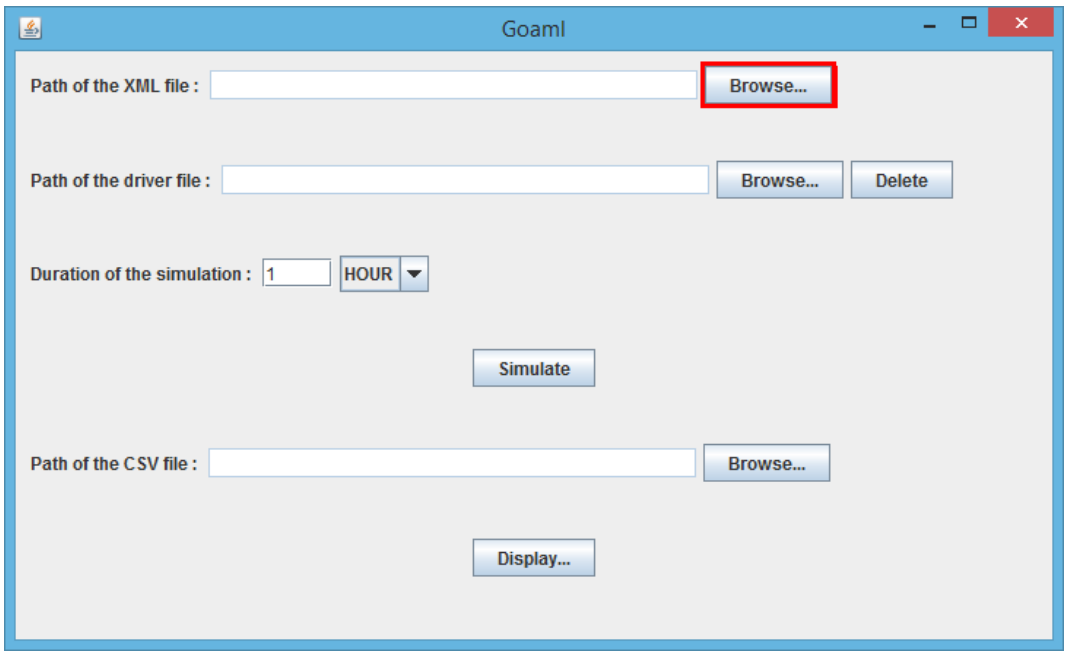

FIGURE 2 – Sélection du modèle à simuler (fichier XML)

1.2.2 Le driver

Il est également possible d'utiliser un driver lors de la simulation. Ce driver contient le protocole expérimental, comme par exemple des injections à réaliser à certains moments de la simulation. Cependant, l'utilisation d'un driver n'est pas obligatoire pour réaliser une simulation. Pour choisir un driver, il faut utiliser le deuxième bouton « Browse... » comme le montre la figure 3. Il est également possible de supprimer le driver qui a été sélectionné à l'aide du bouton « Delete » si l'on souhaite par exemple relancer la simulation sans utiliser de driver.

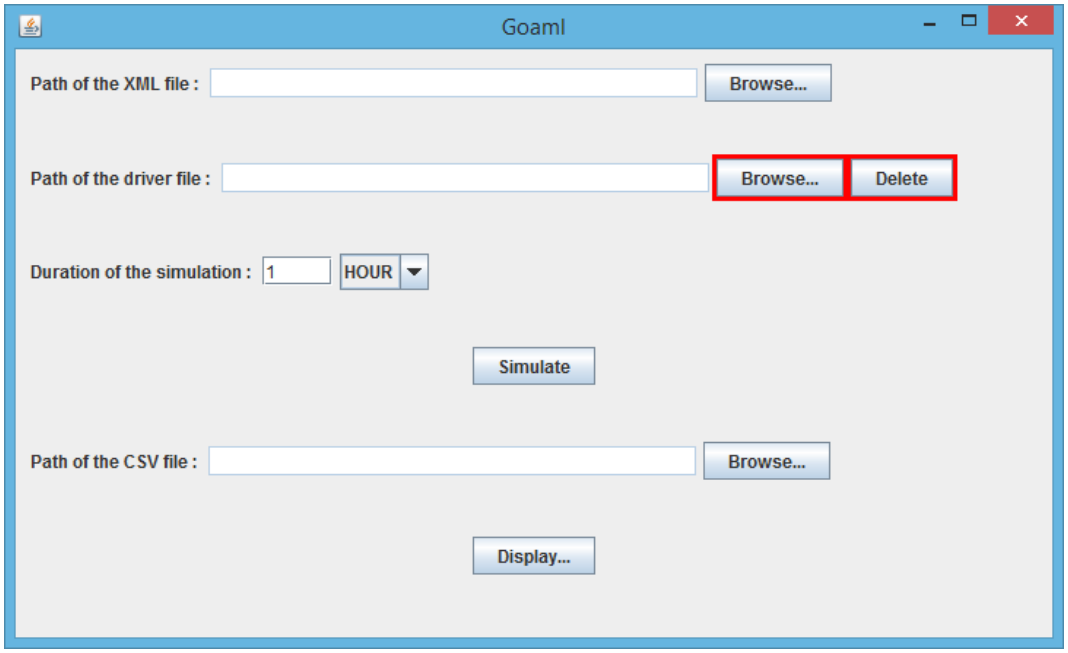

FIGURE 3 – Sélection et suppression du driver (fichier CSV)

1.2.3 Sélection de la durée de la simulation

Il faut ensuite définir la durée de la simulation. Pour ce faire il faut entrer un nombre dans le champ de texte associé au label « Duration of the simulation » comme présenté dans la figure 4. Il est aussi possible d'indiquer si le nombre entré correspond à un nombre d'heures ou bien à un nombre de jours à l'aide du menu déroulant.

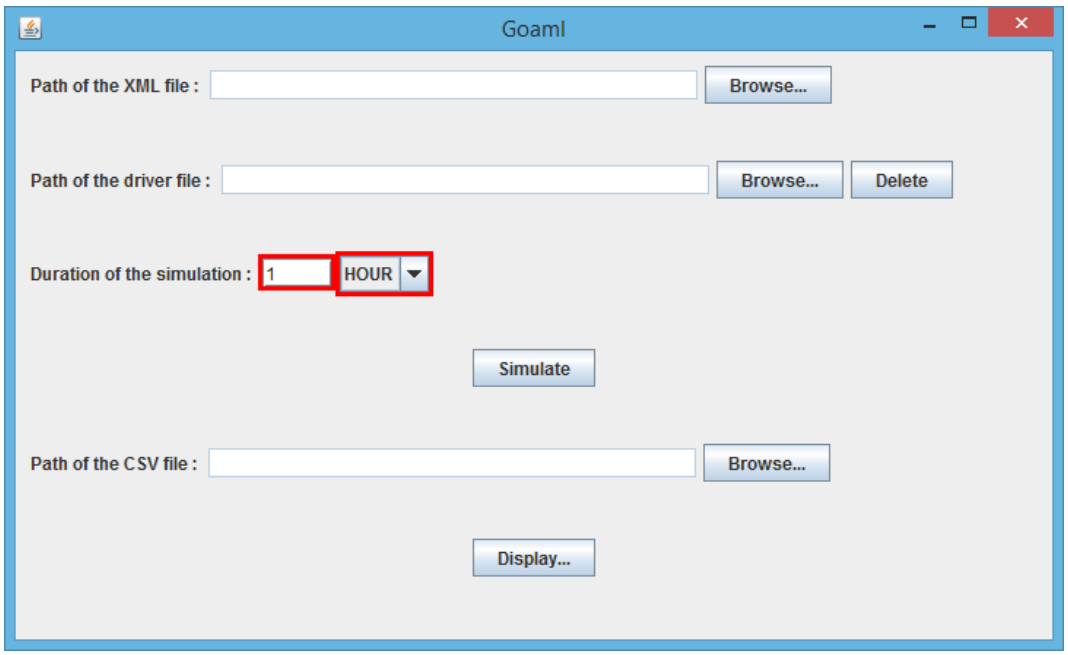

FIGURE 4 – Sélection de la durée de la simulation

1.2.4 Lancement de la simulation

Pour lancer la simulation, il faut cliquer sur le bouton « Simulate » (figure 5). Un fichier au format CSV sera créé dans le même dossier que le fichier JAR permettant de lancer GOAML. Il contiendra toutes les valeurs que les états de chacune des entités présentes dans le modèle ont pris au cours de la simulation.

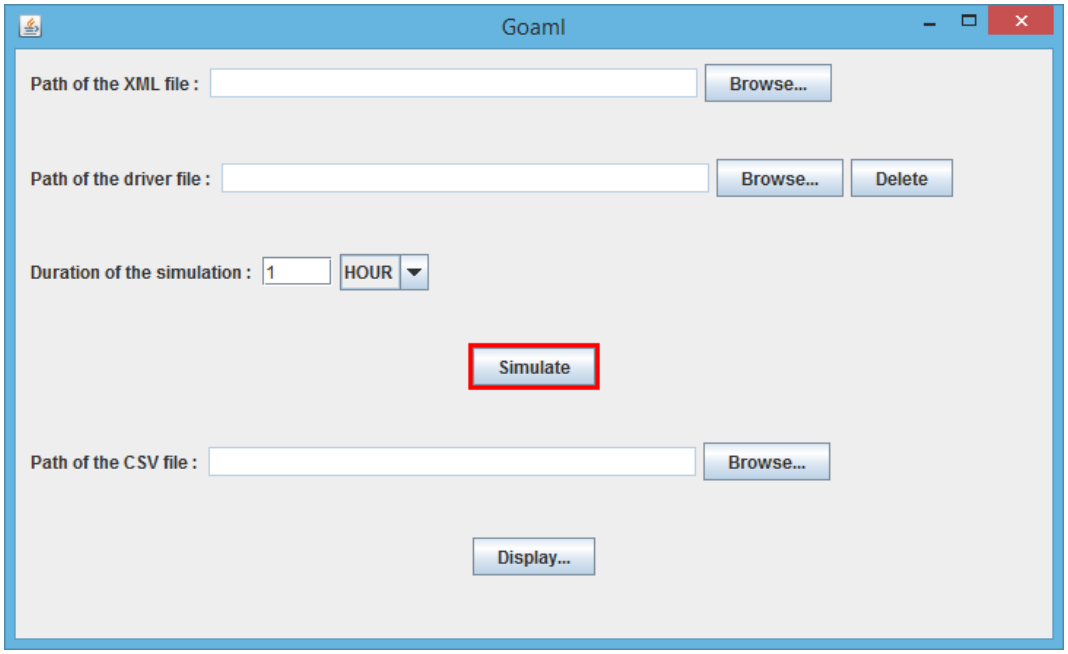

FIGURE 5 – Lancement de la simulation

1.2.5 Affichage graphique des résultats d'une simulation

Il est possible d'afficher graphiquement les résultats de la simulation. Pour ce faire, il faut sélectionner le fichier CSV contenant les résultats d'une simulation, puis lancer l'interface graphique. Les deux boutons pour réaliser cela sont entourés en rouge dans la figure 6.

