Université de Nantes
UFR des Sciences et Techniques
Master ALMA 1

TP Système Temps Réel Embarqué

Robin BONCORPS
Guillaume CHARON
Jérôme PAGES

Nantes, le 13 novembre 2012

Sommaire

Introduction

Le but de ce projet, qui nous a été confié dans le cadre des Travaux Pratiques de Systèmes Temps Réel Embarqués (Master ALMA 1), est de développer un simulateur d'ordonnanceur temps réel implémentant les algorithmes suivants :

- Rate Monotonic (RM) ;
- Earliest Deadline First (EDF) ;
- Background Server (BG) ;
- Total Bandwith Server (TBS).

Deux parties sont à identifier. Étant donné que l'ordonnanceur a besoin d'un jeu de tâches périodiques et/ou apériodiques comme base de travail, la première partie de ce projet sera de créer un générateur de tâches. L'autre partie concernera le simulateur en lui-même. Ces deux parties sont indépendantes, c'est-à-dire que le livrable final est composé de deux exécutables différents.

Le résultat final sera observable sous deux formes : un affichage à l'écran, et un fichier .ktr en sortie écrit selon la syntaxe acceptée par l'outil Kiwi, qui nous permettra de visualiser graphiquement l'ordonnancement des tâches.

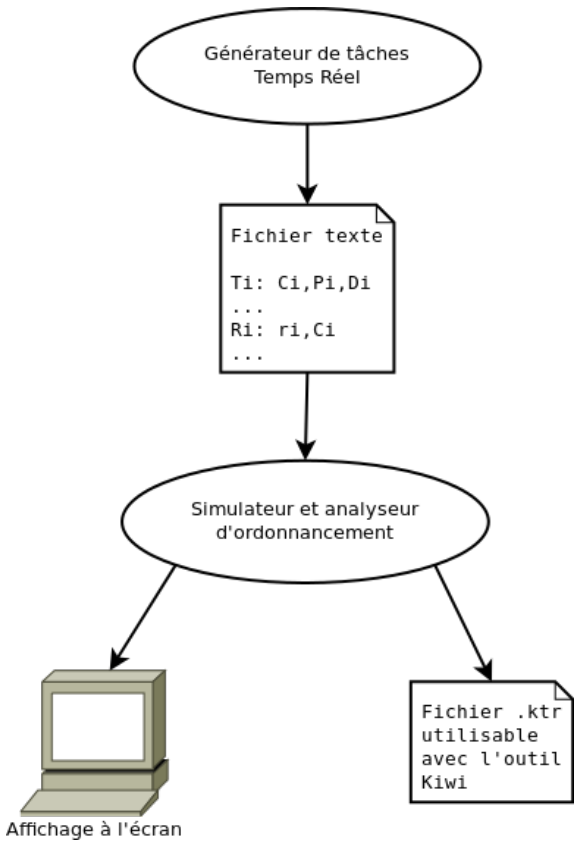

Figure 1 – Résumé

1 Générateur de tâches

1.1 Spécification de l'environnement

Cette application offre la possibilité de créer un fichier contenant un jeu de tâches (périodiques et apériodiques non critiques). Dans un premier temps, le programme permet de générer manuellement ce jeu de tâche en offrant une interface minimaliste. Dans un deuxième temps, il est possible de générer automatiquement ce fichier en fournissant des contraintes à respecter pour les tâches périodiques et apériodiques.

Lorsque le programme est lancé, l'utilisateur doit choisir le mode de génération. Il rentre les données nécessaires au bon déroulement du mode. Le programme crée un fichier qui contient le résultat de la génération de tâches.

1.2 Spécification du logiciel

1.2.1 Explication du choix du langage

Nous avons opté pour le langage C++ pour réaliser ce projet car celui-ci possède un certain nombre d'avantages. Il s'agit d'un langage orienté objet très utilisé dans le monde et de nombreux outils libres sont disponibles pour l'utiliser (comme GCC pour la compilation par exemple). De plus, la documentation correspondante est très facilement accessible dans la littérature ou sur le web.

Bien que le travail qui nous a été demandé ne requiert pas la maîtrise d'une programmation bas niveau, le langage C++ nous apparait en cohérence avec le thème abordé par le projet (les systèmes temps réel embarqués).

1.2.2 Fonctionnalités

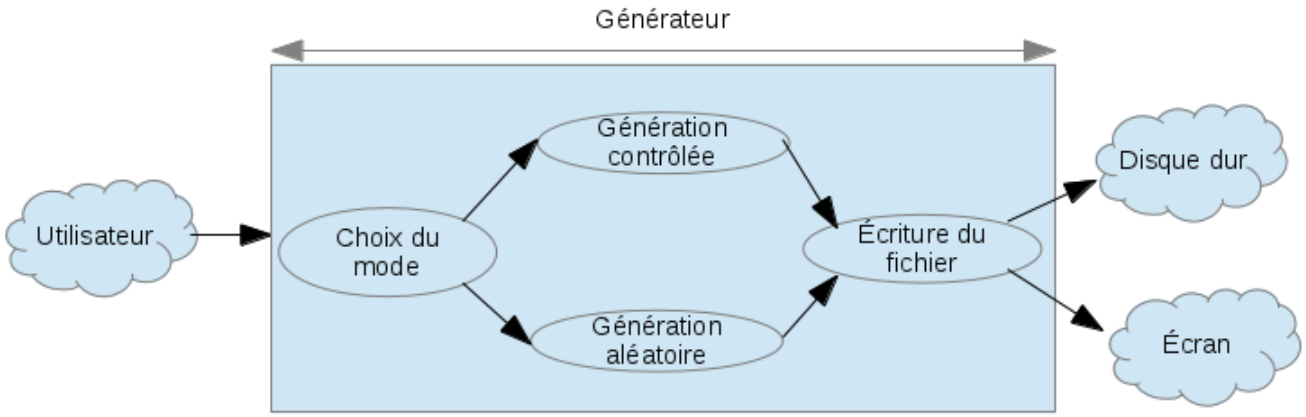

FIGURE 1.1 – Fonctionnement et modules du générateur.

