WoPMaRS: Workflow Python Manager for Reproducible Science

Luc Giffon

Mai 2016

Table des matières

Première partie

Rapport

1 Introduction

1.1 Organisme d'accueil

L'organisme d'accueil est le laboratoire *Technical Advances For Genomics And Clinics (TAGC)*, de l'Institut National De La Santé Et De La Recherche En Médecine (INSERM). Les projets de recherche menés dans ce laboratoire sont axés sur la compréhension de processus biologiques complexes notamment dans le cadre de maladies multi-factorielles chez l'Homme. Le laboratoire propose des approches en biologie des systèmes qui concernent l'identification de variations génétiques qui perturbent le fonctionnement normal de l'organisme et leur impact sur les processus biologiques. Cette approche fortement pluridisciplinaire implique d'assurer la continuité entre la recherche sur la santé d'une part et le développement de méthodes et d'outils informatiques pour traiter les données d'autre part. [1]

1.2 Contexte

Le projet s'inscrit dans le cadre des recherches en Bio-Informatique de M. Aitor Gonz'alez, maître de conférence au *TAGC*[1]. M. González cherche à rassembler puis annoter des données de Polymorphisme génétique [2] pour les soumettre à un outil de classification supervisée (*Support Vector Machine (SVM)* [3], notamment) et construire des modèles de prédiction. Ce type d'analyse peut faire appel à des outils existants de nature très variée, chacun générant une partie du résultat ou un traitement partiel de l'information. Ainsi, le traitement, le rassemblement et l'annotation des données se fait suivant un processus ordonné d'exécution d'outils spécifiques qu'on peut aussi appeler « *workflow* d'outils »ou « *workflow* ». L'exécution successive des outils d'un *workflow* peut être faite manuellement mais l'opération peut rapidement devenir trop fastidieuse dans le cas d'un *workflow* qui regroupe de nombreux outils et dont ceux-ci peuvent être hautement configurables et avoir un temps d'exécution très long, ce qui est courant lors du traitement de données biologiques.

De plus, l'exécution de ce type d'analyse se confronte souvent à des problèmes de reproductibilité car le nombre des outils employés, leur diversité et

1. Acronyme de Directed Acyclic Graph – Graphe Orienté Acyclique : En théorie des graphes, c'est un graphe orienté qui ne possède pas de boucle (ni simple, ni élémentaire). [2]
2. Le concept de polymorphisme génétique désigne la coexistence de plusieurs versions d'un gène au sein d'une espèce. [3]
3. Acronyme de Support Vector Machine : Ensemble de techniques d'apprentissage supervisé destinées à résoudre des problèmes de classification ou de régression. [4]

les paramètres que chacun doit recevoir rendent complexe la conservation de la totalité des informations requises pour reproduire une analyse à l'identique. Ainsi, le besoin de posséder un outil permettant de générer une science reproductible en bio-informatique est actuellement au coeur des préoccupations de la communauté.

1.3 Présentation du projet

Un *workflow* peut facilement être représenté sous la forme d'un *Directed Acyclic Graph – Graphe Orienté Acyclique (DAG)* où chaque nœud représente une étape de l'exécution et chaque arc la relation de dépendance entre deux étapes d'exécution. Nous appellerons ce graphe, le « graphe d'exécution »du *workflow*.

L'objet de mon stage est de développer un gestionnaire de *workflow* capable d'interpréter un *DAG* et de lancer l'exécution ordonnée des outils afin d'obtenir des résultats structurés et persistants tout en assurant un haut niveau de reproductibilité.

Un cahier des charges du projet a été rédigé par le client-encadrant et le co-encadrant. Ce cahier est disponible en Annexe A et en voici un court résumé :

Le logiciel devra permettre à l'utilisateur de définir, grâce à un fichier de définition à la syntaxe simple, une série d'analyses de données biologiques ordonnées suivant un DAG et de l'exécuter. Les étapes de cette série seront conçues comme des classes *Python* qui auront le rôle de *wrappers* d'outils capables de réaliser effectivement les analyses et produire des résultats. Ces classes pourront être développées par des personnes avec des connaissances limitées en *Python* en profitant d'une *Application Programming Interface (API)*[4] simple pour être intégrées à un *workflow*. D'autre part, le logiciel devra offrir un moyen d'assurer la persistance de ces résultats au fil du temps et la reproductibilité des analyses en proposant, par exemple, le moyen d'utiliser une base de données locale au sein des classes *wrapper*.

De nombreux outils [6][7] de gestion de *workflows* existent. Cependant, après des recherches menées conjointement par le client et le développeur, aucun d'entre eux ne semblait correspondre au cahier des charges énoncé, notamment à la partie consistant en la persistance des résultats en base de données. Aussi, une bonne partie de ces outils a été conçue dans l'optique de l'exécution de *workflows* sur le long terme et non de façon ponctuelle pour répondre aux types de problématiques posées par les expériences scientifiques.

4. Accronyme de Application Programming Interface - Interface de Programmation Applicative : Ensemble normalisé de classes, méthodes et fonctions qui sert de façade par laquelle un logiciel offre ses services à d'autres logiciels. [5]

2 Gestion du projet

Le projet étant mené dans un institut de Bio-Informatique et son équipe de développement n'étant constitué que d'un seul développeur, sa gestion et son organisation se sont déroulés de façon non-conventionnelle, mais les bonnes pratiques de génie logiciel ont été appliquées.

2.1 Membres de l'équipe et répartition des rôles

Aitor González. M. González est le porteur du projet, l'encadrant du stage et le client. Il est Bio-Informaticien au TAGC et maîtrise les concepts de base du langage de programmation *Python*. Au sein de l'équipe, il joue le rôle du client et répond quotidiennement aux questions relatives à la façon dont son programme devra être utilisé.

Lionel Spinelli. M. Spinelli est le co-encadrant. Il est ingénieur de recherche en Bio-Informatique au TAGC et au Centre D'Immunologie De Marseille-Luminy (CIML). Il a travaillé pendant sept ans pour le compte de la société *BMC Software* où il a, en plus de ses activités de conception et développement de logiciel, entraîné d'autres développeurs. Au sein de l'équipe, il joue le rôle de médiateur entre le client et le stagiaire-développeur et celui de conseiller et tuteur pour le stagiaire.

Luc Giffon. M. Giffon est le stagiaire et développeur. C'est moi.

2.2 Méthodologie de Développement

Le projet a été mené en deux phases. La première phase suivant une méthodologie à tendance « cycle en V »(figure 1) et la seconde suivant une méthode « agile »(figure 2).

2.2.1 Phase 1 : méthode en V

Le choix de cette méthode a été fait afin d'amener le développeur à produire une réflexion profonde sur le futur développement. L'aspect pédagogique de cette démarche a été avancée par M. Spinelli qui considère qu'avant de se lancer dans des méthodes agiles pures, un développeur doit comprendre l'intérêt et les limitations des autres méthodes. Ainsi, les phases d'analyse des besoins, de spécification et de conception détaillées ont été suivies afin de fixer les idées

dans l'esprit du développeur de manière à démarrer le développement sur des bases solides.

Comme le montre la figure 1, la méthode en V repose sur une première phase d'analyse avant la réalisation effective du produit. Cette phase est découpée en 3 parties qui vont *crescendo* dans le niveau de détails et qui sont :

- L'analyse des besoins, c'est à dire le recensement de l'ensemble des fonctionnalités, plus ou moins évidentes, désirées par le client
- Les spécifications, c'est à dire les caractéristiques détaillées de chacune des fonctionnalités demandées
- La conception, c'est à dire le détail technique de la réalisation des fonctionnalités

Dans le cadre de ce projet, les deux dernières phases ont gardé une dimension très générale, c'est à dire que seules les fonctionnalités de base ont été détaillées.

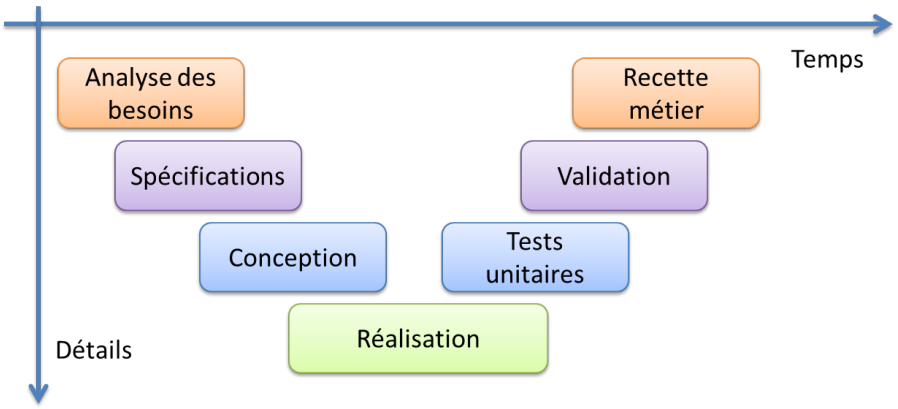

FIGURE 1 – Schéma explicatif du fonctionnement de la méthodologie V

2.2.2 Phase 2 : méthode agile

A partir du démarrage du développement à proprement parler, c'est une méthode agile qui a été choisie. Plus spécifiquement, c'est une méthodologie Scrum que l'équipe a essayé de mettre en place, à sa manière.

Ce choix a été fait afin de profiter de la présence quotidienne du client qui tient le rôle de « *Product Owner*[1] »et aussi, bien entendu, pour commencer à programmer suffisamment rapidement pour pouvoir présenter des résultats préliminaires lors de la soutenance de stage. Le co-encadrant joue le rôle de « *Scrum Master*[2] »en assurant le bon déroulement de la méthodologie.

La figure 2 montre le fonctionnement de base d'une méthode agile dite « conventionnelle »que l'équipe a essayé de reproduire. Le développement a été découpé en « *Sprints* »qui correspondent à l'ajout d'une ou plusieurs fonctionnalités. Avant chaque *Sprint*, l'équipe décide quelles fonctionnalités doivent être

1. Personne représentant le client et connaissant ses besoins au sein d'une équipe de développement agile
2. Responsable de l'équipe Scrum

implémentées pour la prochaine démonstration, c'est le « *Sprint Backlog* ». Ensuite, le développeur estime le temps dû pour vider le Sprint Backlog et si cela convient au Product Owner et au Scrum Master, la date de démonstration est fixée.

Pendant le *sprint*, le développeur découpe chaque fonctionnalité du *Sprint Backlog* en sous-tâches atomiques et chaque jour, il sélectionne certaines de ces tâches et se fixe comme objectif de les « faire »dans la journée. La notion de code « fait »a été définie comme un bloc de code documenté suivant les normes de *Pydoc*[3], respectueux des conventions de syntaxe *Python*[4] et testé à l'aide de Test unitaire[5] et fonctionnels.

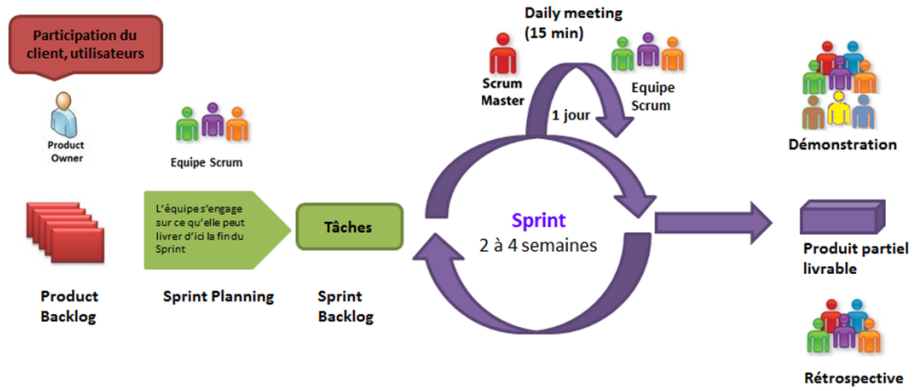

FIGURE 2 – Schéma explicatif du fonctionnement de la méthodologie agile

2.3 Système de gestion des versions

Pour assurer le contrôle de version[6], c'est le système *git* qui a été utilisé car le développeur et le client étaient plus familiers avec ce programme qu'avec *SVN*.

