Documentation

Compilation

Pour compiler gPH dans un environnement Unix, les librairies suivantes sont nécessaires :

- Qt (4.8+)
- Boost (1.48+)
- Graphviz (2.28+)

Pour créer le makefile du projet, il suffit d'exécuter (depuis le répertoire du projet) :

```
qmake pappl.pro
```

On peut alors lancer la compilation avec la commande :

```
make
```

Compilation en mode test

Les prérequis sont les mêmes que pour compiler en mode normal. La seul différence est que `qmake` doit être appelé avec un paramètre supplémentaire :

```
qmake CONFIG+=test pappl.pro
```

Exécution

L'exécution du programme (en mode normal ou en mode test) nécessite d'avoir compilé les utilitaires de Pint et de les avoir placé dans le path du système. Le site de Pint donne des informations succinctes sur les prérequis pour le compiler. Voici une marche à suivre plus détaillée :

- Installer Ocaml (disponible dans la plupart des package managers)
- Télécharger CamlIDL via `http://caml.inria.fr/pub/old_caml_site/distrib/bazar-ocaml/camlidl-1.05.tar.gz`
- Décompresser CamlIDL
- Se rendre dans le dossier `config/` de l'archive en question
- Renommer `Makefile.unix` en `Makefile`
- Éditer ce même `Makefile` pour que la variable `OCAML_LIB` pointe vers votre dossier d'installation de Ocaml. Le mien est `/usr/lib/ocaml`
- De retour dans le dossier de CamlIDL exécuter :

```
make all
sudo make install
```

- Installer le compilateur Fortran g77 (disponible dans la plupart des package managers)
- Télécharger R via `http://www.r-project.org/`
- Décompresser R
- Dans le dossier de R, exécuter :

```
./configure
```

- Depuis le dossier de R, se rendre dans `src/nmath/standalone`
- Exécuter :

```
    run
    make
```

- Copier `libRmath.so` et `libRmath.a` (qui sont désormais dans `src/nmath/standalone`) dans

`/usr/lib`

- Télécharger Pint via `http://processhitting.wordpress.com/download/`
- Décompresser Pint
- Copier `Rmath.h` (qui se trouve dans `src/include` depuis le dossier de R) dans le dossier `/pintlib` du dossier de Pint
- Depuis le dossier de Pint, exécuter :

```
make
```

Documentation des classes du projet

Afin d'avoir une idée générale du fonctionnement de l'application, il est vivement recommandé de commencer par lire la partie « Travail effectué » du rapport de projet d'application. La documentation qui suit n'est pas générée automatiquement et est faite pour être compréhensible par un être humain. Il est toutefois recommandé de consulter les headers des classes décrites ci-dessous pour compléter la lecture (ou vérifier des prototypes).

Package ph

Le package « ph » contient les classes qui servent à construire la représentation en mémoire d'un fichier PH, indépendamment de sa représentation.

Classe PH

Cette classe représente un process hitting complet tel que décrit par un fichier PH. Les actions du PH sont des objets de la classe `Sort` stockés dans une liste appelée `actions` et les `sorts` sont stockées dans une `std::map` qui relie leur nom à des objets de la classe `Sort`. La classe possède plusieurs méthodes (`addSort`, `addAction`) ainsi que des getters pour ajouter des données au PH où y accéder.

Les propriétés `stochasticity_absorption`, `infinite_default_rate` et `default_rate` correspondent aux valeurs par défaut à donner au « rate » et à la « stochaticity absorption » des actions de ce process hitting. La classe possède des getters et setters pour ces propriétés. Elles correspondent précisément aux en-têtes des fichiers PH.

La propriété `scene` est un pointeur vers une `PHScene`, c'est à dire un objet graphique qui représente le process hitting.

La méthode `toString` renvoie une représentation texte du process hitting telle qu'on la sauvegarderait dans un fichier .ph.

