PROJET DE GROUPE
BIOINFO

Rapport Final
V2 - 17/12/2012

Accompagnateurs:

Olivier Roux
Julien Gras
Maxime Folschette

Étudiants:

Jérémy Amar
Gaëtan Girin
Marlène Jaulin
Quentin Servais-Laval

OBJET DU DOCUMENT

Ce rapport a pour objet de présenter le travail de quatre étudiants de l'option informatique de l'Ecole Centrale de Nantes dans le cadre d'un projet de groupe, échelonné d'octobre à décembre 2012.

REMERCIEMENTS

L'équipe projet souhaite remercier M. Roux, M. Gras, et M. Folschette, pour leur disponibilité et l'attention qu'ils ont portée à nos avancées.

Nous remercions Mme Servières pour ses retours sur les diagrammes UML.

Nous remercions également Antoine Gersant, membre de l'équipe ayant développé la première version de l'application gPH, pour ses réponses sur certains points techniques.

Ce projet a permis aux membres de l'équipe de mettre en application des connaissances théoriques étudiées en cours et d'acquérir des compétences sur de nouveaux outils informatiques.

Nous avons pris plaisir à travailler avec les membres de l'équipe MeForBio et espérons que notre travail leur sera profitable.

SOMMAIRE

I PRÉSENTATION DU PROJET 5
I.1 Contexte et Cadre de Travail 5
I.2 Le modèle de frappes de processus 5
I.3 L'application gPH 7
I.3.1 Travail déjà réalisé 7
I.3.2 Le besoin exprimé 8
II OUTILS, LOGICIELS & DÉPENDANCES 10
II.1 *Frameworks* et bibliothèques 10
II.1.1 Axe 10
II.1.2 Boost 10
II.1.3 Graphviz 10
II.1.4 Framework Qt 10
II.2 Programme Pint 11
II.3 IDE : Qt Creator 11
II.4 Gestion de versions 11
II.5 Architecture globale 11
III LIVRABLES 13
III.1 Fonctionnalités développées 13
III.1.1 Déplacement des éléments du graphe 13
III.1.2 Définition des groupes de sortes 15
III.1.3 Réduction et déploiement des éléments du graphe 15
III.1.4 Modulation du facteur de zoom 16
III.1.5 Modification des couleurs 16
III.1.6 Visualisation du fichier texte parallèlement au graphe 17
III.1.7 Réduction et déploiement des panneaux latéraux 17
III.1.8 Persistance des méta-données de présentation 18
III.2 Production de documentation 19
III.2.1 Installation des bibliothèques et programmes 19
III.2.2 Remaniement des commentaires 19
III.2.3 Diagrammes UML 20
III.3 Pistes de solutions sur les demandes non traitées et conseils 20
III.3.1 Les commentaires TODO 20
III.3.2 Correction d'un bug : coïncidence entre une frappe et son saut 21
III.3.3 Coloration syntaxique du texte 21
III.3.4 Edition du graphe à partir de la version texte 21
III.3.5 Importation des préférences utilisateur 22
III.3.6 Rotation à 90° des sortes 22
IV DESCRIPTION DES PAQUETS 23
V ANALYSE CRITIQUE DU PROJET 25
V.1 Difficultés techniques 25
V.2 Management de projet / planning 26

V.3 Retour sur l'accompagnement....27
V.4 Comparaison entre quantité de travail prévue & effective....27
VI CONCLUSION....28
VII BIBLIOGRAPHIE & NETOGRAPHIE....29
VIII ANNEXES....30

I PRÉSENTATION DU PROJET

I.1 Contexte et Cadre de Travail

L'équipe MeForBio (Méthodes Formelles pour la Bio-Informatique) a développé au sein de l'IRCCyN une modélisation novatrice de représentation et d'analyse des réseaux biologiques de régulation (BRN) de grande ampleur : les frappes de processus (*Process Hitting*).

Les frappes de processus permettent de modéliser les mécanismes par lesquels des protéines sont générées depuis différents gènes, et l'influence qu'a l'émission d'une protéine particulière sur un gène donné. La simplicité du formalisme proposé par l'équipe MeForBio permet notamment d'effectuer des analyses statistiques efficaces de systèmes biologiques complexes.

Le logiciel Pint est l'entité qui rassemble divers programmes manipulant des modèles de frappes de processus (simulation stochastique, recherche des états stables, exportation/importation en divers formats, etc.). Des algorithmes efficaces d'analyse du modèle de frappes de processus y sont implémentés. Le modèle informatique développé par MeForBio, très complet, demeure toutefois difficile d'accès aux utilisateurs peu familiers des environnements en ligne de commande. Le logiciel étant conçu pour des biologistes, il n'est en l'état utilisable que par des bio-informaticiens.

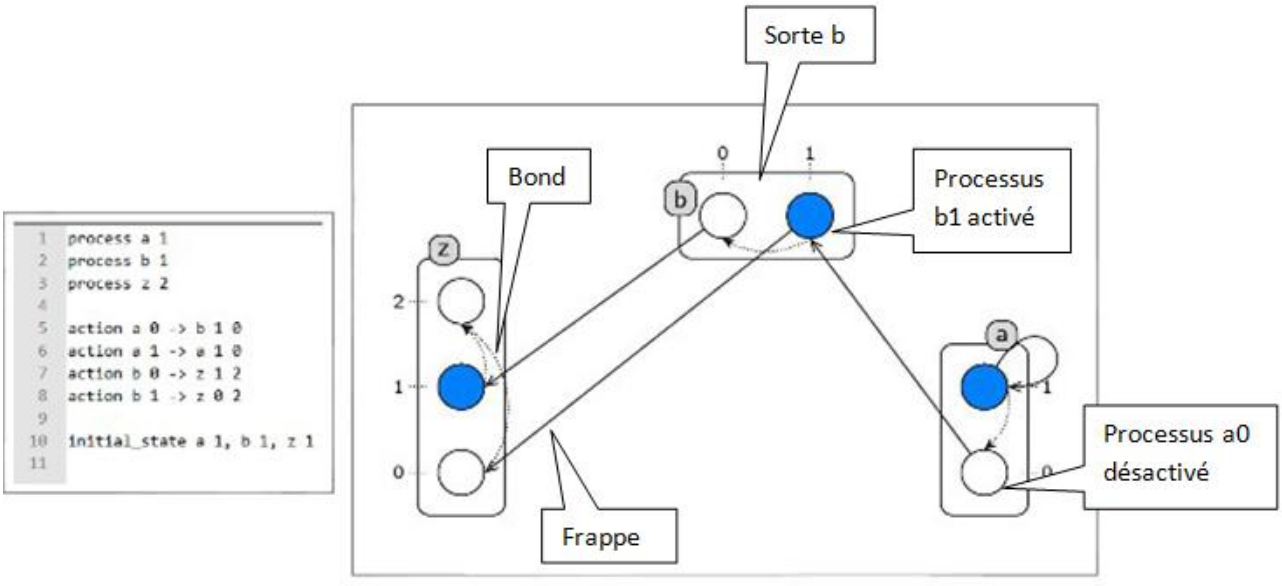
En 2011, l'équipe de MeForBio a fait appel à deux étudiants de troisième année de l'École Centrale de Nantes, dans le cadre d'un projet d'application, pour commencer le développement d'une interface graphique pour le logiciel Pint. Cette collaboration a notamment permis de construire un système d'affichage des frappes de processus, par le biais de la bibliothèque Graphviz.

