M1 ALMA
Université de Nantes
2010-2011

Projet de Travaux pratiques : Systèmes Distribués Mini-projet1

MARGUERITE Alain
RINCÉ Romain

Université de Nantes
2 rue de la Houssinière, BP92208, F-44322 Nantes cedex 03, FRANCE

Encadrant : QUEUDET Audrey

Table des matières

1 Cahier des charges

Introduction : L'objectif est de définir un outil de simulation d'ordonnancement de tâches en temps réel. Parmi ses fonctionnalités, l'outil devra pour tester les contraintes temporelles d'un ensemble de tâches générées au préalable. La génération de ces tâches entre dans la conception de l'outil. Cet outil permettra d'exporter le résultat dans un fichier d'extension *.ktr* pour être exploité directement par l'outil graphique Kiwi.

1.1 Données en entrées

L'outil doit pouvoir permettre à l'utilisateur de rentrer des tâches périodiques et ou apériodiques lui même (en précisant chacun des attributs) ou de demander une génération aléatoire pour les deux catégories.

1.2 Fonctionnement

- Une analyse d'ordonnançabilité. L'outil affichera à l'utilisateur les résultats des différents tests avec les conclusions qui en découlent.
- Un environnement de simulation. L'outil lors du calcul de l'ordonnancement devra afficher les différents événements. Un bilan de ces action sera résumé dans un fichier au terme de l'execution (facultatif).
  L'outil doit pouvoir proposer plusieurs politiques d'ordonnancement. À savoir :
  - Pour les tâches périodiques :
    - Rate Monotonic
    - EDF
  - Pour les tâches apériodiques :
    - BG
    - TBS
- Un fichier d'extension *.ktr* sera généré au terme de l'execution, et contiendra le déroulement de l'ordonancement jusqu'a son terme.

- L'outil doit communiquer, au terme de l'exécution, différents résultats de performance qu'il aura lui même calculés. Les informations à fournir sont les suivantes :
  - Le nombre de violations d'échéances.
  - Le nombre de commutations de contexte et de préemptions.

2 Analyse et solution du problème

2.1 Introduction et choix du langage

Dans le cadre du module Systèmes Distribués, l'opportunité de spécifier et concevoir un générateur de tâches temps réel nous est proposé. Le langage d'implémentation étant libre, notre binome a opté pour Java. Ce choix est basé principalement par le fait qu'il s'agissait du langage le mieux maitrisé. Cela nous a permis de concentrer nos efforts sur la mise en place des algorithmes et non sur des probèmes de langage. L'objectif est de définir un outil de simulation d'ordonnancement de tâches en temps réel.

2.2 Modelisation du problème

Suite à l'étude du cahier des charges nous proposons l'architecture de l'outil suivante :