Choix du mode

Dans ce premier module, on choisit le mode d'utilisation du logiciel en fonction de ce que souhaite l'utilisateur. Pour cela, il peut soit renseigner sous forme d'arguments les différentes données, soit en sélectionnant dans le menu ses préférences.

Parmi les choix qui lui sont offerts, on retrouve les cas présentés dans le sujet du projet (génération aléatoire ou contrôlée et tâches périodiques ou non) mais aussi la possibilité de renseigner un fichier contenant déjà toutes les informations sur les tâches. Dans le premier cas, l'utilisateur devra renseigner ensuite le nombre de tâches concernées tandis que dans l'autre cas, le fichier ne sera pas interprété et ce programme s'arrêtera.

Génération aléatoire de tâches

Le premier type de génération permet de générer automatiquement des tâches. L'utilisateur devra renseigner le nombre de tâches périodiques et apériodiques à générer ainsi que le pourcentage d'utilisation du CPU. Ceci permet de générer des fichiers de test aléatoire qui seront visualisable par la suite. Lors de la génération, le système calcule les différentes valeurs caractéristiques des tâches périodiques (Ci, Pi, Di) et des tâches apériodiques (ri, Ci) en accord avec le pourcentage d'utilisation du CPU.

Génération contrôlée de tâches

Le deuxième type de génération fait appel à l'utilisateur pour renseigner chacune des valeurs de chaque tâche. De cette manière, l'utilisateur va pouvoir tester plus facilement des cas spécifiques et donc observer de manière plus précise le fonctionnement de l'ordonnanceur par la suite.
Lors de la génération, pour toutes les tâches périodiques il devra renseigner les Ci, Pi et Di, c'est-à-dire respectivement les durées d'exécution maximales, les périodes d'activation et le délai critique). Dans le cas où il y a aussi des tâches apériodiques, il devra renseigner aussi ri (la date de réveil) et Ci.

Écriture dans un fichier

Ce module a pour fonction d'écrire dans un fichier les données contenues dans un flux (= stream) et calculées à partir des deux modules précédents de génération. L'intérêt de séparer ces fonctions est

double : il est plus pratique de factoriser le code à travers une seule fonction d'écriture appelée par les différentes générations et cela facilite la maintenance.

1.3 Conception fonctionnelle

1.3.1 Choix du mode

Pré-condition : Le programme vient d'être lancé.
Post-condition : Un choix valide a été effectué.
Objectif : Permettre de choisir entre la génération aléatoire et la génération controlée.
Algo :

```
  Entree : Int choix
  Sortie : -

Si (choix == 1 ) Alors
   Lancer la generation controlee
Sinon
   Si (choix == 2) Alors
      Lancer la generation aleatoire
   Sinon
      Si (choix == 3) Alors
         Quitter le programme
      Sinon
         Afficher erreur de choix
      Finsi
   Finsi
Finsi
```

1.3.2 Génération Aléatoire

Pré-condition : La génération aléatoire a été choisie.

Post-condition : Toutes les valeurs caractéristiques des tâches ont été générées en accord avec le paramétrage de l'utilisateur.

Objectif : Générer aléatoirement les caractéristiques (Ci,Pi,Di) d'un nombre de tâches (determiné par l'utilisateur) en accord avec un facteur d'utilisation du processeur (déterminé lui aussi par l'utilisateur). Ex : pour trois taches et facteur = 75%

Tirage aléatoire de trois nombres (35, 25, 15) dont la somme vaut 75.
Ces nombres représentent le rapport Ci/Pi dans la formule
Il faut ensuite donner une valeur à Ci et Pi. Pour cela, on trouve le pgcd du nombre et de 100. Pour C1/P1 = 35, on obtient 5. on affecte à C1 ← 35/5 et à P1 ← 100/5 , on obtient donc C1 = 7 et P1 = 20. On considère, dans ce programme que Pi = Di.

On obtient donc dans cette exemple :
T1(7,20,20)
T2(1,4,4)
T3(3,20,20)

Toutes les données sont stockées dans 3 tableaux (un pour les Ci, un pour Pi, un pour Di).