En septembre 2012, pour enrichir les fonctionnalités de cette application graphique, le projet a été de nouveau proposé aux élèves de l'option informatique de l'Ecole Centrale de Nantes. L'objectif principal est alors de rendre dynamique la manipulation des différents objets qui rentrent en jeu dans les modèles de frappes de processus.

I.2 Le modèle de frappes de processus

Le modèle de frappes de processus permet la modélisation des réseaux biologiques de régulation (BRN).

L'objectif étant de décrire mais aussi d'établir une interprétation abstraite des scénarios de frappes de processus pour en approximer les états de stabilité, le formalisme employé et les représentations utilisées sont les suivants :



Représentation d'un modèle de frappe de processus

- les sortes regroupent un ensemble fini de processus (exemple : la sorte b regroupe 2 processus, b_0 et b_1),
- à chaque instant, un et un seul processus de chaque sorte est activé,
- les actions sont de la forme "processus de a_k frappe le processus b_i pour le faire passer à b_j" où a et b sont des sortes (une action correspond à une frappe suivie d'un bond (ou saut)).

Le modèle permet de déterminer :

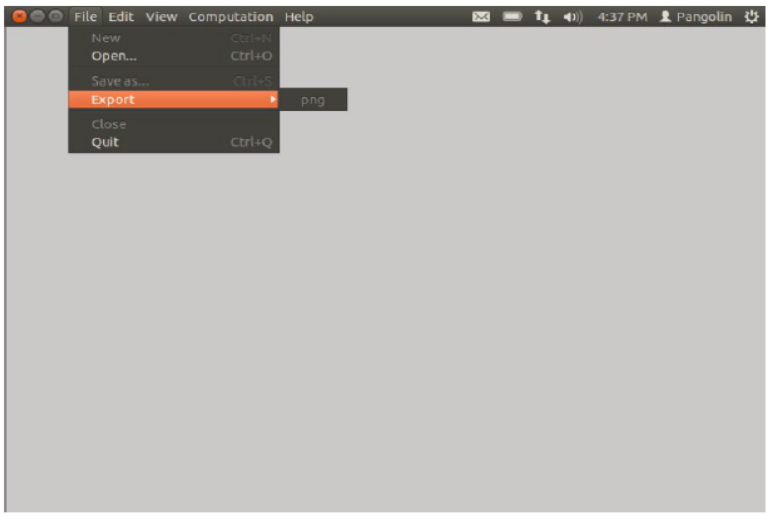
- une liste des états stables (ou points fixes),
- une analyse d'accessibilité d'un état donné,
- des paramètres temporels et stochastiques pour les actions.

I.3 L'application gPH

I.3.1 Travail déjà réalisé

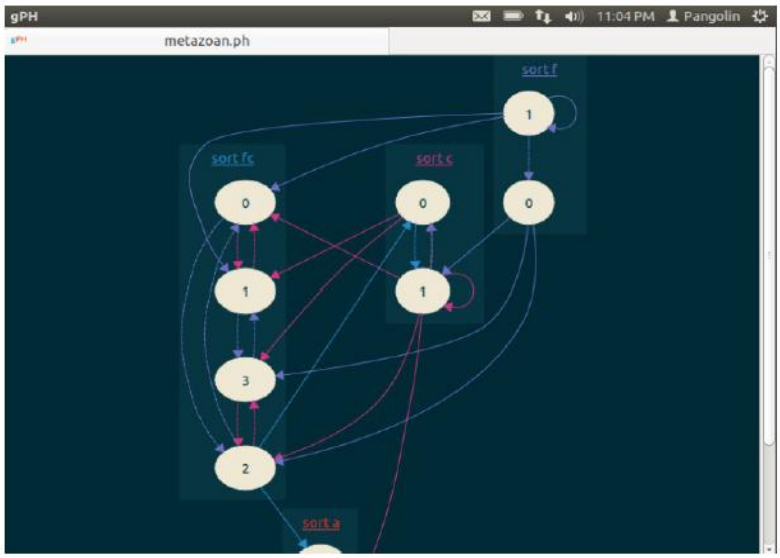
Lors du précédent projet, les bases de l'interface graphique ont été posées. Le *framework* Qt a été choisi pour construire cette interface et la bibliothèque Graphviz pour calculer les graphes. Au terme du projet initié en 2011, l'application gPH permettait d'afficher une représentation graphique statique d'un modèle de frappes de processus, avec les processus, les sortes et les actions. Cette représentation graphique étant construite à partir d'un fichier source .ph (syntaxe décrite ici : http://processhitting.wordpress.com/doc/language/).

Le mécanisme de *parsing* d'un fichier .ph, la construction d'une abstraction objet d'un modèle de frappes, l'appel à Graphviz pour le calcul du graphe, la traduction des sorties de Graphviz en éléments Qt visualisables dans une fenêtre ont été mis en place. Une première architecture des menus de la fenêtre principale a été élaborée et certaines fonctionnalités de Pint y ont été reliées telles que le calcul des points fixes. Les graphes étaient donc générés, non-modifiables mais exportables au format .png.



aucun onglet ouvert

Menu principal avant nos travaux



Visualisation d'un graphe avant nos travaux

L'interface permettait donc l'ouverture d'un fichier .ph et l'affichage du graphe correspondant. Comme indiqué plus haut, le graphe était statique, sans possibilité de modifier les couleurs ou les positions des éléments, et il était impossible de zoomer, ce qui posait problème lors de l'ouverture de grands graphes.

I.3.2 Le besoin exprimé

Les besoins exprimés dans le cadre de notre projet ont principalement concerné l'implémentation de fonctionnalités concourant à plus de dynamisme et d'interactivité des graphes affichés par gPH. Ce logiciel prétend s'adresser aux néophytes de l'informatique, il doit par conséquent disposer d'une interface homme-machine simple d'utilisation.

La quasi-totalité des demandes traitées a eu pour objet de permettre à l'utilisateur d'adapter le graphe à sa convenance, en termes de styles ou de mise en page. Et ce pour accroître la lisibilité et soigner la présentation du graphe. Cela implique de pouvoir déplacer les sortes dans le plan, définir des groupes de sortes et modifier les couleurs et la visibilité de ces éléments.