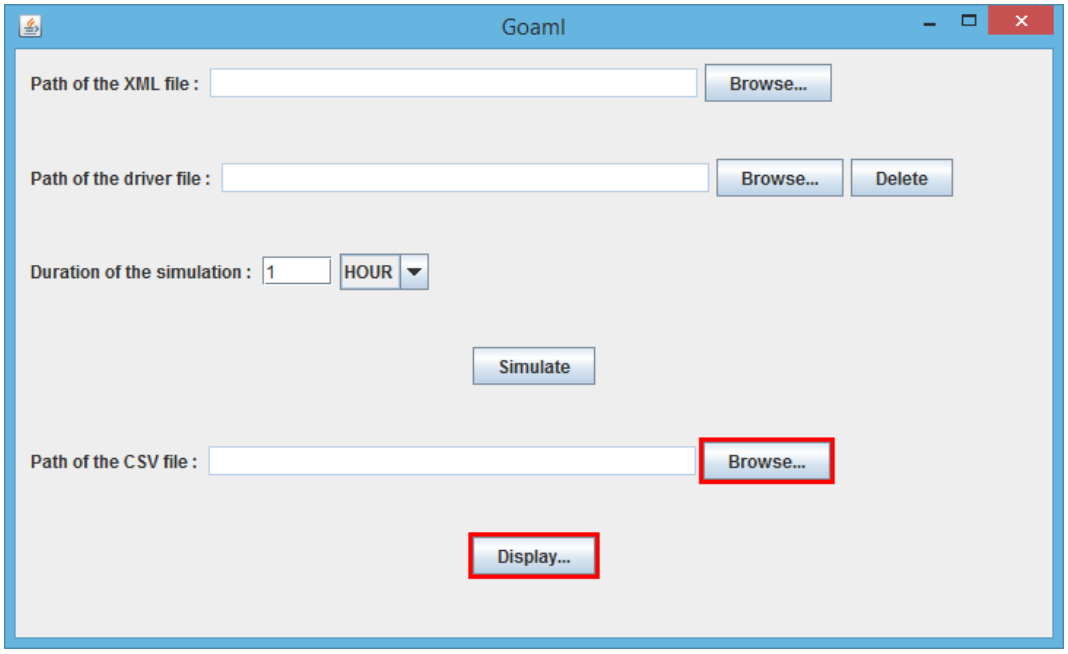

FIGURE 6 – Lancement d'un affichage graphique des résultats

Une nouvelle fenêtre apparaît alors et les valeurs des différents états au cours de la simulation sont affichées au moyen de courbes (figure 7).

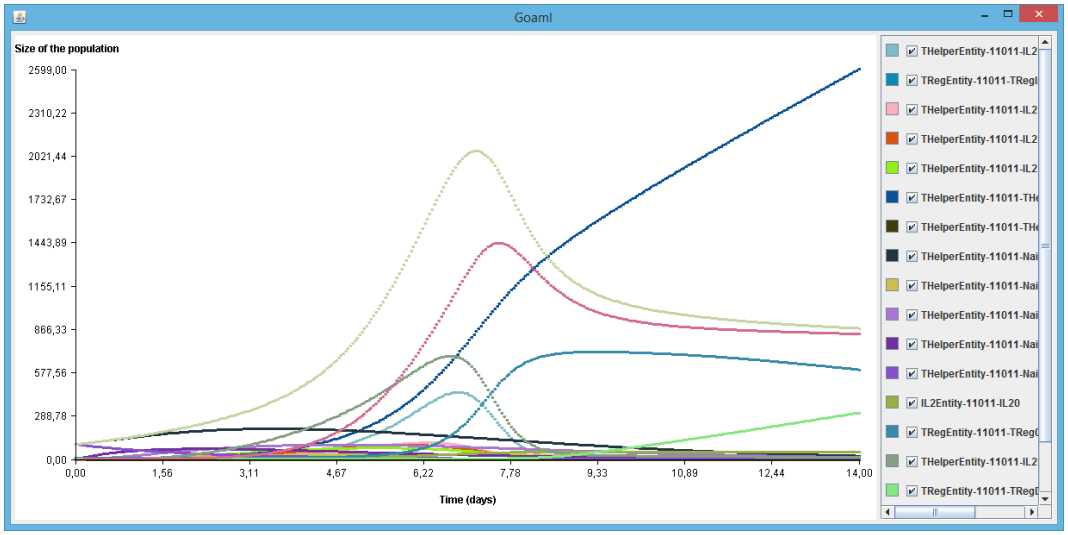

FIGURE 7 – Fenêtre affichant les résultats à l'aide de courbes

Il est possible de changer la couleur de ces courbes en cliquant sur un sélecteur de couleur (figure 8) ou encore d'afficher/cacher certaines courbes (figure 9). Le fait de cacher la courbe possédant la

valeur la plus élevée sur l'axe des ordonnées va mettre à jour l'échelle de cet axe. Il est ainsi possible de voir en détail une courbe ne présentant pas de valeurs élevées en cachant toutes les courbes possédant des valeurs plus élevées afin de réduire l'échelle utilisée et ainsi d'effectuer un zoom.

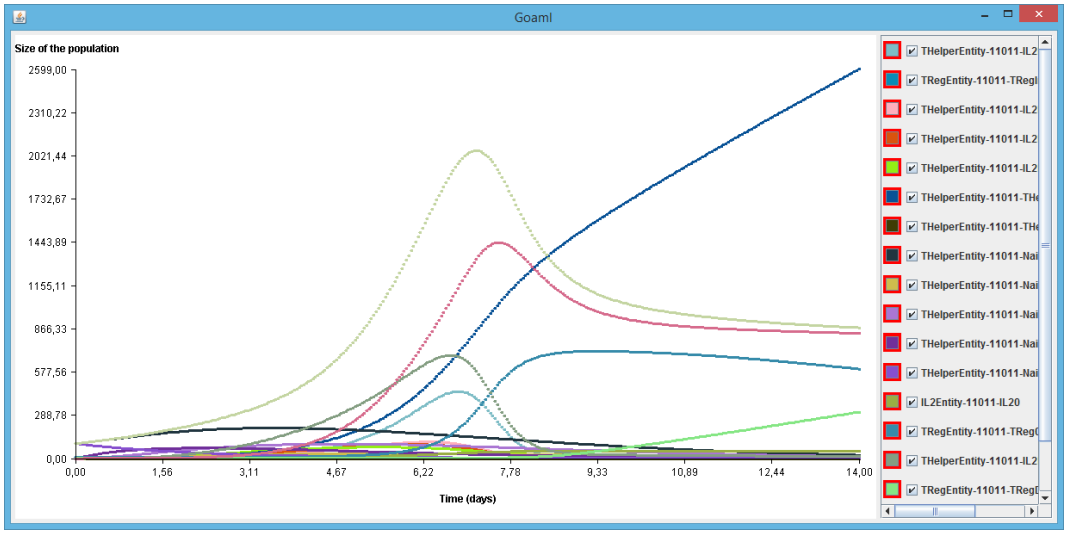

FIGURE 8 – Boutons pour changer la couleur d'une courbe

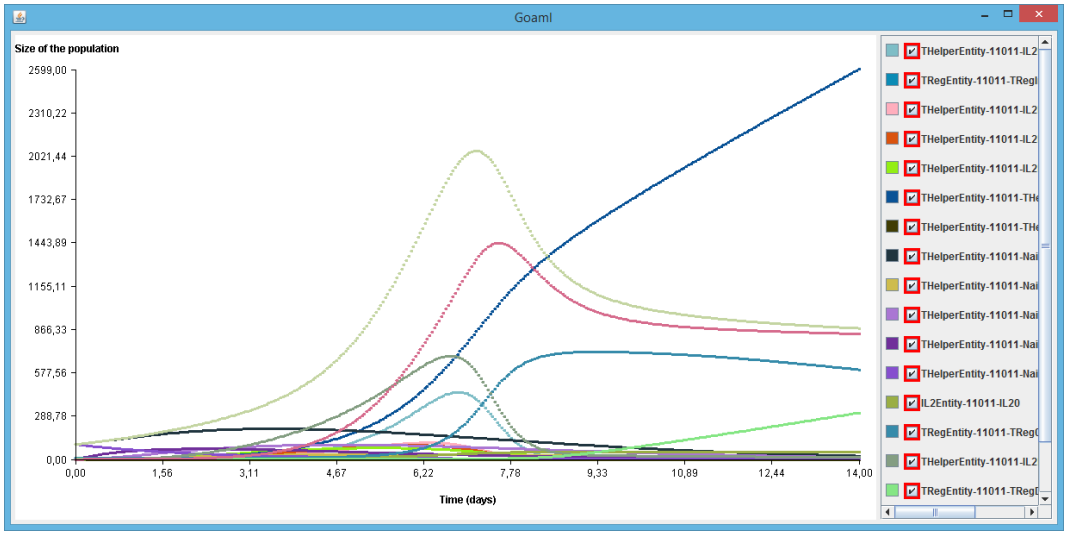

FIGURE 9 – Checkbox pour afficher/cacher une courbe

2 Manuel technique

Ce manuel a pour objectif de décrire le fonctionnement de GOAML et la façon dont le code Java a été pensé pour réaliser une simulation d'un système biologique à partir d'un fichier XML. La version Java utilisée est **JavaSE-1.8**.

