UNIVERSITÉ LIBRE DE BRUXELLES
Faculté des Sciences
Département d'Informatique

Test d'ordonnançabilité pour systèmes à criticité mixte par exploration d'automates

Simon PICARD

Promoteurs :
Prof. Gilles GEERAERTS
Prof. Joël GOOSSENS

Mémoire présenté en vue de
l'obtention du grade de
Master en Sciences Informatiques

Année académique 2015 - 2016

Résumé

Ce mémoire propose un test d'ordonnançabilité CM exact pour un système de tâches CM périodiques ou sporadiques avec un algorithme donné.
La solution de ce problème se base sur une réduction de celui-ci vers l'accessibilité dans un automate fini. Cette solution est ensuite améliorée pour les systèmes de tâches CM sporadiques grâce à la définition d'une relation de simulation.

La première partie du mémoire présente toutes les notions nécessaires pour la compréhension du travail fourni : l'ordonnancement temps réel, l'ordonnancement en criticité mixte, un automate fini, l'accessibilité et les antichaînes basées sur une relation de simulation.

Ensuite, une présentation des algorithmes d'ordonnancement CM pour tâches CM sporadiques est faite, en insistant sur leur condition suffisante d'ordonnançabilité. Les algorithmes présentés sont *Vestal*, *OCBP* et ses extensions, *EDF-VD*, *Greedy* et un algorithme original inspiré de *LLF* est proposé : *LWLF*.

Le chapitre suivant offre une représentation de l'ordonnancement CM d'un système de tâches CM périodiques et sporadiques selon un algorithme sous forme d'automate fini.
Ensuite, la relation de simulation de tâches oisives est présentée.

Par la suite, deux comparaisons sont mises en évidence. La première tient de la complexité du problème d'accessibilité basé sur l'ordonnancement CM de tâches CM périodiques ou sporadiques selon un algorithme, avec et sans antichaîne.
La seconde se concentre sur l'ordonnançabilité des différents algorithmes d'ordonnancement présenté plus tôt dans le mémoire.

Enfin, les conclusions du travail discutent des résultats et offrent des propositions de travaux ultérieurs.

Remerciements

Je tiens à remercier l'Université Libre de Bruxelles, mon Alma Mater, pour m'avoir offert l'accès à un enseignement de qualité durant ces cinq dernières années.

Je tiens à remercier les promoteurs de ce mémoire pour m'avoir proposé le sujet de ce dernier, pour m'avoir guidé durant mon travail et pour leur convivialité.

En particulier, je tiens à remercier Gilles Geeraerts pour le cours qu'il m'a enseigné en premier bachelier, *Fonctionnement des ordinateurs*. La qualité de son enseignement a permis de confirmer ma passion pour les sciences informatiques.

Je tiens à remercier Joël Goossens, pour la précision de son cours *Operating systems II* avec lequel j'ai fait la rencontre de l'ordonnancement temps réel, notion majeure dans ce mémoire.

Je tiens à remercier Emmanuel Filiot, membre du jury pour ce mémoire. Il a été choisi pour l'adéquation de ses travaux au sujet et, car c'est avec son cours *Informatique fondamentale* que j'ai fait la connaissance des méthodes formelles.

Je tiens à remercier Janine Hamel pour la relecture attentive qu'elle a faite de ce mémoire. Ses qualités en matière de syntaxe, d'orthographe, de grammaire, de ponctuation et de style m'ont grandement aidé pour la réalisation de ce travail.

Je tiens à remercier le Cercle Informatique, association étudiante aux multiples facettes. Véritable incubateur de personnalité, il m'a permis de m'épanouir aussi bien socialement qu'intellectuellement durant mon cursus et de développer mes qualités en gestion de groupes ainsi que de projets.

Je tiens à remercier Alice Picard, ma sœur, pour son soutien constant et inébranlable. Ses valeurs et son caractère font d'elle le féal idéal.

Je tiens à remercier mes parents pour l'amour et le temps qu'ils m'ont données.

En particulier, je tiens à remercier Isabelle Primo, ma mère, pour la stabilité qu'elle m'a offerte. J'ai toujours pu compter sur elle et je ne doute pas que ça continuera. Sa

gentillesse a initié l'attention que j'ai pour le bien-être des autres.

Je tiens à remercier Éric Picard, mon père, pour les valeurs qu'il m'a transmises. Les combats qu'il mène avec brio ne cessent de m'inspirer. Sa contribution à la création de l'esprit autonome et débrouillard dont je bénéficie est non négligeable.

Je tiens à remercier Nadine Picard, Mamette, ma grand-mère, pour l'attention qu'elle n'a cessé de me porter. Sa joie de vivre à toute épreuve est pour moi un idéal auquel j'aspire.

Table des matières

Chapitre 1

Introduction

Ce mémoire a été réalisé dans le cadre du second cycle d'études à l'Université Libre de Bruxelles en sciences informatiques. Ce travail est réalisé sous la direction des professeurs Gilles Geeraerts et Joël Goossens.

Les appareils informatiques que nous utilisons tous les jours sont pensés de telle sorte qu'ils supportent une multitude de fonctionnalités. Cette cohabitation de différents services se retrouve dans les *smartphones*, sur les pages internet, sur un réveil matin, etc. Faire fonctionner plusieurs services en même temps n'est pas facile, car ils doivent se partager le processeur dont l'appareil sur lequel ils fonctionnent dispose. Il s'agit donc de donner à chacune de ces fonctionnalités du temps durant lequel elles pourront profiter du processeur, il faut *ordonnancer* les fonctionnalités.

En plus de ces engins qui facilitent la vie, il y a ceux qui nous sauvent la vie, par exemple l'ABS d'une voiture, les radars des avions, les *pacemakers*.
Pour cette seconde classe d'engins, il faut s'assurer que le comportement de leurs fonctionnalités soit correct. Il faut être sûr que leur exécution combinée est possible et assez rapide. Les programmes de ces appareils *temps réel* doivent respecter des contraintes de temps. En effet, un ABS doit s'activer avec un délai minimal, sinon il ne servirait à rien. L'ordonnancement temps réel a pour but d'agencer les différentes fonctionnalités d'un appareil temps réel, de sorte qu'elles respectent leur contrainte temporelle.
Différents algorithmes existent pour créer de tels agencements entre les fonctionnalités d'un appareil.

Aujourd'hui, de plus en plus d'appareils combinent des fonctionnalités *critiques*, qui ne peuvent en aucun cas être défaillantes, et des services de commodité. Par exemple, un avion devra toujours s'assurer que son système de guidage fonctionne, alors que les télévisions des passagers pourraient être défaillantes, sans pour autant compromettre la sûreté de l'appareil volant.

C'est de ce constat qu'est né l'ordonnancement en *criticité mixte*, un ordonnancement dans lequel toutes les fonctionnalités ne seraient pas égales. En effet, certaines seraient plus critiques que d'autres, et devraient toujours respecter leur contrainte temporelle, alors que ce n'est pas le cas pour fonctionnalités moins critiques.

L'ordonnancement en criticité mixte prend de plus en plus d'ampleur. Cet ordonnancement est plus compliqué que l'ordonnancement temps réel classique (en monoprocesseur).
À l'heure actuelle, à ma connaissance, il n'existe aucun moyen de déterminer avec certitude si, pour tout appareil avec des fonctionnalités de différentes criticités, il est ordonnançable sur monoprocesseur selon un certain algorithme, pour tout algorithme. En s'aidant des *méthodes formelles*, ce mémoire répondra à cette question.

Chapitre 2

État de l'art

Ce chapitre présente l'état de l'art ayant pour sujet deux problèmes différents de deux domaines différents de l'informatique.
Premièrement, le chapitre se concentre sur l'ordonnancement à criticité mixte. Pour se faire, il faut commencer par introduire l'ordonnancement classique, puis une explication informelle de ce type d'ordonnancement est donnée, pour enfin terminer par le formalisme de celui-ci.
Dans un second temps, le problème de l'accessibilité est présenté, en commençant par fournir certaines notions de base, ensuite en exposant un formalisme pour finir par expliquer la notion d'antichaîne. Par la suite, l'interaction entre ces deux problèmes est mise au clair.
Enfin, les contributions qui seront faites dans ce mémoire et sa structure seront détaillées.

2.1 Ordonnancement de systèmes à criticité mixte

2.1.1 Notions élémentaires

L'un des domaines majeurs de l'informatique théorique actuelle est celui de l'ordonnancement. Il s'agit d'organiser dans le temps la réalisation de différents travaux, compte tenu de contraintes temporelles. Pour s'exécuter, ces travaux ont besoin d'avoir accès à une ou plusieurs ressources partagées entre eux. En pratique, une fois qu'un travail est généré, il possède un temps d'exécution et une échéance. Dans la plupart des cas, on parle de système comprenant une série de travaux, le but est alors d'agencer ces différents travaux de sorte qu'ils ne ratent pas leurs échéances, il faut partitionner la ou les ressources partagées entre les travaux. L'objet de cette sous-section est de définir un certain modèle pour ce problème ainsi que d'introduire différentes notions de la théorie de l'ordonnancement.

Modèle

Pour permettre l'ordonnancement d'un système, un certain formalisme a été défini, cette partie le présente.

Commençons par une notion fondamentale, le travail. Il s'agit d'un ensemble d'opérations à effectuer nécessitant de prendre possession de la ou d'une des ressources partagées.

Définition 2.1 (Travail [1])**.**
Un travail est représenté par un triplet :

$$J_i \stackrel{def}{=} (r_i, d_i, c_i)$$

— $r_i \in \mathbb{N}^+$ est l'instant auquel J_i est généré.

— $d_i \in \mathbb{N}^+$ est l'échéance absolue de J_i, c'est-à-dire le moment avant lequel le travail doit impérativement avoir été exécuté complètement. On assume que $d_i > r_i$

— $c_i \in \mathbb{N}^+$ définit la durée d'exécution du travail.

