Développement de la model toolbox dans le cadre du projet européen CarboSeq à l'INRAE Centre Val de Loire

Eoghann Veaute

Rapport de stage Master 2 IMIS

Développement de la model toolbox dans le cadre du projet européen CarboSeq à l'INRAE Centre Val de Loire

Du 1 mai au 31 août 2023

Eoghann VEAUTE

Encadrants :

Manuel Martin

Rachid Yahiaoui

I - Remerciements

Avant de commencer ce rapport de stage, je souhaite remercier toutes les personnes avec qui j'ai pu travailler lors de cette expérience. Tout d'abord mes responsables Manuel Martin et Rachid Yahiaoui pour m'avoir suivi tout le long de ce stage, du recrutement jusqu'à la rédaction de ce rapport. Leurs accompagnement, connaissances et patience m'ont permis de pouvoir travailler dans des conditions favorables à l'apprentissage. Ensuite mes collègues, avec Thierry Gboho qui est aussi passé par le master IMIS, pour sa pédagogie et sa patience pour expliquer et aider à résoudre les problèmes des autres. Enfin, Benoît Miege, stagiaire, avec qui j'ai partagé ces derniers mois sur le même projet.

Pour finir, je tiens aussi à remercier l'ensemble des personnes travaillant dans l'unité Info&Sols ici à Orléans pour la convivialité que j'ai pu y trouver. Cela m'a permis de me sentir intégré tout de suite tout en ayant l'occasion de vivre des nouvelles expériences, notamment l'assemblée générale itinérante, de la meilleure des façons.

II - Lexique

INRAE	Institut national de recherche pour l'agriculture, l'alimentation et l'environnement
Site	Endroit où les sols et les systèmes de culture sont étudiés
COS/SOC	Carbone organique des sols
AIAL	Base de données de sites expérimentaux
Century	Modèle de dynamique de carbone organique des sols
DayCent	Modèle de dynamique de carbone organique des sols
RothC	Modèle de dynamique de carbone organique des sols
Usm	Unité de simulation, composée d'une association d'un sol, un système de culture et d'un climat
Reader	Classe effectuant le chargement des données
URI	Uniform Resource Identifier, identifiant d'une ressource par son nom dans un espace de nom
OWL	langage formel permettant la représentation structurée et logique des connaissances dans un domaine
Namespace	Préfixe unique utilisé pour identifier et différencier les termes et les entités au sein de l'ontologie
OBOE	The Extensible Observation Ontology, ontologie formelle pour capturer la sémantique des observations et mesures scientifiques
R6	Paradigme de programmation orientée objet
XMind	Logiciel pour réaliser des cartes mentales
JAR	Fichier ZIP utilisé pour distribuer un ensemble de classes Java
RDFS	Resource Description Framework Schema, langage extensible de représentation des connaissances appartenant au Web sémantique

III - Introduction

Mon stage s'est effectué au sein de l'unité Info&Sols à l'INRAE Val de Loire. Pour m'accompagner dans la partie métier, j'ai pu compter sur Manuel Martin. Pour la partie technique, Rachid Yahiaoui mais aussi Thierry Gboho (durant le premier mois) étaient présents pour répondre à mes questions et m'orienter quant aux choix de technologies.

Le sujet de ce stage était de continuer le développement de la model toolbox qui est un outil faisant partie du projet européen ***CarboSeq***. Ce projet a pour but de quantifier, étudier et simuler les possibilités de la séquestration dans le sol du carbone organique sur le continent européen. Pour permettre aux 24 partenaires européens de pouvoir lancer des simulations avec différents modèles de dynamique du carbone, un des points de ce projet est donc le développement d'un tel outil, la model toolbox. De ce fait, ces partenaires pourront calculer la cartographie du potentiel de la séquestration du carbone organique des sols au sein de leur pays. Le travail effectué peut se résumer en deux parties. La première se concentrait sur la continuation du développement de la model toolbox directement, incluant débogage et ajout de fonctionnalités. Ce travail, principalement effectué lors du premier mois avec l'aide de Thierry Gboho, avait également pour objectif que je puisse me familiariser avec le code de la model toolbox. Quant à la deuxième partie, elle était dédiée à la conception et l'intégration d'une ontologie dans le projet. Pour cela, j'ai travaillé avec Manuel Martin sur la partie métier, plus axée contenu de l'ontologie, et Rachid Yahiaoui sur la syntaxe et l'intégration de cette dernière dans la chaîne d'exécution de la model toolbox.

IV - Contexte

1. L'INRAE

L'Institut national de recherche pour l'agriculture, l'alimentation et l'environnement (INRAE), né le 1er janvier 2020, est issu de la fusion entre l'Institut national de la recherche agronomique (INRA) et l'Institut national de recherche en sciences et technologies pour l'environnement et l'agriculture (IRSTEA). L'INRAE est donc un organisme national dont le but est de réaliser des travaux de recherches scientifiques et technologiques au niveau national mais aussi européen. L'INRAE compte plus de 12 000 agents répartis dans 18 centres de recherche en France (voir *Figure 1*) ainsi que 14 départements scientifiques spécialisés dans des domaines variés (biologie, génétique, numérique, …). L'Institut fait aussi partie de 166 projets de recherche européens.

Figure 1 - Les centres de l'INRAE en France

L'INRAE Val de Loire est présent dans quatre villes de la région : Orléans, Tours, Bourges et Nogent-sur-Vernisson avec un effectif d'environ 800 agents (voir *Figure 2*).

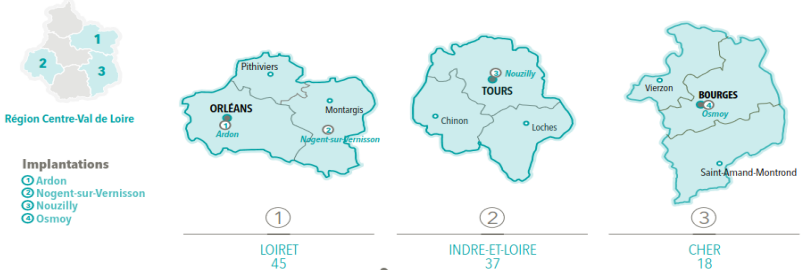


Figure 2 - L'INRAE Val de Loire

2. L'unité Info&Sols

Mon stage a été effectué dans l'unité Info&Sols (anciennement Infosol) sur le site d'Orléans. L'unité de recherche Infosol a été créée en 2000 puis a été renommée Info&Sols après sa fusion avec l'UR Sols au 1er janvier 2023.

L'UR Info&Sols est une unité propre de recherches du département AgroEcoSystem de l'INRAE. La mission principale confiée à l'UR Info&Sols est de développer des travaux de recherche en Science du Sol, sur l'évaluation quantitative du fonctionnement des sols en vue notamment de proposer divers paramètres mesurables ou quantifiables par des modèles et de définir des indices/indicateurs dans l'objectif de les positionner dans un référentiel. Ce besoin s'inscrit pleinement dans une demande générale des politiques publiques agricoles, environnementales et d'aménagement du territoire car tous les acteurs des sols réclament des indicateurs simples et accessibles, accompagnés d'une échelle de valeur pour appuyer leurs décisions dans la gestion des sols. Pour réaliser ses missions, l'unité travaille en interaction avec des réseaux de correspondants nationaux et internationaux. Une autre mission de l'UR est de constituer et de gérer des systèmes d'information environnementaux nationaux.

L'unité est constituée d'environ 70 agents. Elle ne comporte pas d'équipe, mais des pôles d'activité et fonctionne avec un conseil d'unité (consultations régulières entre chercheurs et ingénieurs), des assemblées générales régulières, une cellule QSE et un système de référents individualisés. Chaque agent participe au fonctionnement collectif de l'unité (Voir *Figure 3*).

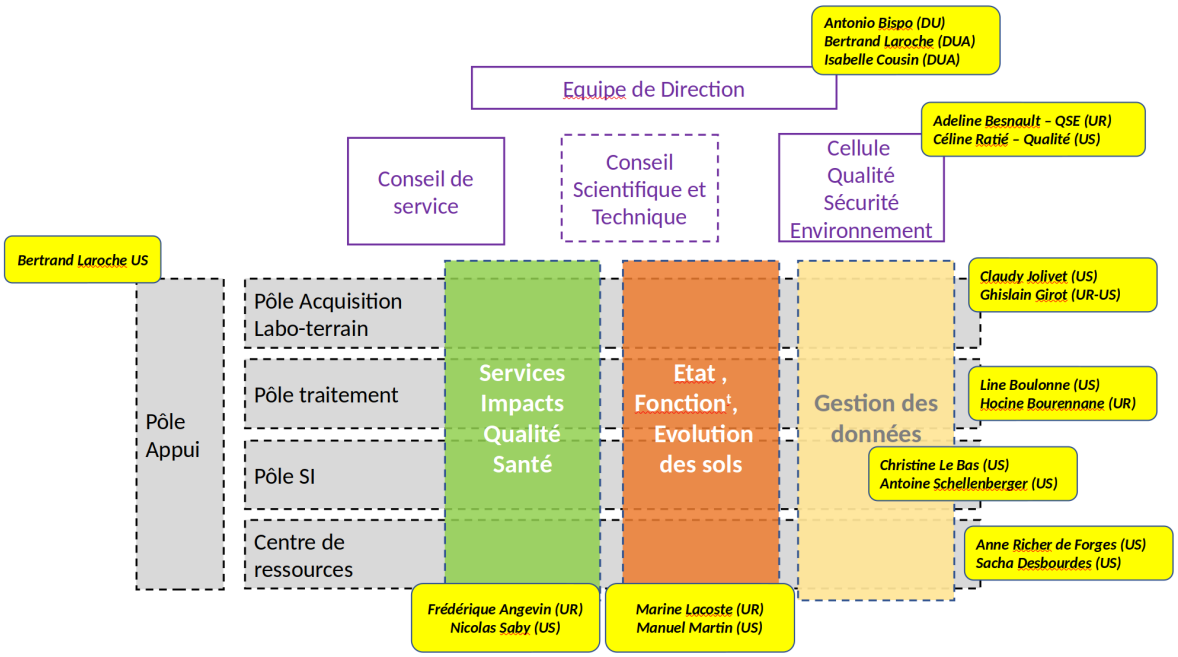


Figure 3 - Organigramme de l'unité Info&Sols

3. Projet et sujet de stage

Le sujet de mon stage vient donc s'insérer dans le projet ***CarboSeq***, qui fait partie des projets européens auxquels l'unité Info&Sols participe. L'unité est principalement impliquée dans le Work Package 9 nommé Modelling (voir *Figure 4*). Il met en œuvre les méthodes d'estimation du potentiel de séquestration du carbone organique dans les sols.

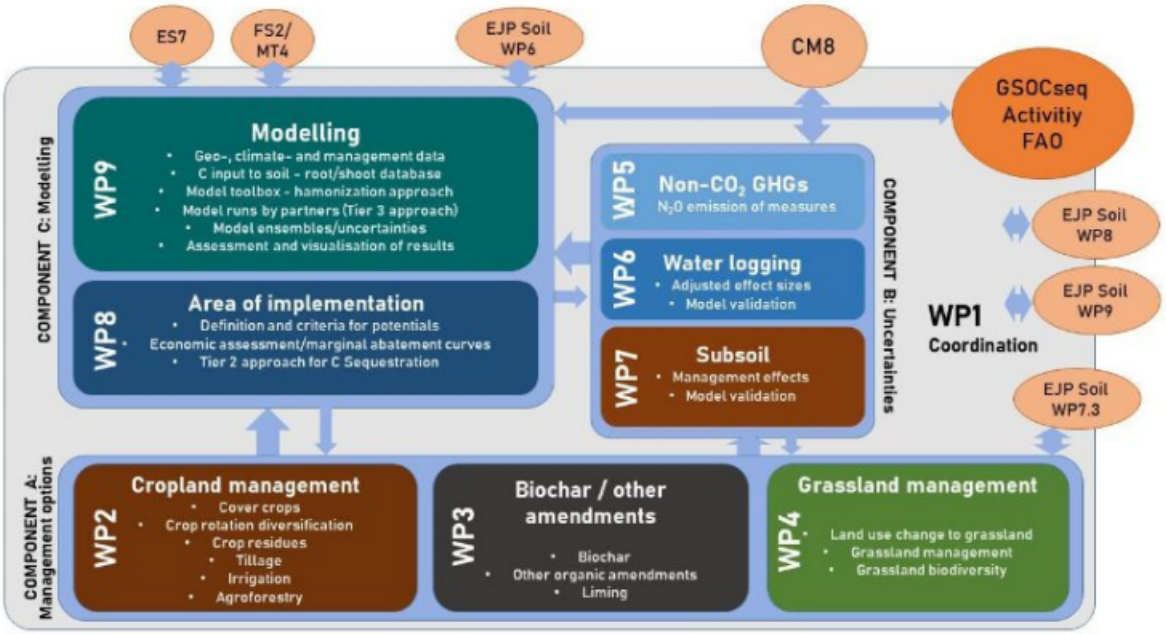


Figure 4 - Organisation du projet CarboSeq

Du fait de leur capacité à stocker une quantité importante de carbone, les sols jouent un rôle crucial dans le cycle global du carbone, ce qui leur confère une importance considérable en tant que réservoirs potentiels, pouvant ainsi induire une réduction considérable des niveaux atmosphériques de CO2 et contribuer activement à la lutte contre les changements climatiques. Cependant, la diversité des sols en Europe engendre des variations de leur capacité à absorber le carbone, une propriété étroitement liée à leur composition spécifique et à une série de critères, tels que les conditions climatiques, le couvert végétal et les activités humaines. Parmi ces dernières, les pratiques agricoles exercent une influence déterminante. Dans cette perspective, l'approche de ***Carboseq*** s'articule notamment autour de la simulation des variations de stock de carbone dans les sols pour une meilleure compréhension et gestion de ces enjeux.

L'approche basée sur la simulation implique l'utilisation de divers modèles de dynamique du carbone présents dans les sols, couvrant des intervalles de temps variés ainsi que des scénarios climatiques divers. Son objectif est de quantifier et de cartographier le potentiel général de séquestration du carbone organique en Europe.

Dans ce cadre, le modèle RothC a été sélectionné afin de mettre en place une preuve de concept et d'ensuite tester l'approche sur d'autres modèles. L'unité Info&Sols a donc pour responsabilité de développer la model toolbox. Un paquet R permettant l'exécution de ce modèle RothC, en anticipant aussi l'utilisation possible d'autres modèles (sous forme de scripts R) ultérieurement, afin de réaliser les estimations. La model toolbox doit donc être assez générique pour répondre à ce besoin mais aussi pour permettre d'utiliser plusieurs sources de données. Pour obtenir cette généricité, il faut se mettre d'accord sur les concepts utilisés dans la model toolbox de manière claire et définie. Cela fait partie du besoin d'améliorer l'interopérabilité de la model toolbox avec les différents jeux de données et modèles. Afin de répondre à tous ces besoins, il a été décidé de créer une ontologie pour ce projet, ce qui permettrait, entre autres, de valider les données en entrée et les transformer dans un format compatible avec la model toolbox peu importe le jeu de données.