La première section utilise des diagrammes de classe afin de décrire le programme java. Plusieurs diagrammes de classes ont été créés pour une meilleure visibilité, mais il y a toujours une classe que l'on retrouve d'un diagramme à l'autre pour faire le lien entre ceux-ci.

La deuxième section quant à elle décrira plus en détail chacune des classes et leur fonctionnement.

Cet manuel ayant pour objectif de décrire les principales classes Java de GOAML permettant de réaliser une simulation d'un modèle biologique, seules les classes présentes dans le package **goaml.lang** seront abordées dans ce manuel. Le package **goaml.mvc** quant à lui, contient les différentes classes utilisées pour la création de l'interface graphique, qui permet de rendre le programme plus facile à utiliser. Cependant du fait que ces classes n'ont aucun lien avec la simulation en elle-même, le fonctionnement de ces classes ne sera pas abordé dans ce manuel.

2.1 Modélisation

2.1.1 Diagramme de classe - Simulation

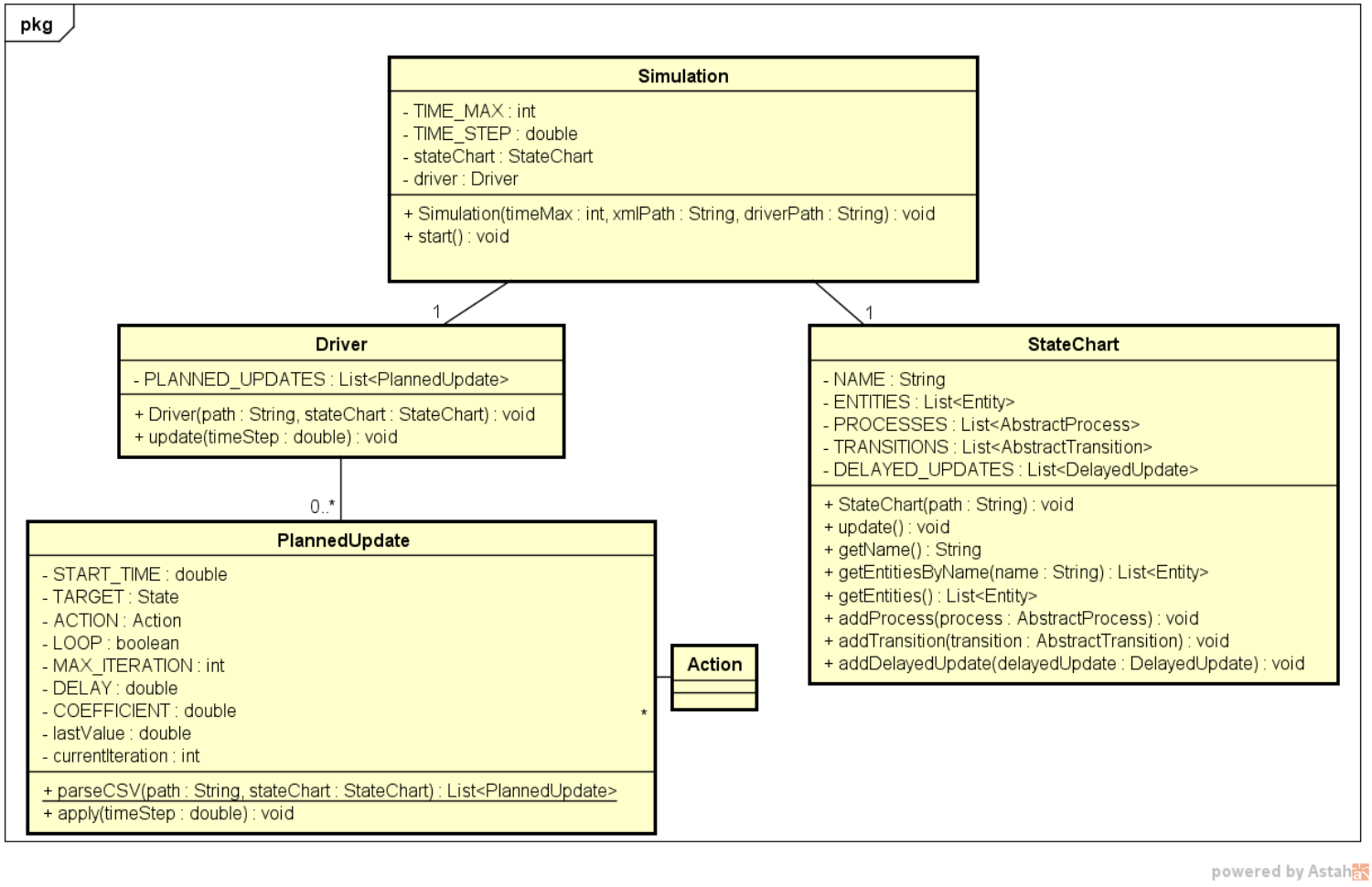

FIGURE 10 – Diagramme de classe - Simulation

2.1.2 Diagramme de classe - Entity

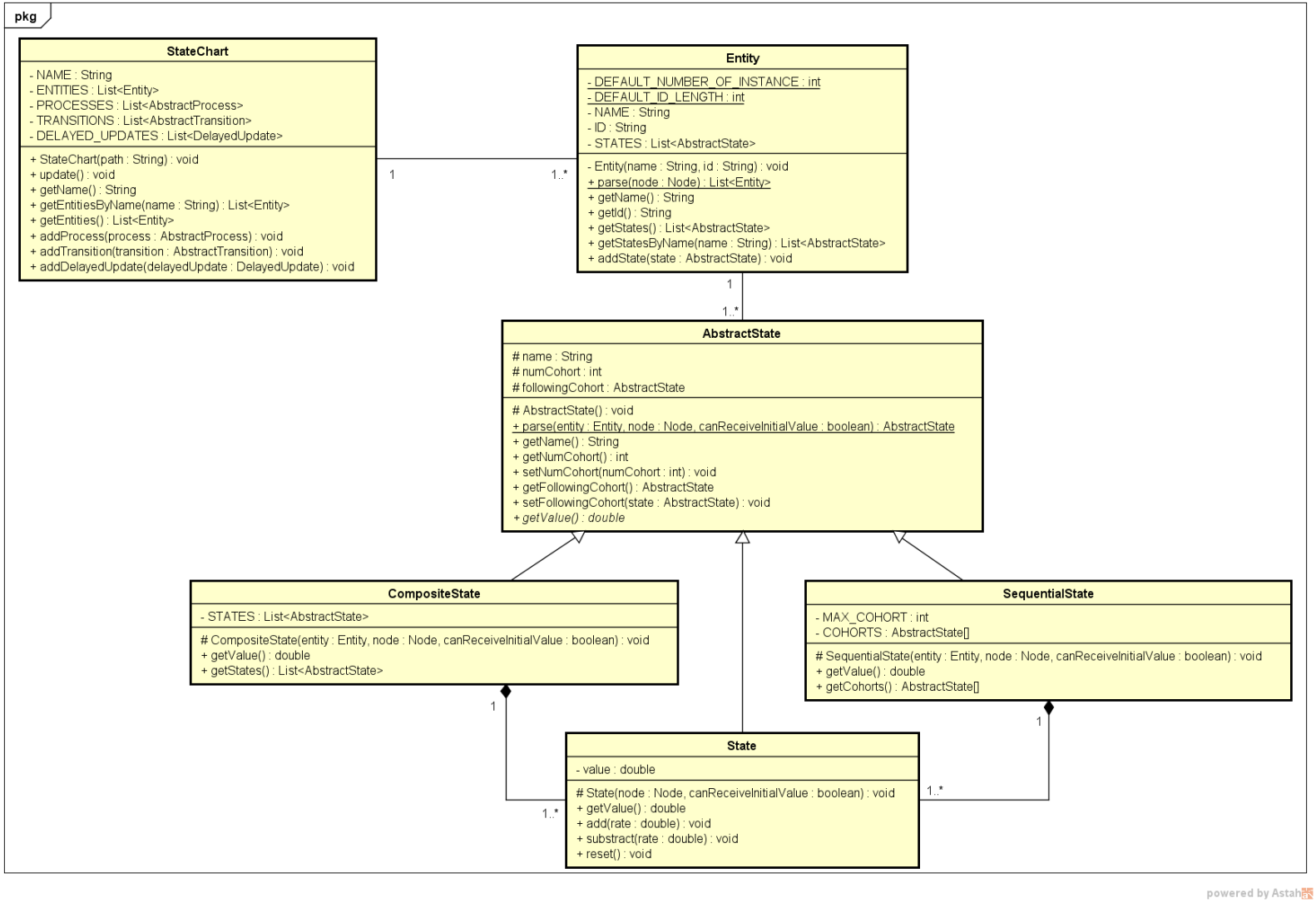

FIGURE 11 – Diagramme de classe - Entity

2.1.3 Diagramme de classe - AbstractProcess

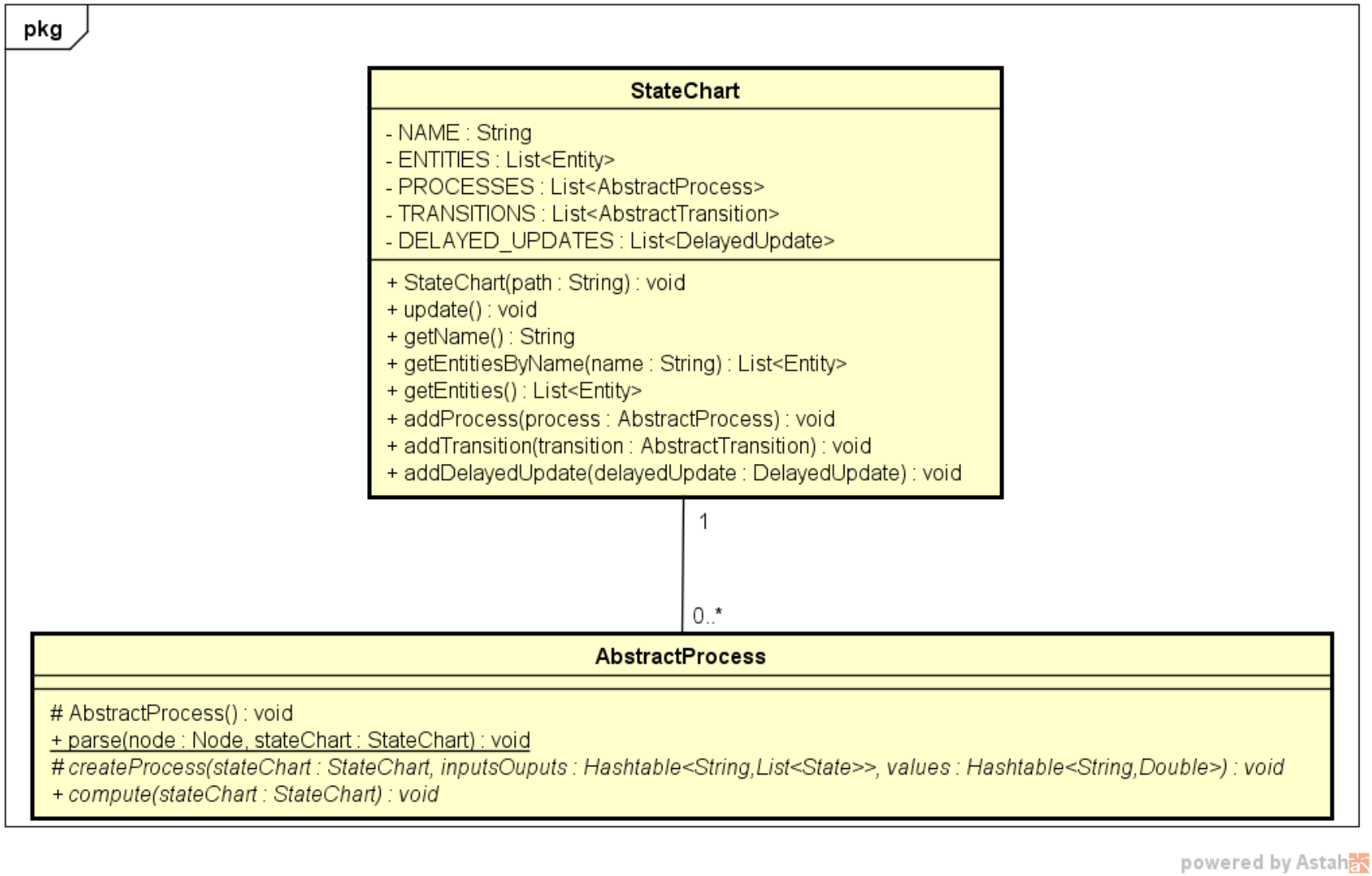

FIGURE 12 – Diagramme de classe - AbstractProcess

2.1.4 Diagramme de classe - AbstractTransition

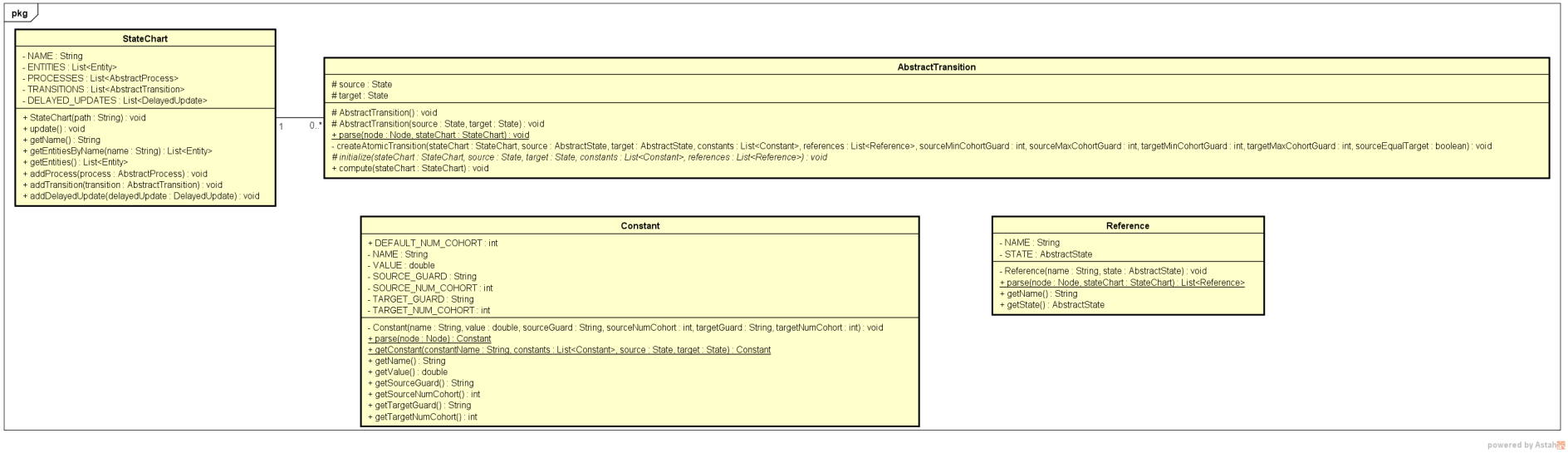

FIGURE 13 – Diagramme de classe - AbstractTransition

2.2 Les classes principales

Cette partie a pour objectif de décrire les principales classes Java de GOAML permettant de réaliser une simulation d'un modèle biologique, c'est à dire les classes qui sont présentes dans le package **goaml.lang**.

2.2.1 Simulation

Il s'agit de la classe principale du programme. Cette classe comme, son nom l'indique, permet de lancer une simulation à partir d'un fichier XML décrivant le modèle à simuler et optionnellement un fichier CSV, le driver, décrivant le protocole expérimental à utiliser (injections, etc.). Les résultats obtenus seront écrits dans un fichier CSV qui pourra par la suite être parsé afin d'afficher les résultats dans une interface graphique.

Les attributs

Nom	Type	Description
stateChart	StateChart	Correspond au StateChart utilisé (voir 2.2.2). Le StateChart contient les différentes entités, processus et transitions du modèle à simuler.
driver	Driver	Correspond au protocole expérimental utilisé lors de la simulation (voir 2.2.3).
TIME_MAX	int	Correspond à la durée de la simulation.
TIME_STEP	double	Correspond au pas utilisé lors de la simulation.

Table 1 – Les attributs de la classe « Simulation »

Les méthodes

