Documentation : *Projet Shogi*

Ou autrement dit : Rapport de Projet

Rédaction : Rakib SHEIKH (NoobZik)

Le rapport a été rédigé en markdown, flemme d'ouvrir Office qui met 5 ans à démarrer pour rédiger un rapport projet. Accrochez-vous bien, car le rapport que vous allez lire n'est pas un rapport comme les autres.

Membre du groupe

Prénom Nom	Numéro Etudiant	Groupe
Rakib Sheikh	11502605	Groupe 9
Ibrahim Kouyate	11507047	Groupe 6
Emeric Bayard	11600611	Groupe 8

Introduction

Le sujet projet à été présenté durant le 20 mars 2017 en cours d'amphi de programmation impérative. Le sujet est donc le **Shogi**.

Tout le projet a été développé à l'aide d'un GIT dans lequel nous avons développé de manière à pouvoir continuer à développer l'application à distance et permettre de retracer entièrement l'historique du projet (rapport de bug, état de développement des branches du programme et des différentes manipulations dans le code). L'avantage principale est de supprimer totalement la fragmentation des versions du projet. Vous voyez, je n'aime pas avoir plusieurs versions.

Cependant, la méthode du projet est de faire une partie du projet chacun de notre côté, pour qu'on puisse avoir une idée du contenue de projet. Une fois qu'on a une idée du contenu du projet, on se rassemble à la fin et on commence à programmer.

La supervision du répertoire git est exclusivement faite par Rakib, en même temps c'est le seul qui a des connaissances dessus et qui n'aime pas attendre les modifications par clé USB ou transfert par G-Drive ou mail. Mais exeptionnelement, je me charge tout seul mettre à jour le code sur le répo.

La méthode de développement est la « méthode agile ». C'est-à-dire de par adaptation systématique de l'application aux changements du besoin détecté lors de la conception-réalisation du projet et par remaniement régulier du code déjà produit (re-factoring). Avec un planning de développement à base de secteurs dans le programme sans réelle précision.

En gros, on prend une fonction au hasard et on fait avec jusqu'à qu'on rencontre un problème.

Le projet est fait en sorte qu'il écrit sous forme de MVC (ou presque ...). Un petit rappel qui ne fait pas de mal : Modèle, Vue, Contrôleur. La priorité absolue du projet est de modéliser la vue. Une fois que c'est fait, on peut commencer à coder en free-style.

Répartition des tâches

Les membres du groupe sont :

- **Rakib Sheikh (NoobZik) : Tous les déplacements + restrictions + tout ce qui en rapport avec promotion / de-promotion + debug + makefile**
- **Ibrahim Kouyaté : Le contenue du game.c - piles.c - files.c**
- **Emeric Bayard : Sauvegardes et chargements de fichiers + Contenue du piece.c**

D'un point de vue sur le MVC :

- **Contrôleur :** Rakib Sheikh (Mouvements + Restrictions) - Ibrahim Kouyate (Fonctions d'appel entre Vue et Contrôleur) - Emeric Bayard (Sauvegardes et gestion de mémoire)
- **Modèle :** Emeric Bayard (Les pièces + L'échiquier)

- **Vue :** Ibrahim Kouyate (Bah tout ce qui concerne l'affichage de l'échiquier) + Rakib Sheikh (Affichage des debugs)

Descriptions des fichiers

La liste des fichiers

Il y a au total 8 fichiers (à l'exclusion du main).

1. piece.c
2. pile.c
3. file.c
4. mouvement.c
5. restriction.c
6. game.c
7. sauvegardes.c
8. debug.c

Description détaillé des fichiers

Cette partie détaille les fonctions qui ont été modifié du sujet ou rajouté.

Pour le chargé de TD, on essaye de vous disposer plusieurs plateaux de sauvegardes pour tester rapidement un peu près tous les fonctionnalités du programme.

Actuellement, le programme peut (liste non exhaustive):

- Jouer une partie jusqu'à la fin (Caractérisé par une capture du roi adverse).
- Quitter une partie à tout moment.
- Sauvegarder le contenu de l'échiquier et l'historique (Capture + Mouvements).
- Charger le plateau.
- Créer des dossiers de sauvegardes s'il n'existent pas.
- Capturer des pièces.
- Gestion des parachutages avec en complément la restriction sur les pions.
- Afficher une **aide-visuel** lors des déplacements.
- Annuler des déplacements.
- Gestion des promotions et de dé-promotions.
- Gestion active de la réserve des deux joueurs.
- Gestion active de la mémoire allouée. (Grâce au pile et files)
- Commandes de développeurs (nommé debug).
- Compiler sans erreurs, warnings et pas de problème de mémoire.

Un MAKEFILE pas comme les autres

La description de l'utilisation du makefile est indiqué dans le README

Cependant, le projet avait l'objectif très strict d'avoir 0 erreurs et 0 warnings à la compilation.

De ce fait, le makefile contient tous les affichages de warnings possible. C'est pour ça que l'affichage du terminal est très surchargé à la compilation du programme. Non, non ce n'est pas des erreurs de compilation ce que vous voyez à la compilation rassurez-vous.

Mais c'est pas du tout marrant de compiler avec tous les warnings dès le début du projet. C'est comme tricher. Je trouve que c'est vraiment marrant de le faire à la fin. C'est-à-dire lorsque le -Wall n'affiche plus rien à la compilation. Entre nous, compiler avec seulement le warning -Wall, ce n'est pas assez jouissif pour régler tous les warnings. -Wall c'est pour les débutants. Nous ce qu'on veut c'est de la zéro tolérance, c'est amusant d'avoir environ 600 Warnings d'un coup.

Ça permet aussi d'avoir un code qui est propre, et optimisé. Pas comme dans les code des cours d'amphis.

---

piece.c

Dans ce fichier, contient tous les fonctions concernant à la gestion des pièces.