Algo :

```
Entree: Int nbTaches, Int factUtProcesseur
Sortie : 3 Tableaux d Int (pour les valeurs de Ci, Pi, Di).
indice du tableau = numero de la tache - 1..

TabCi[ nbTaches ], TabPi[ nbTaches ], TabDi[ nbTaches ]
nbTachesRestantes = nbTaches - 1
nbMax = factUtProcessus

/*
On calcule la valeur maximale que peut prendre le nombre tire aleatoirement :
maximum = Up - somme des Ci/Pi precedents - nombre de taches restantes
Ensuite, on tire au hasard une valeur allantDe 1 a cette valeur maximale.
Exemple : maxT1 = 75 - 0 - 2
          randT1 = 32 (valeur calculee avec le pseudo-hasard)
          maxT2 = 75 - 32 - 1
          randT2 = 18
          maxT3 = 75 - (32 + 18) - 0
          randT3 = 25
Pour resumer, on obtient ici les Ci/Pi : 32, 18 et 25
*/
```

```
Pour i allantDe 0 a (nbTaches - 1)
    nbMaxLimite = nbMax - nbTachesRestantes
    // le nombre aleatoire correspond a (Ci/Pi)
    nbAleatoire = nombre Aleatoire entre 1 et ce nombre maximum


    // On calcule et stocke la valeur des Ci, Pi et Di dans 3 tableaux distincts
    tabCi[i] = nb\_genere / pgcd(nb\_genere,100)
    tabPi[i] =100 / pgcd(nb\_genere,100)
    tabCi[i] = 100 / pgcd(nb\_genere,100)

    /* On abaisse ensuite la limite pour le prochain tirage aleatoire afin de ne jam
    limite\_maj = limite\_maj - nb\_genere
Fin du pour

// Pour la derniere tache, on ne genere pas de valeur. On prend ce qu il reste.
nb\_genere = limite\_maj
tabCi[ nbTaches - 1 ] = nb\_genere / pgcd(nb\_genere,100)
tabPi[ nbTaches - 1 ] =100 / pgcd(nb\_genere,100)
tabCi[ nbTaches - 1 ] = 100 / pgcd(nb\_genere,100)
```

1.3.3 Generation Controlée

Pré-condition : La génération controlée a été choisie.

Post-condition : Toutes les valeurs caractéristiques des tâches ont été générées en accord avec le paramétrage de l'utilisateur.

Objectif : Permettre à l'utilisateur de générer des tâches périodiques ou apériodiques en renseignant les différentes caractéristiques de chacune des tâches (Ci, Pi, Di) ou, respectivement, (ri, Ci).

Algo :

```
Entree : Int nbTaches
Sortie :  3 Tableaux d Int (pour les valeurs de Ci, Pi, Di).
indice du tableau = numero de la tache - 1 .

//generation des taches periodiques
lire(nbTachesPeriodiques)

TabCiP[ nbTachesPeriodiques ], TabPiP[ nbTachesPeriodiques ], TabDiP[ nbTachesPerio

Pour i allantDe 0 a (nbTachesPeriodiques - 1) Faire
    // on demande a l utilisateur de renseigner les differentes valeurs ...
    lire(CiP)
    lire(PiP)
    lire(DiP)
    // ... et on les insere dans les tableaux respectifs
```

```
    tabCiP[ i ] = CiP
    tabPiP[ i ] = PiP
    tabDiP[ i ] = DiP
Finpour

//generation des taches Aperiodiques
lire(nbTachesAperiodiques)

TabriA[ nbTachesAperiodiques ], TabCiA[ nbTachesAperiodiques ]

Pour i allantDe 0 a (nbTachesAperiodiques - 1) Faire
    // on demande a l utilisateur de renseigner les differentes valeurs ...
    lire(riA)
    lire(CiA)
    // ... et on les insere dans les tableaux respectifs
    tabriA[ i ] = riA
    tabCiA[ i ] = CiA
Finpour
```

1.3.4 Ecriture du résultat obtenu

Pré-condition : La génération (qu'elle soit contrôlée ou non) envoie un outputstream contenant les chaînes de caractère à écrire dans un fichier.

Post-condition : L'écriture s'est bien déroulée : le fichier contient bien la chaîne.

Objectif : Enregistrer les données générées par le module précédent de manière pérenne dans un fichier.

Algo :

```
creation du fichier

    outputstream ops = ""
Pour i allantDe 0 a (nbTachesPeriodiques - 1) Faire
    ops << "T" << (i+1) << ": " << tabCiP[i] << "," << tabPiP[i] << "," << tabDiP[i
Finpour

Pour i allantDe 0 a (nbTachesAperiodiques - 1) Faire
    ops << R" << (i+1) << ": " << tabriA[i] << "," << tabCiA[i] << endl
Finpour

ecriture dans le fichier de ops
```

1.4 Réalisation

1.4.1 Choix d'implémentation des structures

1.4.2 Problèmes rencontrés

Nous avons rencontré un problème purement algorithmique à propos de la génération aléatoire. En effet, notre objectif était de générer des nombre le plus aléatoirement possible, tout en contrôlant l'hyperperiode qui découle de ces nombres. Or, ces deux objectifs ne nous paraissaient pas totalement compatibles.

Nous avons implémenté deux solutions. La première, celle qui est présenté précédemment, permet de contrôler l'hyperperiode. Le soucis de cette solution est que les Pi générés ne sont pas totalement aléatoires, puisque ce sont uniquement des diviseurs de 100.

Nous avons donc cherché à implémenté une autre solution permettant de générer des Ci *et* des Pi totalement aléatoire. Après réflexion algorithmique, nous obtenions des chiffres très aléatoires et qui restent cohérents. Cependant les hyperperiodes indues étaient totalement irréalistes (souvent plusieurs centaines de milliers d'unités de temps). Nous avons donc décidé de retenir la première solution, tout en ayant à l'esprit qu'elle doive être améliorer dans une version future.

1.5 Tests

1.5.1 Test de la génération aléatoire

1.5.2 Test de la génération controlée

2 Simulateur d'ordonnanceur

2.1 Spécification de l'environnement

Cette application s'incrit à la suite du générateur de tâches. Elle offre, après récupération des informations du fichier issus du générateur, la possibilité de vérifier l'ordonnançabilité d'un jeu de tâches fournie. Ceci nous permet de savoir si les contraintes temporelles sont respectées.