V - Travail réalisé

1. Analyse de l'existant

1.1. Structure du projet

Dès mon arrivée, j'ai entrepris de comprendre la structure du projet. Ce dernier est constitué de quatre composantes principales, chacune remplissant une fonction spécifique : ***csopratools***, ***csopramapper***, ***csopralibs*** et ***modeltoolbox*** (voir *Figure 5*).

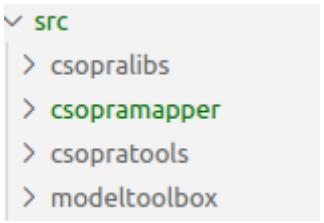


Figure 5 - Arborescence principale du projet

Chacune des quatre composantes correspond à un paquet R. Ces dossiers suivent tous la même structure, à l'exception de ***csopratools***. Tout d'abord le dossier ***inst*** composé des sous-dossiers ***extdata***, ***resources*** et ***run*** (voir *Figure 6*). Le premier dossier englobe non seulement les fichiers de données, mais aussi les fichiers de configuration qui interagissent avec des outils externes à la model toolbox. Ces configurations sont essentielles, car elles contribuent à façonner les résultats obtenus en utilisant ces outils, tels que les modèles. Ensuite, dans

resources, on trouve d'autres fichiers de configuration mais internes au projet. Enfin, le sous-dossier ***run*** regroupe les résultats de l'exécution.

Le dossier ***R*** comporte ensuite tous les fichiers R contenant les fonctionnalités du paquet. Ce dossier suit le patron d'architecture **Boundary-Control-Entity** (**BCE**) afin de bien séparer les responsabilités et de faciliter l'évolutivité et la compréhension du code. On y trouve aussi un sous-dossier ***utils***. A la racine, ensuite, se trouve un dossier ***tests*** qui contient tous les tests pour valider le fonctionnement du paquet. Finalement, il y a trois fichiers utils qui permettent chacun une exécution du programme différente. Le fichier ***src-utils.r*** est utilisé pour utiliser les fonctionnalités en mode développement, ***pkg-utils.r*** sert à la création du paquet R et ***test-utils.r***, quant à lui, permet l'appel aux différents tests présents dans le dossier ***tests***.

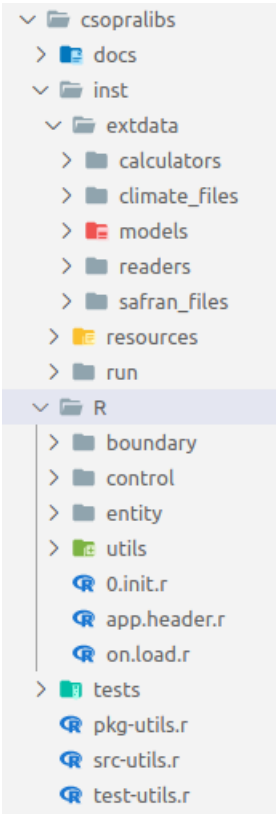


Figure 6 - Arborescence de csopralibs

1.2. Fonctionnalités des composants

1.2.1. modeltoolbox

Pour commencer, le paquet ***modeltoolbox***. C'est à travers ce paquet que les utilisateurs vont interagir. Ce paquet joue le rôle de "coordinateur central". C'est lui qui orchestre les interactions entre les différents paquets de l'application et assure la coordination des fonctionnalités en agissant comme point central d'échange et de gestion des flux d'informations. On peut voir sur le schéma ci-dessous (*Figure 7*) les

différentes intéractions entre les paquets ***modeltoolbox***, ***csopramapper***, et ***csopralibs*** ainsi que l'ordre des opérations dans la chaîne d'exécution.

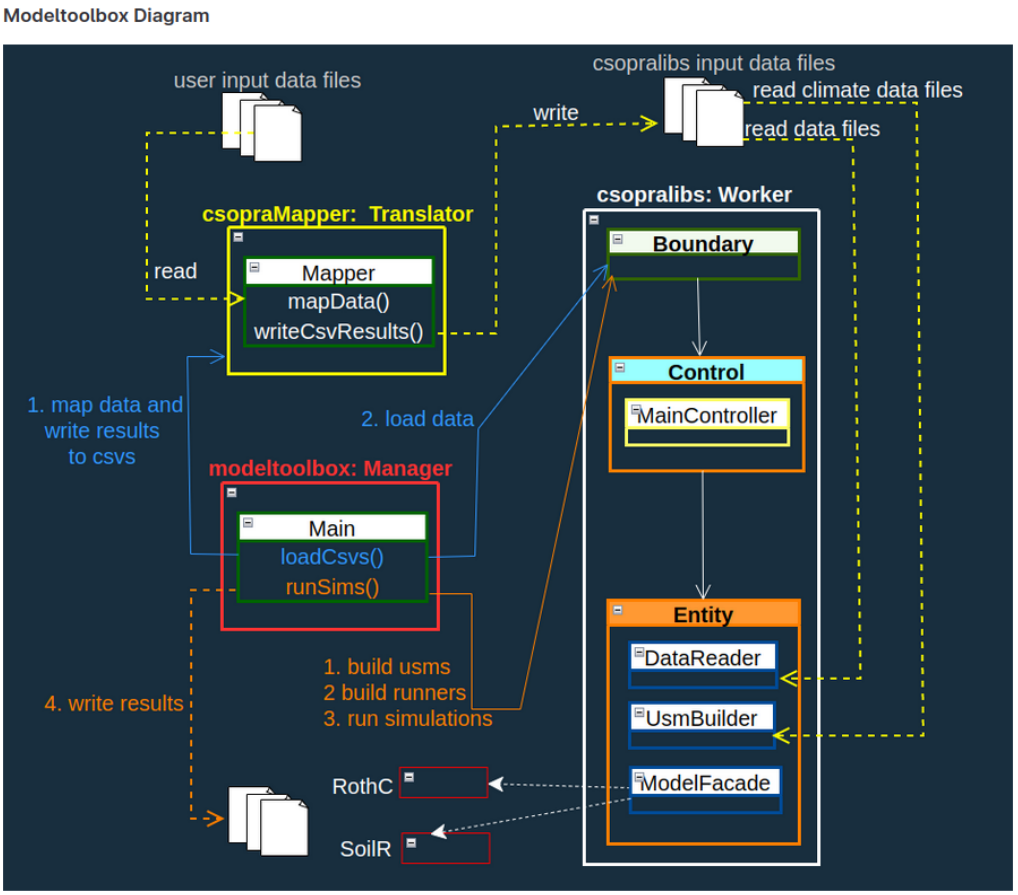


Figure 7 - Diagramme représentant la chaîne d'exécutions de la model toolbox (chargement des données et simulation)

1.2.2. csopratools

Ce paquet regroupe toutes les classes et fonctions utiles au stockage des données en session R. Il y a aussi des classes pour gérer quelques éléments utilitaires comme les logs ou encore la gestion du prompt quand les utilisateurs doivent donner des paramètres. Pour gérer le stockage des données, le paquet défini une classe ***CsopraData***. Les fonctions et la classe ***CsopraData*** de ***csopratools*** seront utilisées par les autres paquets pour tout ce qui touche à l'écriture ou au chargement des données.

1.2.3. csopramapper

Le paquet ***csopramapper*** est le premier appelé par la model toolbox. Sa fonction est de restructurer les données présentes dans les fichiers en entrée vers le format utilisé dans ***csopralibs***, peu importe la source de ces données. L'objectif est de garantir une absence de dépendance entre la model toolbox et le format des

fichiers soumis par l'utilisateur en entrée. Par exemple, dans le cadre du projet ***CarboSeq***, l'utilisateur fournira trois fichiers CSV en entrée : ***soil.csv***, ***crop.csv*** et ***meteo.csv*** (dans la suite de ce rapport, ces trois fichiers seront appelés fichiers CSV utilisateur). Avec un appel à la fonction ***runMapper***, on va générer un fichier météo par site dans le fichier ***meteo.csv*** et neuf autres fichiers CSV (appelés fichiers CSV csopralibs par la suite) à partir de ***soil.csv*** et ***crop.csv***.

1.2.4. csopralibs

Après l'appel à ***csopramapper***, c'est au tour de ***csopralibs*** d'intervenir. Cette partie représente en quelque sorte le cœur de la model toolbox. C'est ici que le chargement des données et la simulation vont se faire. Dans un premier temps, on va charger les données avec la fonction ***loadData*** qui va prendre en entrée les fichiers générés en sortie de ***csopramapper***. Une fois ce chargement terminé, il est maintenant possible d'exécuter la simulation avec ***runSimulations*** ce qui génère des fichiers résultats. Le paquet est décomposé en trois entités : le "reader" qui s'occupe du chargement des données, les "usm" qui regroupe les informations d'un site et d'une culture et enfin les "runner" qui prennent ces "usm" et appliquent les simulations dessus.

1.3. Programmation Orienté Objet

Les fichiers R représentent pour la plupart une classe. Cela est rendu possible par l'utilisation du paquet R6 pour mettre en œuvre le paradigme de programmation orientée objet R6. Grâce à leur structure, les classes R6 permettent de définir des méthodes spécifiques à chaque instance d'objet, facilitant ainsi la création de modèles de données plus complexes et de systèmes orientés objet robustes. Leur nature orientée objet améliore la lisibilité du code, facilite la maintenance et la réutilisation du code, ce qui les rend indispensables pour des projets R qui nécessitent une organisation sophistiquée et une gestion efficace des données et des comportements.

1.4. Généricité de la model toolbox

A mon arrivée, la model toolbox avait déjà été testée avec les modèles de Century et DayCent. Il y avait aussi une autre source de données que celle de ***CarboSeq*** nommé AIAL. Ces autres modèles et sources impliquent l'existence d'autres fichiers R leur permettant de pouvoir être pris en compte dans la model toolbox. Ces fichiers sont spécifiques à une source ou à un modèle, et ils héritent des classes génériques préexistantes correspondant à la fonctionnalité du fichier. Par exemple, on va avoir un fichier ***runner.r*** assez générique représentant l'objet utilisé pour lancer une simulation et un fichier ***rothc.runner.r*** qui va en hériter pour

l’utilisation du modèle RothC. Grâce à cela, peu importe pour quel modèle ou source les fichiers sont créés, ils possèdent tous une même base et vont simplement compléter le traitement à effectuer en fonction de leur spécialité. En procédant de cette manière, nous préservons la généricité, la maintenabilité et l’évolutivité du programme.

1.5. Analyse critique de l’existant

Ma première impression a été que le projet était assez imposant, il y a un bon nombre de fichiers de code et de configuration, parfois très longs. J’ai souvent eu du mal à me retrouver dans l’arborescence et à trouver les passages dans le code durant les premiers jours voire premières semaines. Cependant, au fil du temps, on s’aperçoit que ce grand nombre de fichiers sert un objectif précis car chacun des fichiers assume un rôle distinct et bien défini dans le fonctionnement global du programme. Cette pratique d’attribution des rôles permet donc après un temps d’adaptation, une clarté et une organisation de la structure du projet. Cela a aussi d’autres avantages comme la simplification de la maintenance ou la favorisation de la scalabilité. En effet, cette organisation permet des modifications ciblées et moins susceptibles d'introduire des erreurs en plus de permettre l'ajout de nouvelles fonctionnalités ou modules sans perturber la structure globale.

Je dirai aussi que le projet est bien élaboré. Il est assez bien documenté, bien que certaines documentations ne soient pas à jour. Chaque fonction possède une description détaillée. Ensuite, La gestion des erreurs est réalisée à l’aide d'exceptions. Il y a aussi un logger qui permet de tenir au courant de l’avancée de l’exécution. On y trouve aussi des tests qui permettent de comparer les résultats de la version actuelle à ceux d’une ancienne version agissant comme référence ce qui permet d’assurer la qualité des résultats et que le projet répond aux attentes.

Un des freins à la lisibilité du projet serait le fait d'interagir avec des données dont je ne comprenais pas forcément le sens, ce qui fait que certaines parties du code, notamment tout ce qui touche aux tests des valeurs, étaient compliquées à appréhender.

Un autre point négatif du projet est son utilisation des ressources et son temps d’exécution. Avec des jeux de données plus volumineux, il arrive que la mémoire (RAM) soit entièrement exploitée, ce qui entraîne une interruption de l’exécution du programme. Et dans le cas contraire, l’exécution peut durer plusieurs heures.

2. Analyse des besoins

2.1. Continuation de la model toolbox

Les attentes soumises au début de mon stage concernant le développement de la model toolbox concernaient dans un premier temps :

- des mesures de performances afin de donner un ordre d'idée du temps nécessaire pour une simulation aux partenaires européens du projet
- ajouter de nouvelles fonctionnalités, comme la lecture d'uniquement certaines parties du résultat d'une simulation
- Faire le parallélisme des fonctions importantes pour Windows
- Régler les bugs trouvés au cours des tâches précédentes

Il s'agissait également de profiter du dernier mois de présence de la personne précédemment en stage puis employée sur la model toolbox et de travailler avec cette dernière sur ces points mineurs afin de me familiariser avec le logiciel.

2.2. Intégration d'une ontologie à la model toolbox

2.2.1. Conception de l'ontologie pour la model toolbox

Ensuite, les discussions autour de la réalisation d'une ontologie, qui a constitué le cœur de mon stage, se sont accentuées. L'utilisation d'une ontologie est particulièrement prometteuse pour assurer la généricité de la model toolbox mais aussi son interopérabilité. En effet, cette ontologie servirait d'un référentiel commun partagé entre les différentes parties du projet et peut-être même avec d'autres systèmes à l'avenir, garantissant ainsi une compréhension mutuelle et cohérente des données et des interactions. Grâce à cette structure sémantique unifiée, les échanges d'informations deviendraient plus fluides et les intégrations entre composants hétérogènes seraient facilitées. Cela permettrait aussi aux partenaires de pouvoir partager des significations claires et définies pour les concepts utilisés dans la model toolbox.

Les utilisateurs soumettraient des jeux de données qui sont décrits avec des concepts spécifiques (labels), propres à leur domaine. Grâce à l'utilisation de l'ontologie, la model toolbox sera en mesure de saisir ces concepts et de les traiter de manière claire et sans ambiguïté. Par exemple, pour décrire le blé, la valeur dans les fichiers CSV utilisateur pourrait être “Blé”, “Wheat” ou encore “Grano” et il n'y aurait aucune ambiguïté possible lors de l'exécution car cette valeur serait remplacée par l'URI du concept qui correspond dans l'ontologie. De cette façon, on retire toutes les valeurs, représentant une concept, mises en dur dans le code qui sont utilisées lors des simulations ou pour des tests. Le but a été dans une première

phase de réfléchir à une syntaxe possible pour cette ontologie qui devra décrire tous les concepts utilisés dans le cadre du projet ***CarboSeq*** et plus précisément csopralibs (le nom du projet regroupant toute la model toolbox s'appelle aussi csopralibs, ici je ne parle donc pas du paquet).