Note : Afin d'éviter toute ambiguïté, Joueur 0 et Joueur 1 ont été remplacé par NOIR et BLANC.

Une structure pièce est caractérisé par trois champs :

- Un entier énuméré représentant la couleur, il y en a trois {Blanc, Noir et VIDE}.
- Un entier énuméré type représentant l'ensemble des pièces en plus de VIDE, SELECT* (Plus de détails plus bas).
- Un statut énumérés pour la représentation d'une pièce promu ou non promu.

Il est jugé plus simple de travailler sur des types énumérés que sur des type classique tel que les type unsigned int.

Les fonctions suivantes ont été rajouté ou modifié. Elles sont nécessaires pour la suite du projet.

```
piece.c
piece_couleur(); Initialement piece_joueur(); Prend en argument une pièce piece_t et renvoie sa couleur associé.

piece_t promote_grant       (); // Permet de promouvoir la piece.
piece_t demote_grant_reserve(); // Permet de dépromouvoir la piece en changeant de couleur.
piece_t demote_grant        (); // Exactement la même chose qu'au-dessus mais sans changer de couleur.
piece_t switch_color        (); // Elle permet de changer de couleur avant de placer la pièce dans la réserve.
piece_t piece_cmp_reserve   (); // Elle permet de comparer deux pièces
```

La fonction promote_grant permet de promouvoir les pièces selon les règles en vigueur. Le contenu de cette fonction est découpé en trois partie :

- La partie générale qui donne le choix de promouvoir ou non les pièces sur les deux avant dernières lignes de l'adversaire.
- La partie spécifique qui impose des restrictions de promotion comme pour le cavalier et les parachutages.
- La partie pour impose une promotion directe lorsqu'on arrive à la dernière ligne de l'adversaire.

Lorsqu'on a essayé de modéliser les parachutes, on s'est rendu compte qu'il faut dé-promouvoir les pièces avant de les placer dans la réserve. On a une condition qui permet de vérifier si la pièce est à mettre dans la réserve en vérifiant son statut.

En fonction du résultat de la condition, il y a deux cas possible (soit deux fonctions) :

- demote_grant_reserve();
- switch_color ();

1. **demote_grant_reserve()**

Cette fonction permet de dé-promouvoir une pièce en changeant de couleur. Elle est appelée si cette pièce en question est promue.

2. **switch_color()**

Cette fonction permet juste de changer de couleur si cette pièce en question n'est pas promue.

3. **demote_grant()**

Cette fonction permet de dé-promouvoir une pièce sans changer de couleur. Elle est exclusivement utilisée dans la partie annuler_deplacement

4. **piece_cmp_reserve()**

Cette fonction permet de comparer deux pièces. Elle est spécialement utilisée pour la gestion de la réserve visuelle. Elle permet de comparer le type et le statut des deux pièces. Il faut aussi que les couleurs soient différentes.

---

pile.c / file.c

La *file* représente l'historique des coups joués et ses événement associé (Capture de pièce et promotion).

On a une structure pour pouvoir gérer efficacement les coordonnées :

```
typedef struct movement_s {
    int movement; /* Représente le numéro du mouvement */
    coordinate_t input; /* Représente les coordonnées d'entrée */
    coordinate_t output; /* Représente les coordonées de sortie */
}movement_t;
```

Une fois qu'on à ça en tête de manière temporaire, on peut s'attaquer à la file.

En gros ce qu'un élément de la file contient :

- Un mouvement (On a vu ça au-dessus au passage)
- Un booléen de promotion
- Un booléen de capture
- Et deux champs pour le mouvement suivant et précédent. C'est le principe d'une liste doublement chainée.

Cette élément de la file sera très utile lorsque l'on va traiter les déplacements expliqué plus tard dans cette documentation.

La *pile* représente l'historique brute des pièces capturés. En gros, la pile contient seulement les pièces capturées.

Pour la pile et la file, elle marche comme des listes qui sont doublement chaînées vu en TD. Cependant, deux fonctions ont été introduite pour plus de clarté.

Lors de l'exécution du programme, la pile et la file marchent comme des piles. C'est à dire qu'on fait que d'empiler les deux listes.

Cependant à l'enregistrement, la pile et la file marchent comme des files. C'est à dire qu'on extrait le premier élément, jusqu'au dernier élément.

Les fontions suivantes ont été rajouté pour plus de clarté dans le code:

```
file.c
void file_thread    ();
// Permet d'ajouter un mouvement, un état de promotion et un état de capture dans l'historique.
void file_unthread  ();
// Fait le contraire de file_unthread, sans libérer la mémoire, car on a besoin les données de cette élément extrait.

pile.c
void pile_stacking  (); // Ajoute une pièce dans la pile.
void pile_unstacking(); // Retire la dernière pièce ajouté dans la pile.
```

---

mouvement.c

Les mouvement (autrement dit les déplacements dans le jargon du sujet) sont répertoriés dans ce fichier.

Il s'agit de vérifier si les déplacements sont possibles ou pas, de manière très naïve.

Ce fichier est décomposé en 5 blocs :

1. **Validation de coordonnées d'entré et de sortie :**

- Pour les coordonnées d'entrées, on vérifie qu'ils sont bien dans l'échiquier 11x11.
- En revanche pour les coordonnées d'arrivées, on doit vérifier qu'ils sont bien dans l'échiquier 9x9 (c'est-à-dire sans la réserve visuel). Une valeur de coordonnée est créée pour désélectionner la case, qui est (42,42), si aucun mouvement n'est possible (Notamment lors des parachutes de pions).

2. **déplacement_valide et valide_win**

- déplacement valide est composé de *switch case* pour appeler la fonction de déplacement valide adéquate. Ici il y a aussi un truc pour désélectionner la pièce, il suffit de mettre les mêmes coordonnées de départ et d'arrivée.