Le travail J_i doit donc recevoir c_i unités de processeur durant l'intervalle $[r_i, d_i)$. $[a, b)$ est l'intervalle entre a et b, a est inclus et b ne l'est pas.
En général, on parle d'instance I, il s'agit d'un ensemble de travaux : $I = \{J_1, J_2, ...\}$

Souvent, le programme à exécuter possède une partie d'opération récurrente, en particulier on considère les tâches périodiques et sporadiques.

Définition 2.2 (Tâche périodique [2])**.**
Une tâche périodique génère donc un nombre infini de travaux, elle est représentée par un quadruple :

$$\tau_i \stackrel{def}{=} (O_i, T_i, D_i, C_i)$$

— $O_i \in \mathbb{N}^+$ est le décalage de la tâche, l'instant auquel le premier travail de la tâche est généré.

— $C_i \in \mathbb{N}^+$ définit la durée d'exécution d'un travail généré par la tâche.

— $D_i \in \mathbb{N}^+$ est l'échéance relative de la tâche, c'est donc le laps de temps entre la génération d'un travail et son échéance absolue.

— $T_i \in \mathbb{N}^+$ est la périodicité de la tâche, c'est-à-dire l'intervalle entre deux générations de travail.

Le k^e travail $J_{i,k}$ de la tâche τ_i est donc généré à l'instant $O_i + (k-1) * T_i$ et a pour échéance $O_i + (k-1) * T_i + D_i$.

Définition 2.3 (Tâche sporadique [2])**.**
Une tâche sporadique possède le même modèle que la tâche périodique, la différence est que T_i ne représente plus la périodicité, mais le temps minimal entre deux générations de la tâche. La tâche sporadique est donc imprévisible.

On définit ensuite une série caractéristique [2] qu'une tâche peut avoir :

— Une tâche à échéance implicite est une tâche où l'échéance correspond à la période, $D_i = T_i \; \forall i$

— Une tâche à échéance contrainte est une tâche où l'échéance est inférieure ou égale à la période, $D_i \leq T_i \; \forall i$

— Une tâche à échéance arbitraire est une tâche où il n'y a pas de contraintes en la période et l'échéance de celle-ci.

On définit ensuite, sur un système de plusieurs tâches $\tau = \{\tau_1, \tau_2, ...\}$

— Un système de tâches est synchrone si toutes ses tâches ont un décalage nul, $O_i = 0 \; \forall i$

— Asynchrone sinon.

Ordonnancement

Une stratégie d'ordonnancement est un algorithme permettant le partage d'une ou plusieurs ressources entre plusieurs parties dont elles font la requête de manière simultanée et asynchrone [1]. Les ordonnanceurs sont utilisés, par exemple, dans un système d'exploitation, dans les disques durs, dans les routeurs internet ... Le but d'un ordonnanceur classique est d'éviter le gaspillage de ressources et de partager cette dernière de manière équitable entre ceux qui le demandent. Dans un ordonnanceur temps réel, une contrainte de temps doit être respectée, il faut que les processus se terminent avant leurs échéances, ce type d'ordonnanceur est largement utilisé dans les systèmes embarqués, par exemple dans un cardiostimulateur.

À l'heure actuelle, les constructeurs de systèmes embarqués ont tendance à implémenter plusieurs fonctionnalités sur une même plateforme pour réduire les coûts, la chaleur, l'alimentation... Malheureusement, un tel partage de ressources peut mener à des interférences entre les différents clients (entité voulant avoir accès aux ressources), on cherche donc à savoir si un tel ensemble de clients peuvent partager une ressource sans qu'il y ait de moments où un client n'a pas accès à cette ressource alors qu'il en a besoin.

Typiquement, on parle d'ordonnancement d'un processeur, on divise donc le temps d'exécution disponible grâce aux coups d'horloge, définissant le temps entre deux coups d'horloge successifs comme une unité de temps. L'ordonnanceur peut être préemptif, c'est-à-dire qu'il peut interrompre un travail pour en ordonnancer un autre.

Dans ce document, nous nous concentrerons sur les algorithmes d'ordonnancement préemptif sur un monoprocesseur à vitesse unitaire.

L'algorithme d'ordonnancement devra choisir, parmi plusieurs travaux, lequel pourra jouir de la puissance du processeur. Pour ce faire, il utilisera un système de priorité clair et défini, chaque travail possédera une priorité et, en fonction de laquelle, l'ordonnanceur pourra partager le temps de calcul entre les travaux. Il existe trois classes d'algorithme d'ordonnancement [1] :

— Priorité fixe au niveau des tâches (Fixed Task Priority, FTP) : il s'agit d'un ordonnancement où chaque tâche s'est vue attribuer une priorité avant l'exécution du système. Chaque travail hérite ensuite de la priorité de la tâche qui l'a générée.
— Priorité fixe au niveau des travaux (Fixed Job Priority, FJP) : chaque travail reçoit un niveau de priorité lorsqu'il est généré, celui-ci restera constant pour toute la durée du travail. Dès lors, différents travaux d'une même tâche peuvent avoir une priorité différente.
— Priorité dynamique (Unrestructed Dynamic Priority, DP) : il s'agit du cas général où aucune contrainte ne s'applique à l'assignation des priorités aux travaux. En règle générale, cette nouvelle ligne conduite signifie qu'un travail va changer de priorité durant son existence.

On remarque que ces différentes classifications s'incluent elles-mêmes. En effet, FTP fait partie de la classe FJP qui est elle-même dans la classe DP.

L'ordonnancement est donc avant tout une certaine assignation de priorité. De manière générale, on considère que la valeur de la priorité attribuée à un travail est un naturel et est inversement proportionnel à sa priorité. Si deux travaux ont une même priorité, il faudra choisir de manière déterminée lequel sera ordonnancé. On définit donc la relation de priorité par le symbole $\succ$, par exemple : $\tau_i \succ \tau_j$, ici τ_i a une priorité supérieure à celle de τ_j.

Définition 2.4 (Fonction de priorité [3])**.**
Formellement, une fonction de priorité π assigne à toute tâche τ_i d'un système de n tâches τ (respectivement à tout travail J_i d'une instance de m travaux I) un nombre entier $\pi(\tau_i)$ (respectivement $\pi(J_i)$) compris entre 1 et n (respectivement m) qui représente de manière inversement proportionnelle la priorité de la tâche (respectivement du travail).

$$\pi : \tau \to \{0, ..., n\}$$
$$\text{(Respectivement } \pi : I \to \{0, ..., m\}\text{)}$$
$$\pi(\tau_i) < \pi(\tau_j) \Leftrightarrow \tau_i \succ \tau_j$$
$$\text{(Respectivement } \pi(J_i) < \pi(J_j) \Leftrightarrow J_i \succ J_j\text{)}$$

En temps réel, l'ordonnancement se porte en général sur un système de tâches. On souhaite prouver si un tel ensemble remplit les conditions introduites par ce mode. Deux nouvelles caractéristiques en émergent [1] :

— La faisabilité d'un système de tâches est vérifiée s'il existe un certain partage du temps de calcul entre les différents travaux, de sorte que chacun d'entre eux ait pu exécuter toutes leurs actions sans rater leur échéance.

— Un système de tâches est ordonnançable avec un certain algorithme d'ordonnancement si celui-ci permet d'agencer les travaux de sorte que chacun d'entre eux ait pu exécuter toutes leurs actions sans rater leur échéance.

Une définition analogue peut être faite pour une collection de travaux.

Il est à noter qu'il est possible dans certains cas qu'un système de tâches soit faisable, mais pas ordonnançable. Ces deux notions sont souvent liées à celle de l'utilisation qui représente la charge de travail du processeur : la quantité de travail effectué par rapport au temps dont il dispose pour le faire.

Définition 2.5 (Utilisation [1])**.**
L'utilisation d'une tâche est définie par la fonction $U(\tau_i) \stackrel{def}{=} C_i/T_i$.
L'utilisation d'un système de tâches est la somme des utilisations des tâches le composant : $U(\tau) \stackrel{def}{=} \sum_{\tau_i \in \tau} (\tau_i)$

Pour clôturer cette sous-section, un algorithme d'ordonnancement monoprocesseur est présenté. Il s'agit d'un algorithme à priorité statique au niveau des travaux (FJP) : EDF (Earliest Deadline First). Celui-ci attribue à chacun des travaux une priorité relative à leurs échéances absolues, plus elles sont proches, plus le travail est prioritaire. Formellement, on obtient $d_i < d_j \Leftrightarrow J_i \succ J_j$ et en cas d'égalité, $d_i = d_j \wedge i < j \Leftrightarrow J_i \succ J_j$.

Théorème 2.1 ([1])**.**
Si une collection de travaux est faisable, alors elle est ordonnançable avec EDF.

EDF est donc un algorithme optimal pour un système de tâches à échéance implicite, synchrone ou asynchrone. Un algorithme est optimal si l'agencement qu'il génère permet d'ordonnancer tout système de tâches d'un certain type.

Théorème 2.2 ([4])**.**
Pour un système de tâche périodique à échéance implicite τ, celui-ci est faisable si et seulement si $U(\tau) \leq 1$

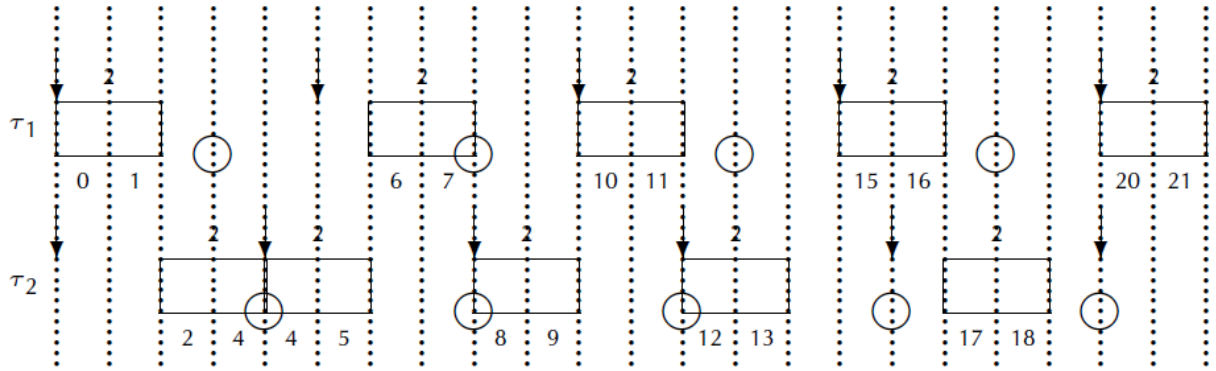

FIGURE 2.1 – Représentation graphique d'un exemple d'ordonnancement par EDF [1]

Il sera fréquent de représenter l'ordonnancement de travaux de manière graphique, la Figure 2.1 en est un exemple, ici il s'agit d'un système de deux tâches.

$$\tau_1 = (0, 5, 3, 2)$$
$$\tau_2 = (0, 4, 4, 2)$$

Sur l'image :

— Chaque colonne en pointillé représente une unité de temps.