Le constructeur de cette classe prend trois paramètres : la durée de la simulation (timeMax), le chemin vers le fichier XML décrivant le modèle à simuler (xmlPath) et le chemin vers le fichier CSV contenant le driver (driverPath).

La classe Simulation ne contient qu'une autre méthode, **start**, qui permet de lancer la simulation. Voici le fonctionnement de cette méthode :

1. Création d'un fichier CSV. Ce fichier est créé dans le même dossier que le fichier JAR permettant de lancer GOAML.
2. Initialisation du fichier CSV :
   (a) Ajout d'une ligne d'en-tête dans le fichier CSV contenant les noms des différents états du modèle.

(b) Ajout d'une ligne dans le fichier CSV contenant les valeurs initiales des états.

3. Pour chaque pas de la simulation :
   (a) Mise à jour des valeurs des états en utilisant le StateChart.
   (b) Mise à jour des valeurs des états en utilisant le Driver.
   (c) Ajout d'une ligne dans le fichier CSV contenant les nouvelles valeurs des états.

2.2.2 StateChart

La classe **StateChart** contient le modèle à simuler. C'est la classe Java représentant le fichier XML qui contient les diverses entités (et leurs états), les processus et les transitions du modèle. Cette classe est donc utilisée pour parser ce fichier et créer les divers éléments de la simulation.

Les attributs

Nom	Type	Description
NAME	String	Correspond au nom du modèle à simuler.
ENTITIES	List<Entity>	Liste contenant toutes les entités présentes dans le modèle (2.2.5).
PROCESSES	List <Abstract-Process>	Liste contenant tous les processus présents dans le modèle (2.2.11).
TRANSITIONS	List <Abstract-Transition>	Liste contenant toutes les transitions présentes dans le modèle (2.2.12).
DELAYED _UPDATES	List <Delaye-dUpdate>	Liste contenant toutes les instances de la classe « DelayedUpdate » (2.2.10).

TABLE 2 – Les attributs de la classe « StateChart »

Les méthodes

Le constructeur prend en paramètre une chaîne de caractères indiquant l'emplacement du fichier XML qui décrit le modèle à simuler. Il s'occupe également de parser ce fichier XML. Pour ce faire, toutes les balises « Entity » vont d'abord être récupérées et pour chacune d'entre elles, la méthode de classe **parse** de la classe **Entity** sera appelée afin de créer les différentes entités. Ces entités sont ensuite ajoutées dans la liste des entités du StateChart (attribut ENTITIES). Une démarche similaire est appliquée pour les balises « Process » et « Transition », la différence résidant dans le fait que ce sont les méthodes de classes **parse** des classes **AbstractProcess** et **AbstractTransition** qui seront appelées, et que ce sont ces méthodes qui s'occupent d'ajouter les processus/transitions au StateChart.