- valide_win Permet juste de savoir que si les coordonnées d'arrivé pointe un roi, bah la partie se termine. (**Remarque**: apparemment d'après plusieurs application shogi, il n'y a aucune annonce d'échec ou échec et mat dans le monde de compétition, en conséquence, ces deux fonctions ne sont pas implémentées).

3. **deplacement_valide**(Inserer type de piece)

- Ce bloc contient tous les déplacements des pièces non promu et des pièces promu

Je vous invite donc à regarder dans les commentaires pour leur fonctionnenement respectif.

4. **movement_valid_helper**

- Ce bloc sert essentiellement pour la restriction qui sera abordée plus en bas.

5. **deplacement_apply**

La fonction deplacement_apply est un peu complexe à comprendre.

A la base, deplacement_apply permet justement d'appliquer de manière naïve les déplacements dès qu'elles sont vérifiés. D'inscrire ce déplacement dans la file d'historique et s'il y a capture, retirer la pièce capturée de l'échiquier et le rajouter dans la pile de capture de pièce.

Mais dès l'introduction de la réserve de parachute, il a fallu le refacturer. Puisqu'il faut afficher cette pièce capturé dans la réserve. Et en plus de cela, il faut récupérer les données de promotion, s'il y a eu ou pas.

Avant de commencer à commenter la réserve, **il faut avoir en tête que**:

- La réserve du haut et de la gauche sans la case 0,9 constitue la réserve Noire.
- La réserve du bas et de la droite sans la case 10,0 constitue la réserve Blanche. Il y a une image d'illustration plus bas de cette documentation.

On commence donc à initialiser la valeur de promotion par

```
int promoted_v = is_promoted();
```

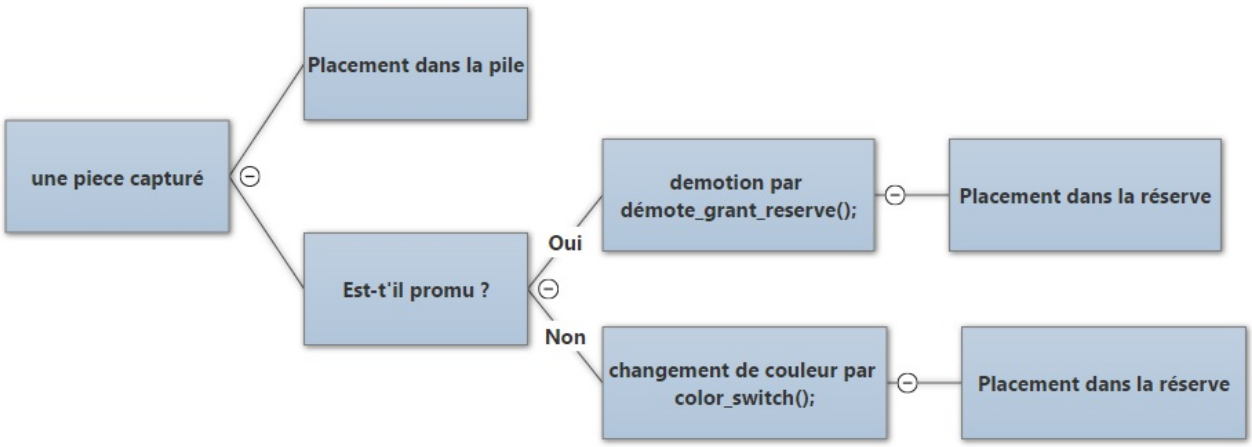
is_promoted est une fonction qui permet de vérifier si la pièce en question peut être promu ou pas. Le résultat est renvoyé sous forme de booléen.

Ensuite, on doit vérifier si les coordonnées d'arrivées contiennent une pièce. Dans le cas où les coordonnées d'arrivées contiennent bien une pièce, on place la pièce dans la capture des pièces qui est la pile, sans la retirer de l'échiquier.

Vient ensuite le placement dans la réserve.

Avant de continuer, normalement, vous devez remarquer que nous avons stocké la pièce directement dans la pile mais pas dans la réserve. La méthode qu'on a employée est le dédoublement de la pièce capturé. On place d'abord la pièce dans la pile, et ensuite on la modifie pour la placer dans la réserve visuelle.

Diagramme décision : Cas d'une pièce capturé



Avant de commencer les boucles while tant attendu, il se peut que la pièce capturé est une pièce promue, dans ce cas, les conditions de promotions sont vérifiées. Si c'est vrai, la pièce est dé-promue en changeant de couleur. Dans le cas contraire, on change juste la couleur.

Les modifications liées à cette pièce sont stocké dans une variable temporaire *p_tmp*

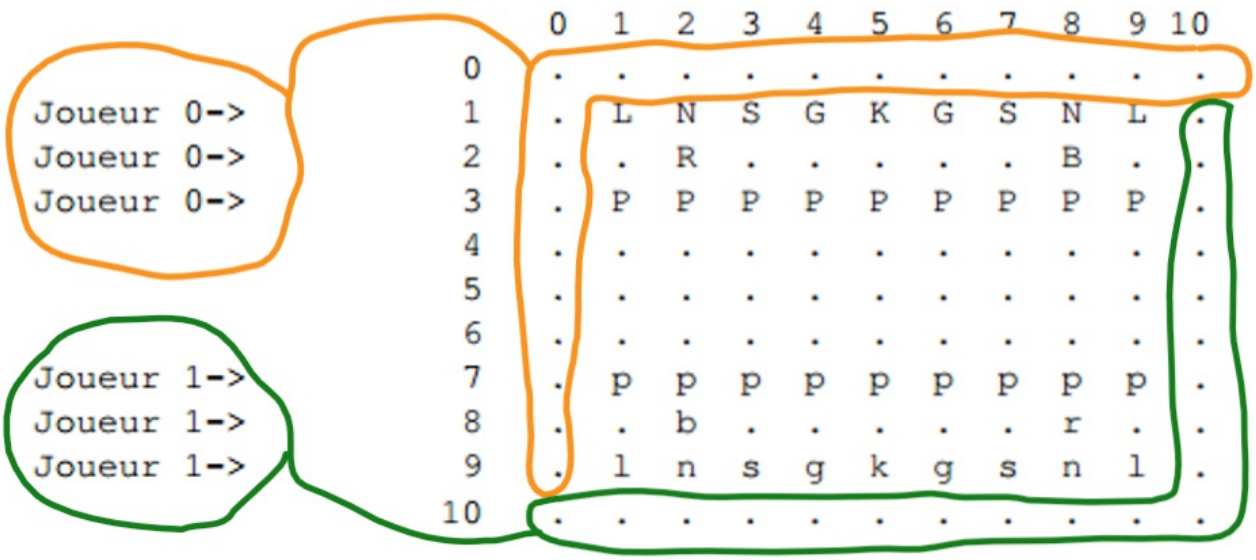
Cette condition est vérifé par la fonction :

```
piece_t demote_grant_reserve();
// Sinon
piece_t color_switch();
```