La deuxième fonctionnalité de ce programme est d'offrir un environnement de simulation selon plusieurs politiques d'ordonnancements :

- pour les tâches périodiques : **RM** et **EDF**
- pour les tâches apériodiques non critiques : **BG** et **TBS**

Cet environnement, après simulation, fourni à l'utilisateur des résultats de performances relatifs à l'ordonnancement :

- le nombre de commutations de contexte,
- le nombre de préemptions
- les temps de réponse des tâches apériodiques.

A la suite de la simulation, le programme doit fournir un fichier de trace de la séquence d'ordonnancement. Ce fichier est au format "ktr" et est exploitable par l'outil Kiwi. Kiwi est un outil graphique développé à l'Université polytechnique de Valence en Espagne. Il permet, à partir d'un fichier texte normé, d'afficher un graphe d'ordonnancement. Nous détaillerons par la suite le contenu du fichier de trace mais il est important pour la suite de savoir que le fichier doit être remplit de manière chronologique.

Le programme final devra pouvoir être exécutable en ligne de commande.

En entrée : le nom du fichier contenant le jeu de tâches périodiques et apériodiques pour lesquelles on veut simuler l'ordonnancement.

En sortie : un ou plusieurs fichiers .ktr utilisable avec l'outil Kiwi et contenant la séquence d'ordonnancement. Le programme affichera également quelques résultats via la sortie standard.

Afin de faciliter la prise en main du programme, on proposera un menu succint présentant les différentes fonctionnalités qui s'offrent à l'utilisateur.

2.2 Spécification du logiciel

Pour les mêmes raisons que celles décrites dans le Chapitre 1.2.1, le langage utilisé est le C++.

2.2.1 Fonctionnalités

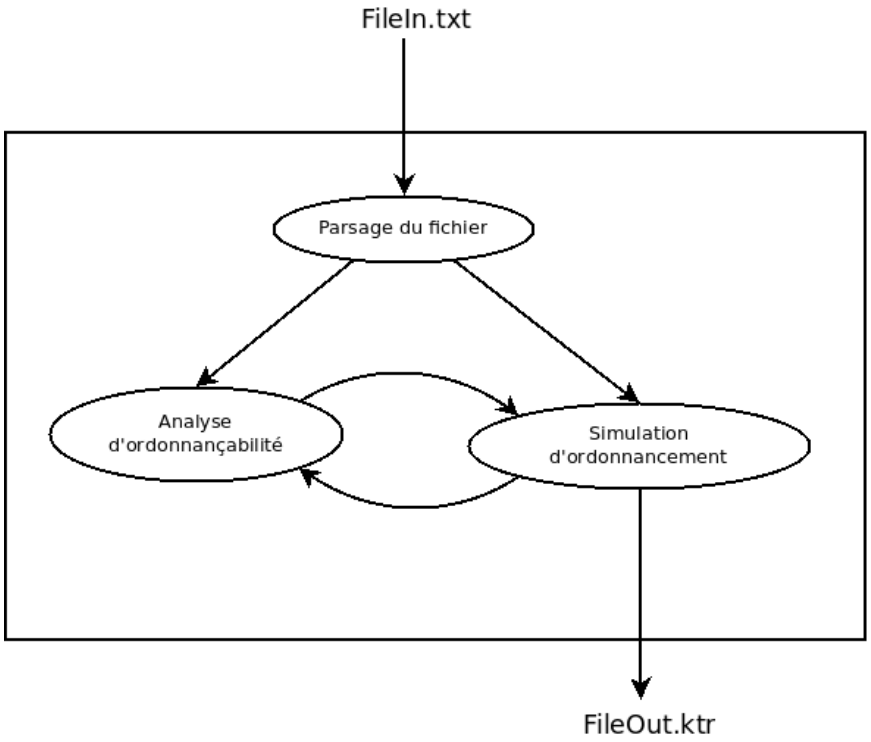

FIGURE 2.1 – Diagramme des fonctionnalités

Parsage du fichier d'entrée

Bien que le fichier en entrée soit, dans le cas général, un fichier écrit par notre générateur de tâche et donc normalement correct, nous ne sommes pas à l'abris de voir arriver en entrée un fichier corrompu. Soit par l'utilisateur lui-même, soit de n'importe quel autre moyen que ce soit. De plus, la syntaxe du fichier en entrée est relativement simple, ce qui permet à un utilisateur lambda de créer son propre fichier sans passer par notre générateur de tâches.

Sachant que le fichier sera parsé suivant le *pattern* de spécification des tâches utilisé par notre générateur, il est indispensable de vérifier la bonne syntaxe du fichier avant de récupérer quoi que ce soit. Cette vérification est effectuée grâce à une expression régulière représentant toutes les syntaxes possible pour une ligne du fichier. Dans les faits, cette *Regex* est divisée en deux :

- Pour les tâches périodiques : `^T[0-9]+: [0-9]+,[0-9]+,[0-9]+$` ;
- Pour les tâches apériodiques : `^R[0-9]+: [0-9]+,[0-9]+$` .

Chacune des lignes du fichier doit donc respecter soit l'une, soit l'autre expression régulière.