— Une tâche est représentée sur une ligne

— La flèche vers le bas représente la période de la tâche et donc le moment où le travail est généré

— Un rond représente l'échéance d'un travail

— Un carré représente l'exécution d'un travail durant une unité de temps

2.1.2 Criticité mixte

Définition informelle

Le problème de criticité mixte est un problème d'ordonnancement. Pour rappel, il s'agit de gérer le partage d'un ensemble de ressources communes entre plusieurs clients.

Dans cette sous-section, nous nous penchons sur un problème plus précis : celui de la certification, dont le but est de prouver qu'un système est ordonnançable.

Précédemment, il a été présenté l'ordonnancement de travaux classique, ici on en déduit un nouveau problème où les clients pourraient avoir une importance plus ou moins élevée en fonction de leur criticité dans leur système. Il est évident que, dans une voiture, le système de freins ABS («Anti-lock Brake System») est plus important, plus critique que l'autoradio, car, en cas de dysfonctionnement, il y a danger pour le conducteur, et donc l'ABS devrait être prioritaire sur l'autoradio, c'est-à-dire que si ces deux clients souhaitent prendre possession du processeur, mais que seul l'un des deux le peut, l'ABS sera privilégié. Ce problème d'ordonnancement avec différentes priorités est très actuel. En effet, c'est un domaine de recherche croissant dans le monde de l'ordonnancement temps réel et des systèmes embarqués, il fait l'objet d'un programme au sein du laboratoire «US Air Force Laboratory», le Mixed-Criticality Architecture Requirements (MCAR) [5].
Dans le milieu aérospatial, le problème a été standardisé par l'organisme RTCA, qui classifie les travaux en fonction de la gravité qu'engendrerait un dépassement de leur échéance, il s'agit du standard DO-178C imprimé en Figure 2.2.

L'objectif de l'ordonnancement serait alors de permettre aux tâches les plus critiques de ne pas souffrir de perte de performance en cas de diminution des ressources, cette facette du problème a permis d'introduire la notion de criticité, mais ne sera pas traité dans ce mémoire.

Il existe un second aspect de l'ordonnancement à criticité mixte où un système devrait être garanti fonctionnel à plusieurs niveaux [7]. Pour illustrer ce concept, nous utiliserons l'exemple des drones, UAV (unmmaned aerial vehicle), de reconnaissance. Nous distinguons deux niveaux de criticité dans ce type de système

— Criticité au vol : ce sont les actions exécutées par le drone pour voler en tant que tel, calculer un chemin pour éviter les obstacles, gérer la puissance des réacteurs, l'inclinaison des ailes ...
— Criticité à la mission : ce sont toutes les actions qui fonctionnent dans le but de mener à bien la mission du drone. Dans notre cas, elles pourraient être le fait de

Niveau	Gravité	Conséquences
A	Catastrophique	Un échec peut causer multiple décès, habituellement avec la perte de l'objet volant
B	Hasardeuse	Un échec a un grand impact négatif sur la sûreté ou la performance, ou réduit la capacité de l'équipage à gérer l'objet volant à cause de détresse physique ou d'une charge de travail plus importante ou bien engendre de sérieuses blessures parmi les passagers.
C	Majeure	Un échec réduit significativement la marge de sûreté ou augmente significativement le travail de l'équipage. Par exemple, il pourrait engendrer un inconfort pour un passager (voire même une blessure).
D	Mineure	Un échec réduit légèrement la marge de sûreté ou augmente légèrement le travail de l'équipage. Par exemple, il pourrait engendrer un désagrément pour un passager dont le personnel devrait s'occuper.
E	Sans effet	Un échec n'a pas d'impact sur la sûreté, les opérations d'aviation ou la répartition du travail du personnel

FIGURE 2.2 – Standard RTCA DO-178C [6]

filmer avec une caméra, de transmettre ces images ou d'autres informations, telles que la température extérieure, de faire de la reconnaissance sur ces images, etc.

Il est évident que le niveau critique de vol est plus important que celui de la mission. En effet, si une des actions en rapport avec le vol du drone dysfonctionnait, cette défaillance pourrait mener à un atterrissage forcé qui entraînerait des dégâts matériels, voire même pire, un réel danger pour un être humain.

Pour introduire les contraintes sur les fonctionnalités, nous introduisons la notion de WCET (Worst-Case Execution Time), qui représente le temps d'exécution dont aura besoin une action pour s'exécuter complètement au pire des cas. Ces estimations sont très importantes, puisque tout le système d'ordonnancement repose sur elles. En effet, si un travail prend plus de temps à s'exécuter que l'estimation, le système complet peut être compromis. L'estimation du WCET doit être [3] :

— sûre. La validité d'une certification repose sur cette valeur, elle doit donc former une borne supérieure la plus juste possible. Il est exclu que celle-ci soit vue à la baisse, elle ne peut être optimiste.

— la moins pessimiste possible. Plus le WCET est pessimiste, plus il induira des instants oisifs durant l'ordonnancement puisqu'en pratique, le temps d'exécution sera fréquemment inférieur à celui-ci. L'estimation ne doit pas être surestimée, sous peine d'engendrer un faux négatif lors d'un test d'ordonnançabilité.

Le calcul d'un WCET sûr et non pessimiste est un exercice difficile et n'est pas à prendre à la légère. Il s'agit d'un domaine à part entière des sciences informatiques, actuellement en plein essor qui fait l'objet de divers concours.

Comme en témoigne le standard DO-178C en Figure 2.2, les conséquences engendrées par un travail qui raterait une échéance varient. Un lien de causalité entre la criticité d'un travail et le pessimisme de l'estimation de son WCET a été mis en évidence [8]. Les estimations du temps d'exécution sont généralement créées grâce à des simulations. Pour les moins critiques, des simulations classiques suffisent, mais plus la fonctionnalité est critique, plus on teste de cas, ce qui amène à simuler ces travaux dans des cas improbables, tels que celui où la mémoire vive du système serait remplie. Les tâches les plus critiques devront pouvoir être ordonnancées malgré une estimation de leur WCET très pessimiste. Le problème est que, comme énoncé ci-dessus, plus une estimation est pessimiste, plus il y aura de gaspillage de ressources. C'est de ce constat que vient l'idée de l'ordonnancement à criticité mixte. En effet, on veut pouvoir garantir que dans les cas d'exécutions classiques, les tâches critiques et moins critiques pourront être ordonnancées, mais dans des cas extrêmes, on souhaite toujours garantir que les tâches hautement critiques restent ordonnancées correctement.

Dans l'exemple du drone, si celui-ci souhaite voler dans l'espace aérien, il est impératif que son logiciel satisfasse les critères d'une autorité de certification (AC) civile qui gère ledit espace aérien, telles que l' «European Aviation Safety Agency» (EASA) en Europe ou bien la «Federal Aviation Authority» (FAA) aux États-Unis. Ces AC se doivent d'être extrêmement prudentes pour éviter tout accident. Dès lors, elles imposent que la certification d'un système soit faite avec des assomptions sur le temps d'exécution des fonctionnalités critiques très pessimistes. En revanche, les AC ne sont pas du tout concernées par les tâches critiques à la mission, leur seule préoccupation est la sûreté du réseau aérien.
Les temps d'exécution des fonctionnalités critiques à la mission sont quant à eux estimés par le client et l'industriel ; ces derniers estiment aussi le WCET des tâches critiques au vol, mais avec des standards moins rigoureux que ceux des AC civiles.

Concrètement, les AC imposeront donc aux tâches critiques au vol des estimations dans des cas extrêmes (il existe d'autres techniques d'estimation que la simulation, comme l'obligation de coder ces tâches critiques en un langage spécifique permettant l'analyse du WCET), tandis que le fabricant simulera l'exécution dans des conditions simples en y ajoutant une valeur de sûreté. Ce résultat sera probablement lui aussi pessimiste, mais moins que celui de l'AC.

Toutes les actions se seront vues attribuer un WCET, mais certaines en auront reçu plusieurs : une au niveau critique de mission et une au niveau critique de vol. Le problème de certification doit donc être valide à plusieurs niveaux. Cependant, toutes les actions n'ont pas besoin d'être certifiées par toutes les instances de vérification. Si c'était le cas, il suffirait de faire un test en sélectionnant les contraintes les plus larges, c'est-à-dire les plus grands WCET, pour chaque action. Seules certaines actions sont soumises aux contraintes de certaines instances de vérification, comme c'est le cas des calculs liés au vol du drone dans notre exemple. D'où, il s'agit de s'assurer que les actions critiques au vol passent le test de l'AC qui gère l'espace aérien, tandis que les actions de l'ensemble du système, donc des deux niveaux critiques, puissent cohabiter selon les WCET estimés par le fabricant.

Reprenons l'exemple du drone de reconnaissance. Supposons qu'il ne possède que deux travaux, J_1 : la manipulation de l'hélice (criticité au vol) et J_2 : la photographie (criticité à la mission). Ces deux travaux peuvent être ordonnancés à partir du temps t_0 et doivent tous deux être terminés au temps t_8. Le WCET de la manipulation de l'hélice a été estimé à 6 alors que celui de la photographie l'a été à 4.