Au delà, la sur-couche de *git*, *git flow*, a été adoptée. Sur la figure 3, on voit que le plus clair du travail du développeur s'organise autour de la branche « *Develop* ». Au début du projet, la branche « *Master* »est crée, ainsi que la branche « *Develop* ». Lors d'un *Sprint*, pour chaque fonctionnalité à implémenter, une branche « *Feature* »est crée et lorsqu'elle est terminée, elle est fusionnée à la branche *Develop*. A la fin d'un *sprint*, lorsque *Develop* est stable, une branche « *Release* »est crée à partir de *Develop*, pour montrer le résultat du *Sprint* au Product Owner. S'il accepte ce résultat, la branche *Release* est fusionnée à la branche *Master*, sinon, on retourne sur *Develop*. Enfin, si un bogue est détecté,

3. Module de documentation pour Python [8]
4. `https://www.python.org/dev/peps/pep-0008/` et `https://www.python.org/dev/peps/pep-0254/`
5. Procédure permettant de vérifier le bon fonctionnement d'une partie précise d'un logiciel. [9]
6. Logiciel de gestion de version : Programme permettant de stocker un ensemble de fichiers en conservant un historique de ses modifications. [10]

une branche « *Hotfix* »est crée à partir de *Master* et est fusionnée à *Master* et *Develop* lorsque le problème est réglé.

FIGURE 3 – Schéma explicatif de l'organisation du versioning du projet, suivant la méthodologie de git flow

2.4 Gestion de la qualité du code

Comme preuve de la qualité du code, un serveur *SonarQube*[7] a été mis en place par le développeur sur sa machine. Les *plugins nosetests* et *coverage* ont été utilisés pour vérifier respectivement le taux de réussite des Test unitaire et leur taux de couverture. En annexe E, les résultats d'analyse obtenus à l'issu du premier *Sprint* sont présentés.

7. Logiciel permettant de mesurer la qualité du code source en continu. [11]

3 Travail réalisé

Le stage a été découpé en une partie d'analyse du produit qui a duré entre trois et quatre semaines et une partie de développement.

Le document d'Analyse détaillée (Annexe C) ne concerne pas le client car il explique les solutions techniques choisies pour répondre au Document de Spécification. Ce document contient différents diagrammes *Unified Modeling Langage (UML)*[1] qui expliquent, entre autres, l'architecture du programme et le déroulement d'une exécution normale.

Tous ensembles, ces documents on été rédigés pendant 3 semaines pendant lesquelles la majorité du temps a été passé à comprendre le fonctionnement d'autres programmes similaires et monter un « Etat de l'Art ». [6][7].

3.1 Analyse des besoins

Comme expliqué précédemment, un cahier des charges (Annexe A) a été réalisé par le client avant le démarrage du projet. La première tâche qui a été réalisée fût l'affinage de ce cahier des charges en discutant avec le client et en essayant de préciser son besoin.

3.2 Spécifications

Une fois cela fait, c'est le Document de Spécification (Annexe B) qui a été rédigé. Ceci a été contrôlé par M. Spinelli et plusieurs documents intermédiaires ont été soumis au client pour demander sa validation. La majeure partie du temps des spécifications a été passée à faire des recherches sur les programmes existants : d'une part, comprendre la façon dont on les utilisait a permis de mieux cerner les besoins du client. D'autre part, comprendre leur fonctionnement du point de vue technique a permis d'amorcer la réflexion pour la phase suivante qu'est la « conception détaillée ».

3.3 État de l'art

Cette partie contient une courte explication des méthodes architecturales employées par certains outils de gestion de *workflow*. L'objectif était de définir les avantages et inconvénients de chaque outil afin, d'une part, de profiter de

1. Acronyme de Unified Modeling Langage: En informatique, est utilisé pour représenter graphiquement des systèmes d'information.

l'expérience d'autres développeurs qui ont eu de bonnes idées et, d'autre part, de ne pas tomber dans les même écueils qu'eux.

Parmi les différents outils existants, différentes stratégies de définition de *workflow* ont été adoptées : certains, comme *Taverna*[12] ou *Galaxy*[13][14][15], sont basés sur une interface graphique pour leur définition alors que d'autres, tels que *SnakeMake*[16][17], utilisent un fichier de définition textuel directement modifiable par l'Homme. Les deux approches présentent chacune leurs avantages : d'un côté, les système de gestion de *workflows* basés sur une interface graphique offrent un meilleur confort d'utilisation, sont plus intuitifs et généralement favorisés par les utilisateurs « non-informaticiens ». De l'autre, les systèmes textuels permettent plus facilement de travailler collaborativement sur un projet en utilisant des systèmes de contrôle de version et sont facilement utilisables sur des Ferme de calcul [2] ou serveurs via l'utilisation du protocole *Secure Shell (SSH)* [3]. Le second type de système, le système textuel, est demandé par le client pour des raisons de reproductibilité et son utilisation sur des machines distantes.

Une autre différence entre les outils précédemment évoqués est d'ordre conceptuelle. En effet, la définition d'un *workflow* peut-être faite de deux façons. La première est explicite, c'est à dire que le graphe d'exécution est construit en fonction des positions absolues des étapes du *workflow* (par exemple en numérotant l'ordre d'exécution des différentes étapes). La deuxième façon est implicite, en considérant que la succession des étapes est définie par leurs Input [4] et Output [5] c'est à dire que le graphe d'exécution est construit en fonction des arcs reliant, dans cet ordre, les outputs d'un outil aux inputs du suivant.

3.4 Taverna

Taverna est l'un des outils de gestion de *workflows* les plus connus. Les développeurs de cet outil ont mis au point leur propre langage de définition de *workflow* qu'ils ont appelé « *SCUFL* ». Ce langage est dérivé de *XML* [6], ce qui le rend très flexible mais aussi difficilement lisible par un humain. Ce choix était judicieux dans le sens où *Taverna* repose sur la définition de *workflow* au travers d'une interface graphique, ce qui signifie que l'utilisateur ne « voit »jamais le fichier de définition *XML*. En revanche, *XML* est très pratique à lire et à modifier par un programme.

2. Appelé aussi « grappe de serveurs »ou « cluster ». Désigne une technique consistant à l'utilisation de plusieurs ordinateurs indépendants pour augmenter la vitesse de calculs. [18]

3. Acronyme de Secure Shell : Protocole de communication sécurisé basé sur un système de cryptographie asymétrique. Il permet, entre autre, l'exécution de tâches sur des machines distantes. [19]

4. Anglicisme permettant de définir les données entrant dans l'exécution d'un programme (paramètres, matière première, etc.) [20]

5. Anglicisme permettant de définir le produit de l'exécution d'un programme. Opposé à Input.[21]

6. eXtensible Markup Language

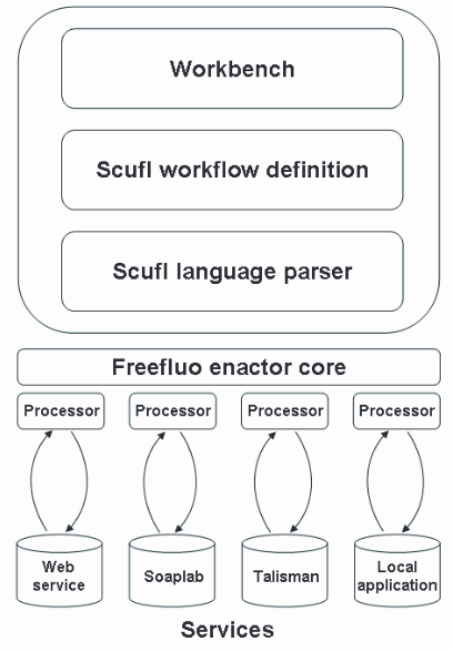

FIGURE 4 – Organisation générale du fonctionnement de *Taverna*

La figure 4 montre l'organisation générale du fonctionnement de *Taverna* :

— L'encadré en haut représente la partie « définition du *workflow* », c'est à dire l'utilisation de l'interface graphique pour le dessin du *workflow*, l'écriture du fichier de définition *SCUFL* qui en découle et enfin son analyse

— « *Freefluo enactor core* »est le moteur d'exécution du *workflow*. Vraisemblablement, l'analyseur de *SCUFL* produit une structure de données contenant les étapes du *workflow* ordonnées et prêtes à l'exécution par ce moteur

— Les cylindres en bas représentent les différents types de services pouvant ordonner l'exécution du moteur et recevant les résultats.

Enfin, la gestion des tâches du workflow se fait grâce au Patron de Conception [7] « *Thread Pool*[8] »qui permet de réaliser les tâches du *workflow* en parallèle sans risquer de dépasser les ressources disponibles. Figure 5.

7. Implémentation générique pour répondre à un problème spécifique

8. Collection de threads qui peut être utilisée pour effectuer plusieurs tâches en arrière-plan. La mise en thread pool représente une forme de multithreading où les tâches sont ajoutées à une file d'attente et automatiquement lancées lorsque les ressources sont disponibles. [22][23]

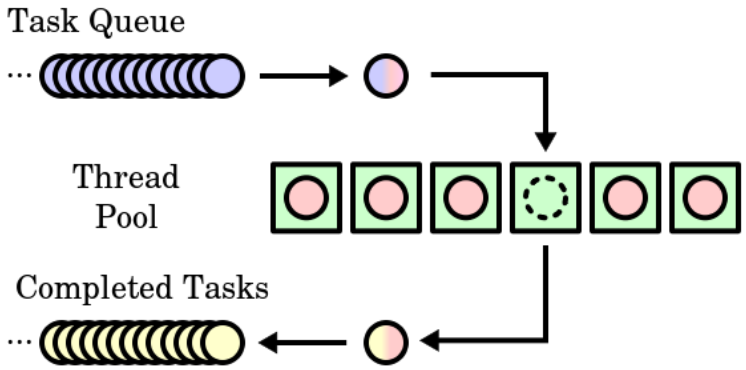

FIGURE 5 – Schéma explicatif du Patron de Conception « *Thread pool* »

Taverna n'était pas utilisable par le client car il repose sur une interface graphique pour la configuration du *workflow* et utilise un langage de définition dérivé de *XML* donc illisible. Cependant, l'utilisation du Patron de Conception *Thread Pool* s'est avérée être intéressante et est largement reproduite parmi les différents systèmes de gestion de *workflows*. D'autre part, la séparation architecturale entre l'analyse du fichier de définition du *workflow* et l'exécution effective du *DAG* offre une bonne stabilité au programme et permet la modification de l'une des parties sans altérer le fonctionnement du tout.

3.5 NextFlow

NextFlow [24] repose sur l'utilisation d'une *API* haut niveau pour la gestion de *workflow*. L'idée est inspirée des *Object Relational Mapping – Mapping Objet-Relationnel (ORM)*[9] où on utilise des classes pour représenter les tables d'un *Relational Database Management System – Système De Gestion De Base De Données (RDBMS)*[10]. Pour *NextFlow*, les classes représentent chaque bloc d'exécution utilisé par ce qu'ils appellent un *Business Process Management System (BPMS)*[11] responsable de l'exécution elle-même.

9. Acronyme de Object Relational Mapping – Mapping Objet-Relationnel
10. Acronyme de Relational DataBase Management System – Système de Gestion de Base de Données
11. Acronyme de Business Process Management System

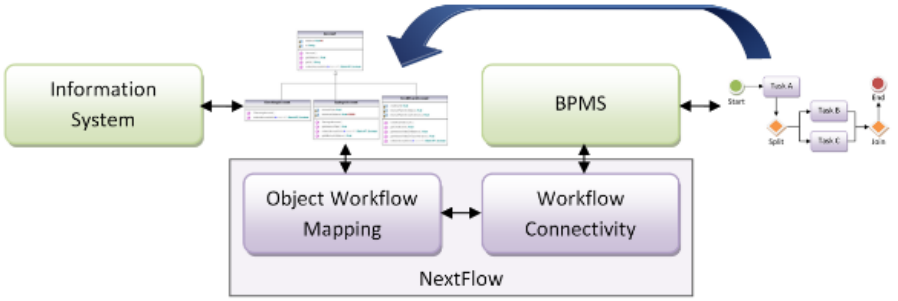

FIGURE 6 – Organisation générale du fonctionnement de NextFlow

La figure 6 décrit le fonctionnement de NextFlow :

— Les différentes tâches possibles sont « converties »en classes répondant à des spécifications précises pour leur intégration à un [12].

— Le système d'information, c'est à dire le service qui définit le *workflow*, manipule ces « classes-tâches »et les organise pour spécifier le *workflow*

— *NextFlow* instancie les objets correspondants à ces classes et organise leur exécution par le BPMS

Le BPMS utilisé par *NextFlow* est jBPM[25], spécifique de *Java*. Cependant, il en existe d'autres pour *Python*, notamment *SpiffWorkflow*[26], qui semble être largement utilisé.