Une fois la syntaxe vérifiée, il est très aisé de travailler le fichier ligne par ligne pour en extraire les informations nécessaires grâce aux fonctions sur les *string*. Chacune des tâches sera ajouter à un Conteneur de tâches qui sera ensuite passé à l'ordonnanceur pour que celui-ci puisse travailler dessus.

Analyse d'ordonnançabilité

TODO ajouter les tests pour les taches périodiques où Di $<$ Pi On fait tous les calculs en même temps :

– Condition nécessaire pour l'ordonnançabilité du jeu de tâches avec RM ;
– Condition suffisante avec RM ;
– Condition suffisante et nécessaire avec EDF.

Simulation d'ordonnancement

Avant même de commencer à réfléchir sur les algorithmes d'ordonnancement, nous avons réfléchi à la structure de ces algorithmes. Nous verrons dans la conclusion que ça n'était pas forcément la bonne solution, et qu'il aurait mieux valu extraire cette structure de l'algorithme d'ordonnancement.

Premièrement, un ordonnanceur a besoin d'avoir plusieurs informations à propos des tâches qui lui demandent du temps d'exécution. Par exemple, l'algorithme EDF, qui élit une tâche selon son Di, doit bien entendu connaître le Di des différentes tâches qui attendent leur exécution. Toutes ces informations dont ont besoin les algorithmes, et qui seront stockées dans différentes structures, devront être initialisées en tout début de fonction afin que l'ordonnanceur puisse s'en servir à sa guise plus tard. Nous appelerons toutes ces informations le « contexte d'exécution ».

Bien entendu, ça n'est pas à l'ordonnanceur de mettre à jour le contexte d'exécution. On réservera donc la fin de chaque fonction d'ordonnancement à la mise à jour de ces structures.

Enfin, l'ordonnanceur ne doit pas être appelé à chaque unité de temps écoulée mais seulement : au réveil d'une tâche, à la fin de l'exécution d'une tâche, et au début d'une nouvelle période d'une tâche périodique.

Voici la structure finale des algorithmes d'ordonnancement :

1. Initialisation du contexte d'exécution ;
2. Jusqu'à la fin de l'hyperpériode :
   (a) Vérifier si l'ordonnanceur doit être appelé ;
   (b) Si oui, exécuter l'algorithme d'ordonnancement ;
   (c) Affichage du résultat à l'écran ;
   (d) Mettre à jour le contexte d'exécution .
3. Affichage des statistiques sur l'ordonnancement (nombre de changements de contexte, de préemptions, temps de réponse des tâches apériodiques) .

Une fois cette structure mise en place, il suffit simplement d'implémenter un algorithme d'ordonnancement qui va tout simplement retourner le numéro de la tâche qu'il a élue en utilisant les informations qui sont mises à disposition par le contexte d'exécution.

Ecriture du fichier de trace

Pendant la simulation de l'ordonnancement, l'application génére les différentes lignes qui composent le fichier de trace exploitable par Kiwi. Ce fichier se décompose en 2 parties :
– Une entête contenant la déclaration des tâches et la durée d'ordonnancement (hyperpériode)
– Les évènements concernants les tâches dans l'ordre chronologique.

2.3 Conception fonctionnelle

2.3.1 Conteneur

Comme nous l'avons vu, les tâches qui sont spécifiées dans le fichier en entrée doivent être stockées dans un conteneur pour que l'ordonnanceur puisse travailler dessus. Voici donc la manière dont nous avons décidé de concevoir notre conteneur de tâches :

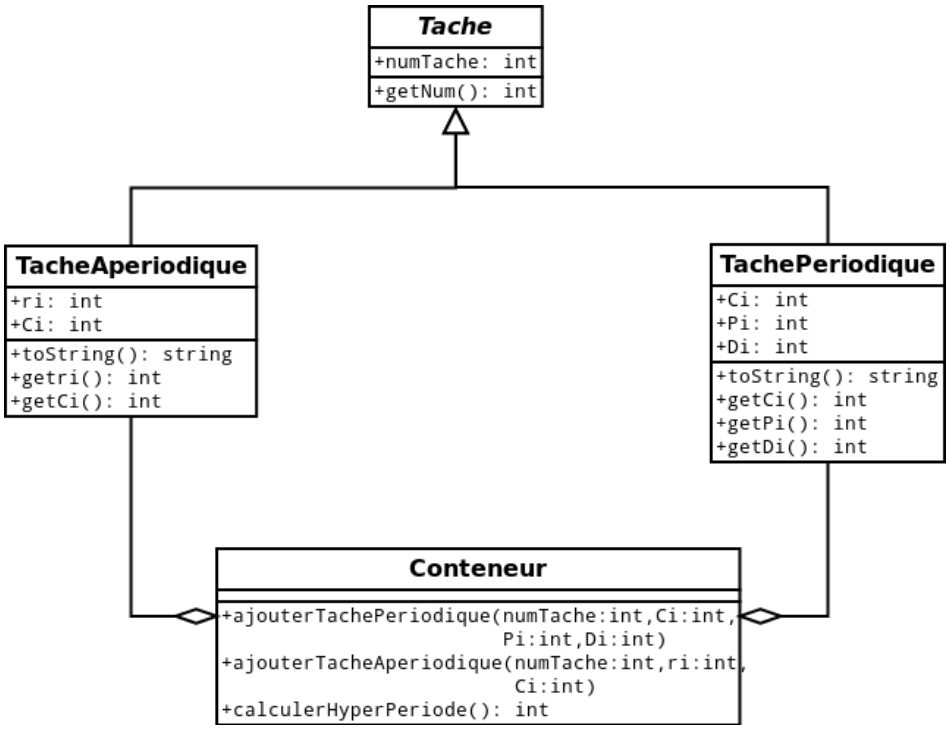

FIGURE 2.2 – Diagramme de classes du conteneur de tâches

2.3.2 Analyse du fichier d'entrée

Pré-condition : Le fichier spécifié existe et est lisible

Post-condition : Le fichier est syntaxiquement valide et toutes les tâches ont été ajoutées au conteneur