Ci-dessous figure la liste des principaux besoins exprimés par l'équipe MeForBio :

- cliquer-glisser (*drag and drop*) des éléments d'un modèle,
- zoom +/-,
- définition de groupes de sortes,
- réduction et déploiement des éléments du graphe,
- modification des couleurs (sortes, groupes de sortes, fond d'écran...),
- visualisation du fichier texte parallèlement au graphe,
- persistance des méta-données de présentation.

Ci-dessous figure une liste d'autres fonctionnalités que l'équipe MeForBio trouvait intéressantes pour le projet :

- coloration syntaxique du fichier texte,
- édition du graphe à partir de la version texte,
- branchement de traitements (toutes les commandes du logiciel Pint ne sont pas disponibles dans l'interface),
- enregistrement en différents formats,
- gestion d'autres modèles de graphes (il existe plusieurs formalismes pour représenter les réseaux de régulation biologiques),
- détermination des états atteignables (utilisation d'une commande Pint, pas encore implémentée dans l'interface).

Pour répondre à ces attentes, nous avons construit un cahier des charges et élaboré un planning prévisionnel comportant 6 phases principales :

- Transfert de compétences,
- Rédaction des spécifications,
- Développement, avec revue régulière des spécifications,
- Recette du logiciel,
- Rédaction du rapport et consolidation de la documentation,
- Préparation de la soutenance.

II OUTILS, LOGICIELS & DÉPENDANCES

II.1 *Frameworks* et bibliothèques

II.1.1 Axe

Axe est un générateur de *parser* écrit en C++11. Il est utile pour parcourir et analyser les fichiers .ph. Dans l'application gPH, il sert à créer une abstraction objet d'un modèle de frappes de processus à partir d'un fichier .ph (cf. rapport final et documentation du projet précédent dans les Annexes).

II.1.2 Boost

Boost est un ensemble de bibliothèques libres écrites en C++ qui étendent les fonctionnalités du langage. Le projet utilise les bibliothèques suivantes :

- *Smart Pointer* : permet l'utilisation de pointeurs avec compteur de référence pour optimiser la gestion de la mémoire,
- *Filesystem* : permet l'accès au système de fichiers,
- *Exception* : étend le fonctionnement de la levée d'exceptions.

II.1.3 Graphviz

Graphviz est une bibliothèque écrite en C qui permet de calculer des représentations graphiques de graphes (au sens mathématique du terme) selon différents algorithmes de positionnement. L'API de cette bibliothèque n'est disponible qu'en C.

II.1.4 Framework Qt

Qt est un *framework* open source écrit en C++ qui permet de développer des interfaces graphiques indépendamment de la plate-forme ciblée. L'application développée requiert la version 4.8 du logiciel Qt.

II.2 Programme Pint

Pint est un *framework* qui fournit des outils en ligne de commande utiles au traitement des graphes de frappes de processus. Pour effectuer ces traitements, l'application fait appel aux exécutables du projet Pint, notamment phc, ph-exec, ph-stat et ph-stable.

II.3 IDE : Qt Creator

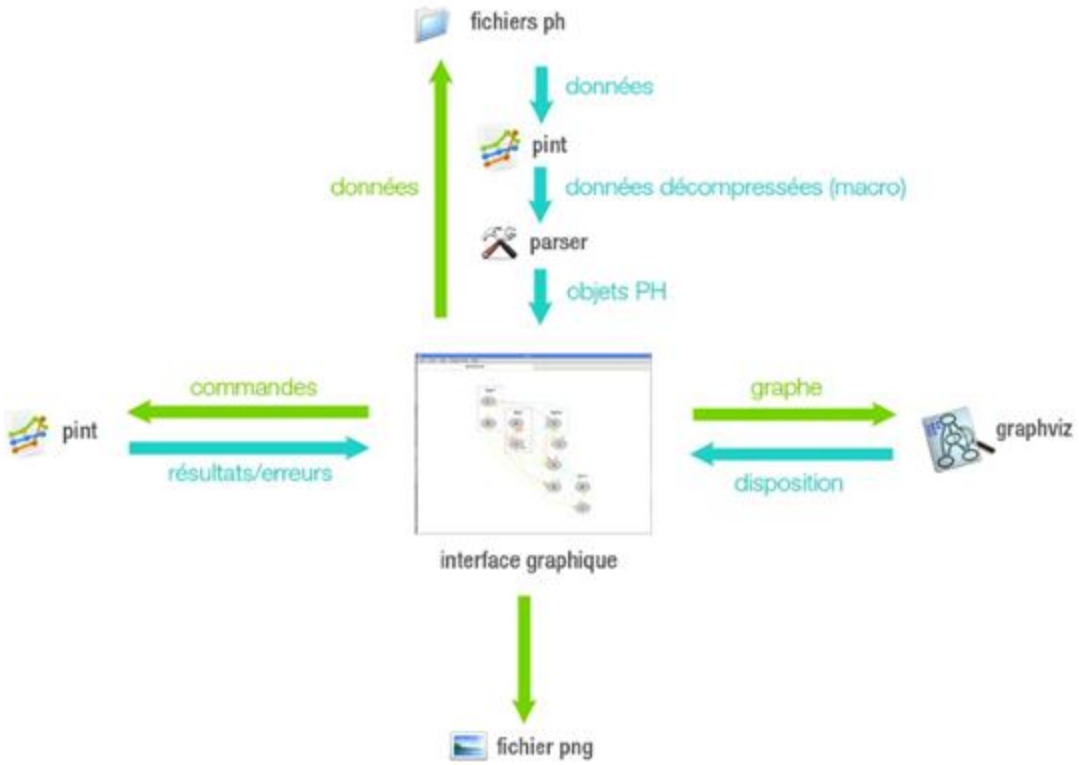
L'utilisation du *framework* Qt entraîne mécaniquement le choix de l'environnement de développement intégré Qt Creator, qui lui est dédié. Nous avons utilisé la version 2.4.1, non commerciale.

II.4 Gestion de versions

Afin de gérer les différentes versions de la base de code et de permettre un développement collaboratif, nous avons utilisé le programme Git avec un dépôt web public sur le site Github.

II.5 Architecture globale

Le schéma ci-dessous récapitule les différents échanges entre modules et bibliothèques qui se produisent lors de l'utilisation de l'application. Le code source de l'application est réparti en différents paquets qui correspondent à ces fonctionnalités.



Vue globale du système

Source : *Projet d'application InterfaceBioInfo*, Antoine Gersant, Karlotcha Hoa, 2012

III LIVRABLES

III.1 Fonctionnalités développées

III.1.1 Déplacement des éléments du graphe

Le développement de la fonctionnalité de glisser/déposer des sortes dans le graphe s'est fait en trois phases.