Ce système semble non ordonnançable, car la somme de ces WCET est plus grande que la limite d'exécution des deux actions. Cependant, il ne faut pas oublier qu'il y a deux niveaux de certification : le premier correspond à l'AC qui gère l'espace aérien, cette dernière se soucie exclusivement de la manipulation de l'hélice et le second est en relation avec la validation par le fabricant qui ne tient pas compte du WCET de la manipulation de l'hélice imposée par l'autre AC.

Supposons que l'industriel ait attribué à la manipulation de l'hélice un WCET de 2. Il est alors possible de passer les deux certifications : pour la criticité au vol, seule la manipulation de l'hélice doit être ordonnancée, l'action est donc exécutée du temps t_0 au temps t_6 et pour la criticité à la mission, la manipulation de l'hélice est exécutée du temps t_0 au temps t_2 et la photographie est exécutée du temps t_2 au temps t_6. Dans les deux scénarios, toutes les actions ont pu être exécutées avant leur échéance.

On remarque qu'utiliser un ordonnancement à fonction de priorité basée sur le niveau de criticité aurait fonctionné ici aussi. Cependant il est aisé de construire un autre système où un tel algorithme d'ordonnancement échoue, alors que le système est ordonnançable.

Exemple 2.1.
Considérons une instance avec deux travaux qui démarrent au même moment, le premier, J_1 a une criticité de 1 , une échéance au temps t_3 et un WCET estimé à 2. L'autre travail, J_2 a une criticité double, une échéance au temps t_8 et un temps d'exécution estimé à 3 et 5.

Si l'ordonnancement se fait par priorité en fonction de la criticité des travaux, il est évident qu'en criticité de niveau 1, le travail J_1 manquera son échéance puisque le travail J_2 qui a une plus grande priorité, due à sa criticité plus élevée, sera exécuté du temps t_0 au temps t_3, alors que l'échéance de J_1 est justement t_3.
Le système est pourtant ordonnançable : en effet, la stratégie serait d'exécuter J_1 puis J_2, elle est correcte pour les deux niveaux de criticité.

La difficulté de cet ordonnancement est qu'il est évalué avant de savoir s'il se trouve dans le cas où il faudrait considérer uniquement les actions de haute criticité et leurs grands WCET. En effet, c'est lors de l'exécution que l'ordonnanceur s'en rendra compte. Les actions signalent à l'ordonnanceur lorsqu'elles se sont complètement réalisées. C'est donc au moment où une tâche d'un niveau critique supérieur prend plus de temps qu'elle le devrait en l'état actuel que le système passe au niveau critique supérieur et ne se soucie plus des actions moins critiques. Le niveau de criticité représente le niveau de pessimisme avec lequel il faut ordonnancer le système.

Motivations supplémentaires

En plus de pouvoir certifier qu'un programme peut fonctionner selon plusieurs WCETs, l'ordonnancement en criticité mixte peut être utilisé pour résoudre d'autres problèmes. Par exemple, Wongsen *et al.* proposent de vérifier qu'un système embarqué puisse être alimenté par batterie, partiellement rechargée en [9] en utilisant l'ordonnancement en criticité mixte. Typiquement, ce genre de systèmes embarqués seront des satellites rechargés à l'énergie solaire. La difficulté de cette question vient du fait de la non-constance de la demande d'énergie du satellite et de la non-constance de la recharge de la batterie.

Modèle

Cette partie définit une série de notions permettant de formaliser le problème de l'ordonnancement à criticité mixte, ces notions seront utilisées dans la suite de ce document.

Définition 2.6 (Travail CM [10])**.**
Un travail dans un système à criticité mixte, un travail CM, de niveau K est caractérisé par un quadruple :

$$J_i \stackrel{def}{=} (r_i, d_i, \chi_i, c_i)$$

— $r_i \in \mathbb{N}^+$ est l'instant auquel J_i est généré.

— $d_i \in \mathbb{N}^+$ est l'échéance absolue de J_i, c'est-à-dire le moment avant lequel le travail CM doit avoir été exécuté complètement. On assume que $d_i > r_i$

— $\chi_i \in \{0, ..., K\}$ est le niveau de criticité du travail CM J_i, plus il est élevé, plus critique est le travail CM.

— $c_i : \{0, ..., K\} \to \mathbb{N}^+$ définit le WCET du travail CM pour chacun des niveaux de criticité du système.

On suppose que $c_i(1) \leq c_i(2) \leq ... \leq c_i(K)$ et que $c_i(m) = c_i(\chi_i)$ pour tout $\chi_i < m \leq K$. De manière générale, la fonction du temps d'exécution d'un travail CM sera souvent représentée comme un vecteur $c_i = [c_i(1), ..., c_i(K)]$

Définition 2.7 (Instance CM [11])**.**
Une instance en criticité mixte, appelée instance CM, est définie comme étant une collection de travaux CM $I \stackrel{def}{=} (J_1, J_2, ...J_n)$. En possession d'une telle instance, nous souhaitons déterminer si oui ou non il existe une ligne de conduite permettant de l'ordonnancer correctement.

Chaque travail CM J_i dans une instance CM $I = (J_1, J_2, ...J_n)$ doit recevoir $\bar{c}_i$ unités de temps d'exécution où $\bar{c}_i$ n'est pas connu à l'avance et se dévoile lorsque J_i signale la fin de son exécution. La collection des temps d'exécution $\bar{c} \stackrel{def}{=} (\bar{c}_1, \bar{c}_2, ..., \bar{c}_n)$ est appelée un scénario.
Le niveau de criticité du scénario $\bar{c} = (\bar{c}_1, \bar{c}_2, ..., \bar{c}_n)$ est défini comme étant le plus petit entier ℓ satisfaisant $\bar{c}_i \leq c_i(\ell)$ pour tout travail CM J_i (s'il n'existe pas de tel ℓ, le scénario est dit erroné).

Dans ce modèle, un scénario de criticité ℓ doit uniquement s'assurer que les travaux CM de criticité ℓ ou supérieurs soient réalisés avant leur échéance. En d'autres termes, une fois la criticité ℓ atteinte, les travaux CM de criticité $\ell - 1$ et inférieurs sont abandonnés.

Définition 2.8 (Tâche CM [10])**.**
Nous considérons ici les tâches dans un système à criticité mixte, tâches CM, de niveau K. Elles sont définies par un tuple :

$$\tau_i \stackrel{def}{=} (O_i, T_i, D_i, C_i, \chi_i)$$

— $O_i \in \mathbb{N}^+$ est le décalage de la tâche CM, l'instant auquel le premier travail de la tâche est généré (au plus tôt pour tâche CM sporadique, directement pour tâche CM périodique).

— $T_i \in \mathbb{N}^+$ est la période de la tâche CM, ou l'intervalle minimal entre deux générations de travaux CM ; selon que la tâche CM soit périodique ou sporadique respectivement.

— $D_i \in \mathbb{N}^+$ est l'échéance relative de la tâche CM, c'est donc le laps de temps entre la génération d'un travail CM et son échéance absolue.

— $\chi_i \in \{0, ..., K\}$ est le niveau de criticité de la tâche CM τ_i, plus il est élevé, plus critique est la tâche CM.

— $C_i : \{0, ..., K\} \to \mathbb{N}^+$ définit le WCET des travaux CM générés pour chacun des niveaux de criticité du système.

On suppose à nouveau que $C_i(1) \leq C_i(2) \leq ... \leq C_i(K)$. La tâche CM τ_i génère un nombre potentiellement infini de travaux CM ayant une criticité et une fonction de temps d'exécution héritée de τ_i, l'échéance aura lieu à la fin de la période T_i en cours.

La fonction du temps d'exécution sera elle aussi souvent représentée sous la forme d'un vecteur $C_i = [C_i(1), ..., C_i(K)]$

Lorsque la criticité K d'un système de tâches CM ou d'une instance CM est double, on aura tendance à utiliser les abréviations LO et HI, pour les niveaux de criticité 1 et 2 respectivement.

Définition 2.9 (Algorithme d'ordonnancement [11])**.**
Un algorithme d'ordonnancement pour une instance CM I (qui peut être générée par un système de tâches CM) définit à tout instant, et pour tout niveau de criticité du scénario résultant, le travail CM à ordonnancer.

Un algorithme est dit clairvoyant s'il connaît à l'avance quel sera le temps d'exécution $\bar{c}_i$ de tout travail CM $J_i \in I$. À l'inverse un algorithme en ligne ne connaît pas ce temps d'exécution, c'est-à-dire qu'il ne possède pas toutes les informations *a priori*, il

doit se contenter d'une partie de celles-ci et ne découvre ce temps d'exécution réel $\bar{c}_i$ seulement lorsque le travail CM J_i signale sa complétion. La criticité du scénario n'est donc révélée qu'une fois celui-ci terminé.

Définition 2.10 (Ordonnançabilité CM [10])**.**
Un algorithme ordonnance une instance CM $I = (J_1, J_2, ...J_n)$ correctement s'il est en mesure, pour toute criticité ℓ, de garantir que tout travail CM J_i de l'instance CM I ayant une criticité $\chi_i \geq \ell$ pourrait être exécuté jusqu'au signalement de sa complétion, entre sa génération et son échéance, soit l'intervalle $[r_i, d_i)$.
Une instance CM est appelée ordonnançable CM si elle admet un algorithme en ligne qui permet de l'ordonnancer.

L'instance CM I peut résulter d'une génération de travaux CM par un système de tâches CM τ. On dit alors qu'un système de tâches CM est ordonnançable CM, si l'instance CM qu'il produit admet un algorithme en ligne qui permet de l'ordonnancer.

Définition 2.11 (Faisabilité CM)**.**
Une instance CM $I = (J_1, J_2, ...J_n)$ est faisable s'il existe un agencement correct de ses travaux CM permettant, pour toute criticité ℓ, de faire en sorte que tout travail CM J_i de l'instance CM I ayant une criticité $\chi_i \geq \ell$ soit exécuté jusqu'au signalement de sa complétion, entre sa génération et son échéance, soit l'intervalle $[r_i, d_i)$.
Comme pour l'ordonnancement CM, si l'instance CM I générée par le système de tâches CM τ est faisable CM, on dit que ce système de tâches CM est faisable CM.