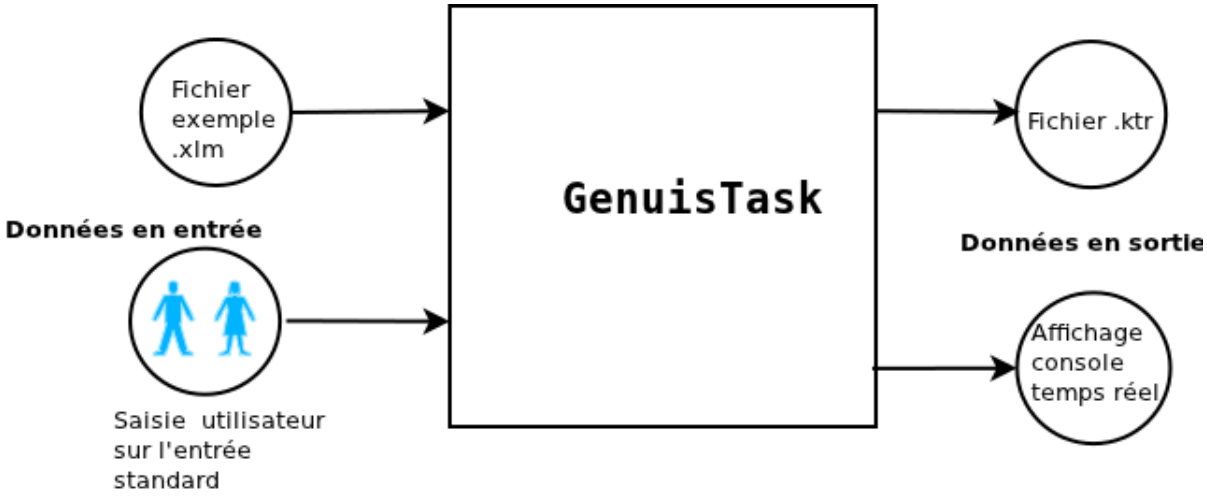

FIGURE 2.1 – Architecture de l'outil

2.3 Génération de tâches dans un fichier

La première partie du projet avait pour objectif d'obtenir un nombre n de tâches périodiques et apériodiques pour de futurs traitements décrits dans la seconde partie (2.4.2). À nouveau le choix du format d'un tel fichier nous était laissé. Nous avons choisi de générer un fichier xml (à nouveau pour des raisons de simplicité) à la syntaxe suivante :

- Des balises `<genTache.AbstractTache-array>` encadrent la totalité du fichier.
- Une tâche périodique sera définie dans une balise `<genTache.TachePeriodique>`
- Une tâche apériodique sera définie dans une balise `<genTache.TacheAPeriodique>`
- Dans une tâche tous ses attributs seront définis de la manière suivante `<nom_attribut>valeur_attribut</nom_attribut>`

Voici un exemple d'un fichier respectant le format décrit ci-dessus :

```
<genTache.AbstractTache-array>
  <genTache.TachePeriodique>
    <Pi>377</Pi>
    <ri>0</ri>
    <id>1</id>
    <Ci>1</Ci>
    <Di>1</Di>
  </genTache.TachePeriodique>
  <genTache.TachePeriodique>
    <Pi>162</Pi>
    <ri>0</ri>
    <id>2</id>
    <Ci>6</Ci>
    <Di>30</Di>
  </genTache.TachePeriodique>
  <genTache.TacheAperiodique>
    <ri>859</ri>
    <id>3</id>
    <Ci>26</Ci>
    <Di>71</Di>
  </genTache.TacheAperiodique>
</genTache.AbstractTache-array>
```

On remarque que les taches périodiques sont identifiées par :

- Pi : période d'activation.

- ri : date de réveil.
- id : L'id de la tâche.
- Ci : durée d'exécution maximale.
- Di : délai critique

Alors que les tâches apériodiques ont seulement :

- ri : date de réveil
- id : L'id de la tâche.
- Ci : durée d'exécution maximale.
- Di : délai critique.