III.1.1.1 Glisser/déposer une sorte

Dans l'état initial de l'application, les objets graphiques représentant les processus (instances de la classe *GProcess*) et ceux représentant les sortes (instances de *GSort*) étaient indépendants. Ainsi, le déplacement du rectangle d'une sorte ne se propageait pas aux ellipses des processus de la sorte. Dans le constructeur de la classe *GSort*, il a donc fallu expliciter le lien entre les objets graphiques des sortes (parents) et ceux des processus (enfants) :

```
process->getGProcess()->getDisplayItem()->setParentItem(this);
```

Nous avons fait hériter directement la classe *GSort* de la classe *QGraphicsItem*, ce qui permet d'avoir accès aux gestionnaires d'événements souris. En effet, pour réaliser le glisser/déposer (*drag&drop*) sur les sortes, le choix a été fait d'implémenter les méthodes virtuelles dévolues à la gestion des clics de la souris, héritées de *QGraphicsItem* : *mousePressEvent()*, *mouseMoveEvent()* et *mouseReleaseEvent()*. En jouant sur les coordonnées de l'objet *GSort* et celles liées à l'événement, il est possible de contraindre l'élément à se déplacer avec la souris. Ces méthodes permettent de déclencher des actions ou de modifier des comportements lorsque l'utilisateur clique ou relâche le bouton de sa souris. Par exemple, le curseur de la souris devient une main fermée lors du clic, grâce à la méthode *setCursor()* héritée de *QGraphicsItem*. En outre, chose plus intéressante, des méthodes de la scène (*PHScene*) sont exécutées : *hideActions()* pour cacher les flèches lorsque le glisser/déposer commence, et *updateGraph()* pour recalculer les positions des éléments lorsque la sorte est déposée. On peut noter également l'appel à *showActions()* s'il s'agit simplement d'annuler le déplacement et de revenir dans l'état initial.

III.1.1.2 Mettre à jour le graphe

Comme évoqué précédemment, c'est la méthode *updateGraph()* de la classe *PHScene* qui déclenche la mise à jour du graphe. Cette mise à jour consiste à créer un nouveau graphe Graphviz par l'intermédiaire des classes des paquets gviz et ph. Cette méthode ressemble à *doRender()*, mais au lieu d'appeler la méthode *PH::toGVGraph()*, elle appelle la méthode *PH::updateGVGraph()*.

Cette dernière s'appuie sur Graphviz pour calculer les positions des nœuds (processus) et des arêtes (flèches des actions) dans un graphe sans clusters (équivalents des sortes). En effet, les clusters sont des sous-graphes, au sein desquels il est possible de contraindre les positions relatives des nœuds ; mais l'algorithme fdp de Graphviz utilisé pour calculer le graphe ne permet pas de contraindre les positions même de ces clusters dans le graphe.

Ce sont donc les coordonnées absolues des processus dans la scène qui sont utilisées pour calculer un graphe où elles sont complètement contraintes (en x et en y), n'entraînant que la mise à jour des flèches. L'opération de mise à jour des flèches est en réalité complexe, car il faut éviter les chevauchements et optimiser les chemins parmi les autres éléments du graphe. C'est pourquoi nous avons fait le choix de conserver l'algorithme de Graphviz pour cette tâche, ce qui de surcroît assure une cohérence esthétique avec le premier graphe calculé. Les positions ainsi calculées sont ensuite adaptées à l'environnement Qt (méthodes *GVSubGraph::nodes()* et *GVSubGraph::edges()*). Enfin, les coordonnées des objets *GSort* (qui contraignent les *GProcess*) et *GAction* sont mises à jour.

III.1.1.3 Empêcher le chevauchement des processus

Bien qu'il soit possible de contraindre les positions des nœuds du graphe, Graphviz traite en priorité d'autres contraintes, comme l'espacement minimum entre les nœuds, fixé à l'aide de l'attribut *sep* dans *GVSubGraph::setGraphAttributes()*. Ainsi, si l'utilisateur place deux processus trop près l'un de l'autre, Graphviz forcera la valeur minimale de leur espacement mutuel en appliquant un coefficient multiplicateur à toutes les coordonnées du graphe. De ce fait, la disposition relative des processus est bien conservée (le graphe garde la même "forme" générale), mais les coordonnées sont multipliées, ce qui provoque un "éclatement" du graphe dans le plan.

Pour empêcher cela, il a fallu restreindre l'utilisateur dans sa liberté de déplacer les sortes. Dans la méthode *GSort ::mouseReleaseEvent()*, on détecte les éventuels chevauchements des zones de marge des processus de la sorte sur le point d'être déposée avec les zones de marge des processus d'autres sortes. Cette détection se fait via la méthode *checkCollisions()* de la classe *GProcess*, qui elle-même fait appel à la méthode *collidingItems()* héritée de *QGraphicsItem*. Pour chaque objet *GProcess*, la zone de marge est matérialisée par l'objet *marginRect* de la classe *QGraphicsRectItem*. Les collisions entre zones de marge sont

détectées parmi toutes les autres collisions grâce à une propriété spécifique des objet *marginRect*, fixée à l'aide de la méthode *setData()* dans le constructeur de *GProcess*.

III.1.2 Définition des groupes de sortes

Pour définir les groupes de sortes nous avons décidé d'utiliser des *QTreeWidget*.
Le lien entre ces *widgets* graphiques et le graphe affiché est le nom des sortes, supposé unique.

Plusieurs fonctionnalités sont possibles pour l'utilisateur :

- Ajouter un groupe en précisant son nom. A ce groupe sera attribuée automatiquement une couleur à partir d'une palette prédéfinie.

- Ajouter une ou plusieurs sortes à un groupe. La couleur du groupe sera attribuée à la bordure du rectangle de chaque sorte ajoutée. Les groupes sont mutuellement exclus, c'est-à-dire qu'une même sorte ne peut pas appartenir à plusieurs groupes.

- Supprimer un groupe ou une sorte au sein d'un groupe. La suppression d'une sorte redonnera automatiquement à sa bordure sa couleur d'origine.

- Chercher une sorte à partir d'une partie de son nom.

Les *QTreeWidget* ainsi que les *QPushButton* et le *QTextEdit* permettant la recherche de sortes sont inclus dans un *widget* global, au même niveau que le graphe dans la fenêtre principale.

Pour manipuler les sortes et les groupes de sortes à partir des arbres, une recherche est systématiquement effectuée entre le nom du *QTreeWidgetItem* sélectionné et la *GSort* du même nom.
Ce système est efficace dans notre cas mais le serait moins dans le cas de graphes plus importants et de méthodes plus lourdes. Un moyen d'optimiser la manipulation des sortes par les arbres serait de lier dans le code ces *QTreeWidgetItem* avec leurs *Sort* associées.
Il serait également intéressant de définir une classe *SortGroup* ou *GSortGroup* et de lier ses éléments avec les *QTreeWidgetItem* de l'arbre permettant la définition et la manipulation des groupes.