La conception de nouvelles tâches utilisées par *NextFlow* nécessitant des connaissances en *Java*, cette solution ne pouvait pas être adoptée. Cependant, ce système de « *mapping* »de tâches sur des objets correspondait aux attentes du client en ce qui concerne la définition de *wrappers* d'outils. Les recherches menées sur *SpiffWorkflow* pour être utilisé n'ont pas été concluantes car cette librairie est en fait prévue pour « pré-définir »des *workflows* de façon explicite mais pas pour réaliser une interprétation dynamique d'un fichier de configuration.

3.6 SnakeMake

SnakeMake étant actuellement utilisé par le client et plus largement par la communauté Bio-Informatique, il a été crucial d'en comprendre le fonctionnement et d'en trouver les points forts et faiblesses afin de mieux cerner les attentes du client.

Parmi les points forts de SnakeMake, les utilisateurs trouvent qu'il est particulièrement facile à prendre en main, notamment au niveau de la syntaxe de son fichier de configuration *Yaml Ain'T Markup Language (YAML)-Like* [13] plus simple que le format *XML*, largement utilisé par les autres outils. D'autre part, le fait de pouvoir définir indépendamment chaque étape d'exécution et de façon customisée en utilisant *Python* était aussi particulièrement apprécié. Cependant,

12.

13. Acronyme de YAML Ain't Markup Language: Format de représentation de données par sérialisation . Il reprend des concepts d'autres langages comme XML. Son but est de représenter des informations plus élaborées que celles du format CSV tout en gardant une lisibilité meilleure que celle du XML. [27]

il semble que le débogage est particulièrement difficile. SnakeMake ne pouvait pas être utilisé dans ntore cas car il n'offre pas de connection à une base de données.

Le moteur d'exécution de *SnakeMake* repose sur la construction d'un DAG au lancement du programme. Ce DAG est produit en utilisant les relations de dépendance entre les inputs et outputs des différentes étapes du *workflow*. La construction de cet arbre se fait « de bas en haut », c'est à dire que le programme cherche le résultat final de la chaîne d'exécution et remonte d'inputs en outputs jusqu'au début pour déterminer le « chemin »à parcourir.

Le langage de définition YAML sera utilisé dans *WopMars* pour à la fois assurer la simplicité de définition du *workflow* et satisfaire les utilisateurs de *SnakeMake* en leur proposant une alternative qui y ressemble. Pour permettre aux utilisateurs de déboguer plus simplement leurs *workflows*, le système d'objets permettra aux développeurs de *wrappers* d'utiliser les exceptions afin d'aiguiller les utilisateurs finaux en cas de problème lors de l'exécution. Enfin, la construction « de haut en bas »du DAG sera privilégiée pour *WoPMaRS*.

3.7 Luigi

Luigi [28] n'est pas un programme spécifiquement conçu pour les expériences scientifiques mais comme il est écrit en *Python* et largement utilisé par de grandes entreprises, comprendre son fonctionnement est évidemment important.

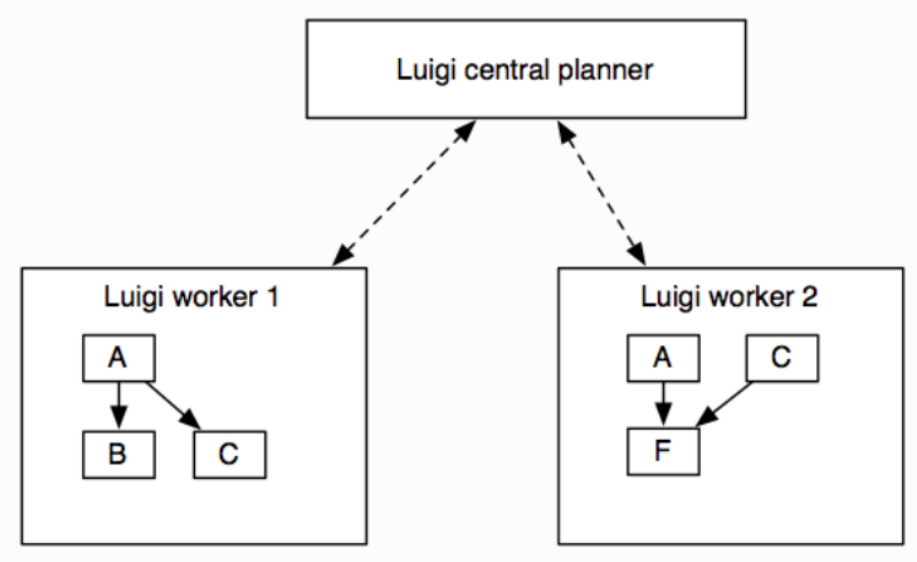

FIGURE 7 – Organisation générale du fonctionnement de *Luigi*

La figure 7 nous montre que *Luigi* utilise une architecture centrée autour d'un « planificateur »qui est chargé d'organiser l'exécution de « travailleurs »autonomes. Cette architecture assure la cohérence de l'exécution car tout est géré à un seul endroit mais elle peut présenter des risques de « surcharge »du pla-

nificateur s'il doit organiser des *workflows* trop lourds, c'est à dire de plusieurs centaines de tâches.

Luigi ne pourra pas être utilisé malgré sa grande plasticité car justement, il est trop généralisable et par conséquent, trop compliqué à utiliser. Cependant, son organisation en « manager-travailleur »pourra être utilisée car elle est très compatible avec le *Thread Pool*.

3.8 Conception détaillée

L'état de l'art réalisé précédemment chevauche la limite entre les spécifications et l'analyse détaillée car il a permit de répondre à des problématiques de conception notamment grâce à la découverte du Patron de Conception *Thread Pool*. Le Document d'Analyse Détaillé (annexe C) explique la façon dont le programme a été imaginé et en voici un résumé (figure 8).

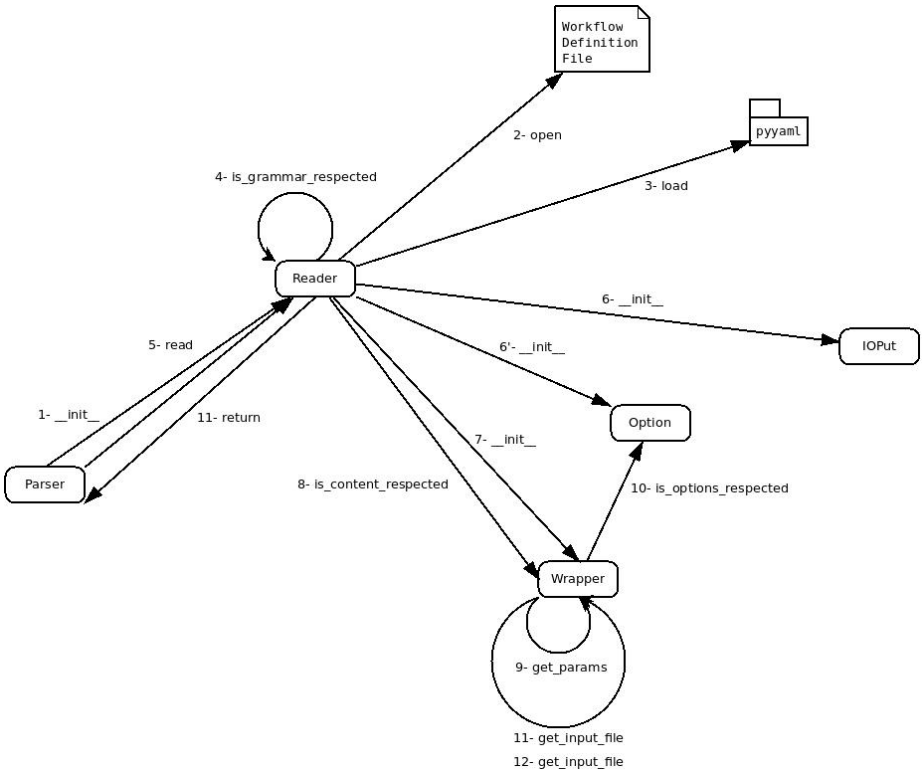

FIGURE 8 – Diagramme de processus général du programme

Sur le diagramme ci-dessus on voit que le programme s'organise autour de 2 parties principales : la partie d'analyse du fichier de définition et l'exécution du *workflow*. Les éléments en commun de ces deux parties sont les objets *Wrapper* contenus dans l'objet *DAG*.

On voit que le `Reader` utilise une méthode `is\_content\_respected` des objets `ToolWrapper` et le `WorkflowManager` une méthode `start`. Ces méthodes ne

devront pas être implémentées par le développeur des objets de classe `ToolWrapper`, en revanche, il devra faire en sorte que ses classes héritent de la classe `ToolWrapper`, qui appartient au gestionnaire de *workflow* et qui elle, implémente ces méthodes. Le développeur de `ToolWrapper` devra implémenter d'autres méthodes triviales qui permettent, entre autres de récupérer le nom des variables d'input ou d'output qu'il utilise.

3.9 Réalisation

La réalisation du programme se fait de *Sprint* en *Sprint* comme le préconisent les méthodes agiles et chaque sous-partie de cette partie correspond au déroulement d'un *Sprint*.

Sprint n° 1. L'objet de ce *Sprint* (annexe D.2) était de lire le fichier de définition du *workflow* et de créer les objets résultants.

Le modèle objet élaboré dans le Document d'Analyse Détaillé (annexe C) a été respecté dans les grandes lignes. En figure 9, on peut voir un diagramme de processus détaillé du fonctionnement réel de cette partie du programme.

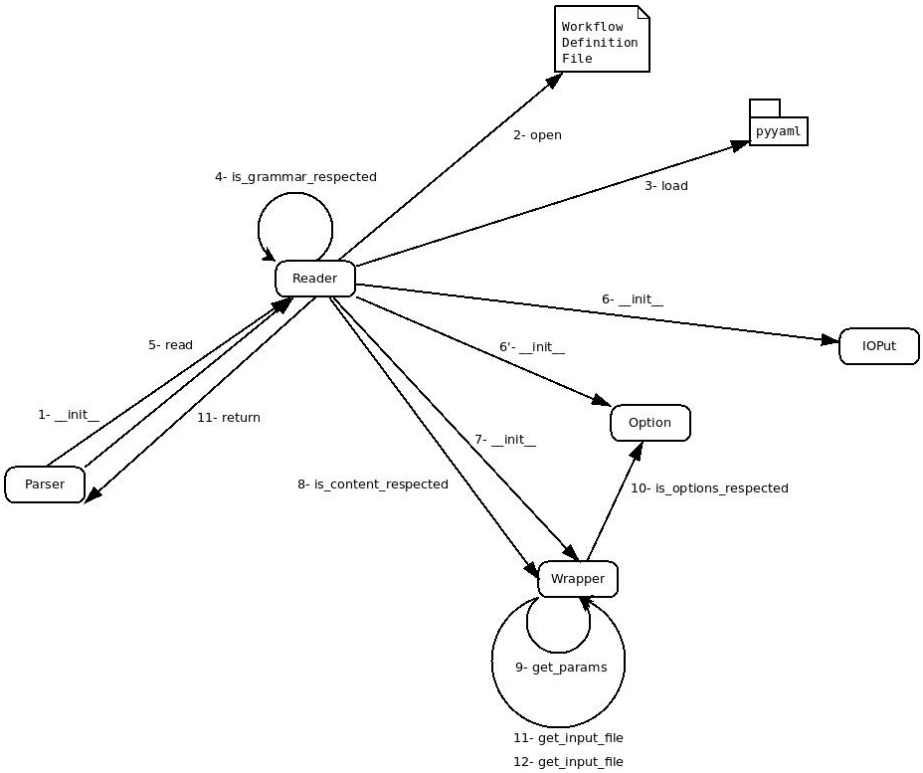

FIGURE 9 – Diagramme de processus de la partie « Lecture du fichier de définition du workflow »

Le `Parser` permet simplement d'initialiser le `Reader` pour le moment. Une

fois que le `Reader` a chargé le fichier de définition grâce à la fonction load de la librairie *pyyaml*[14], il commence par vérifier que ce fichier respecte la grammaire imposée (Annexe C). Ensuite, il crée les objets `IOPut` et `Option` pour instancier les objets `ToolWrapper` correspondant à chaque étape avant de demander à chaque `ToolWrapper` si le contenu est respecté, c'est à dire que les spécifications du développeur métier sont respectées.

Les différentes classes peuvent renvoyer des exceptions de classe `WopMarsParsingException` si jamais elles rencontrent un évènement imprévu comme un fichier de définition de *workflow* mal formé ou dont le contenu est incorrect par rapport aux spécification des classes *wrapper*.