2.4 Algorithmes et structures mises en oeuvre

2.4.1 Commentaires sur la première Partie

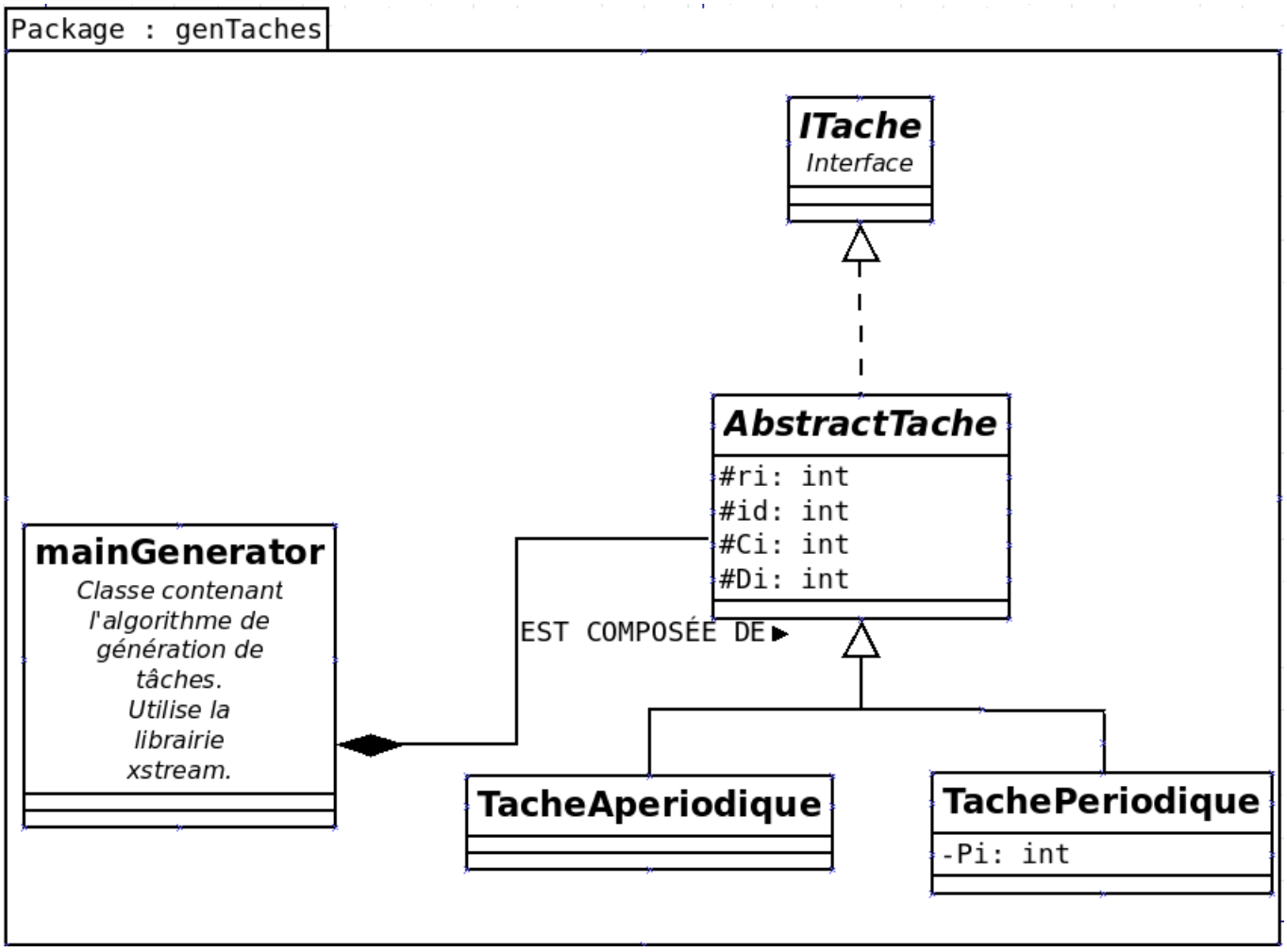

FIGURE 2.2 – Package générateur de tâches temps réel

Calcul des tâches apériodiques Le calcul des tâches ap est effectué selon la formule suivante : $U_a = \frac{\sum_{i=1}^{m} C_i}{ppcm(P_i)}$. L'utilisateur entre la variable Uap et le nombre de tâches apériodiques qu'il désire.

Détail sur la génération aléatoire Le cahier des charges spécifiait que la génération se devait d'offrir la possibilité de générer les tâches aléatoirement ou bien en laissant l'utilisateur choisir les paramètres de chacune d'elle. Cependant la génération de tâches aléatoires pose de nombreux problèmes.

Le premier problème est de savoir qu'elle est la taille maximale que l'on peut attribué a un Ci ou un Pi. Il aurait été possible de laisser le choix à l'utilisateur mais dès que le nombre de tâches dépasse deux ou que leur Pi est grand, l'hyperpériode explose. L'objectif étant par la suite de pouvoir générer un fichier pour kiwi, il est assez génant d'avoir des durées d'ordonnancements qui dépassent les possibilités des valeurs pour kiwi.

Un autre problème étant de savoir comment générer les tâches pour offrir une palette large vis-à-vis des propriétés des tâches.

Au final le programme de génération des tâches aléatoires est peu sûr et laisse la possibilité, à l'utilisateur, d'entrer des informations contradictoires qui peuvent, dans la majorité des cas, faire échouer la génération. Par exemple l'utilisateur peut rentrer un nombre de tâches apériodiques non nulle mais mettre ensuite une utilisation processeur nulle pour ces mêmes tâches, ayant pour conséquence de retourner une erreur lors de la génération.