On en a parlée dans la partie piece.c.

Comme il y a deux bloc de réserve pour chaque joueur, il y aura donc au totale 4 boucles while.

Cas universel: (Noir et vous devinerez pour les blancs qui est presque la même chose).



La première boucle va parcourir la réserve du haut, de la droite vers la gauche. Pendant ce temps, on teste si la case est vide :

- Si elle est vide : On place la pièce capturée modifié qui est stocké dans p_tmp à cette case, et on fait un break pour sortir de la boucle.
- Sinon on passe à la case suivante.

Dans le cas ou toute les cases sont occupées, on change la variable de la deuxième boucle pour qu'on puisse entrer dans la deuxième boucle.

Donc on parcourt maintenant la réserve de gauche, du haut vers le bas. C'est le même algorithme que ci-dessus. Mais on parcourt jusqu'à l'avant dernière case (Car la dernière case c'est pour l'autre joueur).

Une fois qu'on a fait tout ça, il ne reste plus qu'à ajouter les données dans la file et de changer de joueur.

Dans la structure des mouvements, on avait vu qu'il y a un entier qui représente le numéro du mouvement. On inscrit donc le numéro du mouvement dans la file en récupérant la taille de la file + 1.

Les données de la file sont :

- Le mouvement.
- Le booléen de test de promotion.
- Le booléen de capture de pièce.

S'il n'y pas de pièce à l'arrivée, on applique tout simplement les déplacements.

6. **annuler_deplacement()**

Cette fonction permet de faire le chemin inverse de deplacement_apply.

On fait une extraction du dernier élément inséré de la file. Puis en fonctions des données de cette élement on peut soit :

- Dé-promouvoir une pièce s'il y a eu une promotion de la pièce. Grâce aux information de la file, il y a un booléen qui permet de savoir si le mouvement précédent a donné lieu à une promotion. Si c'est vrai, on utilise demote_grant()
- Restaurer une pièce capturé en prenant soin de traiter la réserve grâce aux boucles while inversé de deplacement_apply. (Il est possible qu'il est toujours buggé, Utilisez la fonction Signalement de bug sur le repo du bitbucket). Egalement avec les informations de la file, il y a un booléen qui permet de savoir si le mouvement précédent donne lieu à une capture de pièce. Dans ce cas, il y aura une extraction de la pièce capturée situé dans la pile, avec traitement de la réserve.
- Et dans tous les cas, restaurer la position initiale de la pièce en question avant le déplacement. Il suffit de remettre la pièce dans sa position de départ et de mettre une case vide dans la position d'arrivée.

Remarque : Dans le cas où la pièce capturé est une pièce qui était promu, il n'y a pas besoin de promouvoir cette pièce. En effet, cette pièce existe déjà, elle était tout simplement caché dans la pile. Plus de détail dans la partie. Les détails sont dans deplacement_apply();

On prend aussi le soin de changer de joueur. Sinon ce n'est pas marrant de jouer deux fois d'affiler...