La validation de ce *sprint* a été faite grâce à la mise en place de Test unitaire. Pour chacune des classes, des Test unitaire ont été réalisés et testent chacune des méthodes en fonction du fait qu'elles renvoient une exception ou pas, notamment.

Sprint n° 2. Le deuxième *Sprint* (annexe D.2) aura pour objectif d'améliorer les messages fournis à l'utilisateur par `WopMarsParsingException` afin qu'il puisse mieux situer l'erreur. Ce sprint permettra aussi la création de la classe `DAG` à l'aide de la librairie *NetworkX*[15] qui permet de construire des graphes et de profiter de méthodes des graphes courantes optimisées.

L'amélioration de `WopMarsParsingException` se fera grâce à l'ajout d'une sur-couche lors de l'initiation de l'exception afin que celle-ci spécifie l'étape concernée par le problème.

La classe `DAG` héritera de la classe `DiGraph` de la librairie *NetworkX* qui permet de représenter un graphe orienté. L'amélioration de `DAG` par rapport à `DiGraph` se fera par l'extension de sa méthode d'initialisation en lui permettant de prendre un ensemble d'objets `ToolWrapper` en argument et de se construire lui même à partir de ces objets `ToolWrapper` dont il déduira l'ordre à partir de leurs inputs et outputs.

14. `http://pyyaml.org/`
15. `https://networkx.github.io/`

4 Conclusion

Les premières semaines du stage ont été dédiées à la recherche d'informations sur les programmes existants et à la rédaction du Document d'Analyse Détaillée (Annexe C). Ce travail a permis d'élaborer le fonctionnement général du programme afin d'offrir des bases solides pour le développement du produit.

Le premier *Sprint* était dédié à la lecture et la validation du fichier de configuration ce qui représente la majeur partie du travail concernant l'analyse de ce fichier. Le prochain *Sprint* (annexe D.2) concernera la fin de l'analyse du fichier de configuration, c'est à dire la construction du `DAG` qui devra être exécuté.

Après la fin du stage, il restera trois mois au développeur pour terminer les fonctionnalités de base du programme, c'est à dire le lancement des outils et la gestion de la base de données, en plus du *parsing* du fichier de définition. Ces tâches prendront sûrement moins de temps que trois mois, alors certaines fonctionnalités importantes pourront être implémentées, parmi elles :

— La non-rééxécution des étapes déjà réalisées
— La lancement de l'exécution du *workflow* sur plusieurs *Threads*
— Le lancement de l'exécution du *workflow* sur un *cluster* de calcul

En plus de cela, les fonctionnalités optionnelles suivantes sont envisagées :

— La proposition d'un programme tiers d'aide au développement d'objets `ToolWrapper`

En tant que développeur, ce stage a déjà été riche en expérience même s'il est encore loin d'être terminé. En effet, j'ai demandé à être engagé pour 5 mois afin de profiter d'une première expérience de travail sur un long projet et mieux me rendre compte des problématiques rencontrées.

L'approche mi-cycle-en-V, mi-agile proposée par M. Spinelli m'a permit de me rendre compte que même trois semaines exclusivement dédiées à la réflexion sur le fonctionnement et l'organisation d'un programme ne sont pas suffisantes pour obtenir un résultat parfait et de nombreux problèmes continuent de survenir lors de la réalisation du programme. Cependant, cette période de réflexion permettra probablement de gagner du temps lors du développement du programme car l'idée de base étant posée, des problèmes de logique métier seront évités et théoriquement, aucun réusinage profond de l'architecture ne devra être fait pour permettre l'ajout de nouvelles fonctionnalités.

Deuxième partie

Annexes

A Cahier des charges

A.1 Objectif du logiciel

L'objectif de ce logiciel est de concevoir et implémenter une solution permettant l'exécution d'une série d'analyse de données biologiques ordonnées suivant un graphe dirigé acyclique (DAG). Les analyses doivent pouvoir être conçues comme des wrappers python qui auront en charge l'exécution des analyses et la production des résultats. Le logiciel doit offrir un cadre permettant l'enchainement de ces wrappers ainsi que toutes les facilités pour assurer la cohérence et la bonne exécution de l'ensemble des analyses.

A.2 Fonctionnalités requises

Le logiciel doit implémenter les fonctions listées ci-dessous :

Définition du workflow. Le logiciel doit être capable de lire la définition du DAG d'analyses depuis un fichier texte. Ce fichier doit contenir la définition des dépendances entre analyses (successions) ainsi que pour chaque analyse :

- La définition de l'analyse à exécuter
- La définition des options requises pour l'analyse

Le langage utilisé pour définir l'ensemble de ces éléments dans le fichier texte doit permettre :

- De rédiger facilement le fichier
- De lire facilement la définition de chaque analyse ainsi que de l'ensemble des dépendances

Intégration des analyses. Le logiciel doit permettre l'intégration simple de nouvelles analyses pouvant être exécutées. Un développeur python de niveau moyen doit pouvoir être à même de créer un wrapper d'analyse pouvant s'intégrer dans le workflow en respectant les pré-requis demandés par le logiciel.

Validation. Le logiciel doit être en mesure de valider un workflow avant de lancer son exécution, i.e. de vérifier que les enchainements d'étapes sont cohérents par rapport aux méta-informations dont il dispose sur chaque étape.

Persistance. Le logiciel doit être capable d'assurer la persistance des DAG d'analyse exécutés ainsi que la persistance des résultats de chaque étape d'analyse. Il doit en outre offrir aux wrappers d'analyse un cadre facilitant la per-

sistance de leurs résultats et l'accès à ces résultats persistants. Une solution de type base de donnée sera préférée.

Optimisation. Le logiciel doit être capable de vérifier si un schéma d'exécution (ou une partie du schéma) a déjà été lancé avec les mêmes options d'outils et inputs afin de proposer la récupération des outputs déjà obtenus pour minimiser la consommation de ressources. Il sera bien entendu possible d'outre-passer se comportement en forçant la réexecution complète du workflow.

Flexibilité. Le logiciel doit pouvoir permettre l'exécution complète du workflow (depuis le début du DAG) mais aussi autoriser l'exécution partielle. Un utilisateur doit pouvoir être capable de demander l'exécution à partir d'une étape du DAG. Le logiciel doit assurer la cohérence des analyses selon les dépendances du DAG (vérifier que les étapes en amont ont bien été exécutées) et assurer que les étapes en aval seront exécutées à nouveau. Cette fonctionnalité sera utilisée dans le cas de modifications de classes métier du programme (de telles modifications passeront innaperçues du point du vue du workflow).

Loging. Le logiciel doit permettre de conserver une trace de l'execution du workflow à différent niveaux de précision afin de faciliter la détection de bugs.

Sources de données. Le logiciel doit être capable de recevoir des données à partir de différentes sources (base de données et fichiers textes, notamment) tant que les wrappers d'analyse sont capables de lire les données à partir de la-dite source.

Exemple. Le logiciel doit être capable de mettre en oeuvre un exemple d'application concrète à la demande du client. Il s'agira vraisemblablement d'un workflow sur lequel il a déjà travaillé avec « SnakeMake », qui s'appelle SNPToMatrix et consiste en le regroupement de différentes informations concernant une liste de SNP en une Matrice. Les méthodes dites « métier »des outils de ce workflow ne seront pas à la charge du développeur, en revanche, il devra se charger de l'intégration avec le gestionnaire de workflow.

Innovant. Le développeur devra être capable de montrer en quoi son produit est innovant par rapport aux autres gestionnaires de workflow existants gratuitement et pourquoi les scientifiques (bio informaticiens notamment) ont avantage à s'en servir.

Multiplateforme. Le programme final devra pouvoir être déployé sur des machines UNIX et Windows et être indépendant de python2 ou python3.

A.3 Fonctionnalités optionnelles

S'il reste du temps après avoir implémenté toutes les fonctionnalités obligatoires, le développeur pourra s'attaquer aux tâches suivantes.

Sauvegarde et restauration. Le logiciel doit être à même assurer la sauvegarde des résultats de l'exécution d'un DAG avant toute nouvelle réexécution. Il doit aussi pouvoir assurer la restauration d'un état antérieur sur demande de l'utilisateur.

Execution sur un cluster de calcul. Le logiciel doit fournir le cadre adéquat pour permettre aux étapes d'éxécuter des commandes sur un cluster de calcul (batcher à définir) et récupérer les résultats de ces commandes.

Organisation. Chaque workflow conçu doit être pensé comme un mini-projet avec une arborescence de fichiers propre et logique.

Programme d'aide au développement. Programme tier d'aide au développement dans le workflow. C'est à dire un programme indépendant capable de générer automatiquement le code de base des classes de Wrapping dans l'arborescence des fichiers du projet.

Vérification du type d'Input personnalisable. Le comportement par défaut de vérification des Inputs est simplement de vérifier leur existence et leur « remplissage »(vérifier qu'une table n'est pas vide). Il pourrait être intéressant de permettre au développeur métier d'améliorer ce comportement en vérifiant l'intégrité du contenu des tables ou d'un fichier.

Conception de classes métiers dites « de base ». Le client aimerait que les classes métiers suivantes soient implémentées :

- Une classe permettant l'exécution d'une commande shell non prédéfinie
- Une classe permettant le parsing d'un fichier csv et l'insertion de son contenu en base de données

B Spécifications

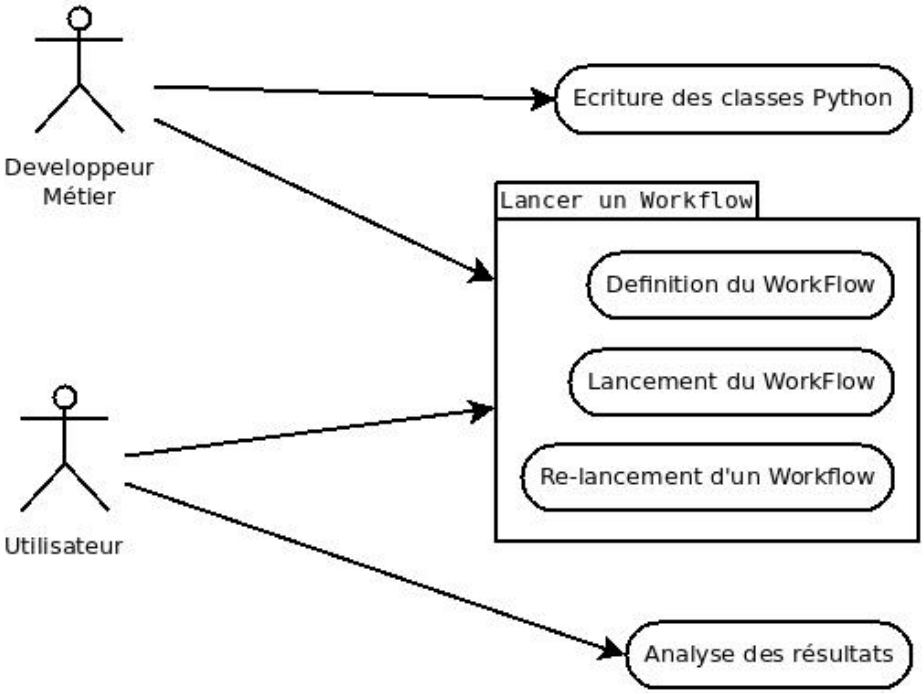

FIGURE B.1 – Diagramme de cas d'utilisations du programme

B.1 Définition du Workflow

B.1.1 Langage de définition du workflow

Le langage choisi pour la définition du workflow est un langage snakemake-like (inspiré du YAML) car le client est habitué à la grammaire de ce langage.

Il s'agit d'un langage de définition de workflow similaire à « Make »qui utilise une syntaxe inspirée de Python, propre, claire et concise. Les étapes du workflow sont définies par des « rules »qui spécifient :

— Un nom qui permet d'identifier une étape
— Un ou plusieurs documents d' »input »

— Un ou plusieurs documents d' »output »
— Un ou plusieurs paramètres

Un exemple de fichier de définition de workflow trivial est présenté comme suit :

```
configfile: "config.yaml"

rule Insert:
    input:
        SnpDefinition: "SNP_definition.txt"
    params:
        -t: 5
rule AnnotRegvar:
    input:
        AnnotationDefinition: "annot_def.txt"
    output:
        Resultat: "result.txt"
```

Le fichier de description du workflow peut être accompagné d'un document de configuration au format YAML. Ce fichier pourra donner des informations comme la localisation du package contenant les classes « Wrapper »ou que le nom donné au « projet-workflow »en cours (le nom par défaut serait celui donné au fichier de configuration). Le fichier config.yaml ressemblerait à quelque chose comme :

```
execution_name: "my_first_workflow"
wrapper_path: "/home/user/Documents/Wrappers/"
```

Nom d'une rule

On peut désigner une rule par son nom afin de demander l'exécution de l'arbre à partir d'une certaine étape. Le logiciel sera capable de vérifier que les étapes préliminaires à cette étape dans l'arbre ont bien été réalisées lors d'une exécution antérieure de telle façon à préserver l'intégrité des résultats obtenus tout en optimisant l'utilisation des ressources. Le nom d'une rule se référera directement au nom de la classe « wrapper »associée, il sera donc crucial d'en respecter l'orthographe et la casse

Inputs et Outputs d'une rule

Les inputs / outputs d'une rule devront répondre à un système de clé = valeur. Tout comme les rules, le nom de la clé est crucial car il se référera directement à la clé de l'input / output lors de son accès, au sein de la classe wrapper. Ce point sera explicité un peu plus tard.