La méthode `toDotString` renvoie une représentation textuelle du process hitting au format dot un des formats d'entrée de graphviz. Cette méthode n'est pas utilisée pour faire les rendus du process hitting dans l'application.

La méthode `render` demande d'effectuer le rendu du process hitting dans sa scène. C'est

une opération coûteuse en temps car elle fait appelle à la méthode `toGVGraph`.

La méthode `toGVGraph` produit une représentation du process hitting sous forme de graphe (objet de la classe `GVGraph`) et utilise graphviz pour calculer une disposition agréable des éléments du graphe.

Classe Sort

Cette classe représente une sort d'un process hitting. Son constructeur est privé car on ne souhaite manipuler que des pointeurs vers des sortes et éviter d'en laisser traîner dans la nature. On peut créer des pointeurs vers des `Sort` via la méthode statique `Sort::make` qui prend un argument le nom de la `Sort` que l'on souhaite créer et son nombre de processes.

Les sortes possèdent un nom (propriété `name`) et un getter associé.

La propriété `processes` est un tableau qui liste les différents processes de la `Sort` (sous forme de pointeurs vers des objets `Process`).

La propriété `activeProcess` est un pointeur vers le `Process` actuellement actif.

La méthode `toString` renvoie une représentation texte du `Sort` telle qu'on la sauvegarderait dans un fichier .ph.

La méthode `toDotString` renvoie une représentation textuelle du `Sort` au format dot (un des formats d'entrée de graphviz). Cette méthode n'est pas utilisée pour faire les rendus du process hitting dans l'application.

Classe Process

Cette classe très simple représente un `Process`.

Elle contient une référence à la `Sort` dont elle fait partie et son numéro au sein du `Sort`.

Les méthodes `getDotName` et `toDotString` sont utilisées pour l'export des process hitting au format dot.

Classe Action

Cette classe représente une action (ou frappe) d'un process hitting.

Elle contient trois pointeurs vers des `Process` qui sont mis en jeu dans la frappe (la source (`source`), le `Process` frappé (`target`) et le résultat de la frappe (`result`)).

Les propriétés `infiniteRate`, `r` et `sa` servent à déterminer le « rate » et la « stochasticity

absorption » de la frappe. Lorsque le booléen `infiniteRate` est vrai, le « rate » de la frappe est infini et la valeur de `r` n'a pas de sens.

Cette classe contient aussi des méthodes `toString` et `toDotString` qui servent le même principe que celles de la classe `PH`.

Package io

Le package « io » qui contient les classes concernant l'entrée et la sortie de fichiers.

Classe IO

La classe `IO` gère les entrées/sorties au niveau le plus bas. Elle ne contient que trois méthodes statiques qui servent à vérifier qu'un fichier existe, lire le contenu d'un fichier et écrire le contenu d'un fichier.

Classe PHIO

Cette classe contient des méthodes relatives aux entrées/sorties de fichiers PH. Elle ne contient que des méthodes statiques.

La méthode `canParseFile` permet de tester qu'un fichier (dont le nom est donné en paramètre) existe et est reconnu par le parser de fichiers PH.

La méthode `parseFile` parse un fichier (dont le nom est donné en paramètre) et renvoie un pointeur vers un objet de la classe `PH`.

La méthode `writeToFile` réalise l'opération inverse, elle sauvegarde un objet `PH` sous forme d'un fichier PH.

La méthode `exportToPNG` permet d'enregistrer au format png la représentation d'un fichier PH telle qu'elle apparaît dans l'interface graphique.

La méthode (privée) `parse` effectue le travaille de parsing sur une donnée brute issue d'un dump de l'utilitaire phc de pint et renvoie un pointeur vers un objet `PH`. C'est la méthode qui est utilisée par son alter ego public `parseFile`.

Package ui

Le package ui permet l'affichage de la fenêtre qui permet l'appel aux différents autres packages et fonctions de pint.