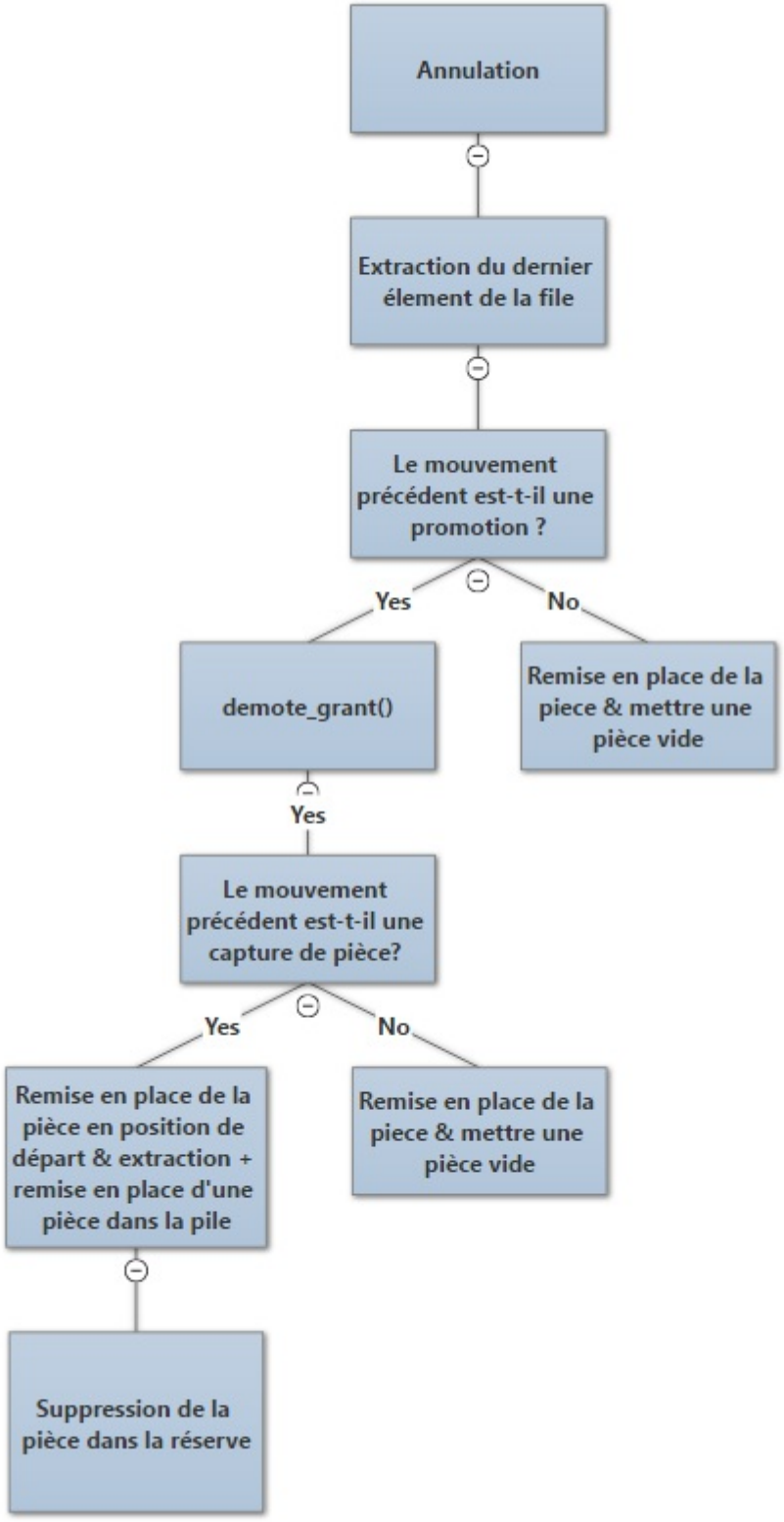
Note : En date du Vendredi 21 Avril, un comportement anormal assez majeur est rapporté.

Description de ce comportement anormale: Après un parachutage d'une pièce, la case est vide. En conséquence, lors des captures de pièces, cette case vide sera à nouveau utilisable pour stocker les pièces capturées. Un conflit majeur intervient lors de l'annulation. En effet, les boucles whiles retire la dernière pièce dans la réserve. Mais puisque la dernière pièce capturée se trouve entre deux pièces capturé, les boucles whiles retirent la mauvaise pièce.

Résolution : La résolution de ce comportement anormale est fait par Rakib Sheikh (NoobZik). Une implémentation d'une nouvelle fonction nommé piece_cmp_reserve() permet de comparer deux pièces, retourne un entier s'il sont égales (avec deux couleur différente) ou pas. On a crée une nouvelle variable p_r_tmp pour stocker la pièce à retirer de la réserve. Cette variable sert de comparaison de pièce.

Pour résumer sous forme de schéma SysML

Diagramme décision : Annulation



restriction.c

Cette partie du code est un peu délicate. Lorsqu'on a développé les déplacements, on s'est rendu compte que les pièces peuvent se balader sur tout échiquier même s'il y a des obstacles sur leur parcours.

Pour contrer ce comportement anormale, il a fallu introduire des restrictions de mouvement. Il y a cependant un nouveau problème. Les pièces pouvaient quand se déplacer partout.

Donc pour contrer encore une fois ce comportement anormale, la fonction

```
mouvement.c

deplacement_valide();
```

est modifié pour que les pièces peuvent se déplacer exclusivement sur des cases de type '*' (SELECT).

C'est ici que se fait le prototype de l'aide visuel, avec les cases '*' (SELECT)

L'aide visuel *permet d'afficher les déplacements possibles sur l'échiquier. Elle est très efficace lors des développements des déplacements de pièces*

Détails du fonctionnement des restrictions

Nous avons atteint les limites du code, en effet, il n'est pas possible d'avoir une fonction qui possède un argument optionnel.

Nous avons donc une coordonnée par défaut qui est donc comme d'habitude :

```
coordinate_t COORDINATE_NULL = {42,42};
```

Pendant l'affichage de l'échiquier, il y a une condition qui permet d'entrer dans la condition qui vérifie si les coordonnées sont différentes de 42,42 justement.

Lorsque l'utilisateur entre une coordonnée d'entrée, la condition est vérifiée et c'est ici qu'entre en jeux les restrictions.

La fonction :

```
restriction.c

movement_restriction();
```

est appelé durant deplacement_valide. Cette fonction appel la restriction adaptée en fonction de la pièce sélectionnée.

Si la restriction n'exige pas de conditions particulière, on fait appel a

```
Situé à mouvement.c

movement_valid_helper();
```

par une boucle for, pour tester les cases.

Entrons dans les détails de ***movement_valid_helper.***

Cette fonction permet de savoir si la case testée par la boucle for, est valide pour effectuer déplacement ou pas. Ils font appel naturellement aux déplacements valide des pièces respectifs, **A l'exception de tour_promu et fou_promu qui font appel à roi.**

Si vous voulez comprendre pourquoi cette exception. Tout simplement car si on ne met pas ces exceptions, alors l'aide visuel va tout simplement permettre de sauter les autres pièces pendant leurs déplacements. Et justement c'est le but des restrictions d'empêcher les déplacements qui saute les autres pièces on le rappel.

Note : Lorsque on a implémenté les restrictions de tour_promu et fou_promu, on les a considérés qu'ils fonctionnent de manière théorique. Mais ce n'était pas le cas, ce comportement anormale à été découvert très tardivement, par faute de temps, elle à été remplacé par les restrictions du roi, en plus des restrictions de tour et fou respectivement.