Les inputs et outputs de type « fichier »doivent être spécifiés avec leur chemin d'accès absolu ou relatif à partir du fichier de configuration par défaut mais le « working directory »peut être spécifié suivant la syntaxe :

```
workdir: "/path/to/the/working/directory"
```

Les inputs et outputs de type « base de données »ne seront pas définis dans le fichier de définition du workflow car ils sont inhérents au fonctionnement de la classe wrapper.

Si une rule n'a pas d'input ou d'output « fichier », alors il est inutile de spécifier la catégorie. Par exemple :

```
rule Rule1:
    input:
        inputName: "/path/to/the/input"

rule Rule2:
    output:
        outputName: "/path/to/the/output"
```

Paramètres d'une rule

Les paramètres d'une rule devront aussi répondre au système de clé : valeur. Comme pour les inputs et outputs, aucune convention de nomage de ces clés n'est imposée au développeur des classes métier. Si un paramètre ne nécessite par de valeur, l'utilisateur peut en référencer uniquement la clé. Encore une fois, le nom de la clé est crucial pour son accession au sein de la classe « wrapper ». Par exemple :

```
params:
    --param: "value_of_a_string_param"
    --paramNumber: 1234
    --paramOnly
```

B.1.2 Analyse du fichier de définition du workflow

Le fichier de définition du workflow sera ensuite passé au programme à proprement parler, en argument (le comportement par défaut pourrait être la recherche d'un document de workflow dans le répertoire courant). Par exemple, la ligne de commande pour appeler le programme sera : `snptomatrix all -w myworkflow.yml`

B.2 Intégration des analyses

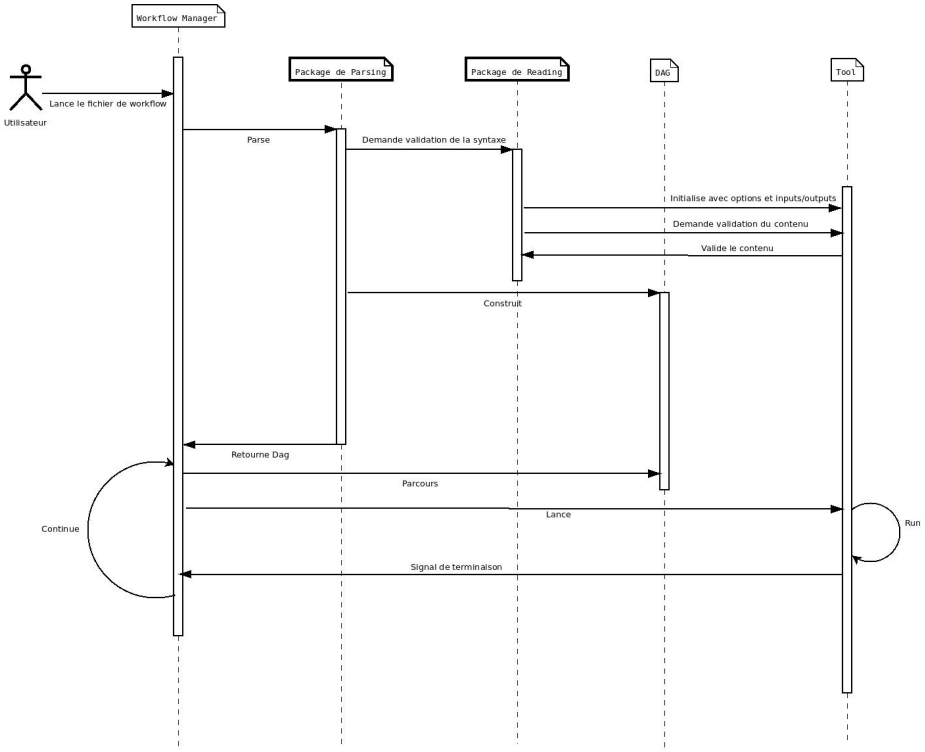

FIGURE B.2 – Diagramme de séquence du programme envisagé

Les outils externes seront encapsulés dans des classes appelées wrappers qui devront porter le nom de la rule qui leur est associée dans le document de définition du workflow.

Ces wrappers devront être capables de fournir des méta-informations quant à leur exécution prévue, dans le but de valider le workflow. C'est à dire que les wrappers devront connaître leurs inputs possibles, les outputs qu'ils vont générer et leurs options pour que l'on puisse les comparer à ceux du fichier de définition du workflow et vérifier leur compatibilité.

Ces informations devront être communiquées par l'implémentation des méthodes suivantes. Les notions d'Input et Output ainsi que de classe « BDD »seront élaborées plus tard :

« get_input_file » retourne la liste des clés d'input de fichiers possibles contenues dans des String

« get_input_db » retourne la liste des tables (objets de classe « BDD ») pouvant être utilisées comme input contenues dans des String

« get_output_file » retourne la liste des clés d'output de fichiers possibles contenues dans des String

« get_output_db » retourne la liste des tables (objets de classe « BDD ») pouvant être utilisées comme output contenues dans des String

« get_options » retourne un dictionnaire associant la clé de l'option à son type :

- **« bool »** pour les options ne nécessitant pas de valeur mais dont leur seule présence est significative
- **« string »** pour les options de type string
- **« int »** pour les options de type entier
- **« float »** pour les options de type réel
- **« default :value »** permet de spécifier une valeur par défaut
- **« required »** permet de définir une option obligatoire

Pour permettre le lancement de la classe, la méthode run devra bien entendu être implémentée par le développeur métier : elle contiendra le code d'exécution du wrapper à proprement parler et permettra au gestionnaire de workflow de lancer l'outil.

Pour accéder aux options depuis le wrapper, le développeur métier aura à disposition la façade « Option »qui pourra être appelé avec, comme indice, la clé associée à la valeur de l'option dans le fichier de définition du workflow en utilisant la méthode d'indexation normale de Python : `self.option("key_name")`

L'accession aux inputs et output se fera de la même façon avant une façade respective « Input »et « Output ».

Le développeur métier devra concevoir un objet « BDD »pour chaque table qu'il voudra voir crée dans la base de données résultante de l'utilisation de son wrapper.

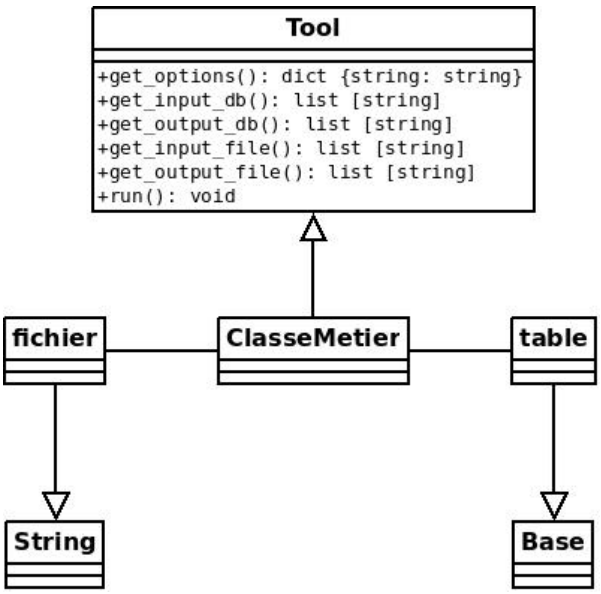

FIGURE B.3 – Diagramme de classe de la partie ToolWrapper

B.3 Validation

Avant de « lancer »un workflow à proprement parler, le programme sera capable de parcourir et d'analyser le fichier de définition du workflow grâce à un package de classes spécialement adaptées. Cette analyse se déroulera en plusieurs phases :

- Validation de la syntaxe : le parser devra déterminer si le fichier respecte la grammaire imposée (similaire à SnakeMake) qui sera détaillée dans le Document d'Analyse Détaillée
- Validation du workflow : le parser devra comprendre la construction du DAG et déterminer si celui-ci est cohérent ou pas :
  - Respect de la définition de DAG (pas de cycle)
  - Les branches de l'arbre sont « possibles », c'est à dire qu'un noeud « outil »a pour input « parent », un (ou plusieurs) input qui peut en effet le précéder. Ce contrôle se fera à l'aide des classes « wrapper »des outils qui connaîtront leurs propres conditions de validations et une Exception sera levée en cas de problème

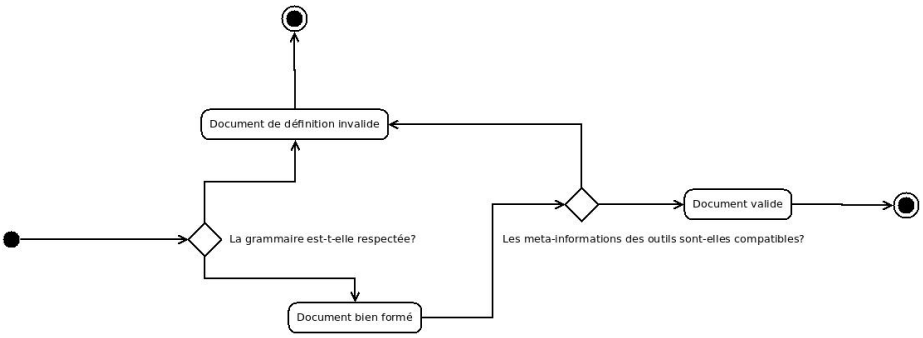

FIGURE B.4 – Diagramme d'états transitions de la validation du fichier de définition

B.4 Persistance

Pour permettre la persistance des données générées par une exécution, une solution « base de données »est proposée. Cet accès à la base de données est géré grâce aux objets de classe « BDD »évoqués précédemment. La solution de gestion des bases de données via l'ORM SQLAlchemy sera adoptée. Pour permettre cela, les objets « BDD »devront hériter de la super classe « Base »fournie par SQLAlchemy (ou hériter d'une classe qui elle même hérite de Base, l'héritage multiple ne sera pas autorisé).

Lors de chaque exécution, une base de données correspondante sera crée (même si aucun résultat n'est stocké en base de données). Celle-ci portera le nom de l'execution du workflow qui peut être spécifié dans le fichier de configuration ou bien porte le nom du fichier de définition du workflow par défaut. Cette base de données contiendra au minimum une représentation relationnelle de l'arbre DAG exécution. Le schéma relationnel utilisé pour la représentation de cet arbre en base de données sera explicité dans le Document d'Analyse Détaillée.

B.5 Optimisation

Avant le lancement du workflow, le programme va rechercher des exécutions antérieures similaires : le parser devra être capable de reconnaître le schéma d'exécution parmi d'autres schéma stockés dans différentes bases de données et proposer, le cas échéant, la ré-utilisation des données communes. Cette comparaison de fichiers se fera par une logique de dates de modifications : si l'input qui a servi à la génération d'un output a une date de dernière modification plus ancienne que celle de la création de l'output, alors re-générer l'output ne donnera pas un output différent.

Dans un second temps, on pourra envisager la comparaison des sommes md5 des fichiers pour offrir à l'utilisateur une comparaison plus robuste et souple. Cependant, cette méthode a l'inconvénient d'être très consommatrice en temps, dans le cas de gros fichiers (environ 3s par GigaOctet sur ma machine).

B.6 Flexibilité

Pour lancer un schéma d'exécution à partir d'une étape donnée, l'utilisateur devra spécifier le nom de l'étape (rule) dans le document de définition du workflow. Cette fonctionnalité correspondra grossièrement au lancement forcé de l'une des rules du workflow, même si le programme estime que l'output sera identique.