Définition 2.12 (Utilisation en criticité mixte [10])**.**
L'utilisation d'une tâche CM τ_i au niveau de criticité ℓ est :

$$U_{\tau_i}(\ell) \stackrel{def}{=} C_i(\ell)/T_i$$

Soit $L_k \stackrel{def}{=} \{i \in \{1, ..., n\} \mid \chi_i = k\}$, l'utilisation des tâches CM de criticité i d'un système de n tâche CM τ au niveau de criticité ℓ est :

$$U_{\tau,i}(\ell) \stackrel{def}{=} \sum_{j \in L_i} U_{\tau_j}(\ell) \qquad i = 1, ..., K;\ \ell = 1, ..., i$$

Définition 2.13 (Utilisation d'un système de tâches CM)**.**
L'utilisation d'un système de tâches CM τ au niveau de criticité ℓ est :

$$U_\tau(\ell) \stackrel{def}{=} \sum_{i=\ell}^{K} U_{\tau,i}(\ell)$$

On définit aussi l'utilisation moyenne :

$$U_\tau^* \stackrel{def}{=} \frac{\sum_{\ell=1}^{K} U_\tau(\ell)}{K}$$

On sait que s'il n'y a qu'un seul niveau de criticité, alors, un système de tâches CM périodiques à échéance implicite τ est faisable sur un processeur si, et seulement si, $U_{\tau,1}(1) \leq 1$ [4]. On en déduit la condition pour la faisabilité CM :

Théorème 2.3 ([10])**.**
Une condition nécessaire pour qu'un système de tâche CM τ soit faisable CM sur un processeur à vitesse unitaire est que, si pour chaque $\ell = 1, ..., K$:

$$U_\tau(\ell) \leq 1$$

Par exemple, si K = 2, il faut que :

$$U_{\tau,1}(1) + U_{\tau,2}(1) \leq 1 \textit{ et}$$
$$U_{\tau,2}(2) \leq 1$$

Théorème 2.4 ([10])**.**
Un système de tâche CM périodique à échéance implicite à criticité unitaire est ordonnançable CM avec l'algorithme EDF sur un monoprocesseur à vitesse unitaire si, et seulement si, $U_{\tau,1}(1) \leq 1$.

Exemple 2.2 ([11])**.**
Considérant une instance CM I contenant quatre travaux CM :

- $J_1 = (0, 3, 2, [1, 2])$
- $J_2 = (0, 3, 1, [2, 2])$
- $J_3 = (0, 5, 2, [1, 1])$
- $J_4 = (3, 5, 2, [1, 2])$

Seul le travail CM J_2 a une criticité de 1, les autres ont une criticité de 2. La fonction du temps d'exécution est représentée explicitement par un vecteur $[c_i(1), c_i(2)]$.
Pour cette instance, tout scénario où les temps réels d'exécution $\bar{c}_1, \bar{c}_2, \bar{c}_3, \bar{c}_4$ inférieures ou égaux à $1, 2, 1, 1$ aura une criticité de 1, tout autre scénario où ces temps $\bar{c}_1, \bar{c}_2, \bar{c}_3, \bar{c}_4$ ne sont pas supérieurs à $2, 2, 1, 2$ aura une criticité de 2. Les autres scénarios sont, par définition, erronés.

Analysons un algorithme d'ordonnancement possible :

— S0 : exécuter J_1 sur $[0, 1)$, si J_1 n'a pas terminé son exécution, c'est-à-dire qu'il ne s'est pas déclaré comme réalisé et donc prend plus d'un coup d'horloge, alors exécuter la stratégie S1, sinon S2.

— S1 : exécuter $J1$ sur $[1, 2)$, J_3 sur $[2, 3)$ et J_4 sur $[3, 5)$

— S2 : exécuter $J2$ sur $[1, 3)$, J_3 sur $[3, 4)$ et J_4 sur $[4, 5)$

L'algorithme d'ordonnancement ci-dessus n'est pas correct pour l'instance I. En effet, le scénario $\bar{c} = (1, 2, 1, 2)$ engendrerait une échéance ratée. Ce scénario est de criticité 2 puisque $\bar{c}_4 = c_4(2) = 2$; dès lors, un algorithme correct se doit d'ordonnancer J_1, J_3, J_4. Dans le cas présent, le travail CM J_4 manquera son échéance d'une unité de temps. Il se trouve que l'instance I n'est pas ordonnançable CM.

2.1.3 Test d'ordonnançabilité

Si un système de tâches CM est déclaré comme non ordonnançable par un certain algorithme, il peut y avoir trois raisons possibles [12] :

— Le problème vient du système de tâches CM lui-même, il n'est pas faisable, c'est-à-dire qu'il n'est pas possible de l'ordonnancer, quelle que soit la ligne de conduite utilisée.

— Le problème vient de l'algorithme d'ordonnancement, la stratégie proposée ne permet pas d'ordonnancer correctement le système même s'il est faisable.

— Le problème vient du test, il ne permet pas d'affirmer que le système de tâches CM n'est pas ordonnançable par l'algorithme d'ordonnancement.

En criticité mixte, déterminer si un système de tâches CM est faisable est possible, la condition de faisabilité d'un système de tâches CM a été donnée précédemment.
Actuellement, pour vérifier qu'un algorithme d'ordonnancement agence correctement un système de tâches en criticité mixte, il faut se baser sur une condition suffisante d'ordonnançabilité CM. Si le système de tâches CM est déclaré comme non ordonnançable pour un algorithme donné, il n'est pas encore possible de savoir si c'est à cause de l'algorithme en lui-même ou parce que les systèmes de tâches ne sont simplement pas ordonnançables CM.

Le but est de distinguer le deuxième cas du troisième, en répondant à la question que pose le dernier cas. Il s'agit donc, étant donné une instance de travaux CM, de déterminer si celle-ci est ordonnançable CM. Ce n'est pas un problème facile.

En effet, il a été prouvé qu'il s'agit d'un problème NP-dur [13], ce qui signifie que, si $NP \neq P$, il n'existe pas de solutions en temps polynomial pour résoudre le problème. La question sera étendue aux systèmes de tâches CM.

Un tel test d'ordonnancement, fiable et permettant de dire avec précision si un algorithme ordonnance correctement un système de tâches CM, serait une première pour certains algorithmes. Ces derniers pourraient alors être utilisés pour un plus grand nombre de systèmes de tâches CM et permettraient une démocratisation de ce type d'ordonnancement.

Dans ce document, nous nous concentrerons sur le problème de l'ordonnançabilité CM d'un système de tâches CM sporadiques sur un monoprocesseur à vitesse unitaire. L'idée principale est d'utiliser des techniques de la communauté des méthodes formelles pour résoudre ce problème. Plusieurs travaux ont été publiés et vont en ce sens [12] [14].

La section suivante décrit une partie du monde des méthodes formelles et définit une série d'outils qui seront utiles pour la résolution du problème.

2.2 Problème d'accessibilité

Le problème de l'accessibilité est un problème fondamental de l'informatique et de la théorie des graphes. En effet, il est applicable sur différentes structures étendues des graphes pour différents problèmes.

Un graphe est une structure permettant de représenter des réseaux au sens large. Il s'agit d'un ensemble de points dont certaines paires sont connectées par un lien. Les points sont appelés sommets. Les liens sont appelés arêtes dans les graphes non orientés, c'est-à-dire que les liens sont utilisables dans les deux sens. Ou alors arcs dans les graphes orientés, où les liens ne sont empruntables que dans un sens et donc orientés. On représente un graphe G par un ensemble de sommets V et un ensemble d'arcs ou d'arêtes E, donc $G \stackrel{def}{=} (V, E)$.

De manière générale, le problème de l'accessibilité revient à décider si, étant donné un graphe $G = (V, E)$ et deux états i et f, il est possible d'emprunter une séquence d'arcs ou d'arêtes $\in E$ successifs en partant de l'état i pour arriver à l'état f. Si tel est le cas, on dit que f est accessible depuis i.

L'accessibilité est largement utilisée, car elle fait l'objet de nombreuses réductions, c'est-à-dire qu'un problème initial est transformé en ce problème, et le résoudre revient à résoudre le problème initial. Annuellement depuis 2007, une conférence sur ce problème en général est organisée, il s'agit de l'«International Workshop on Reachability Problems», l'édition de 2016 aura (ou a eu) lieu en septembre 2016 [1].

2.2.1 Notions élémentaires

En informatique, il est fréquent que le scientifique passe plus de temps à vérifier que son programme est correct plutôt qu'à le développer. Pour pallier ce problème, les méthodes formelles ont émergé.

Il s'agit d'un des sous-domaines des sciences informatiques. Ces méthodes formelles sont des techniques basées sur l'utilisation de différents outils, principalement les mathématiques, dans le but de prouver qu'un certain programme respecte une certaine spécification. Les méthodes formelles s'appuient donc grandement sur des représentations formelles et rigoureuses de la sémantique des programmes. Ces techniques sont largement utilisées dans le domaine de la vérification de programmes qui consiste à prouver qu'un programme fait effectivement ce qu'il est censé faire. Pour de telles questions de validité, on utilise par exemple le model-checking qui fait une analyse exhaustive des différentes exécutions du programme dans le but de déceler la présence (ou de prouver l'absence) d'erreurs dans ces dernières. Il existe différents types de model-checking, une présentation du domaine a été faite par Clarke [15] et aussi par Baier [16].

— La phase de modélisation : il s'agit de la modélisation du système, ainsi que celle des propriétés qu'on souhaite vérifier, le tout selon le formalisme désiré.

— La phase d'exécution : où l'on applique un model-checker (par exemple UPPAAL ou Spin) au modèle du système pour vérifier si la spécification recherchée est présente.

— La phase d'analyse : où l'on tire des conclusions en fonction du résultat obtenu lors de la deuxième phase.

Le model-checking repose donc essentiellement sur la modélisation d'un programme. Le modèle obtenu décrira alors le comportement du système de manière précise et non ambiguë.