III.1.3 Réduction et déploiement des éléments du graphe

Le logiciel permet à l'utilisateur de cacher des sortes, des groupes de sortes qu'il a définis, et que les actions liées aux sortes cachées ne soient pas affichées. Les sortes et les groupes cachés sont reconnaissables par leur police en italique dans les arbres les contenant.

De manière générale cette fonctionnalité a été construite autour des méthodes *hide()* et *show()* des classes *QGraphicsItem*, dont dérivent les éléments qui s'affichent dans le graphe, et *GSort*, qui redéfinit ces méthodes en jouant uniquement sur l'opacité (ceci afin de permettre la détection des collisions des objets *marginRect* des *GProcess* entre eux). Une des difficultés dans le développement de cette fonctionnalité a été la gestion de l'affichage et du masquage des actions liées à des sortes, qui peuvent être visibles ou non.

III.1.4 Modulation du facteur de zoom

Le zoom développé se compose de trois fonctionnalités différentes:

- Zoom positif (respectivement zoom négatif) à partir d'un clic de menu ou de Ctrl++ (Ctrl+-), à valeur fixée *1,2 (*1/1,2), la fenêtre après le zoom étant centrée au même endroit que la fenêtre avant le zoom.
  Cette fonctionnalité utilise simplement la méthode *scale()* de la classe *QGraphicsView*, ainsi que des signaux et slots.

- Zoom positif (respectivement zoom négatif) à partir d'un Ctrl + rotation de la molette dans le sens positif (resp. négatif), la fenêtre après le zoom étant centrée sur la position du curseur dans la fenêtre avant le zoom.
  Cette fonctionnalité utilise également la méthode *scale()* de *QGraphicsView*, mais elle met également en jeu la redéfinition du *wheelEvent* de cette même classe, permettant d'appeler une méthode à partir du mouvement de la molette.

- Adaptation de la vue à partir d'un clic de menu ou de Ctrl+L, qui permet d'adapter la fenêtre pour afficher l'ensemble des sortes.
  Cette fonctionnalité utilise la méthode *fitInView* de la class *QGraphicsView*, avec les arguments adéquats.
  Une précision du cahier des charges était de mettre en place un zoom efficace et sans temps de latence, ce qui est le cas.

III.1.5 Modification des couleurs

La modification des couleurs apparaît à plusieurs niveaux :

- Le choix des couleurs globales d'affichage, permettant de choisir la couleur de fond du graphe, de choisir la couleur des rectangles de l'ensemble des sortes, mais aussi d'utiliser des styles définis par défaut (contraste positif, contraste négatif, impression).

- Le choix de la couleur du rectangle d'une sorte précise, dont l'utilisateur a sélectionné le nom dans l'arbre.

- Le choix de la couleur d'un groupe de sortes, permettant de distinguer ces sortes par la couleur de leur bordure.

Ces fonctionnalités sont construites à l'aide du *QFontDialog* qui permet très simplement de recueillir le choix de couleur d'un utilisateur.
La couleur choisie est alors appliquée sur les éléments correspondant, grâce aux fonctions *setColor(QColor)* et *setBrush(QBrush)* des classes *QGraphicsItem*, dont dérivent les éléments qui s'affichent dans le graphe.

La couleur d'une sorte ou d'un groupe de sortes, lorsqu'elle a été modifiée, est appliquée à la police du *QTreeWidgetItem* correspondant à cette sorte ou à ce groupe. Cet affichage est permis grâce à la méthode *setForeground*.

III.1.6 Visualisation du fichier texte parallèlement au graphe

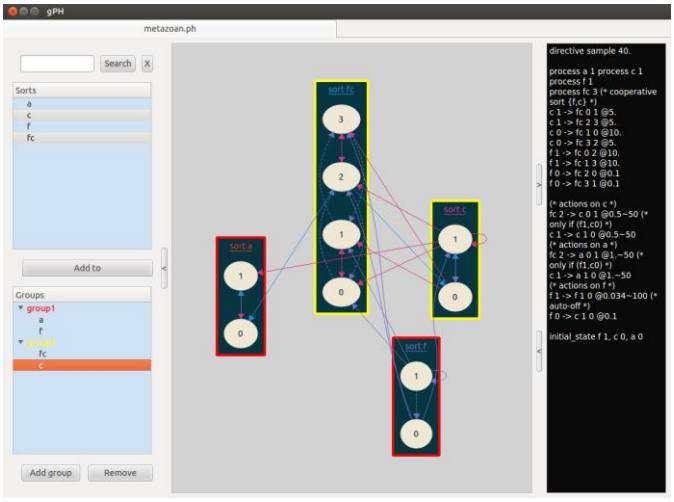
Il est à présent possible de visualiser le fichier texte correspondant au fichier .ph en parallèle de la manipulation du graphe. Le *widget* utilisé est un *QTextEdit*, rempli à l'aide d'un *QByteArray.* Nous avions dans un premier temps permis l'affichage du texte provenant du dump (décompression des macros par la commande phc de Pint), utilisé pour faire calculer à Graphviz la disposition du graphe.

Cependant, il s'avère que ce texte soit incompréhensible pour un scientifique qui manipule les frappes de processus. Nous avons donc permis l'affichage du texte original du fichier .ph, tout en laissant l'ancien code commenté en cas d'une possible réutilisation.

Le *QTextEdit* est placé au même niveau que le graphe et le *widget* contenant les *QTreeWidget* dans la fenêtre principale.

III.1.7 Réduction et déploiement des panneaux latéraux

La modification principale subie par l'affichage graphique est la séparation en trois parties principales à l'ouverture du fichier .ph, comme sur l'impression d'écran ci-dessous.



Affichage de la fenêtre principale avec panneaux latéraux

Nous avons cependant laissé le choix à l'utilisateur d'afficher ou non les deux nouveaux volets, à savoir le volet de gestion des groupes et le volet d'affichage du texte.

Des *QPushButton* sont donc présents aux extrémités du graphe et offrent trois possibilités:

- Cacher le volet de gestion des groupes de sortes (dépendant, l'affichage ou non de ce volet est le même sur tous les onglets).
- Cacher le volet d'affichage de texte (dépendant, l'affichage ou non de ce volet est le même sur tous les onglets).
- Etendre le volet d'affichage de texte (à une autre taille fixée).

La taille de ces volets a été fixée volontairement grâce à des *QVBoxLayout* imbriqués dans un *QHBoxLayout* global. Qt semble ne pas gérer de manière efficace le redimensionnement dynamique des onglets, et nous n'avions pas le temps de nous attarder plus précisément sur cette question. Elle peut cependant, pour le confort de l'utilisateur, être développée lors de projets futurs.

III.1.8 Persistance des méta-données de présentation

Il est intéressant pour l'utilisateur, après avoir manipulé son graphe, de pouvoir sauvegarder ses préférences (définition des groupes, couleurs, positions, etc.). Ces préférences ne pouvant être intégrées au fichier .ph, en vertu du principe de séparation des données et de leur présentation, il a été nécessaire de définir le format des fichiers vers lesquels l'utilisateur peut exporter l'état courant du graphe.