La classe **StateChart** possède plusieurs autres méthodes, la plupart étant des accesseurs ou des méthodes permettant d'ajouter des éléments aux différentes listes présentes dans cette classe. Il y a cependant une méthode très importante, **update**, qui permet de mettre à jour les états des entités présents dans la classe. Cette méthode est appelée dans la classe **Simulation** à chaque pas de la simulation. Voici le fonctionnement de cette méthode :

1. Appel de la méthode **compute** des différentes transitions afin de calculer les ratios à appliquer sur chacun des états des différentes entités.
2. Appel de la méthode **compute** des différents processus afin de calculer les ratios à appliquer sur chacun des états des différentes entités.

3. Appel de la méthode **apply** des différentes instances de DelayedUpdate qui auront été créées lors des deux premières étapes.
4. Suppression des instances de DelayedUpdate créées.

Les instances des sous-classes de AbstractProcess et de AbstractTransition ne modifient pas directement les valeurs des états auxquels elles sont liées. De cette manière, chacune de ces instances calcule les ratios à appliquer dans les mêmes conditions, et l'ordre dans lequel les calculs sont effectués n'a pas d'importance. Chacune de ces instances génère une ou plusieurs instances de DelayedUpdate en leur passant en paramètre l'état qui devra être modifié, l'action à effectuer et la valeur associée à cette action.

Ainsi, une fois que tout a été calculé, la méthode **apply** des différentes instances de DelayedUpdate est appelée, permettant de mettre à jour les différents états impliqués dans les processus et transitions.

2.2.3 Driver

La classe **Driver** est utilisée pour pouvoir simuler un protocole expérimental.

Les attributs

Nom	Type	Description
PLANNED _UPDATES	List <PlannedUpdate>	Liste contenant des PlannedUpdate (2.2.4).

TABLE 3 – Les attributs de la classe « Driver »

Les méthodes

Le constructeur prend en paramètre une chaîne de caractères indiquant l'emplacement du fichier CSV à parser et qui contient le protocole expérimental et il permet de créer des instances de **PlannedUpdate** à partir de ce fichier.

La classe **Driver** possède une seule méthode, **update**, qui se contente d'appliquer la méthode **apply** des différentes instances de PlannedUpdate créées lors du parsage du fichier CSV dans le constructeur.

2.2.4 PlannedUpdate

Cette classe permet de modifier la valeur d'un état à un instant précis. Ainsi, il est possible de simuler une injection d'interleukine2 au bout de cinq heures. Mais il est également possible de l'utiliser pour réaliser une injection à un certain moment et de réitérer cette injection n fois, chacune de ces injections espacées de x timestep. Il est aussi possible de remettre à zéro la valeur d'un état ou de lui soustraire un certain montant. Le mot « impact » a été choisi pour décrire cette injection/extraction.

Les attributs

Nom	Type	Description
START_TIME	double	Correspond au timestep de la première application de l'« impact » .
TARGET	State	Correspond à l'instance de **State** ciblée par l'impact.
ACTION	Action	Correspond à l'action que doit réaliser l'impact. Les valeurs possibles sont l'ajout de population (**ADD**), la mise à zéro (**RESET**) et la suppression d'une partie de la population (**SUBSTRACT**).
LOOP	boolean	Indique si l'impact doit être réitéré après son application initiale.
MAX_ITERATION	int	Correspond au nombre d'itérations APRÈS LA PREMIÈRE APPLICATION. Cet attribut n'est pris en compte que si **LOOP** a pour valeur « true ».
DELAY	double	Correspond au délai (nombre de timestep) entre chacune des itérations. Cet attribut n'est pris en compte que si **LOOP** a pour valeur « true ».
COEFFICIENT	double	Correspond à la valeur avec laquelle **VALUE** sera multipliée lors de chacune des itérations. Une valeur de 1.0 permettra à **lastValue** d'être identique pour chacune des itérations. Cet attribut n'est pris en compte que si **LOOP** a pour valeur « true ».
lastValue	double	Correspond à la valeur associée à l'action. Dans le cas où l'action est **RESET**, cette valeur n'a pas d'importance, la population sera fixée à zéro dans tous les cas. Cette valeur est multipliée lors de chaque itération par **COEFFICIENT**.
currentIteration	int	Mémorise le nombre d'itérations déjà effectuées. Cet attribut n'est pris en compte que si **LOOP** a pour valeur « true ».

TABLE 4 – Les attributs les plus importants de la classe « PlannedUpdate »

Les méthodes

Cette classe possède deux méthodes :

La première méthode **parseCSV** est une méthode « static » qui permet de parser un fichier CSV afin de créer une liste de **PlannedUpdate**, et retourne ensuite cette liste. Les commentaires, représentés par des lignes commençant par un « # », ainsi que les lignes vides, sont ignorés lors du parsage du fichier CSV. Cette méthode prend en paramètre une chaîne de caractères indiquant l'emplacement du fichier à parser ainsi que le **StateChart** afin de pouvoir récupérer à partir de leur nom, les différentes entités concernées par les impacts présents dans le fichier.

La deuxième méthode **apply** ne possède qu'un paramètre, une variable de type double représentant le **timestep**. Cette fonction permet de modifier l'état concerné par l'instance de **PlannedUpdate** qui appelle cette fonction. Cette modification ne se fait que sous certaines conditions :

- SI **timestep** est égal à l'attribut **START_TIME** ;
- SINON SI :
  - **START_TIME** est déjà passé
  - ET que **LOOP** est égal à « true »
  - ET qu'il n'y a pas de nombre maximal d'itérations (**MAX_ITERATION** == -1) OU s'il y a un nombre maximal d'itérations (**MAX_ITERATION** >= 1) ET que ce nombre maximal d'itérations n'a pas encore été atteint (**currentIteration** < **MAX_ITERATION**)
  - ET que **timestep** est égal à **START_TIME** + x * **DELAY**

2.2.5 Entity

Objectifs

Cette classe permet de parser une balise XML afin de créer des entités. Une entité représente un objet biologique pouvant être dans plusieurs états. Par exemple, une entité « lymphocyte » peut être dans plusieurs états : naïf, effecteur, mémoire ou mort.

Une instance d'**Entity** possède donc une liste d'**AbstractState** regroupant tous les états possibles dans lesquels elle peut se trouver. Mais cette liste ne contient pas que les états « finaux » (**State**), elle contient également tous les macro-états (**CompositeState** et **SequentialState**). En effet, cette liste est utilisée par les processus, transitions et références. Ainsi, il est parfois souhaitable d'avoir une référence vers un **CompositeState** ou un **SequentialState** afin d'avoir accès à la valeur de celui-ci, qui est la somme de tous les sous-états. Cela évite de devoir créer une référence pour chacun des états finaux. De la même manière, pour une transition il peut être utile d'avoir une source ou une cible qui soit une instance de **CompositeState** ou de **SequentialState** afin de ne créer qu'une balise XML Transition dans le fichier XML, et non pas une balise pour chacune des transitions qui sera créée à partir et vers des états finaux.

Il peut exister plusieurs entités identiques dans une simulation, à ceci près que ces « clones » possèdent un identifiant différent (**ID**). Dans notre exemple, l'**ID** peut permettre de représenter le récepteur du lymphocyte. Ainsi, il est possible d'avoir dans une même simulation plusieurs lymphocytes mais avec des récepteurs différents. Cet identifiant pourrait être utilisé pour simuler une interaction entre le lymphocyte et une molécule. Une affinité serait alors calculée à l'aide d'une distance (distance de Hamming par exemple) entre l'attribut **ID** de l'entité « lymphocyte » et celui de l'entité « molécule ». Une affinité élevée générerait ainsi une forte interaction tandis qu'une petite affinité créerait au contraire une interaction faible voir nulle.

Les attributs

Nom	Type	Description
DEFAULT_NUMBER _OF_INSTANCE	int	Représente le nombre d'instances par défaut d'une entité présente dans le XML.
DEFAULT_ID _LENGTH	int	Représente la taille par défaut de l'ID d'une entité.
NAME	String	Correspond au nom de l'entité.
ID	String	Correspond à l'identifiant de l'entité. Il s'agit d'une chaîne de caractères contenant des 0 et des 1.
STATES	List <AbstractState>	Correspond à la liste des états dans lesquels peut se trouver l'entité.

Table 5 – Les attributs de la classe « Entity »

Les méthodes

La classe **Entity** possède plusieurs méthodes utiles au bon fonctionnement de GOAML.

La méthode **parse** prend en paramètre un objet de type **Node**, ce qui correspond à une balise XML, et s'occupe de parser cet objet. Elle s'occupe notamment de certaines vérifications comme le fait que la balise XML a bien pour nom « Entity ». Cette méthode récupère ensuite des informations contenues dans la balise comme le nom de l'entité à créer (**name**), le nombre d'instances qui doit être créé (**numberOfInstances**) et la taille de la chaîne de caractères représentant l'identifiant de l'entité (**idLength**). Ensuite, la méthode créée une ou plusieurs entités selon **numberOfInstances** en leur attribuant à chacun un identifiant unique, puis elle appelle la méthode **parse** de la classe **AbstractState** afin de créer les différents états présents dans l'entité. La méthode renvoie ensuite une liste contenant tous les entités créées. Ces entités sont donc strictement identiques si ce n'est leur identifiant qui diffère d'une entité à l'autre.

La méthode **getStates** prend en paramètre un chaîne de caractères. Cette méthode recherche dans la liste des états présents (**STATES**) de l'entité tous les états ayant pour nom la chaîne de caractères passée en paramètre et les renvoie dans une liste. Cette liste sera vide si aucun état n'a été trouvé.

Les autres méthodes permettent d'accéder aux attributs de l'entité (getters) ou d'ajouter un état à l'entité.