2.4.2 Deuxième partie

La deuxième partie du projet, consistait à partir du fichier de tâches (cf Première Partie), de créer un simulateur proposant deux types de fonctionnalités :

- une analyse d'ordonnançabilité,
- un environnement de simulation.

Le premier sujet de réflexion était de savoir dans quelles structures de données stocker les tâches et par quels moyens. Le choix des ArrayList s'est fait naturellement grâce en premier lieu à sa facilité de manipulation. Par ailleurs la fonctionnalité de la bibliothèque xtream permettant d'extraire le contenu d'un fichier xml dans une ArrayList en quelques lignes nous a d'emblée convaincue. Une fois cette première difficulté franchie une seconde de taille est apparue. Nous avons compris qu'il serait délicat de modifier directement les tâches dans les ArrayList. Notre idée initiale était de modifier le Ci d'une tâche après son traitement dans l'algorithme. Ce qui dans le cas des tâches périodiques par exemple, serait problématique. En effet par définition les taches périodiques sont susceptibles d'être traitées à plusieurs reprises. Il faut donc garder leurs propiétés initiales intactes. Nous avons donc choisi d'organiser notre programmation en plusieurs classes pour bien séparer les opérations de

manipulation des données d'une part, les opérations mettant en jeux les algorithmes d'ordonancement et enfin celles écrivant dans le fichier de sortie au format .ktr