La création de l'ontologie devait être réalisée en tenant compte de certaines contraintes spécifiques.Tout d'abord, les partenaires n'étant pas formés sur le domaine des ontologies, ils doivent avoir un moyen de la comprendre et pouvoir la modifier si besoin. Aussi, l'ontologie doit pouvoir être re-générée automatiquement après une modification.

2.2.2. Intégration de l'ontologie

Une fois la conception effectuée, il va falloir intégrer cette ontologie dans la model toolbox. Cette étape consistera à retirer toutes les valeurs textuelles, représentant un concept, écrites en dur dans le code (lors de l'accès aux colonnes par exemple) et à ajouter une étape dans la chaîne d'exécution où les fichiers CSV utilisateur vont passer une validation sémantique. Cette validation pourra entraîner la conversion d'unités de certaines colonnes, le changement des chaînes de caractères par l'URI (Uniform Resource Identifier) correspondant dans l'ontologie et des tests sur le type des valeurs dans les colonnes. Il va falloir trouver à quel endroit de la chaîne d'exécution de la model toolbox placer cette étape de validation sémantique.

3. Model toolbox

3.1. Introduction

Après avoir fini l'installation de la model toolbox sur mon poste de travail, j'ai entamé l'exécution de simulations afin d'observer les résultats. Initialement, j'ai employé de petits ensembles de données, mais lors de la transition vers des ensembles plus vastes, le programme a rencontré un plantage en raison d'une consommation excessive de mémoire RAM. En effet, lors du chargement des données, tout est gardé en session, ce qui entraîne une utilisation plus ou moins importante de la mémoire RAM en fonction de la taille de ces données. Certains fichiers peuvent dépasser les 100 000 lignes, par exemple pour 30 000 sites différents. Ce problème nécessite donc de réduire l'utilisation de la mémoire par la modification de la sortie lors du chargement des données, ce qui entraîne aussi une modification des usms, pour correctement récupérer les résultats.

3.2. Modification du reader

Le problème étant situé à la sortie du chargement des données, il fallait trouver un moyen de réduire l'utilisation de la mémoire RAM tout en gardant l'intégralité de données. Pour cela, nous avons décidé d'écrire ces données dans des fichiers. J'ai commencé par utiliser des fichiers RData pour stocker la sortie mais il a été décidé plus tard de les remplacer par des fichiers CSV afin d'avoir une meilleure visibilité (pouvoir les ouvrir par exemple). Un Usm est relié à un seul couple id/idSucc, où "id" représente l'identifiant d'un site et "idSucc" l'identifiant de la culture dans la succession des cultures du site. Le "reader" génère donc en sortie un fichier par couple ainsi qu'un autre fichier CSV regroupant toutes les paires id/idSucc (voir *Figure 8*) qui sera par la suite stocké en session.

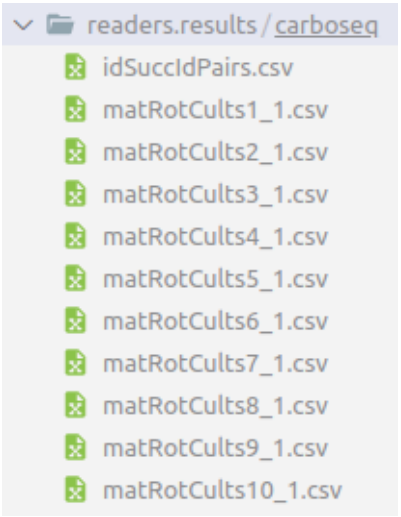


Figure 8 - Exemple de sortie du "reader"

Dans la Figure 8, il n'y a qu'une partie des fichiers générés. On peut y voir les fichiers pour le dataframe matRotCults mais il existe les mêmes pour les huit autres dataframes correspondants aux fichiers CSV csopralibs.

Dans le but d'obtenir uniquement les données pertinentes en sortie du "reader". Il faut maintenant nettoyer le répertoire à chaque exécution. Si les fichiers ne sont pas tous supprimés, cela pourrait engendrer une incohérence, car des données excédentaires pourraient être intégrées dans la simulation.

3.3. Mesures de performances

Suite à ces modifications, j'ai pu effectuer les mesures de performances demandées. L'objectif de ces mesures était de pouvoir informer les partenaires européens des temps d'exécution afin qu'ils aient un ordre d'idée pour les simulations qu'ils feront dans le futur. Ces tests ont fourni des métriques qui ont

permis de comparer les différentes versions de la model toolbox en terme de durée d'exécution ainsi que des ressources utilisées (mémoire). Pour réaliser ces mesures, la model toolbox possède une méthode ***timeRunSims***. Aussi, j'ai utilisé des jeux de données de taille croissante pour observer l'augmentation de l'utilisation de la mémoire RAM au fil des exécution et voir s'il existe une limite de taille. Enfin, la machine utilisée lors de ces mesures possède 16 coeurs et 16 Go de mémoire RAM.

```
utilisateur     système      écoulé
    366.781       1.162     367.978
utilisateur     système      écoulé
    378.014       1.053     380.186
```

Figure 9 - Temps d'exécution pour 290 sites en séquentiel
(avec au dessus l'ancien version et en dessous la version avec le nouveau reader)

```
utilisateur     système      écoulé
    557.538       2.980      82.770
utilisateur     système      écoulé
    742.078       5.004      83.420
```

Figure 10 - Temps d'exécution pour 290 sites avec 8 coeurs puis 15 coeurs
(avec la nouvelle version, pas de temps pour l'ancienne car provoquait un plantage)

Figure 11 - Etat de la machine avec l'ancienne version puis la nouvelle (15 coeurs)

J'ai aussi effectué des tests proches de situations réelles, c'est-à-dire sur des jeux de données de 2 000 puis 30 000 sites. Les mesures sur des plus petits jeux m'ont permis d'observer que l'augmentation du temps d'exécution est linéaire en fonction de la taille des données. Pour 2 000 sites, l'exécution dure environ 6 minutes 10, en totalisant le chargement et la simulation. Grâce à ce temps, j'ai pu estimer la durée pour 30 000 sites, qui serait d'environ 4 heures. Les partenaires étaient satisfaits avec cet ordre de temps.

3.4. Parallélisme

A mon arrivée, le parallélisme des principales fonctions était implémenté avec le paquet **parallel** et la méthode ***mclapply***. Cette méthode utilise le mécanisme de "forking" pour créer des processus enfants et les exécuter en parallèle. Les deux processus (parent et fils) partagent initialement le même espace mémoire et les mêmes descripteurs de fichiers (voir *Figure 12*).

```
idSuccIdPairsList <- parallel::mclapply(1:numberOfBlocks, function(thatblock) {

    leftPos <- (thatblock - 1) * blockSize
    ids <- lapply(1:blockSizes[[thatblock]], function(i) {
        return(usmIds[[i + leftPos]])
    })

    dataReader <- readerCtl$getDataReader()

    currentCsopraData <- csopratools::cloneCsopraDataRetainIds(csopraData = csopraData, ids = ids)

    dataReader$applyCInputRules(csopraData = currentCsopraData,
                                pid = thatblock,
                                inputParams = params)

    CsopLib$mainCtl$logInfo(paste0("Writing data for block ", thatblock, " ..."))
    readerCtl$writeCsopraDataFiles(currentCsopraData)
    return(currentCsopraData$getIdSuccIdPairs())

}, mc.cores = nbCores)
```

Figure 12 - Implémentation du parallélisme pour le chargement des données sous Linux

Inconvénient, elle n'est pas utilisable sur le système d'exploitation Windows qui lui utilise le "threading", un processus peut contenir plusieurs threads, chacun exécutant des parties spécifiques du code. Pour implémenter le parallélisme sous Windows, j'ai donc utilisé la méthode ***parLapply***. Cette implémentation se fait dans le code de ***modeltoolbox*** et appelle les méthodes séquentielles de ***csopramapper*** et ***csopralibs*** (voir *Figure 13*).

```
cl <- parallel::makeCluster(nbCores)
parallel::clusterExport(cl, c("readerInputs", "ids", "splitIndex", "rem", "sessionConfig", "csopraData"), envir=environment())
parallel::parLapply(cl, 1:nbCores, function(i) {
    if (i>rem){
        start <- rem * (splitIndex + 1) + (i - rem - 1) * splitIndex + 1
        end <- rem * (splitIndex + 1) + (i - rem) * splitIndex
    } else {
        start <- (i-1) * (splitIndex + 1) + 1
        end <- i * (splitIndex + 1)
    }

    csopralibs::setReaderOutputDir(readerInputs = readerInputs)
    csopratools::filterCsopraDataRetainIds(csopraData = csopraData,
                                           ids = ids[start:end])

    csopralibs::loadCsopraData(
        readerInputs = readerInputs,
        interactive = FALSE,
        csopraData = csopraData,
        pid = i
    )
})
parallel::stopCluster(cl)
```

Figure 13 - Implémentation du parallélisme pour le chargement des données sous Windows

On remarque que dans le cas de ***parLapply***, contrairement à ***mclapply***, il faut exporter les variables de l'environnement pour pouvoir les utiliser dans les threads.

4. Mise en place d'une ontologie

4.1. Introduction

Comme dit précédemment, il a été décidé de concevoir une ontologie pour le projet dans le but d'assurer la généricité et l'interopérabilité de la model toolbox et permettre à tous les partenaires européens de partager des concepts clairs et définis. Ces concepts concernent aussi les unités et les types qui vont être utilisés à l'intérieur de la model toolbox.

Les ontologies appartiennent au domaine du web sémantique qui est une vision du web visant à donner un sens et une signification aux données en ligne, permettant ainsi aux machines de comprendre et d'interpréter les informations. Il repose sur l'idée de structurer les données de manière à établir des relations sémantiques entre elles, ce qui facilite la recherche, l'analyse et la compréhension automatisée. L'utilisation d'ontologies, de langages sémantiques et de normes spécifiques permet d'enrichir le web en données structurées et cohérentes, ouvrant la voie à de nouvelles applications et à des interactions plus intelligentes entre les utilisateurs et les machines.

Une ontologie est donc un modèle de connaissances structuré et formel qui décrit les concepts, les relations et les propriétés d'un domaine particulier. C'est là que le format OWL entre en jeu. OWL, ou Web Ontology Language, est un langage standardisé pour créer et définir des ontologies. OWL offre une variété de constructions pour modéliser des relations hiérarchiques, des propriétés, des

contraintes et des règles d'inférence. Grâce à OWL, les ontologies peuvent être exprimées de manière précise, ce qui permet aux machines de raisonner sur les données et d'effectuer des tâches d'inférence automatique. Le format RDF (Ressource Description Framework) quant à lui, est utilisé pour représenter des données sous forme de graphes orientés. Il est principalement utilisé pour modéliser des informations sous forme de triplés sujet-prédicat-objet. RDF/XML est l'une des syntaxes utilisées pour représenter des données RDF. C'est une manière de codifier les triplés RDF dans un format XML. Afin de me former sur ces notions, j'ai suivi un MOOC réalisé par l'INRIA appelé "Web sémantique et Web de données".

Pour mettre en œuvre l'ontologie, il a fallu se mettre d'accord sur sa représentation. En effet, les partenaires doivent pouvoir comprendre l'ontologie, ils ne sont pas experts en web sémantique.

4.2. Création du graphe et syntaxe

Afin que l'ontologie soit compréhensible par tous, nous avons choisi d'utiliser un graphe réalisé avec le logiciel XMind pour la représenter. Le graphe élaboré à l'aide de Xmind étant une carte mentale, il présente des similitudes avec le format d'une ontologie en termes de structure hiérarchique et de relations. Il offre une visualisation claire et structurée des concepts présentés.

On y trouve au centre un nœud racine donnant un titre à l'ontologie. Ensuite, les nœuds forment une sorte d'arbre qui définit les concepts (classes) et leurs attributs. Cette hiérarchie omet des parties techniques implicites et cependant nécessaires pour générer le fichier OWL. Cela permet de faciliter la compréhension et la navigation au sein du graphe.

Nous avons développé une syntaxe dédiée pour représenter l'ontologie au format XMind, composé de fichiers XML dont les balises représentent la structure et le contenu de la carte mentale créée. Cette syntaxe surcharge les fichiers XML. L'ensemble est ensuite interprété par un autre outil que nous avons créé, appelé OWLifier, et qui génère l'ontologie au format OWL (sous sa version RDF/XML) à partir de ces données.

4.2.1. Syntaxe d'une classe et ses propriétés

La création de cette syntaxe a impliqué une collaboration entre la partie métier et la partie technique. En effet, cette dernière a subi plusieurs révisions au fil du temps, visant à affiner son contenu pour finalement aboutir à une version qui satisfait aux contraintes convenues par les deux parties. Au départ, la syntaxe était très simple mais à la suite de détails techniques, il a fallu l'enrichir afin de la rendre utilisable. Pour la partie métier, il a aussi fallu discuter avec les partenaires européens afin de déterminer le contenu à mettre et si la syntaxe choisie leur

convenait. Manuel Martin s'est occupé de cet aspect. Pour la partie technique, l'aide de Rachid Yahiaoui a été précieuse pour ses connaissances sur d'autres ontologies dont j'ai pu m'inspirer. Par exemple, pour représenter les concepts qui seront donnés par les utilisateurs, nous avons décidé de reprendre l'ontologie OBOE avec les classes "Observation" et "Measurement" pour définir le type de nos classes. La classe Observation va servir à définir les différentes colonnes des fichiers CSV fournis par les utilisateurs et la classe Measurement va apporter des précisions sur la nature de ces observations comme par exemple le type ou l'unité de cette dernière.

Les classes et propriétés sont définies en fonction du titre des nœuds et de règles d'écriture. Le nom d'un nœud représentant une classe commencera toujours par une majuscule, à l'inverse d'une propriété, qui commence par une minuscule. Aussi, le titre d'un nœud peut comporter plusieurs informations, séparées par le caractère ":". Ces informations peuvent aussi être précédées d'un nombre accompagné d'un "#". Cela permet d'indiquer que le concept ou la propriété provient d'une ontologie extérieure et de préciser laquelle.

Pour une classe, on retrouve le nom de cette dernière et optionnellement son type. Par exemple le noeud "1#Observation:CarbonInputObs" indique que l'on va créer une classe nommée CarbonInputObs de type Observation, où Observation est un concept défini dans une ontologie extérieure identifiée à l'aide du chiffre 1 dans notre cas (voir *Figure 14*).

Dans certains cas, les noms des classes doivent suivre certaines règles supplémentaires afin d'assurer une certaine cohérence, comme le nommage des concepts. On peut observer que toutes les classes de type "Observation" finissent par "Obs". De même pour celles de type "Measurement" qui finissent en "Meas".