2.2.6 AbstractState

Objectifs

Cette classe abstraite permet de parser une balise XML afin de créer des AbstractState. Cette classe abstraite possède trois sous-classes :

— State (2.2.7)
— CompositeState (2.2.8)
— SequentialState (2.2.9)

Ces classes seront décrites plus en détail dans la suite de ce manuel.

Les attributs

Nom	Type	Description
name	String	Correspond au nom de l'état.
numCohort	int	Correspond au numéro de la cohorte de l'état. (voir SequentialState)
followingCohort	AbstractState	Correspond à l'AbstractState « frère » de l'état courant ayant **numCohort** égal au **numCohort** de l'état courant plus un s'il existe, correspond à l'état courant sinon. Cet attribut est notamment utilisé dans les transitions simulant la prolifération d'une cellule.

Table 6 – Les attributs de la classe « AbstractState »

Les méthodes

La méthode **parse** permet de parser un objet de type **Node** représentant une balise XML passé en paramètre. Selon le nom de cette balise, cette méthode va créer une instance de la classe correspondante, qui doit être une sous-classe de AbstractState. L'instance créée s'occupera elle-même de s'initialiser à partir de l'objet **Node**. Cette méthode va ensuite ajouter l'instance à l'entité possédant cette état. Finalement, la méthode retourne l'instance créée ce qui permettra notamment aux CompositeState et aux SequentialState de remplir leur propre liste de sous-états.

Les autres méthodes permettent d'accéder aux attributs de l'entité (getters) ou permettent d'affecter une valeur à un attribut (setters).

2.2.7 State

La classe **State** permet de créer un état simple, atomique, contrairement aux classes **CompositeState** (2.2.8) et **SequentialState** (2.2.9) qui contiennent chacune une liste de sous-états. Elle permet également de stocker le nombre d'individus de l'entité qui sont dans cet état.

Dans le fichier XML décrivant le modèle à simuler, la balise State ne doit pas contenir d'autres balises.

Les attributs

Nom	Type	Description
value	double	Correspond au nombre d'individus de l'entité qui sont dans cet état.

TABLE 7 – Les attributs de la classe « State »

La classe **State** dérivant de **AbstractState** (2.2.6), elle hérite donc également de tous les attributs de cette dernière.

Les méthodes

Le constructeur de la classe s'occupe d'initialiser les attributs de la classe ainsi que l'attribut **name** hérité de **AbstractState**. Cette classe possède également plusieurs méthodes liées à l'attribut **value** :

— **getValue** qui est un accesseur et permet donc de récupérer la valeur de l'attribut **value** ;
— **add** qui permet d'ajouter la valeur passée en paramètre à **value** ;
— **substract** qui permet de retirer un certain montant passé en paramètre à **value** ;
— **reset** qui permet de mettre la valeur de **value** à zéro.

2.2.8 CompositeState

La classe **CompositeState** permet de créer un macro-état contenant un ou plusieurs **AbstractState**. Lorsqu'on cherche à obtenir la valeur d'un **CompositeState**, celui-ci renvoie la somme des valeurs de chacun de ses sous-états **de manière récursive**. Ainsi si un des sous-états est un **CompositeState** ou un **SequentialState**, alors ce sous-état renverra la somme de ses propres sous-états.

Dans le fichier XML décrivant le modèle à simuler, la balise « CompositeState » doit envelopper une ou plusieurs autres balises représentant un état. Ces états peuvent être de n'importe quelle sous-classe de **AbstractState** : des **State** par exemple mais il est également possible que ce soit des **CompositeState** ou des **SequentialState**.

Les attributs

Nom	Type	Description
STATES	List <AbstractState>	Il s'agit d'une liste contenant tous les sous-états **direct** du **CompositeState**

Table 8 – Les attributs de la classe « CompositeState »

Les méthodes

Le constructeur de la classe s'occupe d'initialiser les attributs de la classe ainsi que l'attribut **name** hérité de **AbstractState** à partir d'une balise XML passée en paramètre. Le constructeur utilise également la méthode **parse** de la classe **AbstractState** (2.2.6) sur chacune des balises XML contenue dans celle passée en paramètre et représentant le **CompositeState**. Pour rappel, la méthode **parse** de la classe **AbstractState** permet d'ajouter automatiquement les états créés au StateChart, mais elle retourne également les états créés, ce qui permet au **CompositeState** de remplir l'attribut **STATES** représentant la liste de ses sous-états, et ainsi de connaître la valeur qu'il doit renvoyer lorsque sa méthode **getValue** est appelée.

La classe **CompositeState** possède deux autres méthodes, la première **getValue** renvoie la somme des valeurs de chacun de ses sous-états **de manière récursive**. La méthode **getStates** quant à elle est un accesseur qui permet d'accéder à la liste des cohortes présente dans le SequentialState à l'aide de l'attribut **STATES**.

2.2.9 SequentialState

Dans le fichier XML décrivant le modèle à simuler, la balise **SequentialState** doit envelopper **une unique balise** représentant un état. Cet état peut être de n'importe quelle sous-classe de **AbstractState** : un State par exemple mais il est également possible que ce soit un **CompositeState** ou un **SequentialState**.

La classe **SequentialState** permet de créer une séquence d'**AbstractState**, des cohortes, à partir de la balise XML enveloppée. Les cohortes sont donc identiques, elles possèdent toutes le même nom et la même structure à l'exception de deux attributs, **numCohort** et **followingCohort** hérités de la classe AbstractState. De la même manière que lorsqu'on cherche à obtenir la valeur d'un **CompositeState**, un **SequentialState** renvoie la somme des valeurs de chacun de ses sous-états **de manière récursive**.

La classe **SequentialState** est principalement utilisée pour la division cellulaire en générant plusieurs instances d'un même état, chaque état regroupant tous les individus ayant effectué le même nombre de divisions. Ainsi il est possible de savoir combien de cellules ont effectué une division, deux divisions, etc.

L'autre utilisation de cette classe est lorsqu'on souhaite simuler une transition se déroulant en un certain montant fixe de timestep. Ainsi si un timestep représente une heure, pour simuler une transition se déroulant en cinq heures, il est possible de créer une séquence de cinq **AbstractState**, et à chaque pas de la simulation, la population de chaque cohorte passe dans la cohorte suivante.

Il y a un point commun dans ces deux cas d'utilisation : dans le fichier XML, lorsque la balise qui sera utilisée pour la création des clones possède une valeur initiale (attribut « value »), seule la première cohorte devra être initialisée.

Tous les **State** (la seule classe possèdant un attribut « value ») qui ne sont pas un sous-état (direct ou non) d'un **SequentialState** peuvent avoir une valeur initiale étant donné qu'ils possèdent tous un numéro de cohorte égal à zéro. Dans le cas où un **SequentialState** est présent, seule la première cohorte doit être initialisée. Pour ce faire, il suffit d'utiliser un paramètre de type « boolean » dans le constructeur des **State** (canReceiveInitialValue). Si ce paramètre est égal à « false », alors l'attribut « value » ne sera pas utilisé pour initialiser l'état. Ainsi lors de la création des cohortes, ce paramètre ne sera égal à « true » que pour la première cohorte.

Cependant, un problème se pose si l'état est présent dans un **SequentialState**, lui-même présent dans un **SequentialState**. Le résultat obtenu est visible dans la figure 14. Le premier état de chacune des cohortes a été initialisé à l'aide de « value », même celui étant présent dans le SequentialState « Seq2 » de la deuxième cohorte d'un autre **SequentialState**, « Seq1 ».

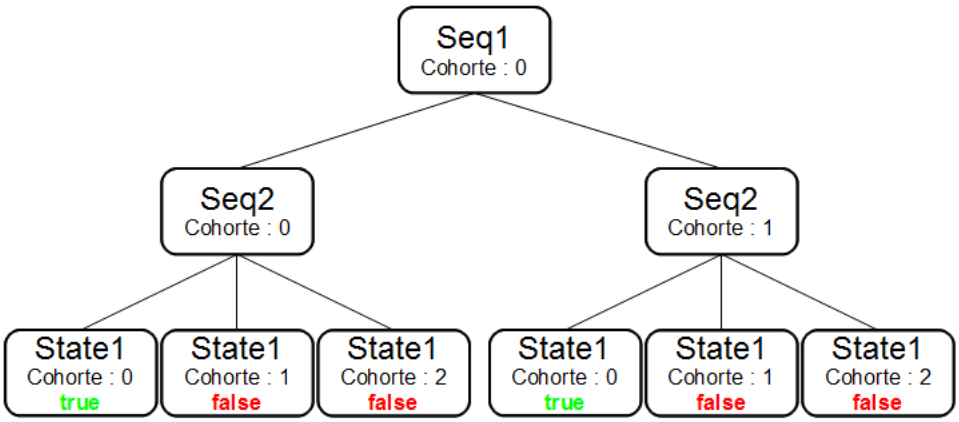

FIGURE 14 – Mauvaise initialisation de l'attribut « value »

Pour régler ce problème, il faut étendre le boolean « canReceiveInitialValue » aux constructeurs de tous les types d'**AbstractState** et modifier la condition permettant à un **SequentialState** de laisser la possibilité d'initialiser ou non une de ces cohortes. Dans la classe **Entity** (2.2.5), lors du parsage des états, ce boolean a la valeur « true ». Lorsqu'un **SequentialState** est créé et qu'il crée à son tour les différentes cohortes, il laisse la possibilité à la première cohorte de s'initialiser que si lui-même appartient à la première cohorte, c'est à dire s'il n'a pas de **SequentialState** au-dessus de lui dans la hiérarchie (parent direct ou non) ou si c'est la première cohorte d'un **SequentialState** parent, direct ou non (voir figure 15).