B.7 Logging

Le niveau de logging sera spécifié par l'utilisateur, habituellement, 4 niveaux sont utilisés :

1. Error : les problèmes aboutissants à l'arrêt du programme, par exemple, un fichier de définition du workflow invalide
2. Warning : les problèmes pouvant aboutir à des résultats inatendus mais ne perturbant pas l'execution du programme, par exemple, deux fichiers avec des contenus identiques mais dont les meta-informations diffèrent (nom, date de dernière modification)
3. Info : les informations relatives au déroulement du programme
4. Debug : la trace d'execution du programme avec des informations détaillées sur le contenu des variables

Le développeur des classes wrapper pourra utiliser un helper pour remplir le document de log suivant le niveau de log qu'il désire.

C Conception détaillée

C.1 État de l'art

De nombreux projets de conception de gestionnaire de workflow existent. Il a donc fallût, dans un premier temps, vérifier si l'un d'entre eux n'étaient pas utilisable tel quel, voir pourrait servir de base pour une éventuelle amélioration et mieux coller aux besoins du client.

Deux dépôts git listant très précisemment ces projets ont été découverts :

— `https://github.com/common-workflow-language/common-workflow-language/wiki/Existing-Workflow-systems`

— `https://github.com/pditommaso/awesome-pipeline`

Il n'a pas été possible d'analyser en profondeur chacun d'eux donc un « pré-pruning »a été réalisé suivant les critères suivants :

— Langage orienté objet (préférablement Python)

— Le programme devait permettre une utilisation modulaire

— Des manuels d'utilisation / développement devaient être disponibles

J'ai pu me rendre compte que les projets les plus intéressants étaient ceux ayant fait l'objet d'une publication scientifique mais que, d'une façon générale, il était très difficile d'obtenir des documents de spécification détaillés sur l'architecture d'un programme, même libre.

Le contenu de l'état de l'art est disponible dans la partie « État de l'Art »du rapport (3.3).

C.2 Définition du workflow

C.2.1 Grammaire

```
NEWLINE configfile
configfile  = "configfile:" stringliteral
SNPToMatrix = rule | workdir
rule        = "rule" (identifier | "") ":" ruleparams
workdir     = "workdir:" stringliteral
ni          = NEWLINE INDENT
ruleparams  = [ni input] [ni output] [ni params]
NEWLINE SNPToMatrix
input       = "input" ":" ((identifier ":" stringliteral),?)+
output      = "output" ":" ((identifier ":" stringliteral),?)+
params      = "params" ":" ((identifier ":" stringliteral),?)+
```

C.2.2 Rules

En général, une rule est composée d'un nom, d'un input et d'un output. Il est aussi souvent composé de params. La rule se référant à la classe wrapper correspondante, il est important d'en respecter l'orthographe est la casse.

Inputs et Outputs

Les inputs et outputs « fichiers »sont spécifiés par un système de « clé : valeur »avec comme clé, l'identifiant de l'input ou de l'output et comme valeur, une chaîne de caractère référant au chemin d'accès vers le document. En ce qui concerne le nomage, comme pour les rules, il faut que la clé de l'input ou de l'output corresponde exactement à celle définie par le développeur de la classe métier correspondante.

```
input:
    input1: "/absolute/path/to/input"
    input2: "relative/path/to/input/from/workdir"
output:
    output1: "/absolute/path/to/output"
    output2: "relative/path/to/output/from/workdir"
```

Paramètres

Les paramètres sont spécifiés de la même façon que les inputs et outputs. Le respect des règles de nomage définies par le développeur métier est toujours indspensable. Cependant, les valeurs peuvent être de type :

- String : spécifiée par des guillemets
- Numerique : float (le séparateur des décimales est un point) ou integer
- Booleen : si la clé est spécifiée, la valeur est True, sinon False

```
params:
    param1: "string value of param1"
    param2: 10.27
    param3
```

Synthèse

```
rule Wrapper1:
  input:
    InputFile1: "path/to/inputfile"
    InputFile2: "path/to/other/inputfile"

  output:
    OutputFile1: "path/to/outputfile"

  params:
    Param1: "value of param 1"
```

C.2.3 Répertoire de travail (Optionnel)

Le répertoire de travail est défini par la clause « workdir »suivie du chemin absolu vers le répertoire. Il peut être spécifié à n'importe quel endroit du fichier entre deux rules (ou au tout début) et prend effet pour toutes les rules suivantes jusqu'à la prochaine modification du workdir.

```
1 workdir: "/absolute/path/to/the/working/directory/"
```

C.2.4 Fichier de Configuration (Optionnel)

Ce fichier est une « idée »pour le moment, il n'est pas forcement utile, mais il est précisé afin de voir venir une éventuelle utilisation.

Le fichier de configuration est défini en début de fichier par la clause configfile suivie du chemin absolu ou relatif à partir du fichier de définition du workflow, vers le fichier de configuration. Ce fichier peut contenir des informations comme le nom de base de données à utiliser (le comportement par défaut est l'utilisation du nom du fichier de définition du workflow) :

```
(NEWLINE configfield)+
configfield   = executionName
executionName = "execution_name" ":" stringliteral
```

C.3 Parsing du fichier de définition du workflow

La classe responsable de l'organisation de la partie « parsing du fichier de définition »est appelée « Parser ». Cette classe utilise trois autres classes :

- une classe « Reader », chargée de lire le fichier de définition du workflow, a proprement parler
- une classe « DAG », chargée de construire le graphe de processus du workflow

C.3.1 Parser

La classe Parser comporte une méthode appelée « parse »qui lui permet de mettre en route le processus d'analyse du fichier de définition. Ce processus se déroule en trois phases :

- dans un premier temps, elle instancie et appelle le « Reader »avec sa méthode « read() »
- elle instancie le graphe d'exécution du workflow via l'initilaisation d'un objet de classe « DAG », en lui passant la liste des ToolWrappers retournées par le Reader

C.3.2 Reader.read()

Le fichier de définition du workflow étant « Yaml-like », il sera possible d'utiliser la librairie de parsing de Yaml « PyYaml »(`http://pyyaml.org/`). Cette librairie permettra de mapper le contenu de l'arbre Yaml à un dictionnaire python.

```
{
    "wrapper_name": {
        "input": {"input_name": value},
        "output": {"output_name": value},
        "option": {"option_name": value}
    }
}
```

A partir de ce dictionnaire, les objets de classes ToolWrapper vont être initialisés avec les objets de classes IOPut : IOFilePut et IODbPut et Option nouvellement créés. Enfin, l'ensemble de ToolWrapper sera renvoyé.

C.3.3 DAG.__init__()

Pour la construction du DAG, la librairie networkX (`https://networkx.github.io/`) va être utilisée.

Lors de l'initialisation de l'arbre, l'ensemble de ToolWrapper est analysé pour déterminer leur enchainement à l'aide de leurs méthodes « follows() » :

```
graph <- initialisation du graphe orient\'{e}
  POUR CHAQUE wrapper1 du set(wrappers) FAIRE
    POUR CHAQUE wrapper2 du set(wrappers) - wrapper1 FAIRE
      SI wrapper2.follows(wrapper1) ALORS
        arbre.addDirectedEdge(wrapper1, wrapper2)
```

Des heuristiques concernant le caractère acyclique du graphe pourront éventuellement être incorporées afin d'optimiser cet algorithme.

C.3.4 ToolWrapper.__init__()

Les classes appelées « ToolWrapper »sont les classes contenant le code métier des outils qui devront être lancés par le workflow. Elles sont appelées « ToolWrapper »par abus de langage. En réalité ce sont des classes qui héritent de la super-classe « ToolWrapper ». La super classe ToolWrapper appartient au gestionnaire de workflow et est l'interface par laquelle doivent passer les classes dites « métier »conçues par les « développeurs métiers »pour pouvoir être intégrées à un workflow.

Python ne permet pas d'implémenter des interfaces ou classes abstraites « façon java »(c'est à dire qui entraînent une erreur de compilation si une méthode n'est pas implémentée) mais ce comportement peut être simulé. En effet, dans le cas où le développeur métier n'a pas implémenté une méthode qu'il aurait du, une exception « NotImplementedException »est levée.

Les objets de classe ToolWrapper font appel aux méthodes définies par l'utilisateur au moment où les paramètres de lancement d'un outil sont initialisés afin de vérifier leur validité (e.g. leur existence et leur type).

« get_input_file » retourne la liste des clés d'input de fichiers possibles contenues dans des String : [String]

« get_input_db » retourne la liste des tables (objets de classe « BDD ») pouvant être utilisées comme input contenues dans des String : [String]

« get_output_file » retourne la liste des clés d'output de fichiers possibles contenues dans des String : [String]

« get_output_db » retourne la liste des tables (objets de classe « BDD ») pouvant être utilisées comme output contenues dans des String : [String]

« get_options » retourne un dictionnaire associant la clé de l'option à son type :

- **« bool »** pour les options ne nécessitant pas de valeur mais dont leur seule présence est significative
- **« string »** pour les options de type string
- **« int »** pour les options de type entier
- **« float »** pour les options de type réel
- **« default :value »** permet de spécifier une valeur par défaut
- **« required »** permet de définir une option obligatoire

Si plusieurs informations doivent être spécifiées par l'utilisateur, le caractère « | »peut être utilisé. exemple :

```
{
    "option1": "int",
    "option2": "float|required",
    "option3": "int|default: 3",
    "option4": "str|default: 'the_default_value'"
}
```

C.3.5 Diagramme de classes de « Parsing du fichier de définition »

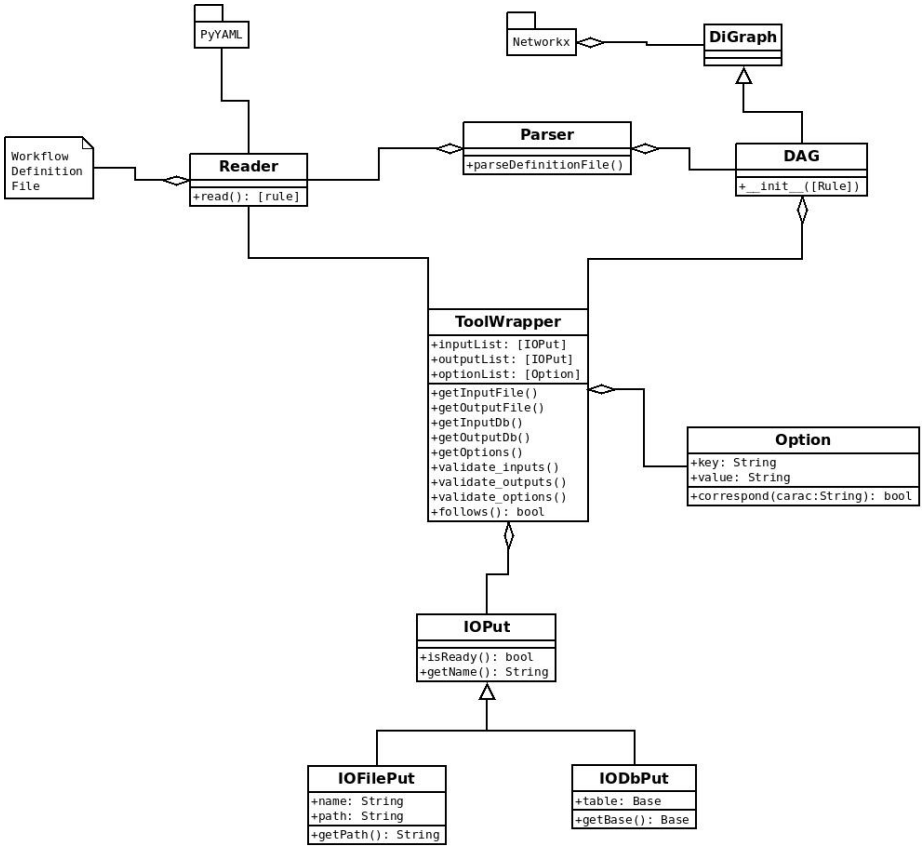

FIGURE C.5 – Diagramme de classe de la partie « Parsing du fichier de définition »

C.3.6 Diagramme d'objets réels d'un bloc « ToolWrapper »

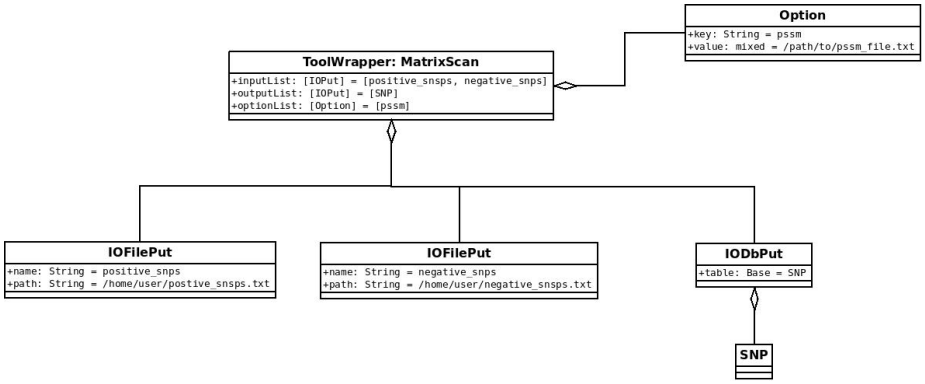

FIGURE C.6 – Diagramme d'objets réels de la partie « Parsing du fichier de définition »