1. http ://rp16.cs.aau.dk/

On utilisera souvent la représentation sous forme d'un automate fini qui définit un ensemble d'états fini et un ensemble de transitions. Les états contiennent alors des informations pour les différentes variables d'un système et les transitions représentent l'évolution du système.

Définition 2.14 (Automate fini [14])**.**
Un automate fini A est un quadruple :

$$A \stackrel{def}{=} (V, E, S_0, F) \text{ où}$$

— V est l'ensemble fini d'état

— $E \subseteq V \times V$ est l'ensemble des transitions

— $S_0 \subseteq V$ est l'ensemble des états de départ

— $F \subseteq V$ est l'ensemble des états finaux

Le problème de l'accessibilité reste adéquat dans les cas des automates finis, car ils peuvent être représentés par des graphes orientés.

2.2.2 Définition et résolution

Pour poser le problème de l'accessibilité correctement, il faut encore définir ce qu'est un chemin.

Définition 2.15 (Chemin [14])**.**
Un chemin dans un automate fini $A = (V, E, S_0, F)$ est une séquence finie d'état $v_1, ..., v_t$ telle que, pour tout $1 \leq i \leq t-1 : (v_i, v_{i+1}) \in E$

La fonction $Reach(A)$ découle de cette définition : soit $V' \subseteq V$ un ensemble d'état de A, s'il existe un chemin $v_1, ...v_t$ dans A tel que $v_t \in V'$, on dit que v_1 a accès à V'. Pour un automate A, on représente l'ensemble des états accessibles depuis un état initial de A par la fonction $Reach(A)$.

Définition 2.16 (États accessibles [14])**.**
L'ensemble des états accessibles d'un automate est défini par

$$Reach(A) \stackrel{def}{=} \{v \in V \mid \exists\ un\ chemin\ v_1, ..., v_t : v_1 \in S_0 \wedge v_t = v\}$$

Dès lors, le problème de l'accessibilité dans un automate pose la question de savoir si, étant donné un automate A et l'ensemble d'états finaux F, il est possible d'accéder à un état final dans l'automate A, c'est-à-dire, de savoir si $Reach(A) \cap F = \emptyset$ ou non.

La complexité en temps, soit la quantité d'opérations nécessaires pour résoudre un problème, de la fonction $Reach$ dépend du modèle utilisé. Dans le cas d'un automate fini, le problème peut se résoudre simplement par une recherche en profondeur ou en largeur. Dès lors, il est résoluble en temps linéaire par rapport au nombre d'états et de transitions, c'est-à-dire qu'au pire des cas il faudra emprunter toutes les transitions et passer par tous les états.

Concrètement, on pourrait utiliser le problème de l'accessibilité pour vérifier qu'un programme ne peut jamais atteindre un état erroné. On créerait alors un automate fini $A = (V, E, S_0, F)$ avec V les états du programme, E les transitions du programme, S_0 comprenant l'état initial et F l'ensemble des états erronés. On voudrait donc obtenir que $Reach(A) \cap F = \emptyset$.

Recherche en largeur

Le problème d'accessibilité peut se résoudre grâce à la recherche en largeur, la suite de cette sous-sous-section présente cet algorithme pour un automate fini.

En premier lieu, il faut définir la fonction de successeurs.

Définition 2.17 (Successeurs [14])**.**
Soit $A = (V, E, S_0, F)$ un automate fini, les successeurs directs d'un état S sont

$$Succ(S) \stackrel{def}{=} \{S' \mid (S, S') \in E\}$$

Les successeurs directs d'un ensemble d'état R sont

$$Succ(R) \stackrel{def}{=} \cup_{S \in R} Succ(S)$$

Cette simple fonction permet déjà de créer l'Algorithme 2.1, offrant la possibilité de faire une recherche en largeur.

ALGORITHME 2.1: Recherche en largeur [14]

```
1  begin
2      i ← 0
3      R_0 ← S_0
4      repeat
5          i ← i + 1
6          R_i ← R_{i−1} ∪ Succ(R_{i−1})
7          if R_i ∩ F ≠ ∅ then return Reachable
8      until R_i = R_{i−1}
9      return Not reachable
```

L'algorithme commence donc avec un ensemble égal à celui des états initiaux de l'automate fini. Cet ensemble représente l'ensemble des états explorés. Par la suite, l'algorithme va visiter tous les successeurs directs des états explorés, vérifier si l'un d'entre eux fait partie des états à atteindre. Si oui, alors l'algorithme annonce que ces états sont atteignables, si non l'algorithme recommence en explorant les successeurs de ces états nouvellement explorés. L'algorithme continue à itérer, tant qu'il découvre de nouveaux états à visiter ; une fois que ce n'est plus le cas, il s'arrête et annonce qu'il n'est pas possible d'accéder à l'un des états à atteindre. En effet, si les successeurs d'un ensemble d'états font déjà tous partie de cet ensemble, alors tous les états atteignables ont été visités.

2.2.3 Antichaîne

Soit un automate A, la fonction $Reach(A)$ est linéaire en la taille de A, mais il est justement possible que celle-ci soit grande. Les automates sont souvent construits pour faire un test exhaustif, comme c'est le cas en model-checking, leurs tailles deviennent alors exponentielles en le nombre de branchements du programme à vérifier, mais, bien souvent, il y a des répétitions dans ces automates fraîchement créés.

Les antichaînes construisent le sous-ensemble des éléments incomparables deux à deux avec une certaine relation d'ordre (autorisant les cycles) d'un ensemble. Nous allons utiliser ces dernières avec une relation d'ordre spécifique pour qu'elles représentent des groupes d'éléments dans l'ensemble de départ. Cette relation d'ordre spécifique est une relation de simulation ; cette notion et celle de l'antichaîne sont définies dans cette sous-section, en commençant par plus générale, le préordre.

Définition 2.18 (Préordres et leurs propriétés [17])**.**
Un préordre sur un ensemble fini V est une relation binaire $\preceq \subseteq V \times V$ (ou $\succeq$) qui est réflexive et transitive. Si $v_1 \preceq v_2$, on dit que v_1 est plus petit que v_2 (ou que v_2 est plus grand que v_1).

Un préordre $\preceq'$ est dit plus grossier que $\preceq$ si pour tout $v_1, v_2 \in V$ si $v_1 \preceq v_2$ alors $v_1 \preceq' v_2$.

La $\preceq$-fermeture vers le haut d'un ensemble $S \subseteq V$ est l'ensemble $Up(\preceq, S) \stackrel{def}{=} \{v_1 \in V \mid \exists v_2 \in S : v_2 \preceq v_1\}$ des éléments qui sont plus grands ou égaux à un élément de S.

Un ensemble S est $\preceq$-fermé vers le haut s'il est égal à sa $\preceq$-fermeture vers le haut, $Min(\preceq, S) \stackrel{def}{=} \{s_1 \in S \mid \forall s_2 \in S : v_2 \preceq v_1 \rightarrow v_1 \preceq v_2\}$ reprend l'ensemble des éléments minimaux de S.

On remarque que $Min(\preceq, S) \subseteq S \subseteq Up(\preceq, S)$.

De la même façon, on définit la $\preceq$-fermeture vers le bas $Down(\preceq, S) \stackrel{def}{=} \{v_1 \in V \mid \exists v_2 \in S : v_1 \preceq v_2\}$ d'un ensemble S et on dit que S est $\preceq$-fermé vers le bas si $S = Down(\preceq, S)$ et $Max(\preceq, S) \stackrel{def}{=} \{s_1 \in S \mid \forall s_2 \in S : v_1 \preceq v_2 \rightarrow v_2 \preceq v_1\}$ est l'ensemble des éléments maximaux de S.

Maintenant que les préordres sont définis, nous pouvons détailler l'antichaîne se basant dessus.

Définition 2.19 (Antichaîne [17])**.**
Un ensemble $H \subseteq V$ est une quasi-antichaîne si pour tout $v_1, v_2 \in S$, soit v_1 et v_2 sont incomparables, soit $v_1 \preceq v_2$ et $v_2 \preceq v_1$.
Ce même ensemble $H \subseteq V$ est une antichaîne si pour tout $v_1, v_2 \in S$, v_1 et v_2 sont incomparables.
On obtient alors, une antichaîne avec la relation $\preceq$ est un sous-ensemble H de V tel que :

$$H \stackrel{def}{=} \{v_1 \in V | \forall v_2 \in V : v_1 \preceq v_2 \Rightarrow v_1 = v_2\}$$

Avec un préordre, les ensembles $Min(\preceq, S)$ et $Max(\preceq, S)$ sont des quasi-antichaînes et avec un ordre partiel, un préordre antisymétrique, ces ensembles sont des antichaînes.

Par abus de langage, on appelle antichaînes les ensembles d'éléments minimaux ou maximaux, même s'il ne s'agit pas d'un ordre partiel, mais d'un préordre.

L'antichaîne nous donne les éléments incomparables d'un ensemble selon un ordre partiel. L'ordre partiel qui nous intéresse est celui de la simulation, c'est celui-là qui nous permettra de représenter plusieurs états d'un automate en un seul.

Définition 2.20 (Relation de simulation [14])**.**
Soit $A = (V, E, S_0, F)$ un automate fini, un préordre $\succeq$ sur V est une relation de simulation pour A si :

— Pour tout $v_1, v_2, v_3 \in V$, si $v_2 \succeq v_1$ et $(v_1, v_3) \in E$, alors il existe $v_4 \in V$ tel que $v_4 \succeq v_3$ et $(v_2, v_4) \in E$.

— Pour tout $v_1, v_2 \in V$, si $v_2 \succeq v_1 : v_1 \in F$ implique $v_2 \in F$.