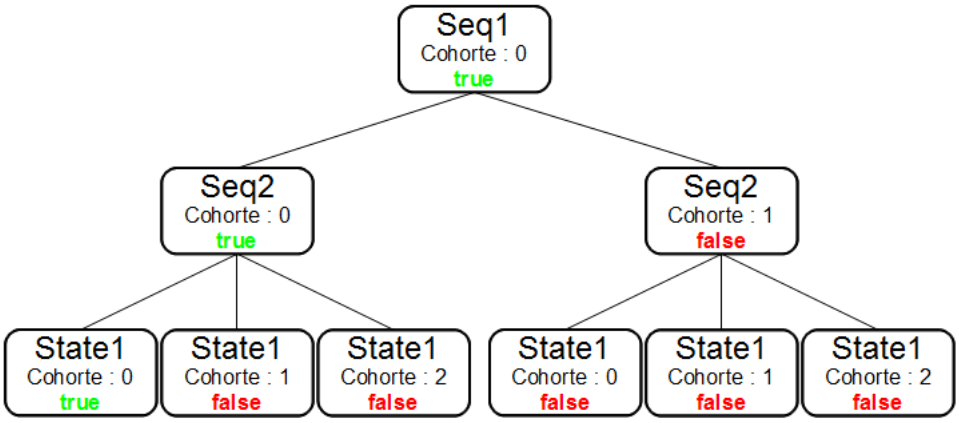

FIGURE 15 – Bonne initialisation de l'attribut « value »

Les attributs

Nom	Type	Description
MAX_COHORT	int	Correspond au nombre de clones qui doit être créé.
COHORTS	AbstractState[]	Il s'agit d'un tableau contenant toutes les différentes cohortes du SequentialState.

TABLE 9 – Les attributs de la classe « SequentialState »

Les méthodes

Le constructeur de la classe s'occupe d'initialiser les attributs de la classe ainsi que l'attribut **name** hérité de **SequentialState**. Il permet également de créer les sous-états du **SequentialState** en utilisant la méthode **parse** de la classe **AbstractState**. Ces états sont cependant identiques à peu de chose près : le numéro de cohorte **numCohort** qui est différent pour chacun d'entre eux, allant de zéro à **MAX_COHORT - 1**. De plus, l'attribut **followingCohort** hérité de la classe AbstractState de chacune des cohortes est défini à la cohorte suivante, sauf pour la dernière cohorte qui se réfère elle-même.

La classe **SequentialState** possède deux autres méthodes. La première, **getValue**, renvoie la somme des valeurs de chacun de ses sous-états **de manière récursive**, de la même manière qu'un **CompositeState** (2.2.8). La méthode **getStates** quant à elle permet d'accéder à la liste des cohortes présente dans le SequentialState (attribut **COHORTS**).

2.2.10 DelayedUpdate

La classe **DelayedUpdate** permet de mettre à jour les états qui sont liés à des processus (2.2.11) ou des transitions (2.2.12).

Sans cette classe, un processus (ou une transition) modifierait directement les états auxquels il est lié et les processus (ou transitions) suivants calculeraient leurs taux à partir de la nouvelle valeur de ces états. Ainsi l'ordre dans lequel les processus (ou transitions) serait appliqué aurait une influence sur la simulation.

L'objectif de cette classe est donc de retenir les actions à effectuer sur un état (ajout, soustraction) ainsi que la valeur associée, et de n'appliquer cette action que lorsque tous les processus et transitions ont fini de calculer les taux à appliquer. De cette manière, les calculs sont tous effectués dans les mêmes conditions.

Les attributs

Nom	Type	Description
STATE	State	Correspond à l'état qui devra être modifié.
ACTION	Action	Correspond à l'action à effectuer sur l'état (ajout, soustraction de population)
VALUE	double	Correspond à la valeur associée à l'action

Table 10 – Les attributs de la classe « SequentialState »

Les méthodes

Le constructeur de la classe ne fait qu'initialiser les attributs à partir des paramètres.

La classe DelayedUpdate ne possèdent qu'une seule autre méthode, **apply**, qui met à jour **STATE** en fonction de **ACTION** et de **VALUE**.

2.2.11 AbstractProcess

La classe **AbstractProcess** est une classe abstraite qui permet de créer des processus, comme par exemple la production d'interleukine2. Un processus peut avoir zéro ou plusieurs entrées, **input**, et zéro ou plusieurs sorties, **output**. Ces inputs/outputs ne sont liés qu'à des instances de **State**, les autres classes héritant de **AbstractState** ne peuvent pas être liées à un input ou un output. Plusieurs valeurs, des nombres réels intervenant dans le calcul des taux, peuvent également être liées à un processus.

Les attributs

La classe **AbstractProcess** ne possède pas d'attributs, ce qui n'est pas le cas des classes concrètes (non abstraite) qui en dérivent. Elle ne s'occupe que de parser une balise XML et de créer des instances des classes concrètes à l'aide de cette balise XML.

Les méthodes

La classe **AbstractProcess** possède trois méthodes, dont deux abstraites qui devront être redéfinies dans les sous-classes.

La première méthode abstraite est **createProcess**. Une classe héritant de AbstractProcess devra donc l'implémenter et cette méthode aura pour but de créer une ou plusieurs instances de cette classe concrète, et d'ajouter ces instances au StateChart.

La deuxième méthode abstraite, **compute**, permettra de calculer les montants à ajouter/soustraire aux divers états liés au processus et de créer des instances de DelayedUpdate (2.2.10) qui s'occuperont de mettre à jour les états une fois que tous les processus et transitions auront effectué leurs calculs.

La méthode non abstraite est **parse** et permet comme son nom l'indique de parser une balise XML afin de récupérer les différents inputs et outputs ainsi que les valeurs présentes dans la balise. Une fois cette étape faite, à l'aide du nom d'une des classes concrètes (héritant de **AbstractProcess**) contenu dans la balise, elle crée une instance de celle-ci. Elle appelle ensuite la méthode **createProcess** de cette instance afin de créer une ou plusieurs instances de cette classe concrète en lui passant en paramètres les inputs/outputs et les valeurs récupérés plus tôt ainsi que le StateChart afin que ces instances puissent s'ajouter au StateChart et être utilisées lorsque la simulation sera lancée.

Voici son fonctionnement global :

1. Vérification sur la balise (non nulle, etc.).
2. Création d'une Hastable **inputsOuputs** contenant les différents « input » et « output » du processus. La clé utilisée est le nom de l'input ou de l'output. La valeur associée à la clé est une liste des états associés.

3. Création d'une Hastable **values** contenant les différentes valeurs fournies par le fichier XML. La clé utilisée est le nom de la valeur, et la valeur associée est sa valeur.
4. Parsage de la balise XML
   (a) Récupération du nom de la sous-classe héritant d'AbstractProcess qui devra être créé (attribut **clazz**) ;
   (b) Pour chacune des balises contenues dans la balise à parser :
      — Si la balise correspond à un **input** ou un **output** :
         — Récupération du nom de l'input/output.
         — Récupération du nom de l'entité concernée et du nom de l'état concerné appartenant à l'entité.
         — Récupération des potentiels minCohortGuard et maxCohortGuard, et vérifications diverses sur ces derniers.
         — Création d'une liste qui contiendra tous les **State** réellement concernés par l'input/output
         — Récupération de tous les états de l'entité grâce au nom de l'entité et au nom de l'état récupéré plus tôt.
         — Vérification sur les numéros de cohortes de chacun de ces états avec **minCohortGuard** et **maxCohortGuard**, et ajout des états passant le test dans la liste créée plus tôt.
         — Ajout d'un élément dans la Hashtable **inputsOuputs** ayant pour clé le nom de l'input/output, et pour valeur associée la liste des états valides.
      — Si la balise correspond à une valeur, **value** :
         — Récupération du nom de la **value**.
         — Récupération de la valeur de la **value**.
         — Ajout d'un élément dans la Hashtable **values** ayant pour clé le nom de la **value**, et pour valeur associée la valeur récupérée à l'étape précédente.
   (c) Création d'une instance d'une classe dérivant de **AbstractProcess** à partir de l'attribut **clazz** ;
   (d) Appel de la méthode **createProcess** de cette classe car seule la classe connait les attributs qu'elle possède et sait comment s'initialiser. Cette méthode va créer une ou plusieurs instances de cette même classe qui seront ajoutées au StateChart ;

2.2.12 AbstractTransition

La classe **AbstractTransition** est une classe abstraite qui permet de créer des transitions. Une transition possède une et une seule source (**source**) ainsi qu'une et une seule cible (**target**). Il est important de noter qu'une transition ne s'effectue qu'au sein de la même entité. Ainsi, la source et la cible appartiennent obligatoirement à la même entité.

Les sous-classes héritant de **AbstractTransition** peuvent avoir des attributs supplémentaires, comme des constantes (2.2.13) ou des références (2.2.14).

Les attributs

Nom	Type	Description
source	State	Correspond à l'état source de la transition.
target	State	Correspond à l'état cible de la transition.

TABLE 11 – Les attributs de la classe « AbstractTransition »

Les méthodes

La classe **AbstractTransition** possède quatre méthodes, dont deux abstraites qui devront être redéfinies dans les sous-classes.

La première méthode abstraite est **initialize**. Une classe héritant de AbstractTransition devra donc l'implémenter et cette méthode aura pour but de créer une ou plusieurs instances de cette classe concrète, et d'ajouter ces instances au StateChart.

La deuxième méthode abstraite, **compute**, permettra de calculer le montant à soustraire à la source et celui à ajouter à la cible, puis de créer des instances de DelayedUpdate (2.2.10) qui s'occuperont de mettre à jour ces deux états une fois que tous les processus et transitions auront effectué leurs calculs.

La première méthode non abstraite est **parse**. Elle permet comme son nom l'indique de parser une balise XML afin de récupérer les différentes informations nécessaires à la création des transitions :

- la source et diverses informations sur elle (numéro de cohorte minimal/maximale) ;
- la cible et diverses informations sur elle (numéro de cohorte minimal/maximale) ;
- la liste des constantes (s'il y en a), obtenues en utilisant la méthode **parse** de la classe **Constant** ;
- la liste des références (s'il y en a), obtenues en utilisant la méthode **parse** de la classe **Reference**.