Nous avons choisi le format XML car il est très connu et il existe des méthodes permettant de le manier dans Qt. Le format XML est également très facilement manipulable et les méthodes d'exportation des préférences pourront être mises à jour rapidement lors du développement des fonctionnalités du logiciel.

Pour ce faire, nous avons utilisé un *QFileDialog* et les méthodes de la classe *QXmlStreamWriter*. Ces méthodes sont très faciles à utiliser et ne nécessitent aucune documentation. Toutefois, leur caractère verbeux rend la génération d'un fichier XML assez fastidieuse.

La structure du fichier XML d'exportation est présentée en Annexe.

Nous n'avons malheureusement pas eu le temps d'implémenter les méthodes d'importation des préférences de l'utilisateur, cette fonctionnalité ayant été la dernière que nous ayons traitée, après avoir pris du retard à cause des phases d'acquisition de compétences et de documentation du code qui ont été plus longues que prévues. Ces méthodes pourront cependant faire l'objet de développements futurs, et représentent un exercice assez formateur sur le maniement de Qt et la compréhension du fonctionnement de l'application gPH.

III.2 Production de documentation

III.2.1 Installation des bibliothèques et programmes

La documentation générale fournie par le groupe précédent ne nous a pas paru suffisante ou assez précise pour prendre en main le projet. Nous avons donc décidé de rédiger plusieurs documentations assez accessibles pour permettre à tout utilisateur souhaitant utiliser l'application gPH d'installer simplement les bibliothèques et les dépendances nécessaires.

III.2.2 Remaniement des commentaires

Nous avons également manqué de documentation lorsque nous avons souhaité comprendre le fonctionnement de certaines parties du code. Pour faciliter la maintenabilité et la compréhension du code dans son ensemble, nous avons décidé de produire une documentation plus riche et structurée.

Pour cela, nous avons utilisé Doxygen sous Ubuntu et avons instauré des conventions d'écriture. La rédaction de la documentation originale du code que nous n'avions pas produit a été un travail assez long et fastidieux. Nous avons ensuite pris l'habitude de documenter toutes les nouvelles classes et méthodes que nous implémentions, ce qui permet d'avoir une documentation complète et exhaustive.

Une documentation pour la génération automatisée de la documentation Doxygen sous Ubuntu est jointe en Annexe.

III.2.3 Diagrammes UML

Lors de la découverte du code, nous ne disposions pas de diagramme de classes et nous en avons manqué. Nous avons spontanément pris la décision d'en rédiger un et de le mettre à jour au fur et à mesure de nos modifications.

Dans le cadre du projet de l'enseignement GELOL, il nous était demandé de rédiger ce diagramme des classes ainsi que d'autres diagrammes UML offrant une bonne compréhension de l'utilisation et du fonctionnement du logiciel.

Nous en avons donc profité pour améliorer le diagramme des classes et pour construire les diagrammes suivants :

- Diagramme des cas d'utilisation : présente une vue globale du système ainsi que les interactions avec les différents acteurs de son environnement. Il présente également les principales fonctionnalités de l'interface graphique ainsi que les bibliothèques exploitées.

- Diagrammes d'états-transitions : décrivent les possibilités de l'Interface Homme-Machine aux niveaux de la fenêtre de l'application, des onglets, des panneaux latéraux et des menus, d'un point de vue utilisateur. Ces diagrammes sont assez génériques.

- Diagrammes d'activités : décrivent les principales opérations de manipulation d'un graphe au sein d'un onglet (manipulation globale, zoom, couleurs globales, export, glisser-déposer et personnalisation de la couleur d'une sort), d'un point de vue utilisateur.

- Diagramme de séquence : décrit le processus d'ouverture d'un fichier PH, de l'analyse du contenu du texte jusqu'à l'affichage, d'un point de vue principalement système.

III.3 Pistes de solutions sur les demandes non traitées et conseils

III.3.1 Les commentaires TODO

Des informations utiles pour le remaniement de certaines parties du code ou la correction de défauts sont disponibles dans les commentaires TODO laissés au sein du code source.

III.3.2 Correction d'un bug : coïncidence entre une frappe et son saut

Avec l'algorithme dot de Graphviz (utilisé initialement dans l'application), une solution a été trouvée pour faire coïncider la flèche d'une frappe avec la base du saut correspondant. Cette solution consiste à utiliser les attributs *samehead* et *sametail* de Graphviz.

En utilisant *samehead* pour faire converger deux arêtes (flèches) au même point, puis en utilisant l'attribut *dir* avec la valeur *"back"* pour retourner le sens d'une des deux flèches, on obtient un résultat esthétique, qui donne l'impression d'un rebond. Cette solution a été testée avec le programme Graphviz en ligne de commande mais elle n'a pas été intégrée à gPH, car l'algorithme dot ne permettant pas de forcer la position des nœuds, il a été abandonné au profit de fdp. Or l'algorithme fdp ne permet pas l'utilisation des attributs *samehead* et *sametail*.

Il est toutefois possible de contraindre la coïncidence des flèches avec les attributs *headport* et *tailport*. Mais cela ne donne pas directement un résultat esthétique, car les flèches traversent les processus et leurs tangentes au point de contact ne sont plus dirigées vers le centre du processus, ce qui supprime l'effet "rebond".

III.3.3 Coloration syntaxique du texte

Cette fonctionnalité serait assez simple à implémenter dans la mesure où la syntaxe du fichier .ph est bien définie. Cette syntaxe est décrite dans la documentation en ligne de Pint. Le gros du travail résiderait dans l'analyse automatique de cette syntaxe, qui peut par exemple être réalisée grâce à Axe (générateur de *parser* utilisé pour la conversion d'un fichier PH en objet).

La coloration du texte serait ensuite assez facilement réalisable grâce à la fonctionnalité *setHtml* de la classe *QTextEdit* que nous avons utilisée pour construire le volet d'affichage. Il faudrait donc *parser* un fichier .ph, l'exporter vers un document HTML et lui appliquer les couleurs choisies, avant de copier le HTML généré dans le *widget* d'affichage de texte.

III.3.4 Edition du graphe à partir de la version texte

Cette fonctionnalité demanderait quant à elle plus de travail. La structure qui a été mise en place et que l'on a enrichie fournit cependant des bases : le graphe est modifiable dynamiquement et le *widget QTextEdit* qui affiche actuellement le texte peut gérer dynamiquement les modifications de texte.

Pour l'instant, ce *widget* est placé en mode *ReadOnly*, ce qui ne permet pas de modifier le texte. Il suffirait de changer ce *flag* et de lier les nouvelles lignes créées à des modifications dans la *MyArea*.