Les propriétés suivent le même principe. La partie avant les ":" représentent le nom de la propriété et celle d'après la plage de valeur. La plage de valeurs représente les types ou les classes que les valeurs associées à cette propriété peuvent avoir. Si le nom est manquant (donc pas de ":") alors un nom par défaut sera donné, "has" concaténé à la plage de valeur. Par exemple, "1#hasValue:real" indique que la propriété hasValue attend un nombre réel. On peut aussi citer le noeud ayant pour titre "1#ofCharacteristic:DepthRelativeToSurface" qui montre que la propriété ofCharacteristic prend un objet de type DepthRelativeToSurface (voir *Figure 14*).

Il existe aussi des propriétés implicites qui permettent de simplifier la lecture du graphe. D'abord la propriété hasContext de l'ontologie OBOE qui permet de lier deux Observations et ensuite la propriété hasMeasurement, provenant aussi d'OBOE, qui lie une Observation et un Measurement. Par exemple, la classe CarbonInputObs possède une propriété hasMeasurement dont la valeur est la

classe CMassMeas. Elle a également une propriété hasContext ayant pour valeur la classe SoilLayerObs. On retrouve le même schéma pour SoilLayerObs et les deux classes de type Measurement qu'elle inclut (voir *Figure 14*).

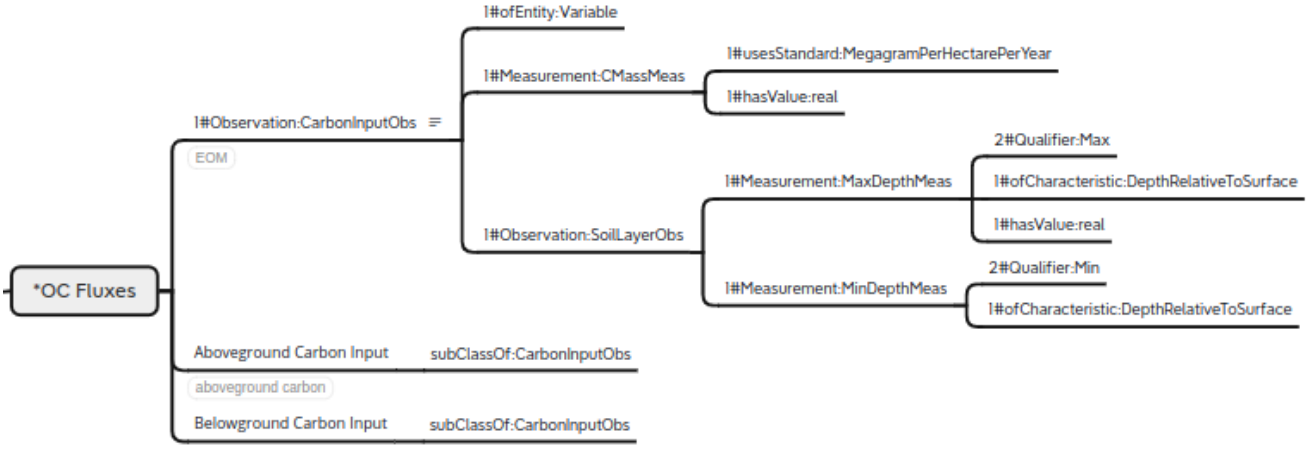


Figure 14 - Représentation de classes (concepts) dans le graphe

4.2.2. Gestion des types et unités

La model toolbox s'appuie sur l'ontologie pour identifier le type des colonnes à l'intérieur des fichiers fournis en entrée. Pour cela, on récupère cette information dans la propriété 1#hasValue des classes de type "Measurement" à l'intérieur des observations.

L'ontologie doit aussi contenir les informations concernant les unités des valeurs à utiliser dans les fichiers d'entrée de la model toolbox (fichiers CSV utilisateur). Dans cette optique, une section de l'ontologie est dédiée à la définition de ces informations. De plus, cette section comporte également d'autres unités considérées comme "acceptables", c'est-à-dire que les utilisateurs pourront utiliser ces unités à l'intérieur de leurs fichiers CSV en entrée. Ces unités autorisées sont explicitement énoncées dans l'ontologie donc aucune autre unité ne pourra être utilisée.

Afin d'établir un lien avec l'utilisation réelle dans le programme, il est essentiel de spécifier les conversions entre les unités utilisées dans la model toolbox et les unités acceptables. Un exemple concret est illustré par la classe "Centimeter". Celle-ci présente deux propriétés "hasConversion", chacune liée à deux objets de type "Conversion". Ces objets possèdent également deux attributs : "conversionTo" pour indiquer l'unité vers laquelle une conversion est effectuée, et "conversionFactor" pour définir le coefficient de conversion entre ces deux unités (voir *Figure 15*). Ceci permet de faire le lien entre les unités utilisées dans la model toolbox et les unités que les partenaires fournissent dans leurs fichiers d'entrée, tant que ces unités utilisées sont déclarées dans l'ontologie.

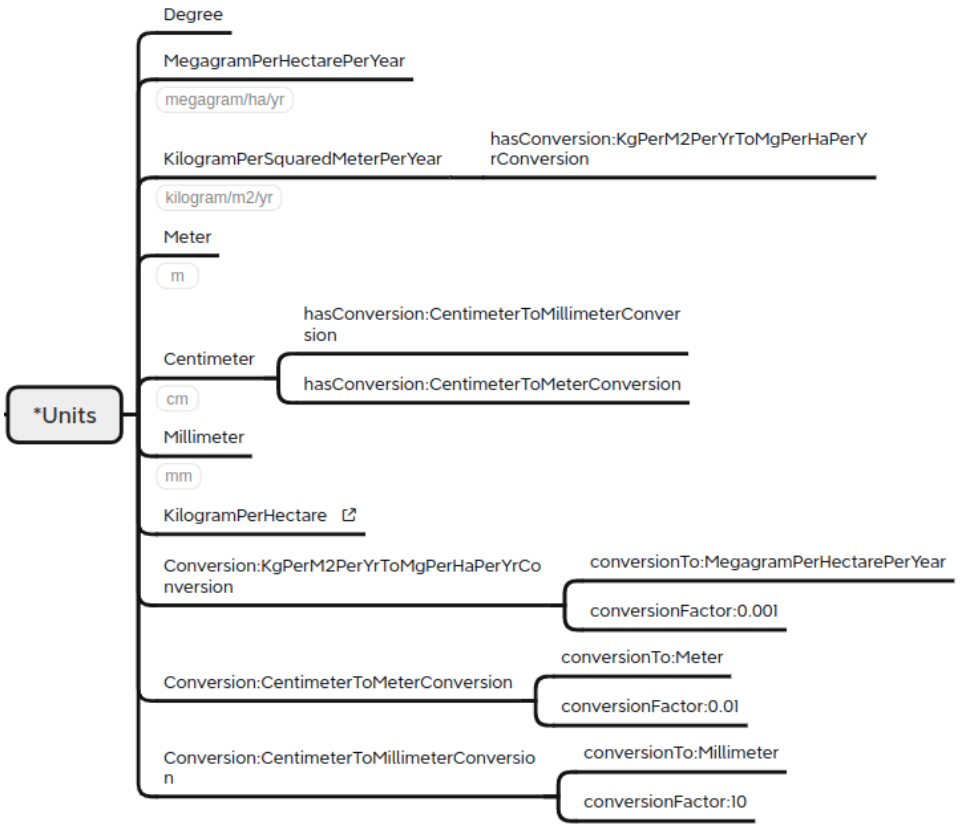


Figure 15 - Définition des unités dans le graphe

4.2.3. Contenu et utilité des classes

Afin de réunir assez d'informations sur les concepts de l'ontologie, les classes sur le graphe possèdent d'autres propriétés, qui ne sont pas représentées sous la forme de nœuds fils mais par des attributs sur le nœud.

On retrouve dans un premier temps les labels. Ce sont ces labels qui vont être utiles lors de l'exécution de la model toolbox en agissant comme identifiants pour les classes. Une classe peut posséder plusieurs labels mais un même label ne peut pas être dans deux classes à la fois. Ils vont permettre aux partenaires européens de pouvoir mettre les données en utilisant leurs propres noms, tant que ceux-ci sont renseignés dans l'ontologie. Par exemple, imaginons une classe représentant le blé, elle aurait un certain URI dans notre ontologie (http://csopralibs.org#Wheat). Maintenant, si on y ajoute les labels "Blé", "Wheat" ou encore "Grano", peu importe laquelle des trois valeurs se trouve dans le jeu de données, grâce aux labels de cette classe, on retrouvera toujours une référence au même URI et donc au même concept. Dans la *Figure 16*, on peut voir que la classe "Bovine solid manure" possède plusieurs labels : "DVB", "FYM-Qua", etc.

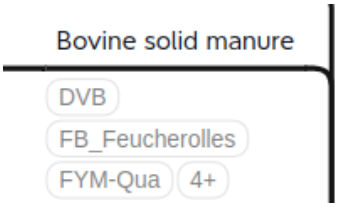


Figure 16 - Exemple de labels pour une classe dans l'ontologie

Il est également envisageable d'affirmer qu'une classe que nous avons créée est équivalente à une autre présente dans une ontologie externe, qu'elles ont la même signification sémantique. Pour réaliser cela, il suffit d'ajouter un lien au nœud ce qui établit une liaison entre les deux classes. Cela permet de pouvoir créer nos propres classes pour certains concepts tout en assurant que les informations associées aux deux classes sont équivalentes et donc quelles peuvent être échangées et intégrées de manière cohérente. C'est utile par exemple si le nom ou l'URI de la classe équivalente ne convient pas et que l'on veut le modifier. Il est aussi possible d'indiquer qu'un noeud est à ignorer grâce au caractère "*" devant son titre.

En outre, il est possible d'apporter des informations complémentaires à une classe en fournissant un commentaire, permettant ainsi de détailler davantage que son nom, en apportant une description du concept.

4.2.4. Liens vers les ontologies extérieures

Pour concevoir cette ontologie, nous avons utilisé certaines classes déjà existantes dans d'autres ontologies, sans avoir besoin d'y ajouter des informations. Par exemple, la classe Observation (entre autres) provient de l'ontologie OBOE. Il a donc fallu trouver un moyen d'identifier l'origine de ces classes extérieures tout en gardant un graphe lisible et compréhensible. Pour ce faire, un nombre suivi d'un "#" est présent devant chaque référence à un concept extérieur (voir *Figure 14* avec "Observation" ou "Qualifier"). S'il n'y a aucun nombre, alors le concept appartient à notre ontologie. Afin de donner un sens à ce nombre, une partie détachée du graphe principal a été ajoutée, représentant les namespaces (voir *Figure 17*).

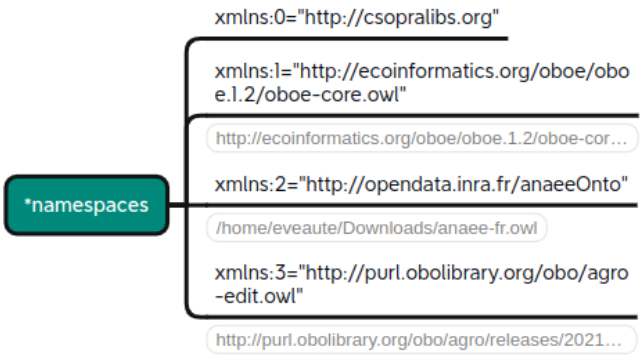


Figure 17 - Définition des namespaces

Ces namespaces permettent d'obtenir l'URI des concepts externes utilisés dans l'ontologie. Les namespaces externes possèdent aussi un label, indiquant où récupérer l'ontologie concernée (sur le web ou en local) afin de pouvoir l'importer dans la nôtre.

4.3. OWLifier

4.3.1. Introduction

OWLifier est le nom que nous avons donné à l'application Java permettant de générer l'ontologie dans un fichier OWL à partir du graphe Xmind. ***OWLifier*** se sert de la combinaison des balises des fichiers XMind et de la syntaxe choisie pour générer l'ontologie avec OWL au format RDF/XML de manière automatique. J'ai opté pour l'utilisation de Maven dans le cadre de cette application, afin de gérer efficacement les dépendances nécessaires et de créer le fichier JAR final. J'ai aussi utilisé la librairie ***picocli*** pour la gestion des arguments en ligne de commande, ***Apache Jena*** pour la création de l'ontologie et ***Junit*** pour la réalisation des tests unitaires. La documentation a été réalisée avec Javadoc.

L'utilisation d'***OWLifier*** n'appartient pas à la chaîne d'exécution de la model toolbox. ***OWLifier*** est appelé en amont, à chaque ajustement de l'ontologie. Étant donné que l'ontologie est prévue pour être stable une fois achevée, ou en tout cas sujette à de rares modifications, l'utilisation fréquente d'***OWLifier*** n'est pas anticipée. Son rôle se limite à établir le lien entre le graphe intelligible par les partenaires du projet et l'ontologie OWL, ce qui garantit une correspondance entre les deux.

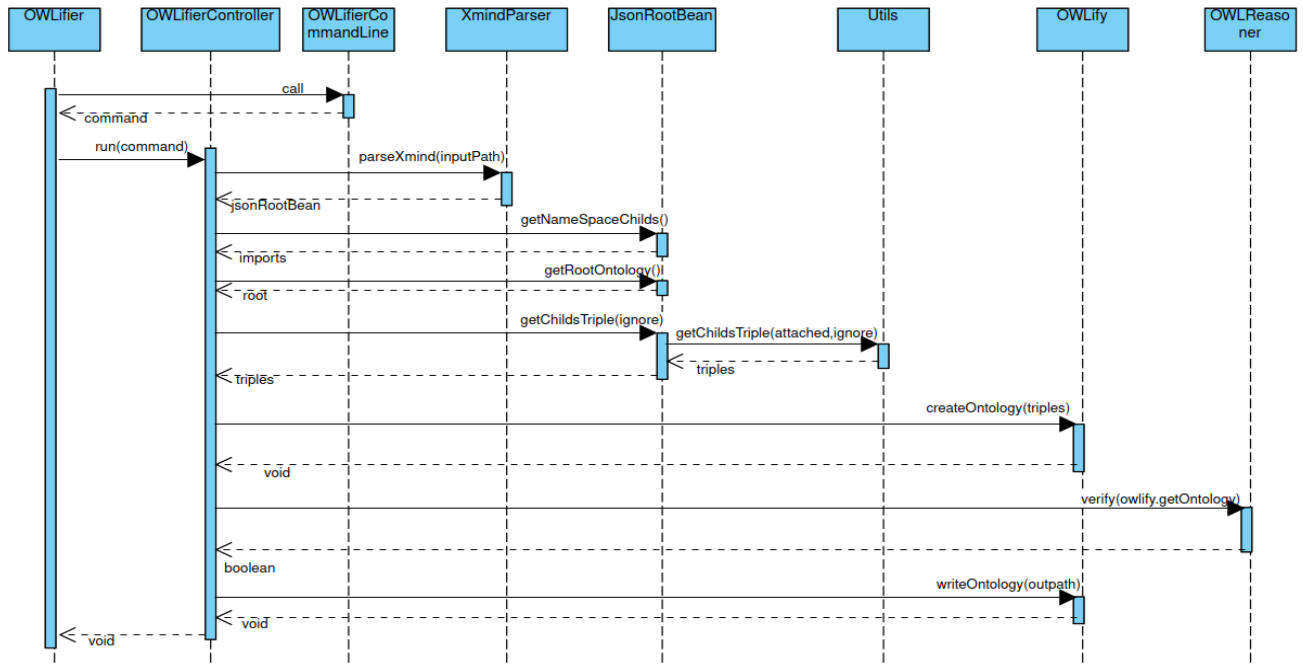


Figure 18 - Diagramme de séquence de OWLifier

4.3.2. Structure et paramètres

Pour ce programme, j'ai décidé d'utiliser un patron d'architecture efficace pour le développement d'une application Java, le patron **Boundary-Control-Entity** (**BCE**). En effet, ce patron offre une structure organisée qui sépare clairement les composantes de l'application en trois couches distinctes. Cette séparation facilite la gestion et la maintenance du code en isolant les responsabilités spécifiques de chaque couche. La couche Boundary gère l'interaction avec les utilisateurs, ici l'exécutable, la couche Control gère la logique métier et la couche Entity gère les données et les entités. Cette modularité renforce la cohésion, réduit les interdépendances et favorise la réutilisation du code. De plus, le patron Boundary-Control-Entity facilite la prise en compte d'éventuelles évolutions et améliorations de l'application, car les changements dans une couche n'ont que peu d'impact sur les autres.