Objectif : Ajouter toutes les tâches spécifiées par le fichier dans le conteneur

Algo :

```
Entree : file fileIn , Conteneur conteneur
Sortie : conteneur

Tantque line = nextLine(fileIn) Faire
    verifierSyntaxe(line);

    elements = eclater(line , ',')
    Si commencePar(line , 'T') Alors
        ajouterTachePeriodique(conteneur , elements[0] , elements[1] , elements[2])
    Sinon si commencePar(line , 'R') Alors
        ajouterTacheAperiodique(conteneur , elements[0] , elements[1])
```

```
    Finsi
Fintantque
```

2.3.3 Analyse d'ordonnançabilité

Condition nécessaire RM

Pré-condition : n représente le nombre de tâches périodiques.
Post-condition : Soit c'est non-ordonnançable, soit on ne peut rien conclure.
Objectif : Vérifier les conditions d'ordonnançabilité nécessaires pour l'algorithme RM.
Algo :

```
U = 0

Pour i de 0 a n Faire
    U = U + Ci/Pi
Finpour

Si U <= 1.0 Alors
    afficher("On ne peut rien conclure")
Sinon
    afficher("Non-ordonnancable")
Finsi
```

Condition suffisante RM

Pré-condition : n représente le nombre de tâches périodiques.
Post-condition : Soit c'est ordonnançable, soit on ne peut rien conclure.
Objectif : Vérifier les conditions d'ordonnançabilité suffisantes pour l'algorithme RM.
Algo :

```
U = 0
UBoundRM = n * (2^(1.0/n) - 1)

Pour i de 0 a n Faire
    U = U + Ci/Pi
Finpour

Si U <= UBoundRM Alors
    afficher("Ordonnancable")
Sinon
    afficher("On ne peut rien conclure")
Finsi
```

Condition nécessaire et suffisante EDF

Pré-condition : n représente le nombre de tâches périodiques et il n'y a pas de tâches apériodiques.

Post-condition : On obtient une réponse binaire : le système est ordonnançable ou non.

Objectif : Vérifier les conditions d'ordonnançabilité nécessaires et suffisantes pour l'algorithme EDF.

Algo :

```
egaliteDiPi = vrai
superioritePiDi = vrai
U = 0

Pour i de 0 a n Faire
    // Verifie si tous les Pi sont bien egaux aux Di
    egaliteDiPi &= (Pi == Di)
    // idem pour Pi > Di
    superioritePiDi &= (Pi > Di)
Finpour

Si egaliteDiPi Alors
    afficher "Test de condition necessaire et suffisante pour EDF : "

    Pour i de 0 a n Faire
        U = U + Ci/Pi
    Finpour

    Si (U <= 1.0) Alors
        afficher "ordonnancable"
    Sinon
        afficher "non-ordonnancable"
    Finsi

Sinon Si superioritePiDi Alors
    afficher "Test de condition suffisante pour EDF : "

    Pour i de 0 a n Faire
        U = U + Ci/Di
    Finpour

    Si (U <= 1.0) Alors
        afficher "ordonnancable"
    Sinon
        afficher "on ne peut rien conclure"
    Finsi
Sinon
    afficher "Erreur : au moins un Pi < Di"
Finsi
```

Condition nécessaire et suffisante EDF-TBS

Pré-condition : n représente le nombre de tâches périodiques et Us est .

Post-condition : On obtient une réponse binaire : le système est ordonnançable ou non.

Objectif : Vérifier les conditions d'ordonnançabilité nécessaires et suffisantes pour l'algorithme EDF.

Algo :

```
Up = 0

Pour i de 0 a n Faire
    Up = Up + Ci/Pi
Finpour

afficher "Test de condition necessaire et suffisante pour EDF-TBS : "
Si (Up + Us <= 1.0) Alors
    afficher "ordonnancable"
Sinon
    afficher "non-ordonnancable"
Finsi
```

2.3.4 Simulation d'ordonnancement

Le simulateur d'ordonnancement est découpé en deux algorithmes principaux :

- RM ;
- EDF.

Nous avons en premier lieu commencé par développer ces deux algorithmes sans tenir compte des tâches apériodiques dans le but ensuite de les "patcher" avec nos algorithmes d'ordonnancement des tâches apériodiques (BG et TBS).

poll_needed

Pré-condition : un algo d'ordonnancement a été lancé

Post-condition :retourne un booléan qui informe du besoin du besoin d'élire un nouvelle tâche à exécuter