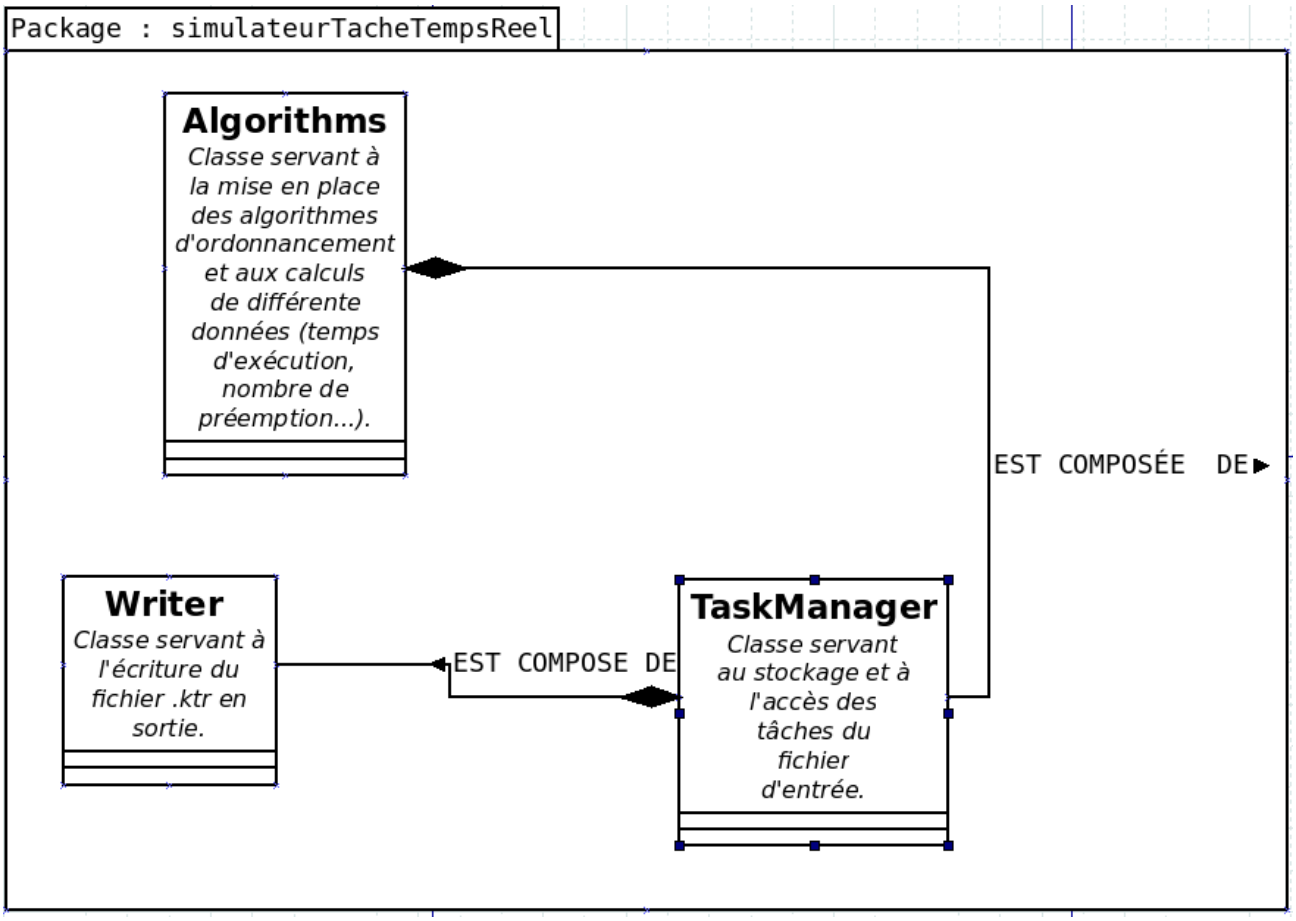

FIGURE 2.3 – Package : Simulateur de systèmes temps réel

2.5 Interface proposée

Toutes l'interface de l'outil est en ligne de commande. L'absence d'IHM graphique est due au manque de temps et au peu d'intêret que cela représentait pour le problème étudié. De même l'outil ne prend en compte aucune option. Même si cet aspect aurait apporté un certain confort à l'utilisateur, nous avons opté pour la solution de lui demandé au fur et à mesure de l'exécution du programme les diverses options et paramètres requis. Nous avons regretté ce choix lors de la phase des tests (la répétition sans cesse des divers options est fastidieuse). Cependant nous avons gagné un temps de conception non négligeable.

3 Manuel utilisateur

4 Jeux d'essais

4.1 Cas de RMBG

L'utilisation de la combinaison des algorithmes Rate Monotonic et BackGround donne les résultats suivants :