L'application prend plusieurs paramètres. Le chemin vers le fichier d'entrée (le graphe Xmind ou un fichier JSON compatible) est obligatoire. La sauvegarde du JSON, le chemin du fichier de sortie ou encore l'activation des logs sont optionnels.

4.3.3. Du Xmind au JSON

La première étape à réaliser était de trouver comment transformer ce graphe en quelque chose d'utilisable en Java. L'option la plus adaptée était de le transformer en un fichier JSON ce qui permet de ne perdre aucune information et garder la structure en arbre. Pour ce faire, j'ai commencé par utiliser l'outil ***xmind2json***, un script python. Après discussions et malgré son bon fonctionnement, même si OWLifier se chargeait de faire cette transformation, cela obligeait à avoir ***xmind2json*** et donc Python d'installé sur la machine. Il a donc fallu trouver un autre moyen. Nous avons donc décidé de prendre un code open-source effectuant la transformation en JSON et de l'intégrer directement dans OWLifier. J'ai tout de même dû faire quelques modifications sur ce code afin de bien retrouver toutes les informations dont nous avons besoin et d'enlever celles qui n'étaient pas nécessaires (voir la *Figure 19* pour un aperçu du fichier JSON).

```
"children" : {
  "attached" : [ {
    "title" : "1#ofEntity:Variable"
  }, {
    "children" : {
      "attached" : [ {
        "title" : "1#usesStandard:MegagramPerHectarePerYear"
      }, {
        "title" : "1#hasValue:real"
      } ]
    },
    "title" : "1#Measurement:CMassMeas"
  }, {
    "children" : {
      "attached" : [ {
        "children" : {
          "attached" : [ {
            "title" : "2#Qualifier:Max"
          }, {
            "title" : "1#ofCharacteristic:DepthRelativeToSurface"
          }, {
            "title" : "1#hasValue:real"
          } ]
        },
        "title" : "1#Measurement:MaxDepthMeas"
      }, {
        "children" : {
          "attached" : [ {
            "title" : "2#Qualifier:Min"
          }, {
            "title" : "1#ofCharacteristic:DepthRelativeToSurface"
          } ]
        },
        "title" : "1#Measurement:MinDepthMeas"
      } ]
    },
    "labels" : [ "CROP" ],
    "title" : "1#Observation:SoilLayerObs"
  } ]
},
```


Figure 19 - Exemple de json généré

4.3.4. De JSON à OWL

4.3.4.1. Parser le JSON

Maintenant que le JSON est obtenu, il faut récupérer son contenu. J'ai pu effectuer cette action grâce à la librairie JSON ainsi qu'en créant des classes représentant les différents champs du fichier et une autre classe nommée ***JsonRootBean***, correspondant à l'entièreté de l'ontologie. Cette manière de faire permet d'obtenir les données du fichier JSON en un appel (voir *Figure 20*)

```
public static JsonRootBean parseJson(String jsonContent) throws DocumentException, ArchiveException, IOException {
    return JSON.parseObject( jsonContent, JsonRootBean.class);
}
```

Figure 20 - Méthode pour récupérer les données du JSON

Une fois les données récupérées, j'ai entrepris de transformer les informations qu'il contenait en triplés, conformément au mécanisme fondamental du web sémantique et de la modélisation ontologique. Ils constituent la base du web sémantique en capturant des relations entre des ressources sous la forme sujet-prédicat-objet. Par exemple, si on revient sur la *Figure 14*, on peut faire correspondre les différentes branches à des triplés : CMassMeas-hasValue-real ou encore AbovegroundCarbonInput-subClassOf-CarbonInputObs. Dans cette structure, le sujet représente l'entité principale à laquelle l'information est attribuée, le prédicat désigne la propriété ou la relation entre le sujet et l'objet, et l'objet

représente la valeur ou la cible de cette relation. Pour cela, j'ai créé une classe ***Triple*** qui contient le concept parent pour garder une trace de la structure et pouvoir ajouter les sur-classes si besoin, les labels, liens et commentaires potentiels en plus des trois informations précédentes. C'est à partir de ces triplés que l'on va créer nos classes et propriétés. Pour faire cette transformation, j'ai utilisé une méthode récursive afin de parcourir les données représentant le graphe en profondeur (voir *Figure 21*). De cette façon, en partant de la racine, on récupère tous les triplés.

```java
public static Triples getChildsTriple(Attached attachedNode , String ignore) throws InvalidNodeException {
    Triples triples = new Triples(new ArrayList<>(), new ArrayList<>());
    if (attachedNode.getChildren() == null) return triples;
    List<Attached> attachedList = attachedNode.getChildren().getAttached();
    if (attachedList == null) return triples;
    for (Attached attached : attachedList) {
        if (attached.getTitle().startsWith(ignore)) {
            triples.getIgnored().add(Utils.formatString(attached.getTitle()));
        }
        triples.getTriples().addAll(Utils.toTriple(attached, attachedNode.getTitle(), ignore));
        Triples rec = getChildsTriple(attached, ignore);
        triples.getTriples().addAll(rec.getTriples());
        triples.getIgnored().addAll(rec.getIgnored());
    }
    return triples;
}
```

Figure 21 - Méthode pour mettre les données du graphe sous forme de triplés

La méthode ***toTriple*** appelée va formater les données en triplés et appliquer des transformations telles que la suppression des espaces et la mise en majuscules des premières lettres dans les mots suivants un espace, afin de rendre l'URI ultérieurement créé valide.

Figure 22 - Triplés générés (à droite) à partir d'une branche du graphe (à gauche)

4.3.4.2. Générer l'ontologie

A partir des triplés, j'ai maintenant tout ce qu'il faut pour créer l'ontologie. Pour ce faire, j'ai utilisé la librairie Apache Jena. Cette librairie est conçue pour faciliter le développement, la manipulation et la gestion de données sémantiques et de graphes RDF. Jena offre de nombreuses fonctionnalités concernant la création, le stockage, le requêtage et la transformation de données RDF, ainsi que pour la mise

en œuvre d'ontologies et d'applications du web sémantique. J'ai donc créé une classe s'appelant ***OWLify*** qui est chargée d'initialiser cette ontologie et les classes qui la composent. ***OWLify*** possède donc une méthode ***createOntology*** qui va parcourir la liste des triplés et créer les classes et propriétés correspondantes tout en renseignant les informations supplémentaires. Pour ajouter plusieurs labels à une même classe, j'ai été amené à créer une nouvelle propriété "hasLabel" car la propriété standard dans le langage RDFS "rdfs:label", servant à attribuer un libellé à une ressource, est limitée à la prise en charge d'une seule valeur. Pour le reste, les liens sont transformés en une propriété owl:sameAs et les commentaires en rdfs:comment. L'un des aspects cruciaux lors de la création des classes et propriétés était d'assurer l'indépendance vis-à-vis de l'ordre de parcours des triplés. Par exemple, retournons sur la *Figure 15* pour prendre l'unité "Centimeter". Si lors du parcours, on arrive sur le triplé concernant cette unité mais que ceux correspondant aux classes dans les propriétés "hasConversion" n'ont pas encore été atteints, si on ne gère pas cette situation, une erreur va se produire car les deux classes de conversion n'existent pas encore. Pour empêcher ce problème de se produire, si on rencontre un objet qui n'existe pas encore dans notre triplé, alors on le crée sans informations. Cette approche repose sur l'idée que lorsqu'une méthode de Jena est invoquée pour créer une classe ou une propriété, elle renvoie l'objet correspondant si celui-ci existe déjà. Quand le parcours arrivera sur un triplé correspondant à une de ces deux classes, on va récupérer cette classe déjà créée et lui rajouter les propriétés et informations.

Afin de s'assurer que l'ontologie ne comporte aucune erreur syntaxique, j'ai créé une classe ***OWLReasoner*** possédant un objet Reasoner fourni par Jena. Grâce à elle, si l'ontologie comporte des erreurs, elles peuvent être affichées. Dans le cas où aucune erreur n'est détectée, alors on finit l'exécution de ***OWLifier*** par la génération du fichier OWL contenant l'ontologie créée précédemment (Un aperçu de l'ontologie générée se trouve en annexe).

5. Validation et Restructuration des fichiers d'entrée

5.1. Introduction

La prochaine étape du stage consistait maintenant à utiliser l'ontologie précédemment créée afin d'effectuer une validation sémantique sur les fichiers d'entrée de l'utilisateur de la model toolbox, c'est-à-dire les modifications nécessaires pour qu'ils correspondent aux concepts décrits dans l'ontologie. Ces modifications comprennent le remplacement des valeurs sous forme de chaîne de caractères en URI correspondant au concept lié dans l'ontologie, la vérification du type des valeurs dans les colonnes et la conversion des valeurs numériques dans les unités de la model toolbox. Il a aussi été décidé que la restructuration des fichiers d'entrée en fichier au format model toolbox se ferait avec cette application et non

plus dans ***csopramapper*** (voir paragraphe 1.2.3). ***Semantifier*** est le nom qui a été choisi pour l'application réalisant ces tâches et implémentée en Java.

5.2. Structure et paramètres

Semantifier implémente le même patron que ***OWLifier*** pour les mêmes raisons. La sortie de l'application est un dossier nommé après la source de données utilisée (carboseq ou aial) et qui possède deux sous-dossiers: input et output. Dans le premier, on retrouve les fichiers d'entrée modifiés (changements des labels et conversions). Le deuxième contient les fichiers CSV restructurés au format de ***csopralibs***. Aussi, ***Semantifier*** va générer un fichier JSON contenant tous les labels et leur URI correspondant dans l'ontologie, afin de faciliter l'intégration de cette dernière dans les paquets R.

Elle prend plusieurs paramètres. Le chemin vers les fichiers CSV utilisateur en entrée et le chemin de sortie sont obligatoires. Le nom de la source de données, le chemin vers l'ontologie et le nombre de lignes à vérifier prennent des valeurs par défaut. Afin de pouvoir travailler sur des fichiers CSV en Java, j'ai utilisé la librairie ***super-csv*** car elle permet contrairement aux autres librairies d'accéder aux valeurs à l'aide du nom de la colonne. J'ai aussi utilisé ***picocli*** et ***Junit*** comme dans ***OWLifier***. La documentation a été réalisée avec Javadoc.

5.3. Adaptation des fichiers à l'ontologie

5.3.1. Interactions avec l'ontologie

Afin de pouvoir utiliser les concepts et informations au sein de l'ontologie, il faut un moyen d'interagir avec. Pour cela, Apache Jena permet d'utiliser SPARQL, un langage de requête conçu spécifiquement pour interroger des données RDF dans le cadre du web sémantique. Il permet d'extraire des informations spécifiques à partir de graphes RDF en utilisant des motifs de triplés et des critères de recherche. J'ai donc créé une classe ***OWLOnto*** qui va gérer toutes les intéractions avec l'ontologie. Lorsqu'une requête SPARQL est soumise, le moteur de requête analyse la requête pour identifier les motifs de triplés spécifiés et recherche dans le graphe RDF les correspondances aux motifs. Le moteur de requête SPARQL parcourt le graphe RDF en cherchant les occurrences qui satisfont les motifs de triplés de la requête.

5.3.2. Conversion des unités

La première étape pour pouvoir faire une conversion est de connaître l'unité de départ et l'unité d'arrivée. Afin de connaître celle de départ, nous avons décidé que l'utilisateur va devoir fournir un fichier CSV supplémentaire, nommé units.csv,

contenant l'unité utilisée pour chaque colonne ayant des valeurs numériques dans les fichiers d'entrée. L'unité d'arrivée est définie dans l'ontologie. Une requête est alors nécessaire pour trouver le coefficient de conversion entre les deux (voir *Figure 23*).

On peut voir que la requête utilise à peu près les mêmes mots clés et la même syntaxe qu'en SQL (SELECT WHERE). Dans la clause WHERE, on écrit les informations sous forme de triplés. Ici, on cherche une unité (représentée par ?baseUnit) qui a une propriété hasLabel ayant pour valeur le paramètre donné ?baseLabel et qui a aussi une propriété hasConversion. Ensuite, on précise que la classe conversion récupérée doit aussi avoir une propriété conversionFactor. C'est cette valeur de la propriété qui va être renvoyée.

```
if (this.conversionFound.get(labels) != null) {
    return this.conversionFound.get(labels);
}
ParameterizedSparqlString queryString = new ParameterizedSparqlString();
queryString.setNsPrefix( prefix: "csopra",  uri: "http://csopralibs.org#");

queryString.setCommandText(
        "SELECT DISTINCT ?conversionFactor WHERE {" +
                "  ?baseUnit csopra:hasLabel ?baseLabel ;" +
                "            csopra:hasConversion ?conversion ." +
                "  ?conversion csopra:conversionTo ?goalUnit ;" +
                "            csopra:conversionFactor ?conversionFactor ." +
                "}"
);

// Set the label value
queryString.setLiteral( var: "baseLabel", baseLabel);
queryString.setIri( var: "goalUnit", goalURI);

// Create a new SPARQL query from the ParameterizedSparqlString
Query query = QueryFactory.create(queryString.toString());

// Execute the query and obtain results
QueryExecution qe = QueryExecutionFactory.create(query, ontology);
ResultSet results = qe.execSelect();
```

Figure 23 - Exécution d'une requête paramétrée SPARQL pour récupérer la valeur de la propriété "conversionFactor" entre deux unités

La conversion va se faire dans une classe ***UnitConverter*** grâce à la méthode ***convert***. Cette méthode va parcourir la liste des fichiers et récupérer l'unité de chaque colonne. Ensuite, pour chaque colonne du fichier, on obtient les coefficients de conversion ce qui permet de modifier les valeurs.