Pour un automate $A = (V, E, S_0, F)$, une fois qu'une relation de simulation $\succeq$ est définie, nous sommes en mesure de créer une antichaîne $H \subseteq V$ simulant des états de l'automate A.
Depuis l'algorithme de recherche en largeur présenté en Algorithme 2.1, on va pouvoir l'adapter pour travailler avec une antichaîne. Par exemple, avec une relation de simulation, on obtient :

ALGORITHME 2.2: Recherche en largeur améliorée [14]

```
1  begin
2      i ← 0
3      R̃_0 ← S_0
4      repeat
5          i ← i + 1
6          R̃_i ← R̃_{i−1} ∪ Succ(R̃_{i−1})
7          R̃_i ← Max(⪰, R̃_{i−1})
8          if R̃_i ∩ F ≠ ∅ then return Reachable
9      until R̃_i = R̃_{i−1}
10     return Not reachable
```

L'Algorithme 2.2 est identique à la recherche en largeur présentée initialement, sauf qu'après avoir récupéré les successeurs des états explorés, celui-ci ne retient que les éléments maximaux. Concrètement, cela signifie que si un état permet de visiter tous les états auxquels un autre a accès, l'algorithme ne prendra pas la peine de visiter ce deuxième état. C'est inutile, puisque tous les états qu'il peut atteindre seront déjà atteignables par le premier état. Le challenge ici est évidement de définir une telle relation de simulation. La preuve de l'exactitude de cet algorithme est consultable en [14].

Cette sous-section se termine en illustrant toutes ces notions par un exemple :

Exemple 2.3.

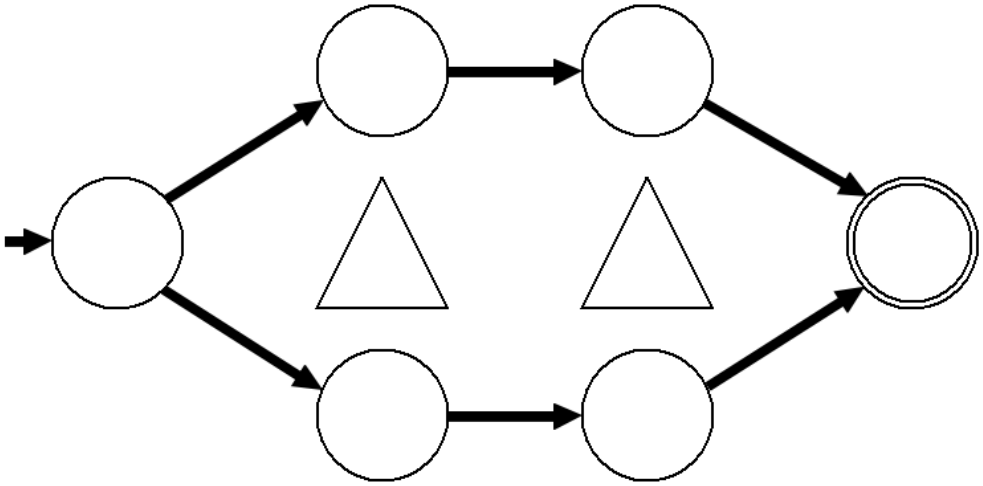

FIGURE 2.3 – Exemple d'utilisation d'antichaîne

Cette illustration représente un automate fini, un cercle est un état, une flèche reliant deux cercles est une transition, l'état initial est celui avec la flèche orpheline pointant vers lui et l'état final est le double cercle.

Sur l'exemple en Figure **??**, l'objectif est de savoir s'il existe un chemin d'un état initial à un état final. Nous avons identifié une relation de simulation $\triangleleft$, sur l'image un état avec la pointe du triangle devant lui simule celui qui a la base devant lui. Nous allons exécuter un algorithme de recherche en largeur améliorée. À la première itération, on obtiendra alors les deux états succédant l'initial, on utilisera la fonction Max avec la relation de simulation $\triangleleft$ pour garder celui ou ceux qui ne sont simulés par aucun autre et nous continuerons à utiliser les fonctions $Succ$ et Max jusqu'à stabilité. Dans l'exemple, nous emprunterons donc le chemin du haut jusqu'à l'état final.

2.3 Réduction

Ont été présentés deux problèmes qui semblent différents, de deux domaines différents de l'informatique, pourtant ils peuvent être corrélés. En effet, le but est de résoudre le problème de l'ordonnançabilité à criticité mixte en résolvant celui de l'accessibilité dans un automate. Ces deux problèmes ne sont pas liés tels quels, il va falloir réduire l'un vers l'autre, en l'occurrence, l'ordonnançabilité à criticité mixte vers le l'accessibilité dans un automate.

Cette réduction est une passerelle entre deux communautés des sciences informatiques qui n'ont *a priori* pas à se rencontrer.

Il s'agit donc de formaliser l'ordonnancement CM en tant que système à transitions étiquetées. La modélisation se basera sur le travail de Baker et Crinei [12], celle-ci formalise toutes les exécutions possibles d'un système de tâches par les différents chemins possibles dans un automate. Cependant, le type d'ordonnancement n'est pas le même. En effet, dans le travail de Baker et Crinei, on retrouve un test d'ordonnançabilité pour un système de tâches sporadiques en multiprocesseur.

Certaines notions restent les mêmes, par exemple on se servira d'état du système, nécessitant de savoir pour chaque travail d'une instance le temps avant leur libération nat et le temps restant de calcul rct.

Définition 2.21 (État du système [14])**.**
Étant donné une instance de n travaux $I = (J_1, ..., J_n)$, l'ensemble des états du système est un ensemble de tuples $S \stackrel{def}{=} \langle nat_S, rct_S \rangle$

- nat_S est une fonction de I vers $\mathbb{N}$ tel que pour tout J_i : $nat_S(J_i) \leq O_{max}$ avec $O_{max} \stackrel{def}{=} max_i\ O_i$
- rct_S est une fonction de I vers $(1, ..., C_{max})$ avec $C_{max} \stackrel{def}{=} max_i\ C_i$

Les états possibles d'une instance I sont représentés par $States(I)$.
Ces états semblent infinis, car il n'y a pas de borne inférieure pour nat, mais il sera démontré plus tard qu'il y en a bien une.

Nous remarquons ici que cette définition telle quelle n'est pas adaptée à une système de tâches CM, car elle ne tient pas compte des différents niveaux de criticité, mais elle donne déjà une idée de l'allure qu'elle prendra. Il faudra définir les différents évènements possibles, comme un coup d'horloge ou le passage à une criticité supérieure, tout ceci se retrouvera dans le modèle.

2.4 Contribution et structure du mémoire

Le but de ce mémoire est de trouver un test d'ordonnançabilité CM exact pour un système de tâches CM sporadiques.

Dans le chapitre 3, un éventail des algorithmes d'ordonnancement pour systèmes de tâches CM sporadiques et leur condition suffisante d'ordonnançabilité sera présenté.

L'objectif de celui-ci sera double. Premièrement il permettra d'utiliser certains de ces algorithmes dans l'automate explorant les scénarios générés par un système de tâches CM et un algorithme d'ordonnancement. Deuxièmement, il offrira la possibilité de comparer la performance des tests d'ordonnançabilité déjà existants et le test exact créé dans ce mémoire.

Le chapitre évitera de rentrer trop dans les détails, le but est de comprendre comment fonctionnent les différents types d'algorithmes et quel est leur test. C'est pourquoi les preuves seront souvent omises.

Enfin, comme une majorité d'algorithmes ne fonctionnent qu'avec une criticité double, le chapitre se concentre sur cette sous-classe d'algorithme. C'est-à-dire que s'il existe une extension de l'algorithme présenté pour une criticité quelconque, elle ne sera pas présentée.

Ensuite viendra le chapitre 4, celui-ci offrira la réduction du test d'ordonnançabilité CM vers l'accessibilité dans un automate.

La première section se concentre sur un système de tâches CM périodiques. Cette première réduction permettra de mettre en évidence l'incertitude générée par l'ordonnancement en criticité mixte. De plus, bien que plus restrictives que les tâches CM sporadiques, les tâches CM périodiques restent tout de même intéressantes.

Il s'en suivra la réduction du test d'ordonnançabilité CM pour un système de tâches CM sporadiques vers l'accessibilité dans un automate.

Le chapitre se terminera en présentant une relation de simulation pour l'automate créé à partir d'un système de tâches CM sporadiques. La preuve de l'exactitude de cette relation de simulation formera le cœur de la section.

Le pénultième chapitre de ce mémoire, le chapitre 5, présentera les résultats du travail fourni. Il s'agira, dans un premier temps, de comparer les complexités des solutions aux problèmes de l'ordonnancement CM pour un système de tâches CM périodiques, sporadiques, avec et sans antichaîne. Et, en second lieu, de comparer les tests d'ordonnancement des algorithmes présentés dans le chapitre 3 entre eux, mais aussi avec le nouveau test exact, par exploration d'automate.

Enfin, le chapitre 6, exposera les conclusions de ce mémoire, basées sur le chapitre consacré aux résultats. De plus, il proposera différents travaux pour étendre le travail fourni ou pour l'exploiter plus amplement.

Chapitre 3

Algorithme d'ordonnancement

3.1 Introduction

Le but de ce chapitre est de présenter différents algorithmes d'ordonnancement d'un système de tâches CM sporadiques. Cette approche remplit deux objectifs.
Premièrement, elle permettra de présenter différentes conditions suffisantes d'ordonnançabilité, autrement dit, des tests d'ordonnancements déjà existants pour ce type d'un système de tâches CM.
Ensuite, le modèle qui sera créé dans ce travail vérifiera précisément si un système de tâches CM est ordonnançable avec un certain algorithme. Il faut donc définir cesdits algorithmes.

Le chapitre commence par l'introduction de l'algorithme trivial *CAPA*, puis présente ceux basés sur *Audsley* au niveau des tâches CM, ensuite au niveau des travaux CM, enfin, passe en revue ceux basés sur *EDF*, mais modifiant les échéances, pour conclure, un algorithme original *LWLF*.

3.2 CAPA

L'algorithme le plus simple est de définir la priorité des tâches CM en fonction de leur criticité, d'où le nom de cette méthode *Criticality As Priority Assignment, CAPA*. L'idée intuitive de l'algorithme est que l'on souhaite éviter de donner du temps d'accès à la ressource à des tâches CM qui pourraient ne plus être considérées, si un autre ordonnancement avait déclenché une criticité supérieure. Cependant, cet algorithme peut mener à l'échec d'ordonnancement de systèmes de tâches CM même avec une utilisation très faible. Cette faible performance vient de la rigidité de l'assignation des priorités [5].