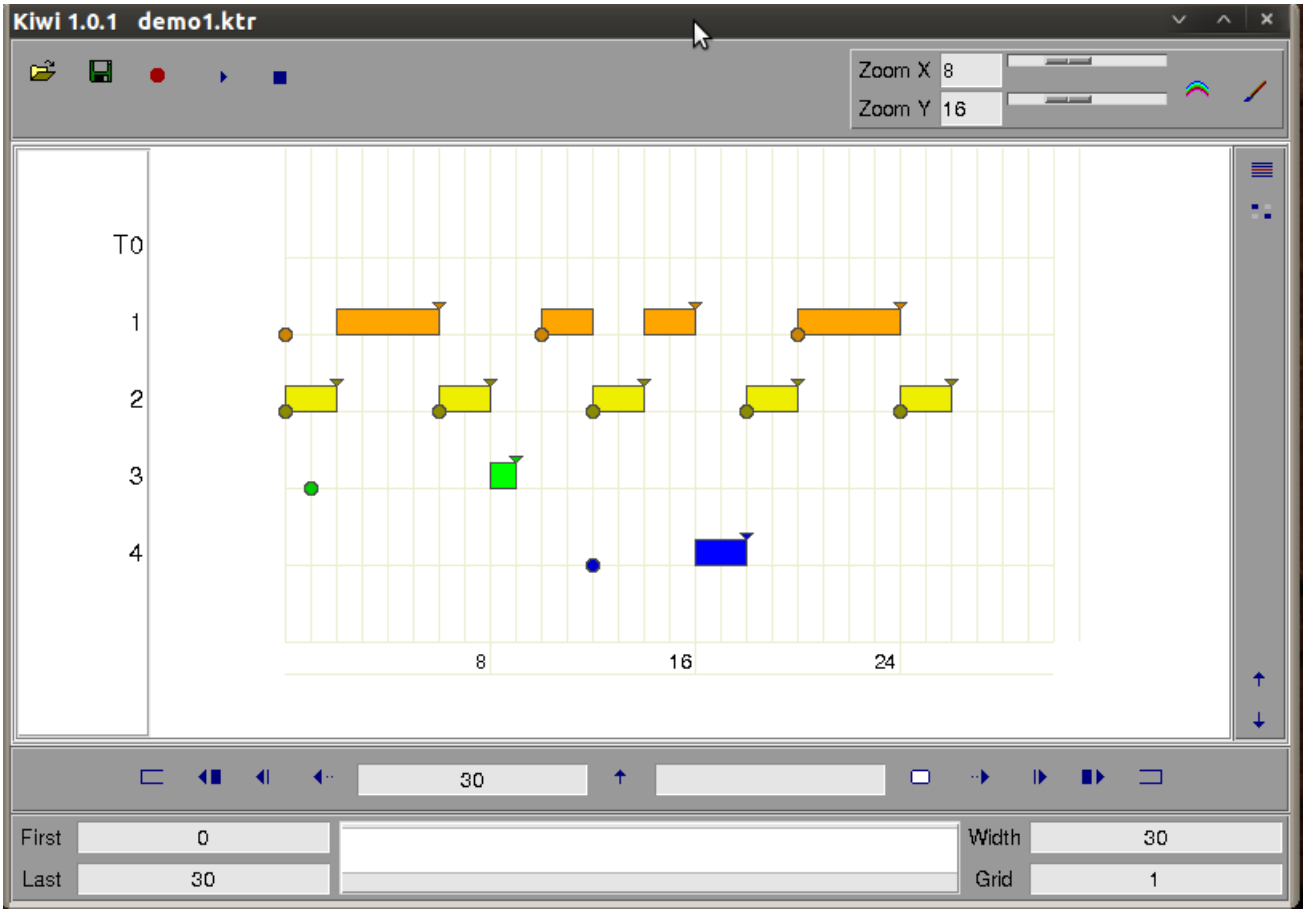

FIGURE 4.1 – Résultat de l'appelle à RMBG

```
PPCM : 30
Resultat du test de faisabilité : 0.73333335
```

```
Déroulement de l'algorithme

****BILAN ET ANALYSE****
Temps d'execution : 30
Temps creux : 5
Utilisation du processeur :83
Nombre de préemptions :1
****TacheAp****
Temps de réponse min : 4
Temps de réponse max : 7
Temps de réponse moy : 6.5
```

4.2 Cas de EDFBG

L'utilisation de la combinaison des algorithmes Earliest Deadline First et BackGround donne les résultats suivants :

```
Resutalt du test pour Ci<Pi selon EDF U= : 0
U<=1 condition suffisante vérifiée

Déroulement de l'algorithme

****BILAN ET ANALYSE****
Temps d'execution : 20
Temps creux : 0
Utilisation du processeur :100
Nombre de préemptions :2
****TacheAp****
Temps de réponse min : 6
Temps de réponse max : 8
Temps de réponse moy : 8.0
```