5.3.3. Remplacement des labels et vérification des types

Maintenant que toutes les valeurs sont converties, il est temps de remplacer les valeurs textuelles et de vérifier que les colonnes contiennent des valeurs connues au sein de l'ontologie. Tout d'abord, il a été décidé que les utilisateurs pourraient choisir les colonnes à valider sémantiquement (où changer les valeurs

textuelles). Pour cela, ma première idée fut d'ajouter un argument dans la ligne de commande par fichier. Ne travaillant qu'avec trois fichiers CSV pour ***CarboSeq***, cette solution n'était pas dérangeante mais il fallait éviter de devoir paramétrer l'exécution du programme pour chaque source de données, car le nombre de fichiers peut varier. Pour remédier à ça, les colonnes concernées sont directement écrites dans le nom du fichier à l'aide de leur indice. Par exemple, si je veux valider les colonnes d'indice 3 et 5 dans le fichier crop.csv, alors j'appellerai le fichier crop_3_5.csv. Afin de faciliter l'utilisation pour les partenaires, les indices commencent à 1, comme dans R. Il est aussi possible d'indiquer uniquement les colonnes à ne pas valider en ajoutant un "-" devant l'indice ou encore de toutes les valider en écrivant "all" après le premier underscore.

Une fois les colonnes récupérées, la main va passer à la classe ***Semantify*** et sa méthode ***matchLabelsAndURIs***. Cette méthode est décomposée en plusieurs étapes. Premièrement, elle vérifie si les indices des colonnes fournis par l'utilisateur sont corrects et dans ce cas, remplace les noms des colonnes par l'URI correspondant dans l'ontologie. Ensuite, pour chaque ligne de ces colonnes, on va vérifier si la valeur correspond au bon type renseigné et remplacer les valeurs textuelles par un URI. Il y a un argument optionnel au lancement de l'exécution du programme qui permet de définir le nombre de lignes où le type doit être vérifié. Cela évite de faire cette vérification sur l'intégralité des lignes dans les fichiers volumineux. Enfin, la méthode renvoie un objet Map (objet clés/valeurs) contenant l'ensemble des données après transformation par fichier. Aussi, comme on modifie le fichier en même temps que la lecture, il a fallu mettre le résultat dans un fichier temporaire afin de ne pas écraser les lignes suivantes.

Pour obtenir les URI correspondant aux valeurs textuelles, j'ai utilisé une méthode similaire à celle pour récupérer un coefficient de conversion (voir *Figure 23*). J'ai aussi mis en place une solution de stockage des résultats des requêtes SPARQL en utilisant une structure de données de type Map. Cette approche s'apparente à un système de cache, où les résultats obtenus sont enregistrés. Cette stratégie vise à améliorer les performances en évitant de ré-exécuter la même requête SPARQL si les données correspondantes sont sollicitées à nouveau (voir *Figure 24*). Dans un premier temps, je faisais une requête par valeur mais lors de tests sur des gros jeux de données, je me suis rendu compte que cela prenait énormément de temps. Pour régler ce problème, j'ai utilisé la méthode du "batch processing" qui consiste à exécuter la requête pour toutes les valeurs de la ligne simultanément plutôt qu'une par une. Ces deux optimisations m'ont permis d'avoir un temps d'exécution drastiquement réduit.

```java
public Map<String, String> getURIsFromLabels(List<String> labels) {
    Map<String, String> uriMap = new HashMap<>();
    Iterator<String> iterator = labels.iterator();
    while (iterator.hasNext()) {
        String label = iterator.next();
        if (Utils.isNumeric(label)) {
            iterator.remove();
        }
        else if (Collections.frequency(labels, label) > 1) {
            iterator.remove();
        }
        else if (this.found.get(label) != null) {
            uriMap.put(label, this.found.get(label));
            iterator.remove();
        }
    }
    try{
        uriMap.putAll(this.getURIsFromQuery(labels));
    } catch (MultipleConceptsForLabelException e) {
        e.printStackTrace();
    }
    return uriMap;
}
```

Figure 24 - méthode renvoyant les URI correspondant aux labels fournis

Concernant la récupération du type des colonnes, j'ai utilisé exactement le même principe que pour les valeurs textuelles, le nom de la colonne correspondant à une classe de type Observation dans l'ontologie.

5.4. Restructuration des fichiers

Après discussions, il a été décidé de déplacer certaines fonctionnalités de ***csopramapper*** vers l'application ***Semantifier***. ***csopralibs***, le paquet central de la model toolbox prend en entrée des fichiers bien définis. C'est pourquoi il y a une phase de restructuration qui permet d'obtenir ces fichiers à partir de ceux fournis en entrée. Cette fonctionnalité nécessite d'être adaptée à chaque source de données car le traitement à effectuer peut grandement différer. Pour cela, j'ai migré le composant "Mapper" de ***csopramapper*** vers Java. J'ai donc créé une classe ***Mapper*** assez générique dont hériteront les classes dédiées aux sources (voir *Figure 25* et *26*).

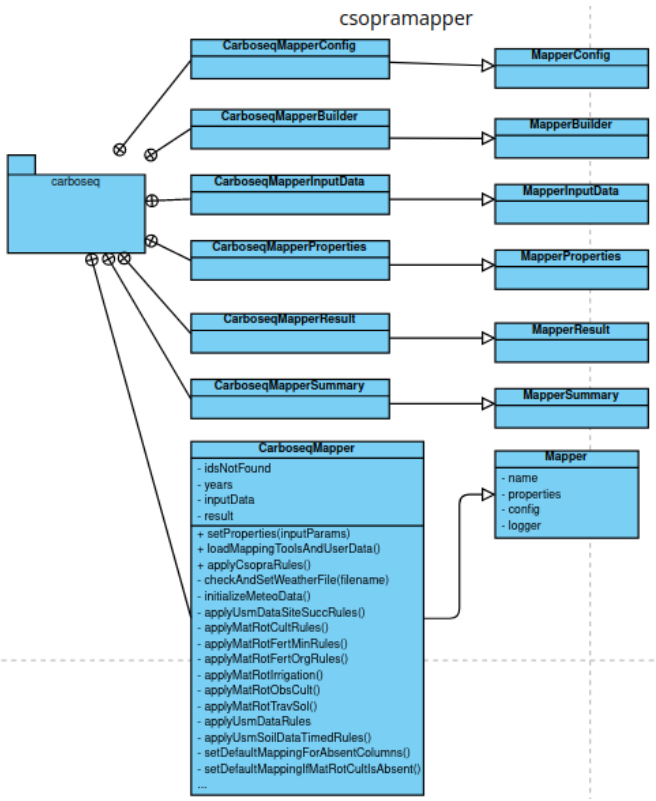


Figure 25 - Diagramme de classes simplifié (csopramapper avant Semantifier)

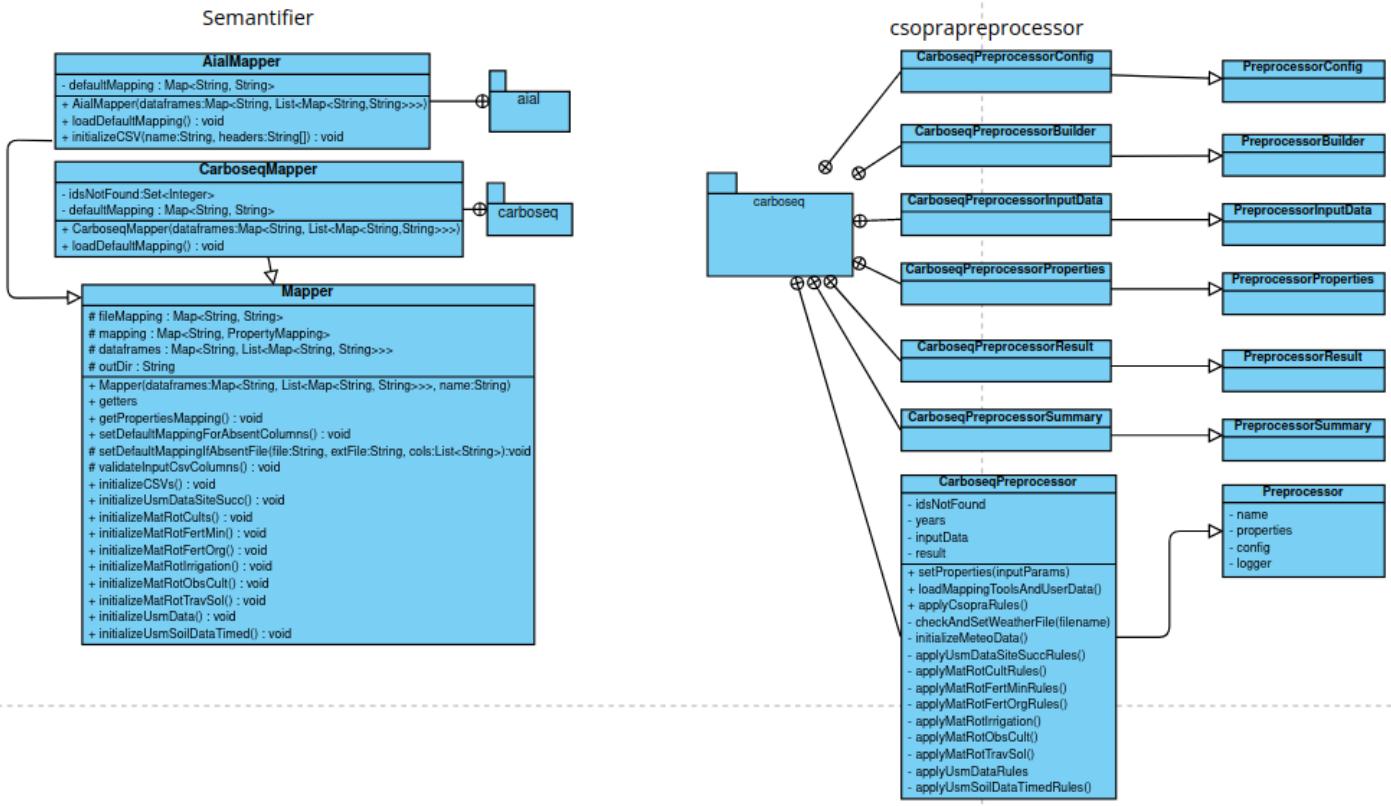


Figure 26 - Diagrammes de classes simplifiés de Semantifier (partie restructuration) et de csopraprepreprocessor (renommage de csopramapper, de plus il n'y a que carboseq de décrit mais il y a la même chose pour AIAL)

Avant, la première étape était, quel que soit la source des données, de faire la liaison entre les colonnes des fichiers CSV utilisateur en entrée et les colonnes attendues dans ceux au format ***csopralibs***, il y avait un fichier mapping.csv qui définissait ces correspondances. Maintenant, on connaît déjà les bonnes colonnes à utiliser car elles sont sous forme d'URI. On va donc pouvoir générer les fichiers de sortie en recopiant les valeurs de ceux en entrée. Aucun calcul n'est effectué dans l'application à ce stade, car elle se contente de recopier les valeurs dans les colonnes correspondantes et parfois d'ajouter des lignes suivant la valeur lue d'une colonne. Le but de cette démarche est de ne laisser à ***csopramapper*** uniquement la responsabilité de la partie métier, en lui fournissant des fichiers avec le bon nombre de lignes et lui laisser cette partie d'inférence (remplissage des cases vides et valeurs par défaut) (voir *Figure 27*). Les parties laissées vides concernent les dates ou alors des valeurs calculées en fonction de celles d'autres colonnes.

```
private void initializeCSV(String name, String[] headers) throws EmptyFileException {
    String file = this.getFileMapping().get(name) + ".csv";
    List<Map<String, String>> inputData = this.getDataframes().get(file);
    if (inputData.size() == 0) {
        throw new EmptyFileException(name + " is empty [empty input file: " + file + "]");
    }

    try(ICsvMapWriter csvWriter = new CsvMapWriter(new FileWriter( fileName: this.outDir + name + ".csv", StandardCharsets.UTF_8), CsvPreference.STANDARD_PREFERENCE)) {
        controller.checkColumnsAndReplaceHeaders(headers, new Integer[]{}, cols: null, name);
        csvWriter.writeHeader(headers);

        for (Map<String, String> row : inputData) {
            Map<String, String> rowContent = new HashMap<>();
            for (String header : headers) {
                rowContent.put(header, row.get(header));
            }
            csvWriter.write(rowContent, headers);
        }
    } catch (IOException | ColumnOutOfIndexException e) {
        e.printStackTrace();
    }
}
```

Figure 27 - Méthode pour générer les fichiers CSV csopralibs avec simple recopie (pour Aial)