III.3.5 Importation des préférences utilisateur

Tout comme l'a été l'exportation des préférences utilisateur (méta-données de présentation), le développement de la fonctionnalité d'importation des préférences utilisateur serait grandement facilité par la classe *QXmlStreamReader*, mais sa construction resterait assez laborieuse.

Pour certaines des préférences, il suffirait de reprendre la construction de la méthode d'export des données de présentation et de remplacer les *getters* par des *setters*. Mais pour d'autres, il serait nécessaire d'implémenter des nouvelles méthodes, notamment pour forcer la construction du graphe selon plusieurs caractéristiques.

III.3.6 Rotation à 90° des sortes

Les objets graphiques qui matérialisent les éléments du graphe sont des instances de la classe *QGraphicsItem* ou de classes dérivées. Ils disposent donc de méthodes de transformation géométrique comme *setRotation()*.

Toutefois, il semble plus judicieux d'utiliser les contraintes de Graphviz pour recalculer le graphe, en contraignant les processus d'une même sorte à s'afficher horizontalement. Cette méthode a le mérite de recalculer les trajectoires des flèches. Mais elle fait appel aux algorithmes de Graphviz dont l'exécution peut geler l'application au-delà d'un temps acceptable pour l'utilisateur lambda.

Pour que la rotation des sortes soit facilement déclenchée par l'utilisateur, il suffirait de lier cette action aux arbres du panneau latéral gauche ou à un clic droit direct sur les objets de la classe *GSort*.

IV DESCRIPTION DES PAQUETS

ph

Le paquet ph contient les classes qui servent à construire la représentation en mémoire d'un fichier .ph. Il contient les classes *PH*, *Sort*, *Process* et *Action*.

gfx

Le paquet gfx contient les classes qui permettent de fabriquer la représentation graphique d'un *Process Hitting* qui est synthétisée dans la classe *PHScene*. La classe *GSort* (respectivement *GProcess* et *GAction*) agrège un objet *Sort* (respectivement *Process* et *Action*) et les données d'affichage correspondant. *GSort* comprend aussi des méthodes pour la gestion des événements souris qui réalisent la fonctionnalité de glisser/déposer des sortes du graphe.

gviz

Le paquet gviz est largement inspiré de l'article *Using libgraph to represent a* Graphviz *graph* (http://mupuf.org/blog/article/34) qui en fournit une description presque complète. Ce paquet contient des structures de données qui permettent de manipuler la structure de graphes utilisée par Graphviz selon un modèle orienté objet (Graphviz étant écrit en C, il ne le permet pas directement). Ce paquet se veut indépendant de la problématique des frappes de processus. Dans ce paquet, les objets correspondant aux éléments du graphe sont appelés des *nodes* (équivalents des processus), des *clusters* (pour les sortes) et des *edges* (les actions).

io

Le paquet io contient les classes concernant l'entrée et la sortie de fichiers. La classe *IO* gère les entrées/sorties au niveau le plus bas tandis que *PHIO* contient le *parser* de fichiers .ph et les fonctions d'export au format .png et au format .xml.

ui

Le paquet ui regroupe toutes les classes relatives à l'interaction avec l'utilisateur. Il était auparavant composé uniquement de la *MainWindow*, qui représente la fenêtre principale, et de la *MyArea*, qui dérive de *QGraphicsView* et qui permet l'affichage du graphe.

Les fonctionnalités que nous avons développées ont apporté des modifications importantes à ce paquet. La *MainWindow* contient dorénavant un système d'onglets contenant chacun une *Area*. Cette *Area* contient quant à elle une *TreeArea*, une *MyArea*, ainsi qu'une *TextArea*, auxquelles viennent s'ajouter plusieurs *widgets* intermédiaires permettant la modification de l'affichage global.

Chaque *Area* est construite selon un *QHBoxLayout* qui contient (dans l'ordre) les classes suivantes, qui contiennent elles-mêmes les *widgets* présentés dans la deuxième ligne du tableau:

TreeArea	QWidget	MyArea	QWidget	TextArea
QTextEdit QPushButton QPushButton QTreeWidget QPushButton QTreeWidget QPushButton QPushButton	QPushButton	QGraphicsView	QPushButton QPushButton	QTextEdit

test

Le paquet test contient les classes relatives aux tests unitaires du programme.

V ANALYSE CRITIQUE DU PROJET

V.1 Difficultés techniques

La première difficulté technique a été de maîtriser le langage C++ et le *framework* Qt, qu'aucun de nous n'avait utilisé auparavant. Fort heureusement, ce *framework* est très utilisé, simple à prendre en main et nous avons pu trouver une aide précieuse sur les forums et dans la documentation officielle, qui est riche et complète.

La bibliothèque Graphviz nous a posé de nombreux problèmes dans les développements autour de la mise à jour du graphe après déplacement d'une sorte par l'utilisateur. L'implémentation ne correspond pas toujours à la documentation officielle et cette dernière s'avère parfois peu explicite voire incomplète : on peut ainsi trouver des informations essentielles sur des forums, que l'on ne retrouve que sous forme sibylline dans la documentation. Bien qu'offrant de nombreux paramétrages en termes de styles ou de disposition du graphe, les possibilités de Graphviz sont très restreintes en ce qui concerne la construction de graphes "dynamiques". C'est en effet une bibliothèque pensée pour générer et optimiser des graphes, potentiellement de grande taille, en un seul bloc et en prenant le temps de calcul nécessaire. Pour ce qui est de rendre le graphe interactif, les algorithmes utilisés semblent beaucoup trop lourds pour les modèles complexes (plusieurs dizaines de sortes).

De même, les possibilités de Graphviz en termes de contraintes (positions des nœuds, des clusters et des extrémités des arêtes) sont assez limitées. La multiplicité des algorithmes de calcul proposée par la bibliothèque complique la recherche de propriétés pertinentes car ces dernières sont souvent spécifiques à un nombre restreint d'algorithmes.

Dans une moindre mesure, l'utilisation des pointeurs intelligents de la bibliothèque Boost a également constitué une difficulté. Ils permettent certes une gestion efficace de la mémoire mais s'avèrent difficiles à manipuler quand il s'agit par exemple de leur appliquer des opérations de transtypage.

Nous avons regretté que l'équipe encadrante n'ait aucune connaissance de Qt, de Boost et de Graphviz, car ils ne comprenaient parfois pas les problèmes que nous rencontrions et n'étaient pas en mesure de nous apporter une aide technique.

V.2 Management de projet / planning

Lors de la négociation du cahier des charges, nous avions eu la prudence de distinguer deux lots de fonctionnalités à implémenter : un premier lot sur lequel nous nous engagions formellement et un deuxième lot facultatif. Il convient de souligner que nous avons développé la quasi-totalité des fonctionnalités du premier lot. En effet toutes les demandes de ce lot ont été traitées, en revanche les spécifications détaillées telles que convenues au début du projet n'ont pas toujours été suivies à la lettre. Certaines divergences s'expliquent par une revue des exigences, convenue d'un commun accord avec l'équipe encadrante.