Pour en revenir à ***movement_restriction.***

Pour les quatre cas *tour, fou, lance et parachute et leurs promotions respectifs* ont leurs propres restrictions spécifiques, qui sont définit dans le même fichier. Je vous invite donc à regarder les commentaires pour leur fonctionnement.

- **Note** : Les parachutes, sauf le cas des pions, tous les cases sont changées en ' * '.

Après avoir effectué les restrictions, le joueur saisie les coordonnées d'arrivé. La condition dans

```
mouvement.c

void deplacement_valide (); et
int restriction_detector();
```

doivent être tous les deux vérifier pour pouvoir appliquer les déplacements.

```
restriction.c

restriction_detector();
// permet de tester si la case d'arrivée est vide ou occupée par une pièce de couleur différente.
restriction_detector_parachute(); // Exactement la même chose mais pour les parachute. Case exclusivement vide.
```

Un commentaire sur les restriction parachutes

Tous les pièces qui sont situé dans la réserve (sauf pion qui est un cas particulier) peuvent se placer sur n'importe quel case vide de l'échiquier. On pourrait se dire que restriction_detector suffit. Mais ce n'est pas le cas. Il y a un cas de figure où on peut parachuter sur une case occupé par l'adversaire qui faut traiter. D'où la fonction restriction_detector_parachute qui restreint aux case qui sont exclusivement vide.

Un commentaire pour pion parachute

Donc les pions ont une restriction particulière, ils ne peuvent pas se placer sur la colonne où il y a déjà son compatriote du même type.

Pour ça, il y a 3 Boucles for. Il faut aussi noter que pour acceder à la troisième boucle il faut que la condition de test soit vérifiée (Elle est changée par la deuxième boucle si besoin).

- La première permet de parcourir les colonnes
- La deuxième permet de parcourir chaque ligne de la colonne actuelle, s'il y a son compatriote, on change la variable conditionnelle pour ne pas rentrer dans la troisième boucle
- La troisième boucle permet de changer tous les case de la colonne en case SELECT pour l'aide visuel.

game.c

Pas grand-chose à commenter ici, vu que game.c se focalise exclusivement de l'interaction utilisateur et machine par une interface graphique, appel les fonctions en fonction de l'interaction de l'utilisateur.

Cependant, on a rajouté une liste chainée qui concerne la liste des pièces capturée. Du coups la structure game (ou partie) ressemble maintenant à

```
game.h
typedef struct        game_s {
  piece_t             board[11][11];
  file_list_t   *     file;    // Liste des coups jouées.
  pile_list_t   *     capture; // Liste des pièces capturée.
  unsigned int        player;
}                     game_t;
```

```
int game_selector();
```

Permet de comparer deux chaines de caractères.

La fonction

```
void partie_jouer();
```

Contient une boucle évènementielle, c'est ici que se font l'interaction entre le contrôleur et le modèle, et le contrôleur et la vue.

Après pour le reste, il suffit de lire les commentaires dans le fichier game.c associé. Je ne vois pourquoi je détaillerai cette partie dans ce document...

---

sauvegardes.c

Donc déjà on commence très fort par :

```
struct stat st = {0};
```

Va nous servir à vérifier plus tard, l'existence du dossier de sauvegarde par la fonction :

```
int stat();
```

La fonction :

```
void partie_sauvegarder();
```

est divisé en deux sous fonctions

```
void game_save_board(); Qui permet de sauvegarder l'échiquier
void game_save_meta (); Qui permet de sauvegarder le contenue de la pile et file.
```

Aussi, cwd permet de récupérer le chemin du projet actuel pour pouvoir créer un dossier qui n'existe pas et de sauvegarder dans ce dossier.

Le fonctionnement de game_save_meta et de game_save_board sont assez similaire au vu de la structure du code.

Ne soyez pas choqué si vous voyez des return qui sont tout seul. C'est normal, on veut quitter la fonction dès qu'il y a une erreur. La première partie du bloc des sauvegardes est tout simplement une partie de concaténation. On va concaténer dans l'ordre suivant:

- cwd
- /partie
- /
- "nom du fichier"
- Extension du fichier

1. game_save_board

Il suffit d'écrire PL au début du fichier. Et ensuite il faut juste afficher l'échiquier dans le fichier avec fputc.

2. game_save_meta

Il suffit d'écrire PR au début du fichier. Et ensuite on commence à écrire tous les éléments de la file dans le fichier (respectivement pile).

Un commentaire pour partie charger :

J'avais eu un problème de chargement de partie. Apparemment, il y a eu des problèmes touchant à la bibliothèque string qui a entraîné pour la première fois les erreurs de segmentations et les erreurs qui sont directement pointé les fichiers includes du système (En même temps, si on code comme un dieu, on casse les includes du code C). Un mail à été envoyé au chargé de TD du groupe, mais resté sans réponse. J'ai dû réfléchir à un autre moyen de coder pour éviter de recasser la bibliothèque source du compilateur.

L'erreur associé se trouve sur ces deux liens :

- Les logs du terminal
- Le code en question

Je m'en rappel plus comment je l'ai réglé ce bug d'ailleurs, vu que je l'ai recodée à l'aide des cours (Non pas de Galilée mais de openclassroom).

Aussi, les pièces ont été affecté à la mauvaise position lors du chargement du fichier. Il s'avère qu'en changeant la boucle de 0 à 12 (Initialement de 0 à 11) cela règle le problème mais je n'ai pas compris pourquoi... Ne me demandez pas pourquoi, je répondrai 'proute' avec les deux paumes de main en l'air.

---

debug.c

C'est peut-être la partie la plus fun de tout le projet. Il permet de moduler le programme en fonction de ce qu'on fait.