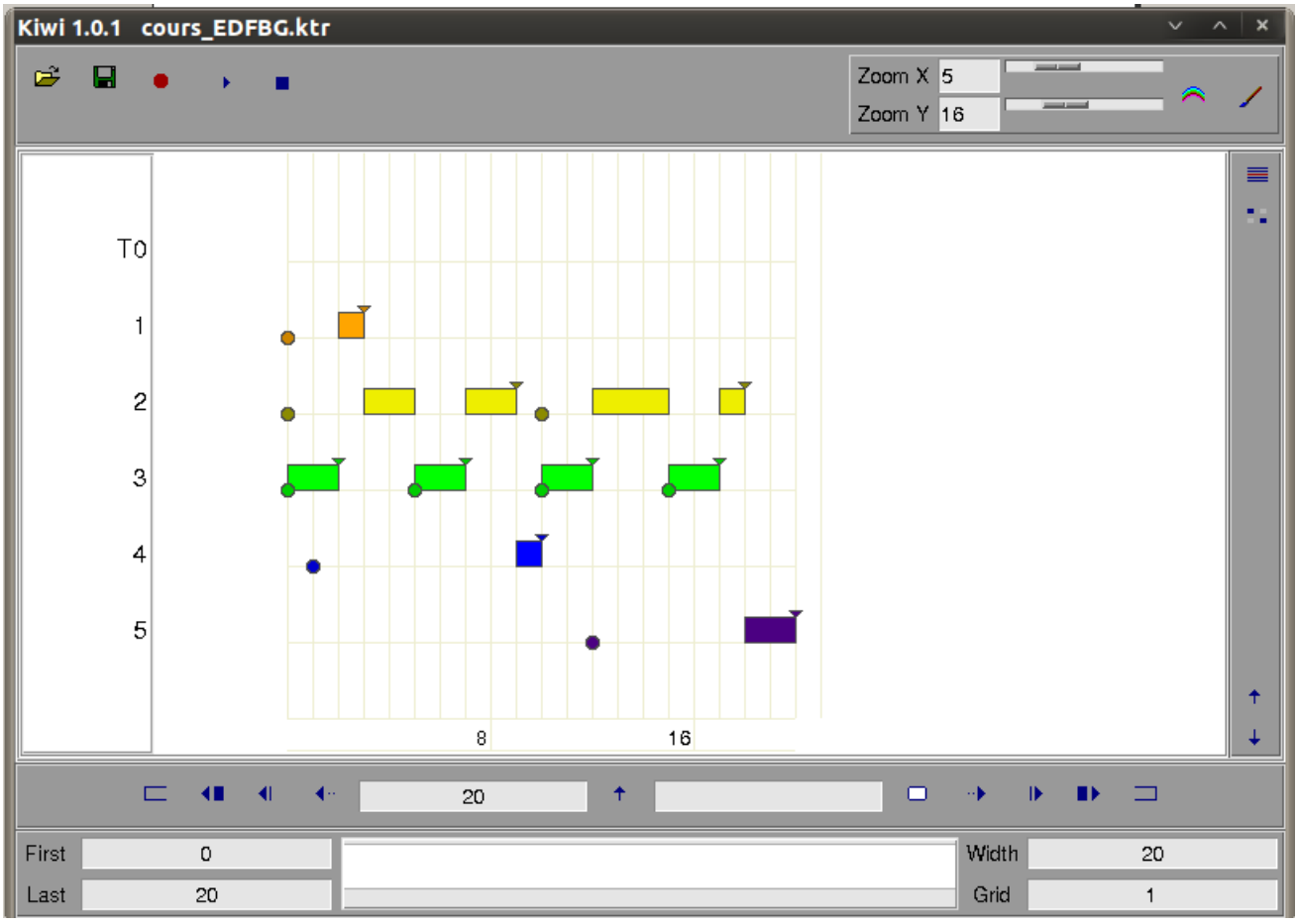

FIGURE 4.2 – Résultat de l'appelle à EDFBG

4.3 Cas de EDF-TBS

L'utilisation de la combinaison des algorithmes Earliest Deadline First et Total Bandwith Server donne les résultats suivants :