C.3.7 Diagramme de processus

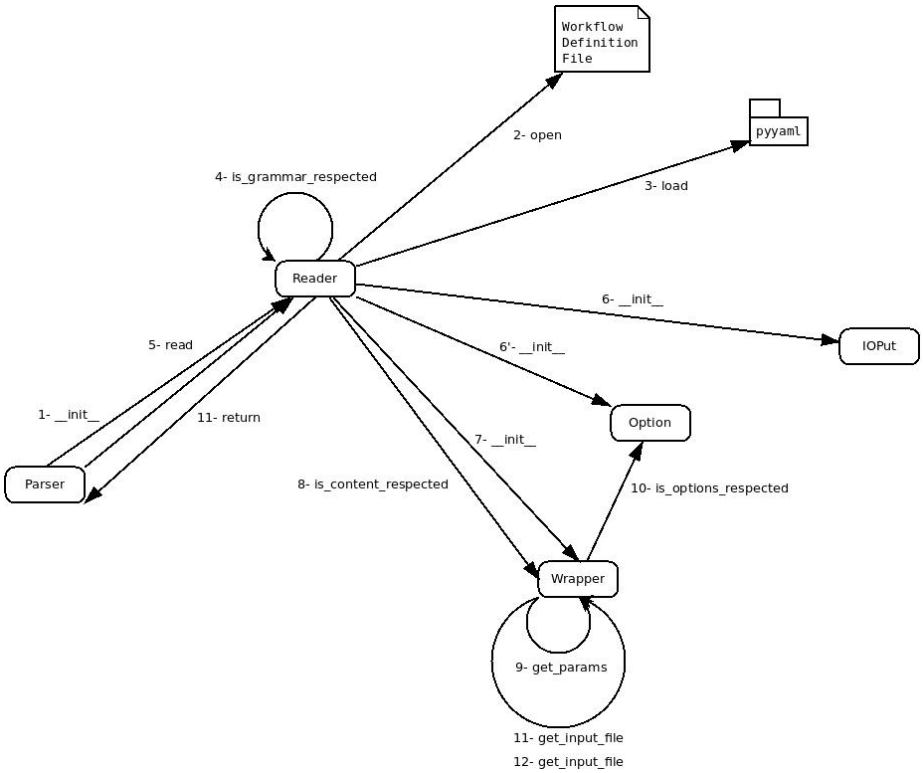

FIGURE C.7 – Diagramme de processus de la partie « Lecture du fichier de définition du workflow »

C.4 Gestion du workflow

La classe « WorkflowManager »est chargée de l'organisation du workflow. Elle utilise les classes suivantes :

— Une classe « DAG », construite dans la partie « Parsing du fichier de définition du workflow »qui lui permet d'accéder aux ToolWrappers à lancer, en descendant le Graphe d'exécution

— Une classe « Queue », qui lui permet de mettre les ToolWrappers à lancer en file d'attente, le temps que leurs Inputs soient prêts

C.4.1 WorkflowManager.run()

La classe WorkflowManager comporte une méthode « run »qui sera appelée par le main. Cette méthode permet d'organiser le déroulement du workflow :

— Tout d'abord, elle appelle la méthode « getRootNodes() »du graphe DAG pour obtenir les ToolWrappers à exécuter

— Ensuite, elle remplit la Queue avec les objets ToolWrappers obtenus

— Grâce à sa propre méthode « runQueue() », elle prend les éléments de cette Queue dans l'ordre et tente de les exécuter en utilisant leur méthode « start() »

Le WorkflowManager est ensuite en attente des signaux renvoyés par les ToolWrapper exécutées sur deux méthodes : « wrapperSucceeded() »et « wrapperNotReady() ».

C.4.2 WorkflowManager.runQueue()

Cette méthode est utilisée pour executer les wrappers contenues dans la Queue.

```
TANT QUE Queue.length > 0:
    wrapper <- on prend le premier \'{e}l\'{e}ment de la Queue
    on utilise la m\'{e}thode start de wrapper
```

C.4.3 WorkflowManager.wrapperSucceeded()

Cette méthode est appelée par une ToolWrapper si il a pu terminer son exécution. Le WorkflowManager reçoit donc l'objet et il obtient ses descendants directs dans le graphe d'exécution via la méthode successors(). Les successeurs sont ajoutés en fin de Queue et le WorkflowManager relance l'exécution des ToolWrappers de la Queue.

C.4.4 WorkflowManager.wrapperNotReady()

Cette méthode est appelée par un ToolWrapper qui a « invalidé »ses inputs et avorté son lancement. Le WorkflowManager remet le ToolWrapper en fin de Queue et relance l'exécution des ToolWrapper de la Queue. Le comportement de cette méthode sera différent dans le cas d'un environnement sériel et multiThreadé, mais en théorie, il devrait fonctionner dans les deux cas.

C.4.5 DAG

Le DAG est donné en argument lors de l'instanciation du WorkflowManager, il a été construit au cours de l'étape d'analyse du fichier de définition de workflow et il est supposé être correct (tous les tests ont été effectués au préalable).

Les successeurs d'un noeud sont obtenus grâce à la méthode héritée de DiGraph, « succesors() ».

D'autre part, une méthode getRootNodes() permet d'obtenir les noeuds dis « racine »du DAG, c'est à dire ceux vers lesquels ne pointent aucun arc.

C.4.6 Queue

La Queue est initialisée par le WorkflowManager. Il s'agit d'une classe built-in de Python (`https://docs.python.org/3/library/queue.html`), spécialement conçue pour les environnements multiThreadés. Ce n'est pas encore le cas du programme mais son utilisation permet d'anticiper cette amélioration et elle n'est pas incompatible avec un environnement sériel.

Les objets récupérés par le WorkflowManager grâce à la méthode successors() du DAG sont ajoutés à cette Queue à l'aide de la méthode « put() ».

Lors de l'exécution de la Queue avec la méthode « runQueue() », la méthode « remove() »est utilisée afin de retirer ET récupérer l'élément en tête de Queue.

C.4.7 ToolWrapper.start()

Pour permettre l'exécution de leur classe, les développeurs métiers devront implémenter la méthode « run() ». Cette méthode contiendra le code d'exécution de l'outil et la détection d'erreur relative à l'exécution même de l'outil revient au développeur métier qui devra lever une exception de type « ToolExecutionException »en cas de problème.

« run() »sera appelée par une méthode « start() »de l'objet « ToolWrapper »qui aura pour rôle de vérifier l'intégrité des inputs et de renvoyer des signaux au WorkflowManager pour lui indiquer l'état de l'outil. Ces signaux seront envoyés via l'appel des fonctions suivantes du WorkflowManager :

— wrapperNotReady() si les inputs n'étaient pas prêt
— wrapperSucceeded() si l'exécution a eu lieu

C.4.8 Diagramme de classes

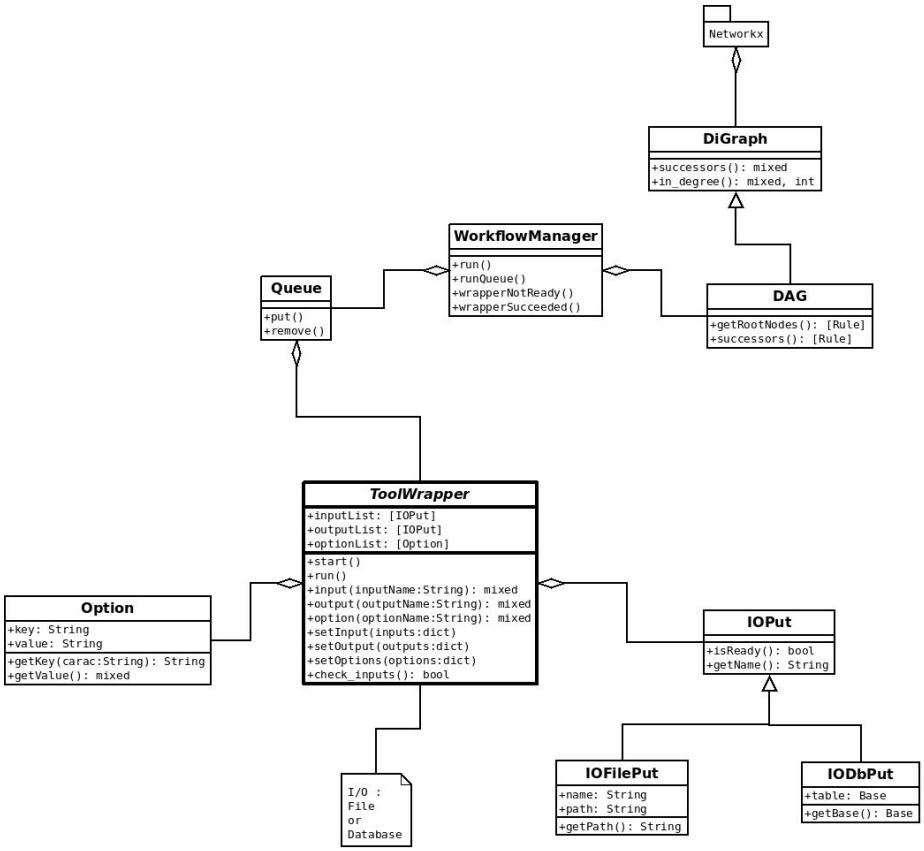

FIGURE C.8 – Diagramme de classe de la partie « Gestion du workflow »

C.4.9 Diagramme de processus

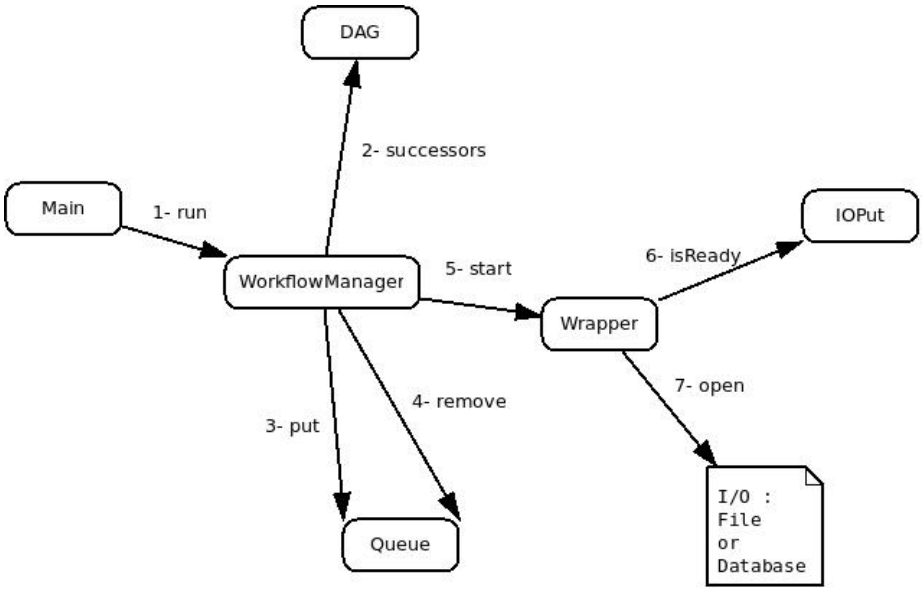

FIGURE C.9 – Diagramme de processus de la partie « Gestion du workflow »

C.5 Logging

La librairie utilisée pour le loging sera celle préconisée par défault pour Python : Logging. (`https://docs.python.org/3/library/logging.html`)

C.6 Base de données

Une solution de type « base de données »est utilisée pour assurer la persistance des données d'exécution. Cette base de données sera utilisée pour stocker les résultats mais aussi pour stocker le graphe d'exécution du workflow.

Le SGBD utilisé sera SQLlite. Ce choix est dû au fait que le fichier contenant la base est unique et donc facilement transportable et partageable. Ainsi, l'exécution d'un workflow conduit à la production d'un fichier de base de données.

Pour accéder à la base, l'ORM SQLAlchemy sera utilisé. Au delà du fait qu'il s'agit de la solution la plus utilisée avec Python, l'utilisation d'un ORM permet l'ouverture d'une session et son partage entre différents objets au sein d'un programme. Cette situation est particulièrement pratique dans notre cas car les classes héritantes de ToolWrapper devront être capable d'écrire dans une même base de données de façon simple et facilitée.