C'est-à-dire :

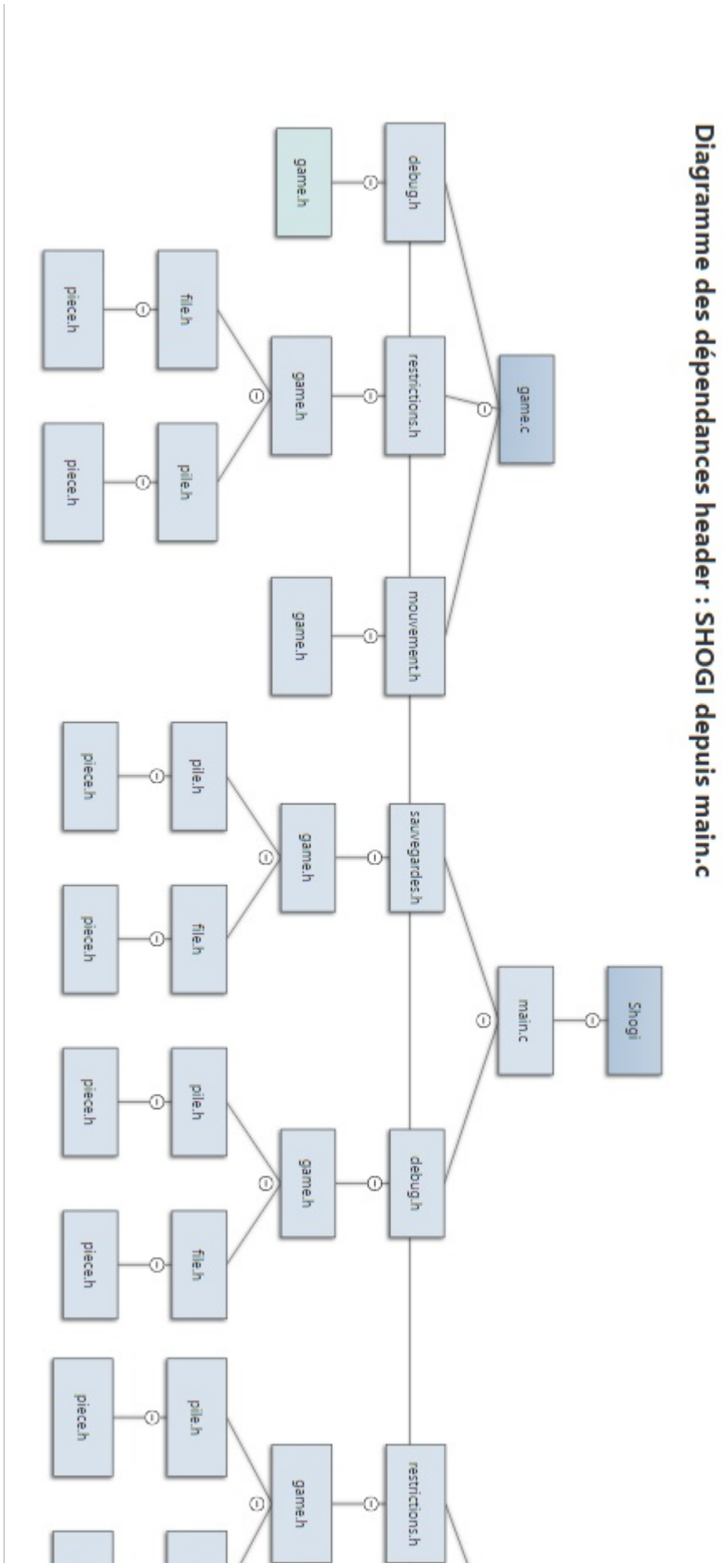
- Inspecter les cases de l'échiquier. Il se peut que des cases ne sont pas complètement vide. Inspecter une cellule permet de vérifier les champs de cette case.
- Prendre connaissance du contenu de la pile et file. Lorsque on a développé les piles.
- Créer un plateau SANS toucher au game.c

Le debug permet aussi de créer une partie sans toucher au game.c De cette manière, il est alors très simple de créer des situations de partie, pour pouvoir finir de programmer les fonctions. Par exemple, pour modéliser les déplacements du fou, on mettait un fou vide en plein milieu de l'échiquier. Lorsque l'on sélectionne la pièce, l'aide visuel s'affiche. Et en fonction de l'aide visuel, on définit les conditions de déplacement.

C'est vraiment cool pour programmer le contrôleur. Plus besoins de refaire une partie à zéro pour tester un truc en rapport avec les déplacements. Tout ça, sans toucher au code principale.

Annexes

Diagramme SysML de dépendances.