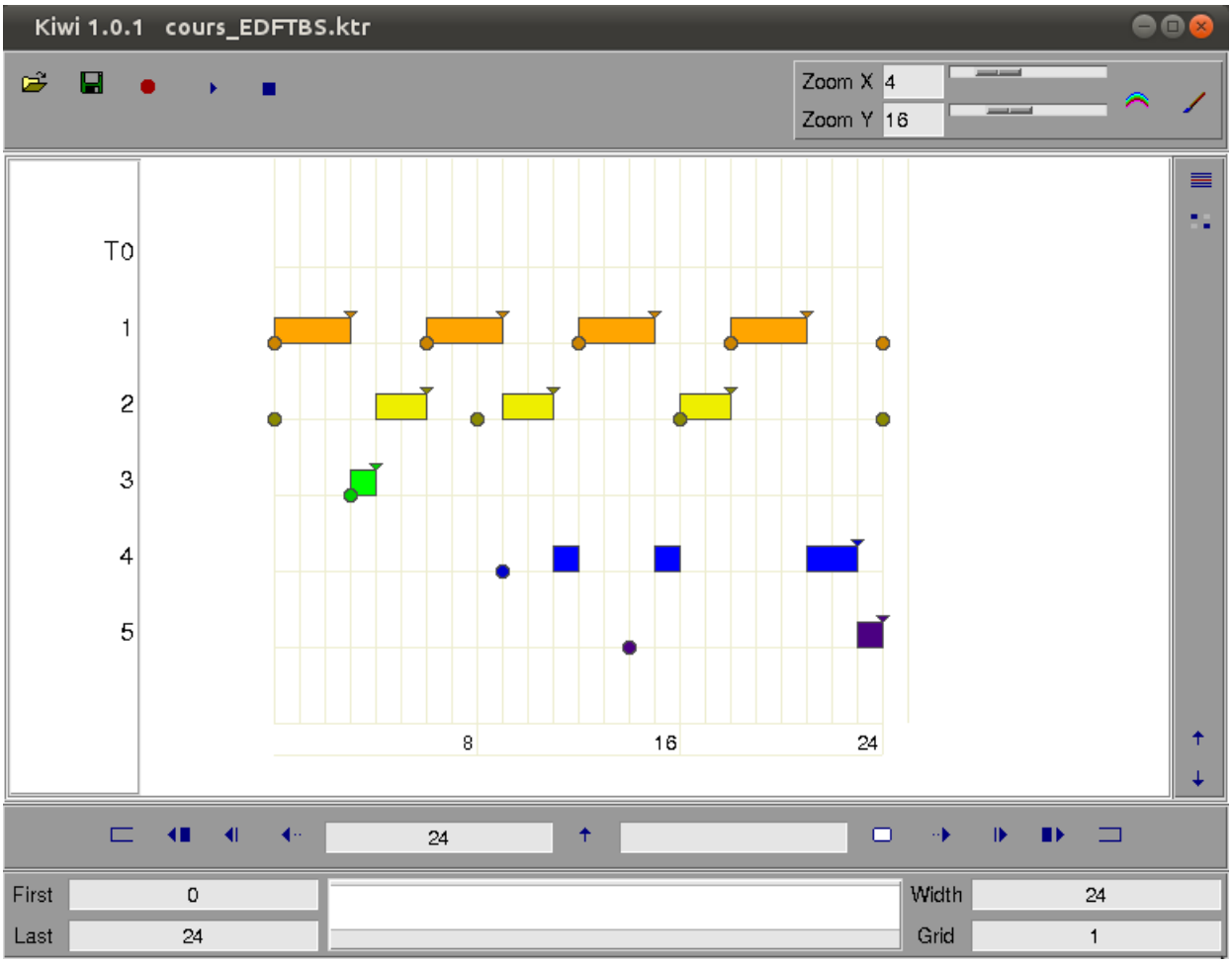

FIGURE 4.3 – Résultat de l'appelle à EDF-TBS

```
Déroulement de l'algorithme

****BILAN ET ANALYSE****
Temps d'execution : 24
Temps creux : 0
Utilisation du processeur :100
Nombre de préemptions :2
****TacheAp****
Temps de réponse min : 1
```

```
Temps de réponse max : 14
Temps de réponse moy : 8.333333
```

5 Bilan et conclusion

5.1 Génération des tâches

Bien que la partie de génération ne fut pas la partie la plus ardue, il a été nécessaire de revenir dessus pour tenir en compte les divers informations et spécifications grapillées sur le sujet auprès des enseignants. Le code à donc souvent été modifié à la volée le rendant petit à petit illisible voir erroné.

On peut aussi regretter que la génération aléatoire de tâches entraine l'impossibilité d'étudier leur ordonnancement dans kiwi à cause du problème de l'hyperpériode précédemment évoqué 2.4.1.

5.2 Mise en oeuvre des algorithmes

Les résultats des différents algorithmes sont plutôt satisfaisants. En effet pour chacun d'entre eux les tests de vérifications ont été concluants. Nous regrettons cependant de ne pas avoir eu le temps d'effectuer plus de tests à partir de la génération automatique des tâches. Les problèmes rencontrés sur la partie génération automatique (cf section précédente) en est la principale cause.

À propos des algorithmes RMBG et EDFBG. Nous avons rencontré des difficultés à produire un code court et clair. Nous avons perdu beaucoup de temps à débugger un code lourd. Avec du recul, même si les résultats sont positifs, il aurait été peut être été plus judicieux de repartir de zéro et repenser le design de ces deux algorithmes. L'algorithme EDF-TBS a d'ailleurs été le plus rapide à impléménter vu qu'il a été fait sans se servir de RMBG et EDFBG.

Table des figures