Classe MainWindow

Pour les méthodes "public":

`MainWindow()` est le constructeur de `MainWindow`, il crée la fenêtre avec ses menus et initialise ses différentes caractéristiques (titre de la fenêtre, icône, activation des menus de base, définition et caractéristiques de la zone centrale.)

`getCentraleArea()` renvoit le pointeur vers la zone centrale de la fenêtre (de type `QMdiArea*`)

`std::vector<QString> getAllPaths()` permet d'obtenir tous les chemins des fichiers ph actuellement ouverts. Attention au type de sortie : on renvoie des `Qstring` (type de string utilisé par Qt)

La méthode `compute` prend trois arguments : `QString program`, `QStringList arguments`, `QString fileName` (ce dernier argument est optionnel). Il s'agit de la méthode principale du menu computation: elle permet d'appeler la fonction de pint (program), avec les arguments voulus. `fileName` permet de transmettre le nom du fichier ph concerné.

`enableMenu` permet d'activer les menus nécessitant un onglet ouvert et actif : à mettre à jour dès qu'une fonction est ajoutée à l'application !

Concernant les attributs "protected":

`centraleArea` est un pointeur vers la zone centrale.

`MainWindow` possède également des pointeurs vers ses menus dans ses attributs. La liste est longue mais nécessaire à leur activation et désactivation depuis les méthodes `enableMenu` et `disableMenu`

Il n'y a pas de `signals` (qui est une particularité de Qt)

Concernant les `slots`, il s'agit d'une autre particularité de Qt: les `slots` sont des méthodes un peu spéciales, activables par des `signals`. Ici il s'agit des actions activées par l'appui sur des boutons dans le menu principal (ou l'utilisation de raccourcis

clavier).

`openTab` permet l'ouverture d'un nouvel onglet. Plusieurs vérifications sont faites: le fichier peut-il bien être parsé et n'est-il pas déjà ouvert ?

`save` permet de sauvegarder

`closeTab` permet de fermer l'onglet actif

`exportPng` permet l'export en png (très proche de la méthode de sauvegarde mais le format de sortie est en .png)

Pour le menu computation, ces slots appellent ensuite `MainWindow::compute` avec les bons arguments et appelant la bonne fontion de pint.

`findFixpoints` permet d'appeler ph-stable.

`computeReachability` permet d'appeler ph-reach.

`runStochasticSimulation` permet d'appeler ph-exec.

`checkModelType` n'est pas encore implémentée.

`statistics` permet d'appeler ph-stat.

Pour l'ajout de nouvelles fonctions utilisant pint, il suffit de se baser sur ces fonctions - En particulier l'appel à phc pour l'export peut facilement se réaliser en créant une autre méthode compute légèrement modifiée (car les sorties sont un peu différentes) et en créant une méthode pour chaque format différent, qui sera appelée par un choix différent dans le menu d'export (dans le menu File).

`DisableMenu`, en parallèle à `enableMenu`, permet de désactiver les menus qui doivent être indisponibles lorsque tous les onglets sont fermés. Le `connect` (connexion entre signal et slot) se réalise dans le constructeur de `MainWindow` avec cette ligne :

```
Qobject::connect(this->centraleArea,SIGNAL(subWindowActivated(QMdiSubWindow*)),
this, SLOT(disableMenu(QMdiSubWindow*)));
```

La seule chose à mettre à jour dans cette fonction est donc le choix des menus à désactiver.

Attention: il semblerait que sous Ubuntu (version d'octobre 2011) cette fonction ne

marche pas! Elle fonctionne néanmoins très bien sous Arch-linux. D'après nos recherches internet, il s'agirait d'un bug propre à Ubuntu. Il faut donc toujours partir du principe que des menus puissent être activés sans nécessairement devoir l'être. Une vérification de l'état des onglets est donc nécessaire à chaque appel de fonction sensible.