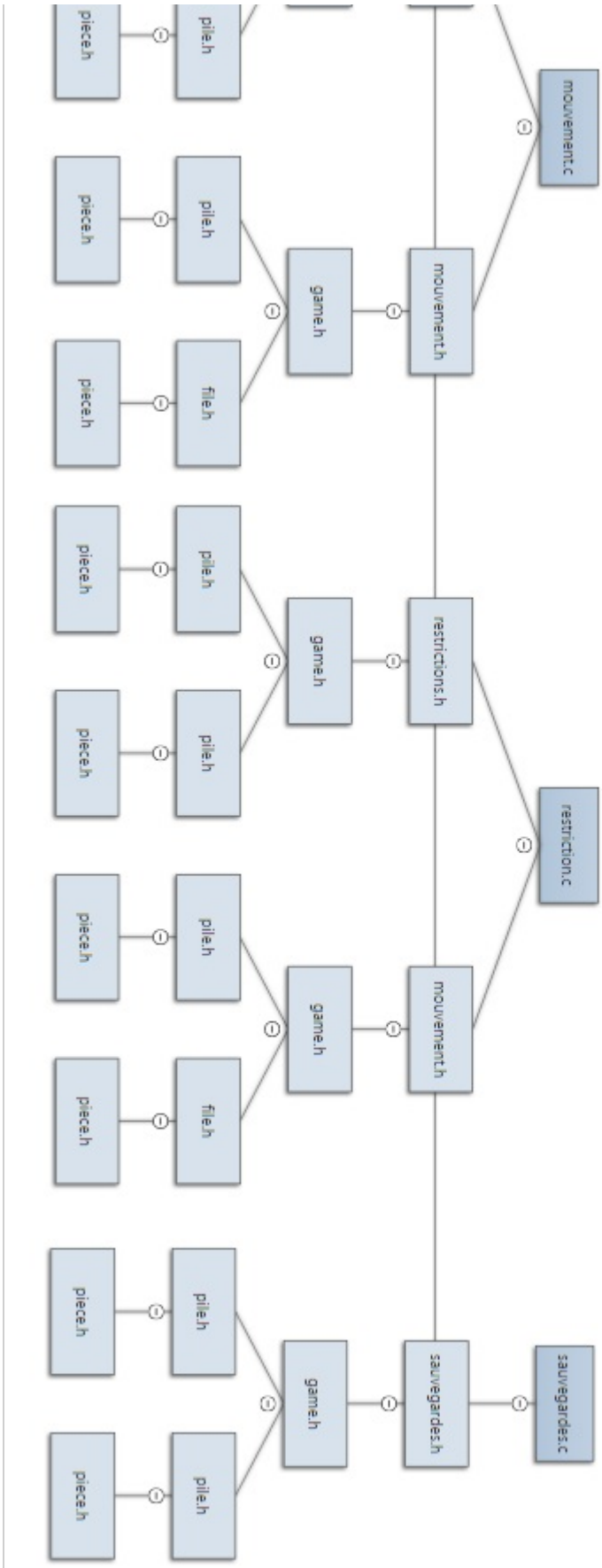
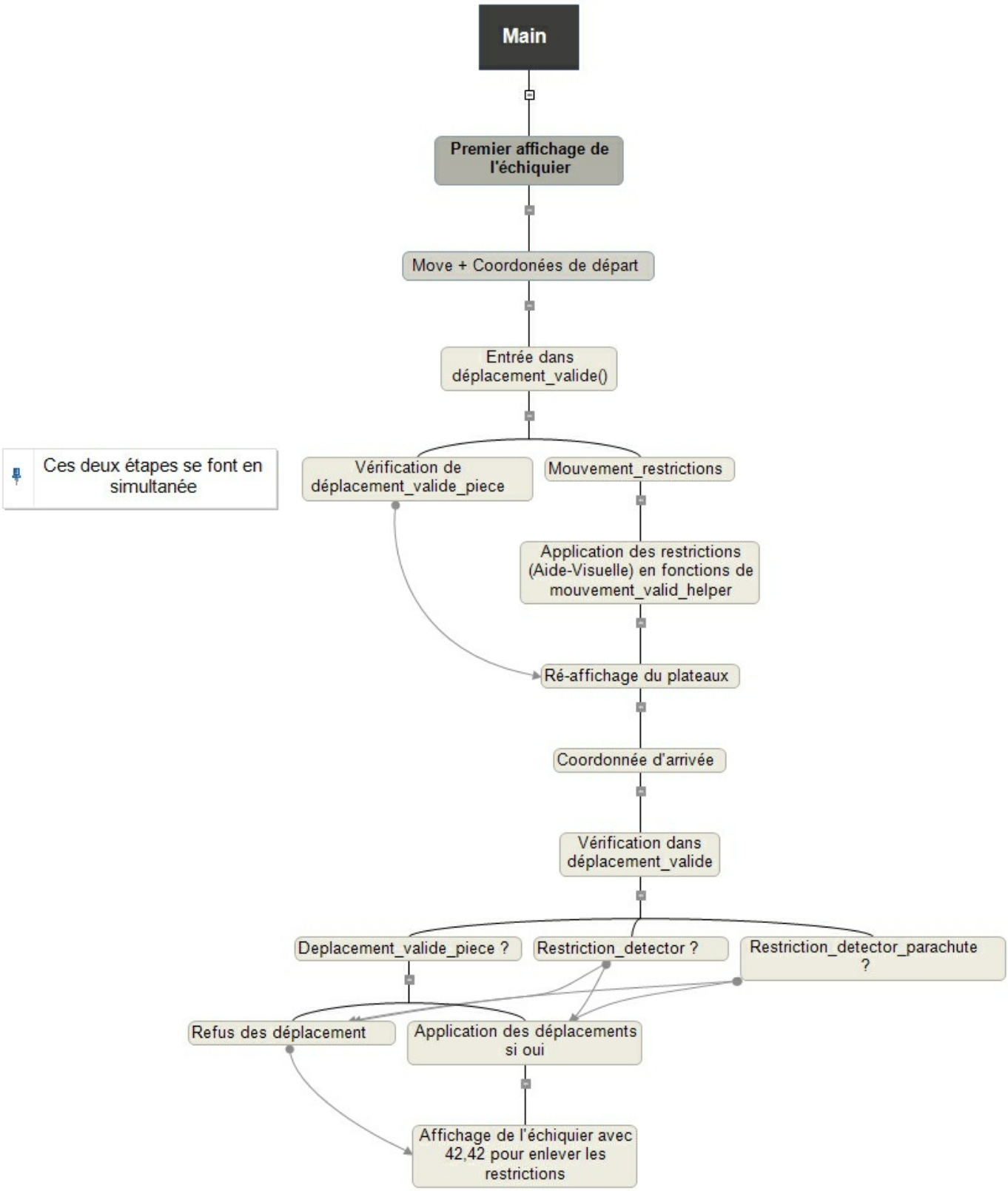


Diagramme SysML de fonctionnement du programme pour un mouvement



C'est la fin de cette documentation ou rapport de projet.