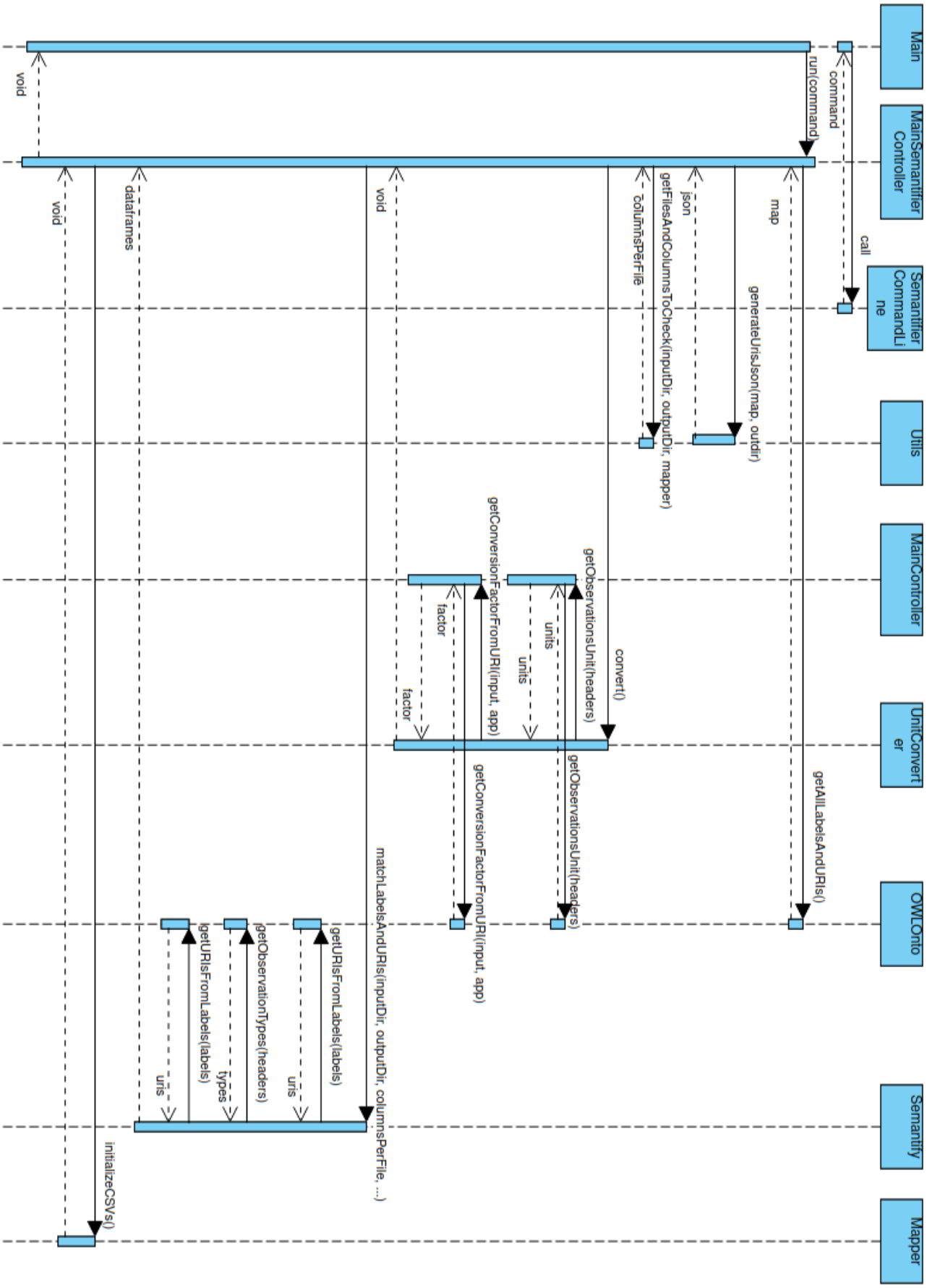


Figure 28 - Diagramme de séquence simplifié de Semantifier

6. Adaptation des paquets R

6.1. Manipulation des classes de l'ontologie dans le code

La dernière étape est donc de modifier les paquets R pour répondre à l'un des objectifs initiaux, utiliser les URI au sein de la model toolbox et donc enlever les valeurs textuelles en dures dans le code, que ce soit dans les fichiers d'entrée ou le code lui-même. Il a d'abord fallu décider des critères de changements. J'ai donc décidé de modifier uniquement les endroits où l'on accède à une donnée d'un fichier CSV csopralibs ou objet qui en est tiré. Les objets internes et variables de la model toolbox qui possèdent un nom correspondant à un label dans l'ontologie ne subissent pas ce changement. De ce fait, on garde une certaine lisibilité et la compréhension en est facilitée. Par exemple, avant, on utilisait une liste appelée colMapping qui pour chaque fichier donnait les correspondances entre les colonnes des fichiers CSV utilisateur et les fichiers CSV csopralibs dans ***csopramapper*** (voir *Figure 29*).

```
results <- data.frame(
        ID = NA, ID_SUCC = NA,
        DATE = NA, SOC = NA
)
```

```
idCol <- colMapping[["id_profil_csopra"]]
start_yearCol <- colMapping[["start_year"]]

latitudeCol <- colMapping[["latitude"]]

longitudeCol <- colMapping[["longitude"]]
```

Figure 29 - A gauche, une variable interne (on ne change pas les noms), à droite, des références aux CSV csopralibs (on change les valeurs par les URI)

J'ai donc commencé par modifier le paquet ***csopratools***. Afin de pouvoir récupérer l'URI à partir d'un label depuis n'importe quel endroit dans le code, j'ai ajouté une fonction dans l'environnement Utils nommée ***getUri*** qui sera aussi exportée lors de la création du paquet. Cette fonction utilise le fichier uris.json généré par ***Semantifier***. Dans ce paquet on trouve la classe ***CsopraData*** qui va contenir toutes les données d'entrée et qui va être utilisée dans les autres paquets. Suite à cela, j'ai modifié tous les accès aux dataframes des classes ***CsopraData*** et autres objets tirés directement des fichiers CSV csopralibs en appelant la fonction avec comme paramètre le nom de la colonne précédemment écrit (voir *Figure 30)*. Pour que cela fonctionne, le nom de la colonne doit correspondre à un label dans l'ontologie, sinon une erreur est renvoyée.

```
getIdSuccessions = function(id = NULL) {
    if (is.null(id)) {
        CsopTools$Utils$throwInvalidArgException("id cannot be null")
    }
    lines <- self$usmDataSiteSucc[self$usmDataSiteSucc[[CsopTools$Utils$getUri("id_profil_csopra")]] == id,]
    if (nrow(lines) == 0) {
        CsopTools$Utils$throwException(message = paste0("Invalid ID: ", id))
    }

    return(unique(lines[[CsopTools$Utils$getUri("id_succession")]]))
},
```

Figure 30 - Exemple de fonction pour obtenir l'URI

Concernant ***csopramapper***, le paquet doit maintenant utiliser les fichiers CSV csopralibs générés par ***Semantifier***. Pour cela, j'ai ajouté un sous-dossier dans le répertoire des données en entrée contenant ces fichiers (en plus des fichiers utilisateur, on a un dossier "csopra" avec les fichiers générés en plus). Les anciennes méthodes qui s'occupaient de générer les fichiers CSV csopralibs avant l'existence de l'ontologie ont donc été modifiées. Maintenant, elles ne contiennent plus que le remplissage des valeurs nulles comme les dates et les valeurs calculées. Il a aussi fallu remplacer les labels en URI en utilisant la fonction de ***csopratools*** présentée précédemment lors de l'accès aux colonnes des fichiers CSV csopralibs.

Pour ce qui est de ***csopralibs***, outre le remplacement des labels en URI pour accéder aux colonnes, un problème s'est posé. La model toolbox doit pouvoir accueillir de nouveaux modèles de simulation et calculateurs (scripts qui traitent des colonnes des d'entrée de ***csopralibs*** avant les simulations à proprement parler). De plus, ces outils pourraient être fournis par des personnes externes au projet, n'ayant donc pas connaissance de l'ontologie. Ces applications externes manipulant des labels pouvant être différents de ceux de l'ontologie, impliquent que les valeurs et concepts représentés sous forme d'URI dans les données ne pourraient donc pas être reconnus et pris en compte pour réaliser les estimations. Pour remédier à ce problème, j'ai décidé de faire un mapping dans le sens inverse, c'est-à-dire de l'URI vers le label. Étant donné que plusieurs labels peuvent correspondre à un seul URI, il sera nécessaire de sélectionner le label approprié afin de concorder avec le modèle, et de garantir qu'il n'y ait pas d'utilisation de labels correspondant au même URI dans l'ontologie pour définir deux concepts distincts. Un fichier est ajouté dans le dossier de configuration des applications externes nommé "uriMap.json", les modèles et calculateurs étant stables, il ne devrait pas subir de modifications une fois créé. La classe de configuration de ces outils se voient aussi complétés par une méthode ***getLabelFromUri*** afin de récupérer ces valeurs textuelles (voir *Figure 31*).

```
cropName <- private$..cinestCalculatorConfig$getLabelFromUri(cropName)
cropType <- private$..cinestCalculatorConfig$getLabelFromUri(cropType)
```

Figure 31 - Exemple d'utilisation de getLabelFromUri

6.2. Schéma d'une exécution

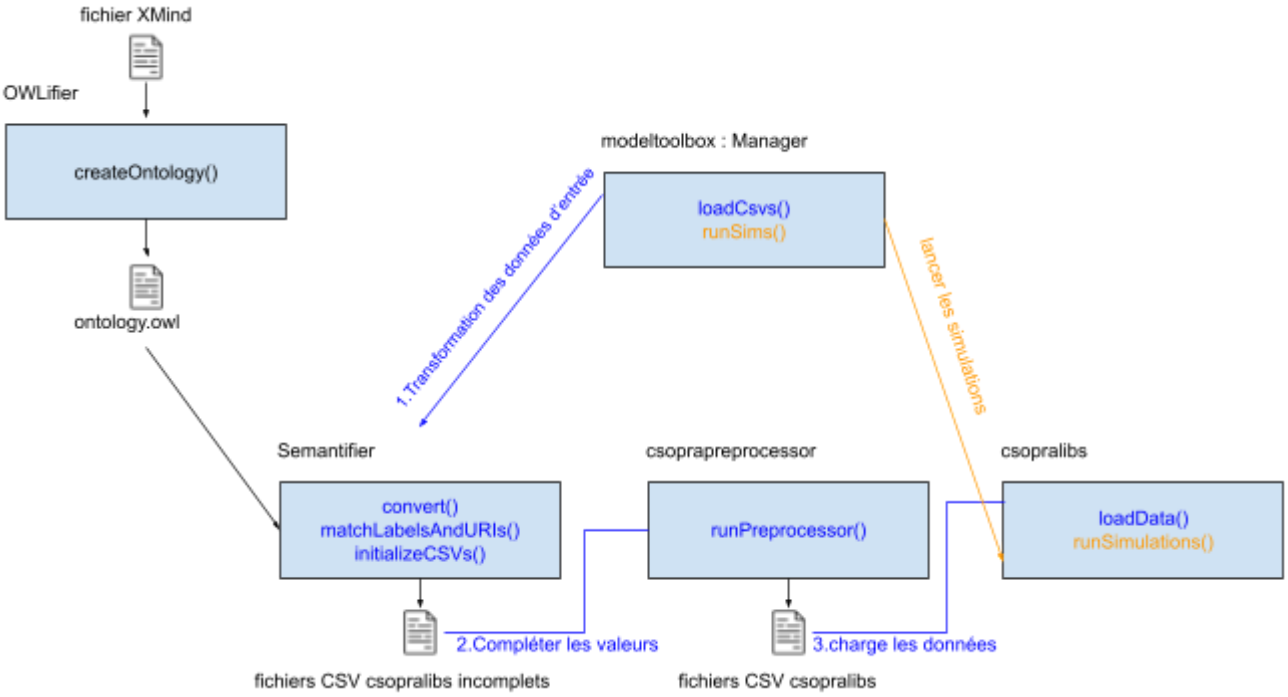


Figure 32 - Schéma d'une exécution de la model toolbox après les modifications

Dans cette figure, que l'on peut comparer à la *Figure 7* dans l'analyse de l'existant, on remarque plusieurs changements. Premièrement, l'utilisation de ***OWLifier*** pour fournir la dernière version de l'ontologie (Elle est exécutée à chaque changement du fichier XMind). Ensuite, on voit que ***Semantifier*** est venu se greffer entre ***modeltoolbox*** et ***csoprapreprocessor***. Enfin, pour ce qui est de ***csopralibs***, le déroulé reste le même.

7. Organisation de l'équipe, structuration et état du projet

Afin de s'organiser, l'équipe se réunissait une fois par semaine dans le but de montrer les avancées qui ont été faites, en discuter, puis se mettre d'accord sur les besoins et donc les nouvelles tâches à faire. Ces réunions permettaient donc d'avoir un suivi de l'avancement du projet mais aussi un retour de personnes plus expérimentées, Manuel Martin pour la partie métier et Rachid Yahiaoui pour la partie technique. Les discussions entre ces deux parties aidaient aussi à mieux comprendre les objectifs à atteindre.

Du côté plus technique, pour gérer le versionnage avec Git, les trois projets suivent une même structure. Il y a une branche principale "main" qui contient une version stable du projet, une branche "develop" où le projet est plus avancé mais

pas encore stable. Ensuite, lorsqu'une fonctionnalité doit être ajoutée, on crée une nouvelle branche "feature/ma_feature". La CI (Continuous Integration) contribue à garantir la qualité et la cohérence du code réalisé. Une fois le développement de cette fonctionnalité terminé et testé, on va fusionner la branche sur "develop".

Concernant l'état du projet, toutes les applications du projet sont interconnectées dans le flux CI/CD, où le processus va les parcourir de manière séquentielle et cohérente en suivant l'ordre de la chaîne d'exécution. Cependant, suite aux modifications, certains tests du paquet ***csopralibs*** ne passent plus, ceux comparant les résultats de la version actuellement en développement à une ancienne version agissant comme référence. Cela est dû à certaines décisions de modifier le contenu des fichiers CSV utilisateur, ce qui entraîne une différence dans les résultats obtenus et donc bloque l'intégration.

VI - Conclusion

Le sujet central de ce stage était d'inclure la création et l'utilisation d'une ontologie au sein du projet ***CarboSeq*** et plus précisément la model toolbox. Les deux programmes Java, ***OWLifier*** pour la création et ***Semantifier*** pour son utilisation, associés aux changements effectués sur le code R existant ont permis de répondre à ce besoin. ***OWLifier*** permet de générer l'ontologie à partir d'un graphe Xmind offrant une certaine simplicité, pour faciliter la compréhension des partenaires européens, tout en gardant assez de détails techniques pour une utilisation efficace de celui-ci. Quant à ***Semantifier***, il permet de pouvoir faire correspondre les fichiers d'entrée à l'ontologie et la model toolbox, cela en préservant la modularité de ***csopramapper*** afin de pouvoir l'utiliser avec plusieurs sources de données.

Mon travail a donc permis à l'INRAE d'avancer dans le projet ***CarboSeq*** et d'avoir ces deux applications supplémentaires. Elles répondent aux besoins et contraintes évoqués au cours de ce rapport et ce de manière générique. En effet, ces deux applications peuvent être utilisées au sein d'un autre projet ou en utilisant une autre ontologie et cela avec peu de changements à apporter.

Ce projet a constitué une expérience extrêmement enrichissante, qui m'a apporté une multitude de connaissances et de compétences. Tout d'abord, j'ai pu découvrir le domaine du web sémantique et des ontologies. En apprenant à créer et à manipuler des ontologies, j'ai développé une compréhension approfondie de la manière dont les informations peuvent être structurées de manière sémantique pour améliorer la recherche, l'analyse et l'interprétation des données. Ensuite, cela m'a aussi permis d'améliorer mes compétences, que ce soit de la conception en passant par du développement Java mais aussi d'avoir une vraie première expérience de développement en R. Sur un plan plus personnel, c'était une opportunité d'en

apprendre un peu sur le domaine de la recherche scientifique, ici plus particulièrement, l'étude des sols.