Objectif : Savoir si on doit élire une nouvelle tâche ou non

Algo :

```
bool need_to_poll = false;
//Ordonnanceur appele que si reveil ou terminaison de tache
    //verification du reveil d'une tache
        //Taches Periodiques

        Pour tachePeriodique dans context Faire
            Si time = (date Reveil de tachePeriodique) Alors
                ajout de tachePeriodique dans tabPeriodiquesPretes
```

```
                need_to_poll = true
            Finsi
        Finpour

        //Taches Aperiodiques
        Si (un serveur de taches a ete specifie) Alors
            Pour tacheAperiodique dans context Faire
                Si time = (date Reveil de tacheAperiodique) Alors
                    ajout de tacheAperiodique dans tabAperiodiquePretes
                    need_to_poll = true
                    Si (le serveur de tache aperiodique choisi est TBS) Alors
                        calcul de dk pour tacheAperiodique
                        stockage dans context
                    Finsi
                Finsi
            Finpour
        Finsi

    //verification de la terminaison de la tache courante executee

    Si (la tache courante executee n'est PAS un temps creux) Alors
        Si (la capacite restante de tache courante executee = 0) Alors
            need_to_poll = true
            //Suppression de la tache d'un des tableaux de taches pretes
            Si (la tache courante executee est une tache Aperiodique) Alors
                suppression de la tache de tabAperiodiquesPretes
            Sinon
                suppression de la tache de tabPeriodiquesPretes
            Finsi

        Finsi
    Finsi

    retourne need_to_poll
```

init_context

Pré-condition :
Post-condition :
Objectif :
Algo :

```
//Initialisation du context :
    - creation d'un tableau contenant toutes les taches (taches periodiques puis ta
```

```
    - le tableau context est un tableau de 3 case de large et nbTaches de long
    - pour les taches periodiques :
        . indice -> Capacite restant | prochaine date de reveil | prochaine date ec
    - pour les taches aperiodiques :
        . indice -> Capacite restante | date de reveil | date echeance ( si TBS sin

 context[ nbTaches ] [ 3 ]

 Pour i allantDe 0 a nbTachePeriodiques Faire
    context[i][0] = Ci de tachePeriodique_i
    context[i][1] = Pi de tachePeriodique_i
    context[i][2] = Di de tachePeriodique_i
 Finpour

 Si (il y a un serveur de precise) Alors
    Pour i allantDe nbTachesPeriodiques a nbTaches Faire
        context[i][0] = Ci de tacheAperiodique_i
        context[i][1] = ri de tacheAperiodique_i
        context[i][2] = 0
    Finpour
 Finsi
```

MAJ_context

Pré-condition :
Post-condition :
Objectif :
Algo :

```
Pour tachePeriodique dans context Faire
    //verification du depassement d'echeance Alors
    Si (time = (date echeance tachePeriodique) ) Alors
        Si ( (capacite restante de tachePeriodique) > 0 ) Alors
            Message("Depassement d'echeance pour la tache : T" + tachePeriodique)
            quitter
        Finsi
    Finsi

    Si (time = (date reveil de tachPeriodique) ) Alors
        capacite restante tachePeriodique = Ci de tachePeriodique
        date de reveil tachePeriodique = date de reveil tachePeriodique + Pi de tac
        date echeance tachePeriodique = date echeance tachePeriodique + Di de tache
    Finsi
Finpour
```

```
//Verification de depassement d'echeance pour les taches aperiodiques
//Si serveur de taches aperiodiques est TBS Alors
    Si (serveur = TBS) Alors
        Pour tacheAperiodique dans context Faire
            Si (time = (deadline de tacheAperiodique) ) Alors
                Si ( (capacite restante tacheAperiodique) > 0 ) Alors
                    Message("Depassement d'echeance pour la tache : R" + tacheAperi
                    quitter
                Finsi
            Finsi
        Finpour
    Finsi
Finsi
```

Rate Monotonic + Serveur

Pré-condition :
Post-condition :
Objectif :
Algo :

```
Entree :
Sortie :

//Initialisation du context :
    - Creation d'un tableau de taches periodiques par ordre de prio : tabPrioPeriod
    - Creation d'un tableau de taches aperiodiques par ordre de prio : tabPrioAperi
    - Creation d'un tableau qui contient la capacite restante de chaque tache : tab
init_context()

int t := 0
int task_executed := -1
bool tache_elue = false;

Tantque (t < getHyperPeriode()) Faire

    Booleen need_to_poll = poll_needed()

    Si(need_to_poll) Alors
        tache_elue = false
        Pour taches dans tabPrioPeriodique Faire
            Si (tabTpsRestant[tache] >0 && tache_elue == false) Alors
                task_executed = tache;
                tache_elue = true;
```

```
            Finsi
        Finpour
        // si Aucune tache n'a ete choisie -> temps creux
        Si tache_elue == false Alors
            //temps_creux
        Finsi
    finsi

    tabTpsRestant[task_executed]--;
    maj_context()
    t++

Fintantque
```