Une fois cette étape faite, elle crée une instance d'une des classes concrètes héritant de AbstractTransition à partir de l'attribut **class** contenu dans la balise XML, qui contient le nom de la classe concrète à utiliser pour la création de la transition. Cette instance ne sera pas ajoutée au StateChart, elle est simplement utilisée car elle connaît la manière qu'il faut utiliser pour créer les vraies transitions.

La méthode **parse** appelle ensuite la deuxième méthode non abstraite, **createAtomicTransition**, qui permet de créer plusieurs transitions dans le cas où la source est une instance de **CompositeState** ou de **SequentialState**, et/ou que la cible est une instance de **CompositeState** ou de **SequentialState**. Par exemple, si la source est un **CompositeState** contenant trois **State**, et que la cible est un **CompositeState** contenant deux **State**, alors il y aura six transitions créées :

— State-source-1 vers State-target-1
— State-source-1 vers State-target-2
— State-source-2 vers State-target-1
— State-source-2 vers State-target-2
— State-source-3 vers State-target-1
— State-source-3 vers State-target-2

La méthode **createAtomicTransition** appelle ensuite la méthode abstraite **initialize** décrite ci-dessus afin de créer les vraies transitions (instances d'une sous-classe concrète de AbstractTransition et qui seront ajoutées au StateChart).

2.2.13 Constant

Cette classe permet de parser une balise XML afin de créer des constantes. Une constante est utilisée dans les transitions lorsqu'une équation doit utiliser un certain taux **fixe et qui reste constant tout au long de la simulation**. Une transition peut posséder une ou plusieurs constantes.

Une balise **Constant** peut contenir une balise **sourceGuard**, ce qui correspond à une condition sur la source de la transition, ainsi qu'un **targetGuard** utilisé pour définir une condition sur la cible. Ces gardiens, ou « guards », **ne peuvent cibler qu'un état de type State**, les **CompositeState** ou les **SequentialState** ne peuvent pas être la cible d'un gardien. En effet, lorsqu'une balise XML Transition est parsée, plusieurs transitions sont créées et ajoutées au StateChart, chacune de ses transitions ayant obligatoirement pour source un état de type **State** et une cible de type **State**, même si dans le fichier XML les attributs « source » et « cible » d'une transition peuvent cibler un **CompositeState** ou un **SequentialState** (voir la méthode « parse » et « createAtomicTransition » de la classe AbstractTransition 2.2.12).

Les gardiens sont généralement utilisés lorsque la source d'une transition est un **CompositeState** ou un **SequentialState** et/ou que la cible est un **CompositeState** ou un **SequentialState**. Si une constante possède un sourceGuard sur un des State contenu dans le Composite, cette constante ne sera utilisée que par la transition dont la source est égale à l'état ciblée par le sourceGuard. Au contraire, si une constante est présente dans la balise Transition mais qu'elle ne possède ni sourceGuard, ni targetGuard, alors la constante sera utilisée pour chaque transition, exceptée s'il y existe des constantes plus spécifiques placées **avant** cette constante dans le fichier XML.

Il est également possible de cibler un état avec un numéro de cohorte particulier à l'aide de l'attribut **numCohortGuard** de la balise XML **sourceGuard** (idem pour targetGuard). Il faut cependant faire attention à un détail : lorsqu'un SequentialState crée des cohortes (voir 2.2.9), si le nombre utilisé pour décrire le nombre de clones qui doit être créé est **quatre**, les cohortes créées auront des numéros de cohorte allant de **zéro à trois** !

Un autre point important auquel il faut faire attention est l'ordre des constantes dans le fichier XML. Il faut en effet placer les constantes les plus précises en premier, c'est à dire celles possédant des gardiens sur un état et sur une cohorte, puis celles ne possédant que des gardiens sur un état, et enfin celles ne possédant aucun gardien.

De plus, dans le cas où une transition a pour source un état X et pour cible un état Y, et qu'il y deux constantes, la première possédant un **sourceGuard** sur X et avec une valeur V1, et la deuxième constante possédant un **targetGuard** sur Y et avec une valeur V2, un problème survient. En effet, la transition ne saura pas quelle valeur utiliser (V1 ou V2 ?), il faut donc créer une constante ayant un sourceGuard sur X et un targetGuard sur Y, et préciser la valeur à utiliser dans ce cas (V1, V2 ou une autre valeur). Cette constante devra également figurer avant les deux autres constantes dans le fichier XML.

Si une balise Transition cible un CompositeState ou un SequentialState possédant trois sous-états,

et qu'elle ne possède qu'une constante ayant un sourceGuard sur le premier des états, il n'y aura qu'une transition qui sera créée, celle ayant pour source le premier des états. En effet, étant donné qu'il n'y a pas de constantes de disponibles pour les autres états (par exemple une constance sans gardien ou alors deux constantes, la première ciblant le deuxième état, et la seconde constante le troisième état), il n'y aura pas de transition qui sera créée ayant pour source le deuxième ou le troisième état.

Les attributs

Nom	Type	Description
DEFAULT_NUM _COHORT	int	Attribut static et constant contenant la valeur par défaut des numéros de cohortes présent dans le sourceGuard et le targetGuard. Sa valeur est à -1.
NAME	String	Correspond au nom de la constante.
VALUE	double	Correspond à la valeur de la constante.
SOURCE_GUARD	String	Correspond au nom du State ciblé par le sourceGuard.
SOURCE_NUM _COHORT	int	Correspond au numéro de cohorte de l'état ciblé par le sourceGuard.
TARGET_GUARD	String	Correspond au nom du State ciblé par le targetGuard.
TARGET_NUM _COHORT	int	Correspond au numéro de cohorte de l'état ciblé par le targetGuard.

Table 12 – Les attributs de la classe « Constante »

Les méthodes

Le constructeur de cette classe ne fait qu'initialiser les attributs. Cette classe possède beaucoup de méthodes, la plupart étant des simples accesseurs permettant de récupérer les attributs de la classe.

Cette classe possède cependant deux méthodes importantes. La méthode, **parse**, est une méthode de classe (static) qui permet de parser un objet **Node** passé en paramètre et représentant une balise XML. Voici le fonctionnement de cette méthode :

1. Vérification sur la balise (balise non nulle et son nom est bien **Constant**) ;
2. Récupération du nom de la constante (**name**) ;
3. Récupération de la valeur de la constante ;
4. Pour chacune des balises contenues dans cette balise :
   — Si la sous-balise a pour nom « sourceGuard », alors le nom de l'état ciblé par le gardien sur la source ainsi que le numéro de cohorte associé, si celui-ci est présent, sont récupérés.
   — Si la sous-balise a pour nom « targetGuard », alors le nom de l'état ciblé par le gardien sur la cible ainsi que le numéro de cohorte associé, si celui-ci est présent, sont récupérés.
5. Une constante est créée avec les différentes valeurs récupérées puis retournée.

L'autre méthode importante est **getConstant** qui s'occupe de retourner la constante qui doit être utilisée en fonction d'un état source et d'un état cible passé en paramètre, parmi une liste de constantes passée en paramètre. Pour que cette fonction puisse renvoyer la bonne valeur, il faut veiller à ce que l'ordre des constantes dans le XML soit correcte (CF premiers paragraphes de cette section).

2.2.14 Reference

Cette classe permet de parser une balise XML afin de créer des références. Ces références sont utilisées dans les transitions lorsqu'une équation doit utiliser la valeur de la population d'un AbstractState. On retrouve notamment ce besoin pour la transition de « suppression » décrite dans le modèle d'Almeida [1]. Cette transition s'applique sur un lymphocyte T helper (THelper) et le fait passer d'un état « Effecteur » à un état « Mémoire » sous l'influence des lymphocytes T régulateurs (TReg). Il n'est pas possible d'utiliser une constante (2.2.13) car la population de TReg peut évoluer au cours de la simulation. La référence permet donc de créer un lien vers un état et permettre ainsi de connaître sa valeur actuelle.

Les attributs

Nom	Type	Description
NAME	String	Correspond au nom de la référence.
STATE	AbstractState	Correspond à l'AbstractState auquel la référence est liée.

Table 13 – Les attributs de la classe « Reference »

Les méthodes

Le constructeur de cette classe ne fait qu'initialiser les attributs. Cette classe possède trois méthodes dont deux servant à récupérer les attributs (getters), **getName** et **getState**.

La troisième méthode, **parse**, est une méthode de classe (static) qui permet de parser un objet **Node** passé en paramètre et représentant une balise XML. Voici le fonctionnement de cette méthode :

1. Vérification sur la balise (balise non nulle et son nom est bien **Reference**) ;
2. Récupération du nom de la référence (**name**) ;
3. Récupération du nom de l'entité contenant l'état sur lequel se porte la référence, puis récupération d'une liste (**entities**) contenant les entités possédant ce nom à l'aide du StateChart passé en paramètre ;
4. Récupération du nom de l'état ciblé (**stateName**) ;
5. Pour chacune des entités récupérées :
   (a) Pour chacun des états ayant pour nom la valeur contenue dans **stateName** et présents dans l'entité parcourue actuellement par la boucle :
      i. Création d'une instance de Reference ayant pour attribut **NAME** le nom récupéré (**name**), et pour attribut **STATE** l'état parcouru actuellement par la boucle ;

Ainsi pour une seule balise XML **Reference**, il y aura plusieurs références qui seront créées, chacune de ces références ayant le même nom mais ciblant des états différents (mais avec le même nom).

Bibliographie

[1] Afonso R.M. Almeida et al. « Quorum sensing in CD4+ T cell homeostasis : a hypothesis and a model. » In : *Frontiers in Immunology* 3.125 (2012). ISSN : 1664-3224. DOI : 10.3389/fimmu.2012.00125. URL : http://www.frontiersin.org/t_cell_biology/10.3389/fimmu.2012.00125/abstract (visité le 30/05/2016).