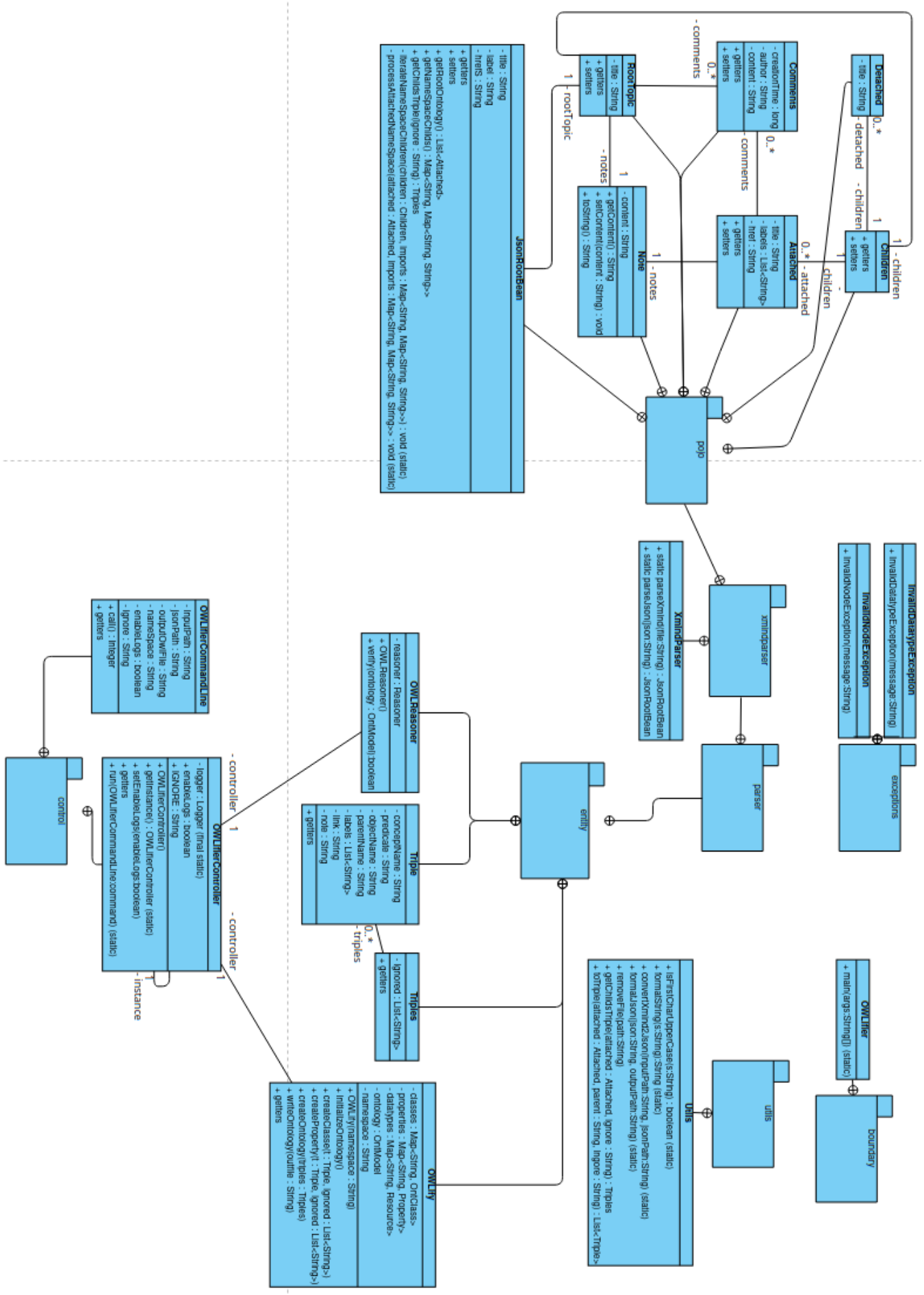
Plusieurs pistes sont considérées afin de continuer le développement des outils développés. Premièrement, il est envisagé que ***OWLifier*** permette de publier l'ontologie en ligne. Pour l'instant, on ne génère qu'un fichier donc celui-ci doit être fourni à ***Semantifier***. L'utilisation de ce fichier nécessite qu'il soit présent sur la machine exécutant le programme, ce qui peut entraîner des problèmes de versions par exemple, si l'ontologie vient à être modifiée. En publiant l'ontologie, on résoudrait ce problème. Concernant ***Semantifier***, le programme va être disponible sous forme de service web. En effet, si les partenaires du projet n'ont pas Java d'installé, alors ils ne peuvent pas l'exécuter. Pour remédier à cela, ***Semantifier*** sera aussi accessible en ligne. Cela entraîne aussi des changements futurs de la model toolbox afin d'intégrer l'appel à ce service dans la chaîne d'exécution du paquet.

Mon intégration dans le monde du travail s'est caractérisée par une expérience enrichissante et formatrice. Tout d'abord, ce stage m'a permis d'acquérir une certaine autonomie dans les décisions prises par les libertés données par mes responsables. Ce fut aussi une occasion d'apprendre en continu, en passant par la programmation objet en R et la découverte du web sémantique. Enfin, les réunions hebdomadaires et autres moments de communications m'ont permis de découvrir la dynamique d'équipe dans le milieu professionnel.

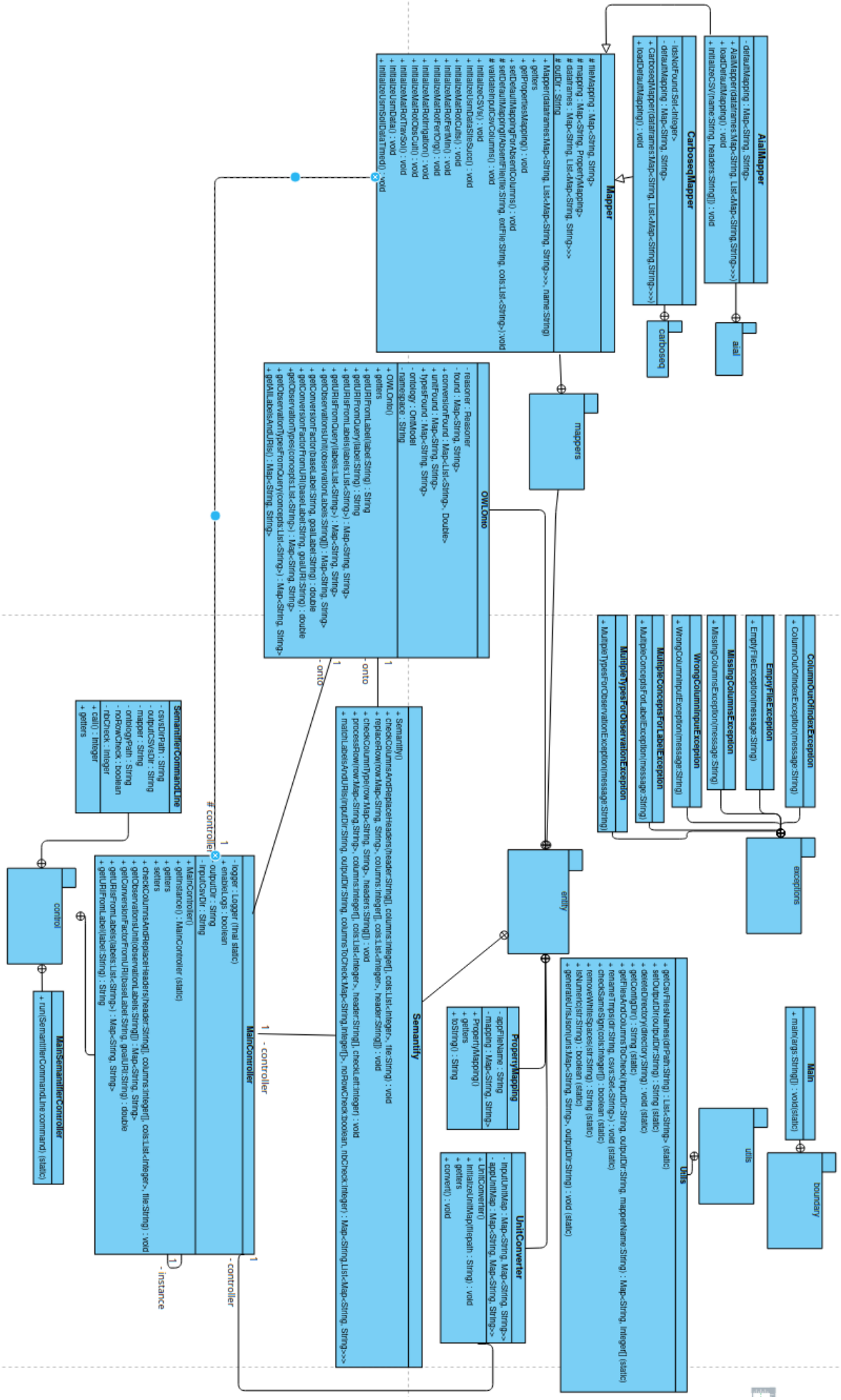
Pour terminer, le sujet de mon stage accompagné des différentes tâches que j'ai pu réalisé lors de ce dernier comme la conception d'une ontologie, la conception et le développement de programmes l'utilisant, la recherche de solutions et l'adaptation au langage m'ont permis d'utiliser les différentes compétences que j'ai pu acquérir au sein de ma formation universitaire.

VII - Annexes

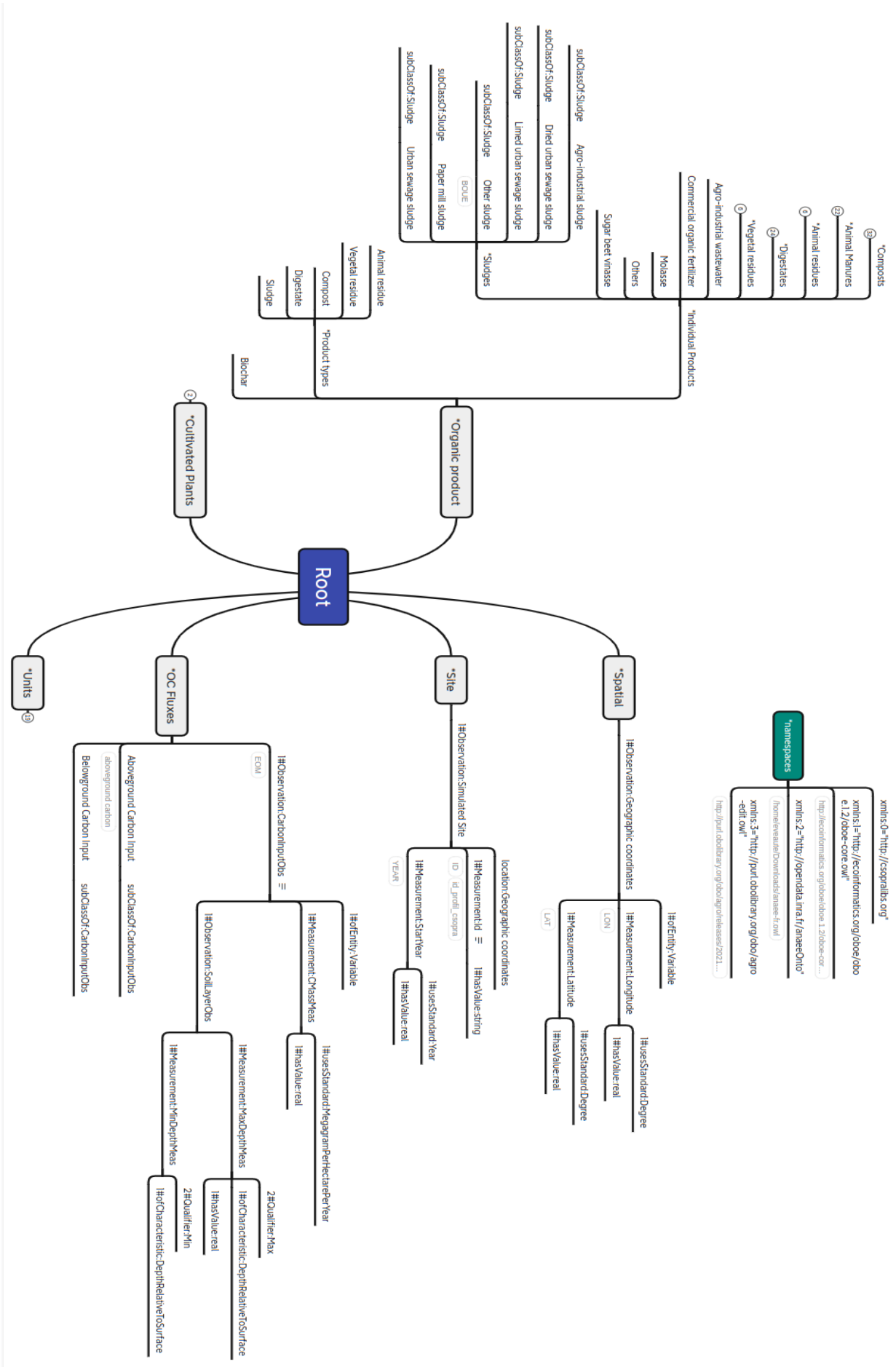
1. Diagramme de classes de OWLifier



2. Diagramme de classes de Semantifier



3. Vue d’ensemble du graphe Xmind



4. Partie de l’ontologie générée au format OWL

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:csopra="http://csopralibs.org#"
    xmlns:j.0="http://opendata.inra.fr/anaeeOnto#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:oboe="http://ecoinformatics.org/oboe/oboe.1.2/oboe-core.owl#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#">
  <owl:Ontology rdf:about="http://csopralibs.org#">
    <owl:imports rdf:resource="http://purl.obolibrary.org/obo/agro/releases/2021-11-05/agro.owl"/>
    <owl:imports rdf:resource="file:///home/eveaute/Downloads/anaee-fr.owl"/>
    <owl:imports rdf:resource="http://ecoinformatics.org/oboe/oboe.1.2/oboe-core.owl"/>
  </owl:Ontology>
  <owl:Class rdf:about="http://csopralibs.org#RawDigestateOfMunicipalSolidWaste">
    <rdfs:label>RawDigestateOfMunicipalSolidWaste</rdfs:label>
  </owl:Class>
  <owl:Class rdf:about="http://csopralibs.org#VegetalResidue">
    <rdfs:label>VegetalResidue</rdfs:label>
  </owl:Class>
  <owl:Class rdf:about="http://csopralibs.org#LiquidPhaseOfDigestateOfBiowasteAndOtherWastes">
    <rdfs:label>LiquidPhaseOfDigestateOfBiowasteAndOtherWastes</rdfs:label>
  </owl:Class>
  <owl:Class rdf:about="http://csopralibs.org#CompostedChickenManure">
    <csopra:hasLabel>compost de fumier de volailles</csopra:hasLabel>
    <rdfs:label>CompostedChickenManure</rdfs:label>
  </owl:Class>
  <owl:Class rdf:about="http://csopralibs.org#CompostedBovineSolidManure">
    <csopra:hasLabel>compost de fumier de bovins</csopra:hasLabel>
    <csopra:hasLabel>compost de fumier</csopra:hasLabel>
    <csopra:hasLabel>CFB</csopra:hasLabel>
    <rdfs:label>CompostedBovineSolidManure</rdfs:label>
  </owl:Class>
  <owl:Class rdf:about="http://csopralibs.org#GreenWasteAndSludgeCompost">
    <rdfs:label>GreenWasteAndSludgeCompost</rdfs:label>
  </owl:Class>
```