Earliest Deadline First + Serveur

Pré-condition :
Post-condition :
Objectif :
Algo :

```
Entree :
Sortie :


init_context()
int t := 0
int task_executed := -1

Tantque (t < getHyperPeriode()) Faire

    Booleen need_to_poll = poll_needed()

    Si(need_to_poll) Alors
        Si (aucune tache periodique prete) Alors
            Si (aucune tache aperiodique prete) Alors
                temps_creux
            Sinon
                task_executed = Selection de la tache Aperiodique prete dont le ri
            Finsi
        Sinon
            deadlineProche = hyperperiode
            task_executed = temps_creux

            //election de la tache Periodique prete la plus prioritaire
```

```
        Pour tachePeriodiquePrete dans tabPeriodiquesPretes Faire
            Si ( deadlineProche > (prochaine date echeance tachePeriodiquePrete
                deadlineProche = prochaine date echeance tachePeridiodiquePrete
                task_executed = tachePeriodiquePrete
            Sinon
                Si ( deadlineProche = (prochaine date echeance tachePeriodiqueP
                    //Conflit --> on compare les dates de reveil
                    Si ( ri_task_executed > ri_tacheAperiodiquePrete) Alors
                        task_executed = tachePeriodiquePrete
                    Sinon
                        Si (ri_task_executed = ri_tacheAperiodiquePrete) Alors
                            // Conflit --> on compare les numeros de tache
                            Si (num_task_executed > num_tachePeriodiquePrete) A
                                task_executed = tachePeriodiquePrete
                            Finsi
                        Finsi
                    Finsi
                Finsi
            Finsi
        Finpour

        //election de la tache Aperiodique prete la plus prioritaire
        Si (serveur == TBS) Alors
            Pour tacheAperiodiquePrete dans tabAperiodiquesPretes Faire
                Si ( (dk de tacheAperiodiquePrete) < deadlineProche ) Alors
                    //La tache Aperiodique est plus prioritaire
                    task_executed = tacheAperiodiquePrete
                Sinon
                    Si ( (dk de tacheAperiodiquePrete) = deadlineProche ) Alors
                        Si ( (ri de tacheAperiodiquePrete) < (ri de task_execut
                            task_executed = tacheAperiodiquePrete
                        Finsi
                    Finsi
                Finsi
            Finpour
        Finsi
    Finsi
Finsi

Capacite_restante de task_executed--
t++

MAJ_context()
```

fin tant que

2.4 Réalisation

2.4.1 Choix d'implémentation des structures

,

2.4.2 Problèmes rencontrés

2.5 Tests

3 Notice utilisateur

Vous disposez normalement des sources de l'application ainsi que des Makefile répartis dans deux répertoires : *Generateur* et Ordonnanceur. L'application est en effet divisée en deux partie, l'une générant un jeu de tâche périodiques et apériodiques dans un fichier texte, l'autre simulant l'ordonnancement des tâches générées.

Étape 1 : compilation

```
$ cd Generateur
$ make
```

4 Conclusion

4.1 Problèmes rencontrés

4.1.1 Combiner l'aléatoire et le contrôle des valeurs générées

Nous avons rencontré un problème purement algorithmique à propos de la génération aléatoire. En effet, notre objectif était de générer des nombre le plus aléatoirement possible, tout en contrôlant l'hyperperiode qui découle de ces nombres. Or, ces deux objectifs ne nous paraissaient pas totalement compatibles.

Nous avons implémenté deux solutions. La première, celle qui est présenté précédemment, permet de contrôler l'hyperperiode. Le soucis de cette solution est que les Pi générés ne sont pas totalement aléatoires, puisque ce sont uniquement des diviseurs de 100.

Nous avons donc cherché à implémenté une autre solution permettant de générer des Ci *et* des Pi totalement aléatoire. Après réflexion algorithmique, nous obtenions des chiffres très aléatoires et qui restent cohérents. Cependant les hyperperiodes indues étaient totalement irréalistes (souvent plusieurs centaines de milliers d'unités de temps). Nous avons donc décidé de retenir la première solution, tout en ayant à l'esprit qu'elle doive être améliorer dans une version future.

Exemple de résultat obtenu avec la deuxième solution pour la génération de 5 tâches périodiques avec 75% d'utilisation processeur, sous la forme $Ti : Ci, Pi, Di$:

```
T1: 7,78,78
T2: 6,19,19
T3: 17,123,123
T4: 4,83,83
T5: 12,63,63
```

Ce jeu de tâche nous donne une hyperpériode de 105 908 166 unités de temps, ce qui est absolument disproportionné.

4.2 Améliorations possibles

4.2.1 Fonction de génération aléatoire

Comme nous l'avons vu dans la partie 3.1.1, nous avons rencontré un problème pour développer une fonction de génération aléatoire qui soit optimale. Une amélioration possible pour la suite serait de trouver une solution algorithmique permettant de combiner les avantages des deux méthodes :

un résultat vraiment aléatoire, et une hyperpériode *contrôlée*, c'est-à-dire raisonnable. Nous pensons qu'une hyperpériode raisonnable se définit approximativement entre 0 et 1000 au vu de l'objectif de l'application.

4.2.2 Factoriser les éléments redondants des algorithmes d'ordonnancement

Nos fonctions d'ordonnancement ont un défaut : elles contiennent bien plus que l'algorithme d'ordonnancement qui leur correspond. Par exemple, dans la fonction RM(), il y a tout un tas d'initialisation et de calculs avant d'arriver au coeur de l'algorithme. De plus, les éléments algorithmiques extérieurs à l'ordonnancement sont similaires pour chacune des deux fonctions. Il serait donc intéressant de factoriser ces éléments en créant par exemple une fonction *init_ context()* qui s'occupe d'initialiser toutes les variables et structures représentant le contexte d'éxecution, une fonction *maj_ context()* pour mettre à jour ce contexte, ainsi qu'une fonction *need_ to_ poll()*.

Pour des raisons de lisibilité, nous avons d'ailleurs choisi de représenter ce découpage dans la description des algorithmes d'ordonnancement (partie 2.3.4).

4.2.3 Ajout d'algorithmes

Lorsque nous étions à mi-chemin dans l'avancement du projet, nous avions l'ambition d'ajouter quelques autres algorithmes d'ordonnancement afin d'enrichir notre application. Malheureusement, nous n'avons pas eu le temps d'aller plus loin. Nous proposons donc l'ajout de ces nouveaux algorithmes à la liste des améliorations possibles.