Le projet étant encadré par un institut de recherche, le deuxième lot regroupait des fonctionnalités qui semblaient intéressantes pour le logiciel mais sur le développement desquelles nous ne nous engagions pas dans le cadre du projet de groupe. Nous avions toutefois à cœur d'apporter des éclairages techniques utiles pour les développements futurs, dans la mesure de nos connaissances.

Nous n'avons malheureusement pas eu le temps de commencer le développement de ce deuxième lot de fonctionnalités et ce pour trois raisons principales :

- Nous avions minimisé la durée de la phase d'acquisition de compétences en C++ et sur Qt. Nous avons chacun passé plus de 20 heures à découvrir ce langage et ce *framework*, souvent en parallèle de la découverte du code de l'application, soit plus d'un tiers du temps théoriquement alloué à ce projet. Sans compter le temps de prise en main d'outils tels que Boost ou Graphviz.

- Nous n'avions pas pris en compte l'important travail de documentation qu'il était nécessaire de fournir. En effet, nous avons rédigé le cahier des charges avant même de nous plonger dans le code, alors qu'une découverte préalable nous aurait permis d'identifier les carences qui subsistaient, d'autant plus qu'une documentation complète est un élément fondamental dans le cadre d'un projet de recherche, sur lequel est amenée à travailler une multitude de personnes.
  Essayer de comprendre un code non documenté, dans un langage que l'on connaît peu, est une mission délicate. Notre travail a néanmoins le mérite de faciliter cette tâche pour nos successeurs.

- Nous nous sommes rendus compte d'un bug important pour la compréhension scientifique des frappes de processus, et l'équipe encadrante nous a demandé de nous atteler à la correction de ce bug, qui ne figurait pas dans le cahier des charges. Ce qui nous semblait un travail assez simple à première vue a monopolisé l'attention d'un membre du groupe pendant la quasi-totalité du projet.

Ces éléments ont eu pour conséquence le fait qu'après plus de la moitié des 10 semaines allouées au projet, nous n'avions pas commencé à coder une seule des fonctionnalités

présentes dans le cahier des charges, et avions plus de 3 semaines de retard. Notre force de travail nous a néanmoins permis de rattraper ce retard dans les dernières semaines en recherchant, concevant et développant de manière efficace.

Nous regrettons de n'avoir pas pu établir une recette en bonne et due forme de l'application développée. Cette phase de tests formalisés et de confrontation avec les spécifications détaillées faisait partie de notre planning initial. Mais le retard accumulé tout au long du projet ne nous a pas permis de passer cette étape.

V.3 Retour sur l'accompagnement

De manière générale, l'équipe encadrante a été très disponible et à l'écoute. Nous tenions des réunions tous les 10 jours en moyenne, ce qui nous permettait d'avancer avec des jalons mais aussi d'obtenir des remarques et des validations de la part des encadrants.

Dès les premières semaines du projet, les encadrants ont mis en avant l'importance du rapport d'avancement, et nous ont conseillé de détailler nos actions et difficultés au maximum. Nous avons cependant parfois regretté de devoir relancer l'équipe encadrante pour obtenir des réponses aux questions levées dans ces rapports.

Nous avons eu la chance que l'équipe ait encadré la première version du projet, ce qui leur permettait de faire preuve de réalisme et d'estimer avec justesse la capacité de travail d'un groupe. Son ambition est donc restée mesurée, mais suffisamment élevée pour rendre le projet challengeant et appeler l'équipe à fournir un travail conséquent.

V.4 Comparaison entre quantité de travail prévue & effective

Le cahier des charges et le planning étaient construits sur la base d'une équipe de cinq étudiants devant chacun allouer un minimum de 60 heures pour le projet.

L'équipe a finalement été réduite à quatre membres, dont la plupart n'avaient que peu de connaissances en développement, d'autant plus en C++ et Qt. Nous avons également été retardés par la production chronophage de livrables qui n'avait pas été intégrée au cahier des charges : l'important travail de documentation et la correction d'un bug.

Au final, le travail délivré par l'équipe dépasse les prévisions, le temps alloué au projet dépasse même les 120h pour certains d'entre nous.

VI CONCLUSION

Ce projet nous a permis de nous confronter à un projet informatique de taille modeste, répondant à un vrai besoin pour le monde de la recherche en bio-informatique. C'est d'ailleurs ce cadre concret qui nous avait donné l'envie de participer à ce projet.

Nous avons pu approfondir nos connaissances en informatique, notamment en C++. De plus, reprendre un code existant est un exercice intéressant, qui était nouveau pour la plupart d'entre nous. Au cours de cette étape, nous avons pu nous rendre compte de la nécessité de bien documenter son code.

Ce projet nous a appris à être attentifs à la qualité des sources héritées : nous veillerons notamment lors de nos futurs projets à découvrir le code, à en évaluer la lisibilité et la maintenabilité, avant de livrer un cahier des charges et un planning prévisionnel.

Au niveau de la gestion du projet, nous avons pu nous confronter à la rédaction d'un cahier des charges engageant. Un travail d'équipe a été mis en place avec une répartition des tâches et la gestion d'un planning.

Le manque d'accompagnement technique au niveau du code a néanmoins rendu notre tâche plus difficile. Mais les ressources disponibles en ligne et nos efforts d'ingénierie logicielle nous ont permis de surmonter les écueils.

Nous avons beaucoup appris lors de ce projet. Nous espérons que notre travail sera utile à l'équipe MeForBio et que la documentation que nous avons construite permettra à nos successeurs une compréhension plus rapide du fonctionnement de l'application, du code ainsi que des différentes bibliothèques utilisées.

VII BIBLIOGRAPHIE & NETOGRAPHIE

http://mupuf.org/blog/article/34
Using libgraph to represent a Graphviz *graph.* Article dont s'inspire le paquet gviz.

http://www.graphviz.org/
Présentation et documentation de Graphviz.

http://www.graphviz.org/pdf/libguide.pdf
Manuel d'utilisation de la bibliothèque Graphviz.

http://qt-project.org/doc/qt-4.8/
Documentation du framework Qt.

http://www.qtfr.org/
Forum de la communauté francophone de développeurs Qt.

http://www.qtcentre.org/
Forum de la communauté internationale de développeurs Qt.

http://stackoverflow.com/
Site de questions/réponses ouvertes à la communauté internationale de développeurs.

VIII ANNEXES

SOMMAIRE DES ANNEXES :

1. Rapport de GELOL – Diagrammes UML
2. Documentation
   2.1. Installation de Pint et gPH sur Ubuntu
   2.2. Génération de la documentation avec Doxygen
   2.3. Installation des programmes Graphviz en ligne de commande
   2.4. Format d'export XML
   2.5. Documentation du projet précédent
   2.6. Rapport final du projet précédent