C.7 Patrons de Conception

C.7.1 Thread Pool

Le patron de conception Thread Pool sera utilisé pour stocker les outils prêts à être lancés dans une Queue.

Ce patron de conception est largement utilisé par les autres gestionnaires de workflow car il permet de facilement contrôler la distribution des ressources pour les différents outils.

La Queue utilisée est fournie par la librairie asyncio de Python et sera de type FIFO (First In First Out), c'est à dire que les premiers éléments ajoutés seront traités en premier, par opposition aux Queue LIFO (Last In First Out).

C.7.2 Observeur - Observé

Le patron de conception Observeur-Observé est particulièrement pratique dans le cadre d'une architecture de type « 1 Manageur - N travailleurs »où le Manageur joue le rôle d'Observeur et les wrappers, celui d'Observés.

C.7.3 Singleton

Le patron de conception Singleton est très connu et pratique, notamment dans le cadre de la configuration d'un programme. Pour économiser de la mémoire et éviter de transmettre les options à toutes les parties du code, il sera utilisé pour contenir les arguments passé en ligne de commande par l'utilisateur et les transmettre aux différentes parties du programme qui le nécessitent.

D Sprints

D.1 Sprint nº 1

D.1.1 Tâche Globale

Parsing du fichier de définition du workflow.

D.1.2 Sous tâches

Intitulé	Classes touchées	Volume de Temps	Risques	Priorité
Lecture du document - Yaml	Reader Parser	$\frac{1}{2}$ journée	librairie PyYaml inadaptée	**MAXIMUM**
Validation de la syntaxe du document	Reader	$\frac{1}{2}$ journée		minimum
Construction des Objets	Reader ToolWrapper IOPut IODbPut IOFilePut Option	$\frac{3}{4}$ journée		MEDIUM
Validation du contenu	Reader ToolWrapper IOPut IODbPut IOFilePut Option	$\frac{3}{4}$ journée		minimum

D.1.3 Date de démonstration

Jeudi 12/05 - 14h00 : Validé

D.1.4 FeedBack

Demande d'amélioration : Améliorer les exceptions qui déclarent les erreurs de définition du workflow

D.2 Sprint n° 2

D.2.1 Tâche Globale

Parsing du fichier de définition du workflow - construction du DAG.

D.2.2 Sous tâches

Intitulé	**Classes touchées**	**Volume de Temps**	**Risques**	**Priorité**
Amélioration des messages d'Exception pour qu'ils spécifient la rule qui pose problème	Reader ToolWrapper	$\frac{1}{2}$ journée	Impossibilité d'accéder à des informations plus précises à cause de la façon dont les choses sont faites actuellement	MEDIUM
Construction du DAG	DAG	$\frac{1}{2}$ journée	Librairie NetworkX innadaptée	**MAXIMUM**
Affichage du DAG	DAG	$\frac{1}{2}$ journée		minimum

D.2.3 Date de démonstration

Lundi 23/05 - 9h30 : inconnu

D.2.4 FeedBack

E Qualité du code

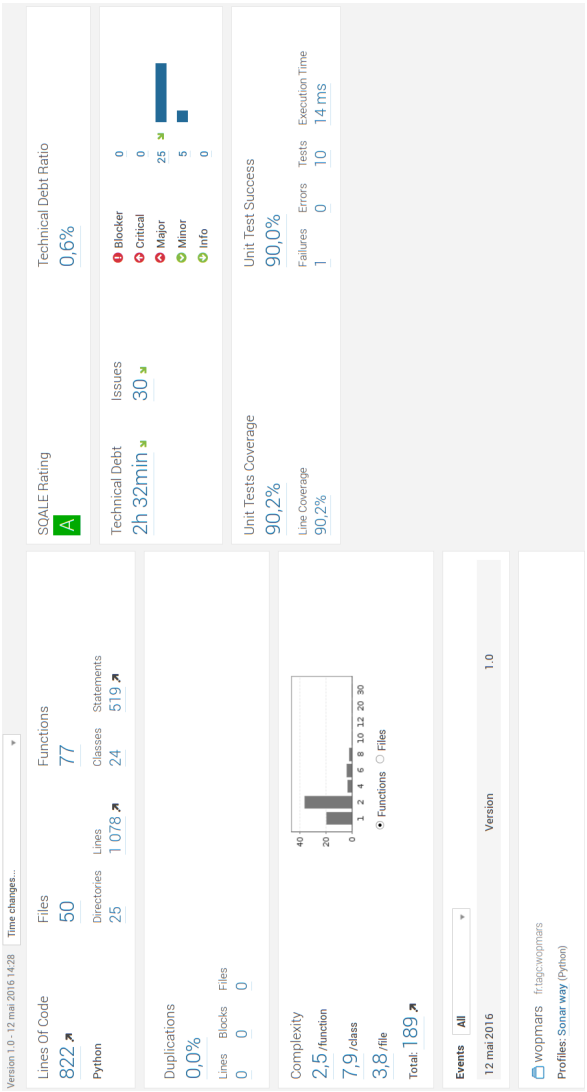

FIGURE E.10 – DashBoard obtenu après analyse par SonarQube à la fin du premier Sprint

F Bibliographie

[1] http ://tagc.univ-mrs.fr/tagc/.

[2] https ://fr.wikipedia.org/wiki/Graphe_orient%C3%A9_acyclique.

[3] https ://fr.wikipedia.org/wiki/Polymorphisme_g%C3%A9n%C3%A9tique.

[4] https ://fr.wikipedia.org/wiki/Machine_%C3%A0_vecteurs_de_support.

[5] https ://fr.wikipedia.org/wiki/Interface_de_programmation.

[6] https ://github.com/common-workflow-language/common-workflow-language/wiki/Existing-Workflow-systems.

[7] https ://github.com/pditommaso/awesome-pipeline.

[8] https ://en.wikipedia.org/wiki/Pydoc.

[9] https ://fr.wikipedia.org/wiki/Test_unitaire.

[10] https ://fr.wikipedia.org/wiki/Logiciel_de_gestion_de_versions.

[11] https ://fr.wikipedia.org/wiki/SonarQube.

[12] Donal Fellows Alan Williams David Withers Stuart Owen Stian Soiland-Reyes Ian Dunlop Aleksandra Nenadic Paul Fisher Jiten Bhagat Khalid Belhajjame Finn Bacall Alex Hardisty Abraham Nieva de la Hidalga Maria P. Balcazar Vargas Shoaib Sufi Katherine Wolstencroft, Robert Haines and Carole Goble. The taverna workflow suite : designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic Acids Research*, 2013.

[13] Nekrutenko A Taylor J Goecks, J and The Galaxy Team. Galaxy : a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biol*, 2010.

[14] Coraor N Ananda G Lazarus R Mangan M Nekrutenko A Taylor J Blankenberg D, Von Kuster G. Galaxy : a web-based genome analysis tool for experimentalists. *Current Protocols in Molecular Biology*, 2010.

[15] Hardison RC Burhans R Elnitski L Shah P Zhang Y Blankenberg D Albert I Taylor J Miller W Kent WJ Nekrutenko A Giardine B, Riemer C. Galaxy : a platform for interactive large-scale genome analysis. *Genome Research*, 2005.

[16] Johannes Köster. Parallelization, scalability, and reproducibility in next-generation sequencing analysis. *TU Dortmund*, 2014.

[17] Johannes Köster and Sven Rahmann. Building and documenting bioinformatics workflows with python-based snakemake. *Proceedings of the GCB*, 2012.

[18] https ://fr.wikipedia.org/wiki/Grappe_de_serveurs.

[19] https ://fr.wikipedia.org/wiki/Secure_Shell.

[20] http ://www.larousse.fr/dictionnaires/francais/input/43243.

[21] http ://www.larousse.fr/dictionnaires/francais/output/56941.

[22] https ://en.wikipedia.org/wiki/Thread_pool.

[23] https ://msdn.microsoft.com/fr-fr/library/3dasc8as.aspx.

[24] Andrew S. Tanenbaum Henri E. Bal, Jennifer G. Steiner. Dataflow variables are spectacularly expressive in concurrent programming.

[25] http ://www.jbpm.org/.

[26] https ://github.com/knipknap/SpiffWorkflow.

[27] https ://fr.wikipedia.org/wiki/YAML.

[28] https ://github.com/spotify/luigi.

[29] https ://fr.wikipedia.org/wiki/Hypertext_Transfer_Protocol.

[30] http ://www.openarchives.org/ore/1.0/toc.

Glossaire

API

Accronyme de Application Programming Interface - Interface de Programmation Applicative : Ensemble normalisé de classes, méthodes et fonctions qui sert de façade par laquelle un logiciel offre ses services à d'autres logiciels. [5].

BPMS

Acronyme de Business Process Management System.

CIML

Acronyme de Centre d'Immunologie de Marseille-Luminy.

CSV

Acronyme de Coma Separated Version.

DAG

Acronyme de Directed Acyclic Graph – Graphe Orienté Acyclique : En théorie des graphes, c'est un graphe orienté qui ne possède pas de boucle (ni simple, ni élémentaire). [2].

Ferme de calcul

Appelé aussi « grappe de serveurs »ou « cluster ». Désigne une technique consistant à l'utilisation de plusieurs ordinateurs indépendants pour augmenter la vitesse de calculs. [18].

HTTP

Acronyme de Hypertext Transfer Protocol : Protocole de communication client-serveur. [29].

Input

Anglicisme permettant de définir les données entrant dans l'exécution d'un programme (paramètres, matière première, etc.) [20].

INSERM

Acronyme de Institut National de la Santé et de la Recherche en Médecine.

Logiciel de gestion de version

Programme permettant de stocker un ensemble de fichiers en conservant un historique de ses modifications. [10].

ORM
Acronyme de Object Relational Mapping – Mapping Objet-Relationnel.

Output
Anglicisme permettant de définir le produit de l'exécution d'un programme. Opposé à Input.[21].

Patron de Conception
Implémentation générique pour répondre à un problème spécifique.

Polymorphisme génétique
Le concept de polymorphisme génétique désigne la coexistence de plusieurs versions d'un gène au sein d'une espèce. [3].

Product Owner
Personne représentant le client et connaissant ses besoins au sein d'une équipe de développement agile.

Pydoc
Module de documentation pour Python [8].

RDBMS
Acronyme de Relational DataBase Management System – Système de Gestion de Base de Données.

Scrum Master
Responsable de l'équipe Scrum.

SonarQube
Logiciel permettant de mesurer la qualité du code source en continu. [11].

SSH
Acronyme de Secure Shell : Protocole de communication sécurisé basé sur un système de cryptographie asymétrique. Il permet, entre autre, l'exécution de tâches sur des machines distantes. [19].

SVM
Acronyme de Support Vector Machine : Ensemble de techniques d'apprentissage supervisé destinées à résoudre des problèmes de classification ou de régression. [4].

TAGC
Acronyme de Technical Advances for Genomics and Clinics.

Test unitaire
Procédure permettant de vérifier le bon fonctionnement d'une partie précise d'un logiciel. [9].

Thread Pool
Collection de threads qui peut être utilisée pour effectuer plusieurs tâches en arrière-plan. La mise en thread pool représente une forme de multithreading où les tâches sont ajoutées à une file d'attente et automatiquement lancées lorsque les ressources sont disponibles. [22][23].

UML

Acronyme de Unified Modeling Langage: En informatique, est utilisé pour représenter graphiquement des systèmes d'information..

YAML

Acronyme de YAML Ain't Markup Language: Format de représentation de données par sérialisation . Il reprend des concepts d'autres langages comme XML. Son but est de représenter des informations plus élaborées que celles du format CSV tout en gardant une lisibilité meilleure que celle du XML. [27].

Acronymes

API
Application Programming Interface.

BPMS
Business Process Management System.

CIML
Centre D'Immunologie De Marseille-Luminy.

CSV
Coma Separated Version.

DAG
Directed Acyclic Graph – Graphe Orienté Acyclique.

HTTP
Hypertext Transfer Protocol.

INSERM
Institut National De La Santé Et De La Recherche En Médecine.

ORM
Object Relational Mapping – Mapping Objet-Relationnel.

RDBMS
Relational Database Management System – Système De Gestion De Base De Données.

SSH
Secure Shell.

SVM
Support Vector Machine.

TAGC
Technical Advances For Genomics And Clinics.

UML
Unified Modeling Langage.

YAML
Yaml Ain'T Markup Language.