Classe MyArea

La classe MyArea est la classe instanciée lors de la création d'un nouvel onglet (l'onglet contient un instance de MyArea).

Cette classe hérite de `QgraphicsView` permettant ainsi l'affichage d'une scène. Elle contient également quelques informations utiles : `myPHPtr` qui pointe vers un objet `PH` et `path` qui contient le chemin vers le fichier associé.

Package gfx

Le package « gfx » contient les classes qui permettent de fabriquer la représentation graphique d'un process hitting qui est synthétisée dans la classe `PHScene`.

Classe PHScene

Cette classe est un objet graphique (elle hérite de `QGraphicsScene`) qui représente un process hitting.

Elle dispose de 3 structures de données (`sorts`, `processes` et `actions`) qui ne contiennent non pas des pointeurs vers des objets des classes `Sort`, `Process` et `Action` mais des classes `GSort`, `GProcess` et `GAction` décrites ci-dessous.

La classe dispose d'une méthode `doRender` qui la force à recalculer la disposition du rendu.

La méthode privée `draw` se contente de vider le contenu graphique de la scène puis d'y ajouter un par un les éléments qui composent la représentation du process hitting.

Classe GSort

Cette classe agrège un objet de la classe `Sort` et un struct `GVCluster` qui contient les données calculées par graphviz pour savoir où et comment dessiner la `Sort` en

question. La classe contient également les objets graphiques du dessin de la `Sort` (un rectangle, un texte et un conteneur `QGraphicsItem*` pour le tout).

Enfin, la classe contient une propriété `color` qui est renseignée à la création de l'objet en bouclant à travers une palette prédéterminée. Cette couleur sera utilisée pour colorer les actions qui démarrent depuis cette sorte.

Classe GProcess

Cette classe agrège un objet de la classe `Process` et un struct `GVNode` qui contient les données calculées par graphviz pour savoir où et comment dessiner le process en question. La classe contient également les objets graphiques du dessin du process (un texte, une ellipse et un conteneur `QGraphicsItem*` pour le tout).

Clase GAction

Cette classe agrège un objet de la classe `Action` et deux struct `GVEdge` (un par flèche à dessiner) qui contiennent les données calculées par graphviz pour savoir où et comment dessiner `l'Action` en question. La classe contient également les objets graphiques du dessin de l'Action : deux courbes et deux polygones (les pointes des flèches).

La méthode privée `makeArrowHead` sert à construire les pointes de flèche en tant qu'objets graphiques à partir des informations stockées dans les `GVEdge`. L'opération consiste à dessiner un triangle que l'on fait ensuite tourner (via une matrice de rotation) pour l'aligner avec la tangente à la flèche en son extrémité.

Package gviz

Le package « gviz » 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). Ce package ne part pas du principe que l'on va représenter des process hitting et a une valeur très générale.

Ce package 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.

Notre principal apport par rapport à la classe proposée l'article est la prise en charge de sous-graphes (appelés clusters dans graphviz) qui seront nécessaires pour forcer les `Process` d'un

même `Sort` à être positionnés dans un même rectangle.

Package test

Le package « test » contient les classes relatives aux tests unitaires du programme.

Classe TestRunner

Cette classe qui n'en est pas une fait office de `main` lorsque le projet est lancé en mode test. Elle se contente de lancer l'ensemble des tests disponibles - c'est à dire les tests unitaires du parser de fichiers PH.

Classe PHIOTest

Cette classe de test utilise la méthode `PHIO::canParseFile` pour vérifier que des fichiers PH élémentaires ainsi que des exemples issus du site de pint sont parsés avec succès. Les exemples élémentaires ont pour but de tester le parser sur les opérations suivantes :

- déclaration de process
- utilisation de cooperativity
- déclaration d'actions
- utilisation de la macro knockdown
- utilisation de la macro rm
- utilisation de headers
- utilisation du footer
- utilisation de la macro grn
- présence de commentaires