Exemple 3.1 ([3]).
Considérons le système de tâches CM τ composé de :

— $\tau_1 = \{0, 3, 3, 1, [1, 1]\}$

— $\tau_2 = \{0, 5, 5, 2, [3, 4]\}$

L'algorithme *CAPA* assignera statiquement la priorité $\tau_2 \succ \tau_1$, car $\chi_2 > \chi_1$.

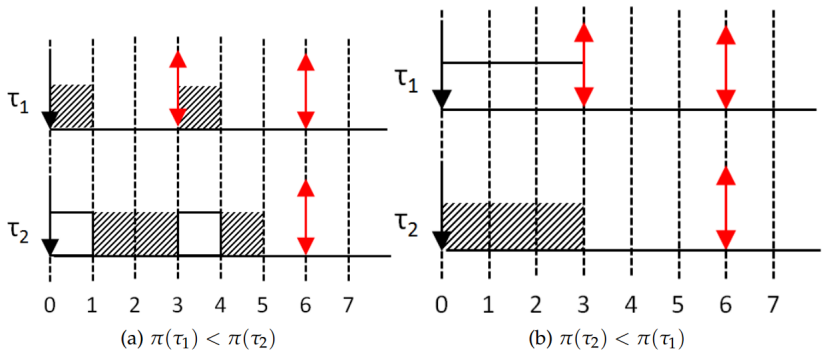

FIGURE 3.1 – Non optimalité de CAPA [3]

Dans la Figure 3.1, il est apparent que *CAPA* mène à un ordonnancement erroné, alors qu'il est possible d'ordonnancer τ.

3.3 Vestal

Hypothèses

— $K \in \mathbb{N}^*$

— $D_i \leq T_i \; \forall \; i$

Vestal propose un autre approche [8], basée sur la méthode d'Audsley [18]. L'algorithme se base sur deux observations, Vestal annonce qu'elles sont maintenues en criticité mixte.

Lemme 3.1 ([8]).
Le pire temps de réponse pour une tâche peut être déterminé en sachant quels sous-ensembles de tâches ont une priorité supérieure à elle, mais sans avoir besoin de savoir quelle est l'assignation spécifique de priorité.

Lemme 3.2 ([8]).
Si une tâche τ_i est ordonnançable selon une certaine assignation de priorité, elle reste ordonnançable avec une priorité supérieure (les priorités des autres tâches restent identiques).

L'algorithme commence avec le système de tâches CM sans priorité. Les priorités sont ensuite assignées de la plus faible à la plus forte, de sorte que la première tâche CM assignée sera le plus faiblement prioritaire.

À chaque étape, l'algorithme sélectionne la tâche CM plus faiblement prioritaire, lui assigne cette priorité et recommence avec l'ensemble des tâches CM sans cette dernière. Si à un moment, aucune tâche CM n'est éligible à recevoir la priorité la plus faible, alors cet algorithme ne permet pas d'ordonnancer ce système de tâches CM.

Il est donc nécessaire de déterminer si une tâche CM peut recevoir la plus faible priorité. Pour ce faire, Vestal propose un test basé sur le temps de réponse, faisable par le Lemme 3.1.

Définition 3.1 (Pire temps de réponse CM [8])**.**
Le pire temps de réponse CM R_i d'une tâche CM τ_i est la taille maximale de l'intervalle compris entre la génération d'un de ses travaux CM et sa complétion. Le temps de réponse ne peut s'obtenir trivialement. Il s'agit de la somme des WCETs des travaux CM de plus haute priorité que celui de la tâche CM durant l'intervalle $[t, t + R_i)$ où t est l'instant où le travail CM est généré :

$$R_i \stackrel{def}{=} \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j(\chi_i) \tag{3.1}$$

Où $hp(i)$ est l'ensemble des tâches CM avec une priorité égale ou plus élevée que τ_i

Cette équation récursive peut se résoudre à l'aide d'une itération de point fixe. Il faut commencer par assigner $R_i := C_i(\chi_i)$ et ensuite, évaluer la partie droite de l'équation jusqu'à stabilisation. À tout instant, si $R_i > D_i$ alors la tâche CM ne peut avoir la priorité qui lui a été assignée, sous peine d'un ordonnancement défaillant. Si, lorsque l'itération est terminée, $R_i \leq D_i$ alors τ_i peut recevoir la priorité la plus faible.

3.3.1 AMC-max

Hypothèses

- $K = 2$
- $D_i \leq T_i \; \forall \; i$

Baruah *et al.* ont étendu les travaux de Vestal et ont trouvé un test pour la viabilité d'une tâche CM à une certaine priorité plus permissif. [19]
Baruah *et al.* distinguent l'analyse de la viabilité d'une tâche CM à une certaine priorité en trois phases :

- Vérifier l'ordonnançabilité en criticité LO
- Vérifier l'ordonnançabilité en criticité HI
- Vérifier l'ordonnançabilité du changement de criticité

Pour effectuer ces tests, une extension du pire temps de réponse CM est faite, prenant en compte la criticité.

Définition 3.2 (Pire temps de réponse CM en criticité ℓ [19])**.**
Le pire temps de réponse CM R_i^ℓ d'une tâche CM τ_i au niveau ℓ est la taille maximale de l'intervalle compris entre la génération d'un de ses travaux CM et sa complétion lorsque ce travail CM est généré quand la criticité est ℓ. Il s'agit de la somme des WCETs au niveau ℓ des tâches CM toujours présentes et de plus haute priorité que celle de la tâche CM durant l'intervalle $[t, t + R_i^\ell)$ où t est l'instant où le travail CM est généré, et de $C_i(\chi_i)$, le pire temps d'exécution de τ_i :

$$R_i^\ell \stackrel{def}{=} C_i(\chi_i) + \sum_{\tau_j \in hp(i,\ell)} \left\lceil \frac{R_i^\ell}{T_j} \right\rceil C_j(\ell) \tag{3.2}$$

Où $hp(i, \ell)$ est l'ensemble des tâches CM de criticité au moins ℓ, avec une priorité plus élevée que τ_i

Pour vérifier l'ordonnançabilité en criticité LO et HI, on utilise les tests standards : $R_i^{LO} \leq D_i \wedge R_i^{HI} \leq D_i$. Ici R_i^ℓ est assigné à zéro pour la première itération.

Le pire temps de réponse CM d'une tâche CM τ_i lors d'un changement de criticité est noté R_i^*. Pour le trouver, on définit l'interférence maximale d'une tâche CM hautement critique s'il y a un changement de criticité, généré par une tâche CM τ_s à un instant s. Le passage au niveau de criticité supérieur est déclenché lorsqu'une tâche CM τ_s s'exécute pour plus longtemps que $C_s(LO)$. Si cet évènement impacte τ_i, cela signifie que $s < R_i^{LO}$, et la priorité de τ_s est supérieure ou égale à celle de τ_i ; sinon τ_i se serait complétée avant le changement de criticité. On crée alors R_i^s, le pire temps de réponse de la tâche CM τ_i, lorsqu'un changement de criticité se passe au temps s, relativement au temps de génération de τ_i.

$$R_i^s \stackrel{def}{=} C_i(HI) + I_L(s) + I_H(s) \tag{3.3}$$

où $I_L(s)$ est l'interférence venant des tâches CM avec une criticité plus faible que τ_i et $I_H(s)$ est l'interférence venant des tâches CM avec une criticité supérieure ou égale à celle de τ_i.

Les tâches CM de criticité plus faible sont abandonnées après s et donc leur interférence est bornée par :

$$I_L(s) \stackrel{def}{=} \sum_{j \in hp(i) \wedge \chi_j = LO} (\left\lfloor \frac{s}{T_j} \right\rfloor + 1) C_j(LO) \tag{3.4}$$

La valeur $plancher + 1$ est utilisée, car dès qu'un travail CM est émise, il faut considérer son interférence. Attention, ici on n'utilise pas $hp(i, LO)$, car on souhaite ne considérer que les tâches CM de basse criticité.

Pour l'interférence $I_H(s)$, on supposera que tous les travaux CM actifs au temps s, s'exécutent pour $C(HI)$ unité de temps. D'où, seuls les travaux CM avec une échéance avant s utilisent leurs estimations pour la criticité LO.

Considérons l'interférence d'une tâche CM τ_k au temps t avec $t > s$. Le nombre maximum d'émissions que la tâche CM peut faire est $\lceil t/T_k \rceil$. Le nombre maximum d'émissions qu'une tâche CM peut faire durant l'intervalle t - s est borné par

$$\left\lceil \frac{t - s - (T_k - D_k)}{T_k} \right\rceil + 1 \tag{3.5}$$

Cependant, si s est petit et D_k trop proche de T_k alors l'Équation 3.5 peut inclure plus de travaux CM qu'il n'y en a vraiment. On définit donc

$$M(k, s, t) = min \left\{ \left\lceil \frac{t - s - (T_k - D_k)}{T_k} \right\rceil + 1, \left\lceil \frac{t}{T_k} \right\rceil \right\} \tag{3.6}$$

L'interférence au temps t est alors

$$I_H(s) \stackrel{def}{=} \sum_{k \in hp(i,HI)} \left(M(k, s, t) C_k(HI) + \left(\left\lceil \frac{t}{T_k} \right\rceil - M(k, s, t) \right) C_k(LO) \right) \tag{3.7}$$

Et donc

$$R_i^s = C_i(HI) + \left(\sum_{j \in hp(i) \wedge \chi_j = LO} (\left\lfloor \frac{s}{T_j} \right\rfloor + 1) C_j(LO) \right) +$$

$$\sum_{k \in hp(i,HI)} \left(M(k, s, R_i^s) C_k(HI) + \left(\left\lceil \frac{R_i^s}{T_k} \right\rceil - M(k, s, R_i^s) \right) C_k(LO) \right) \tag{3.8}$$

Avec

$$R_i^* \stackrel{def}{=} max(R_i^s) \forall s \tag{3.9}$$

Il reste à trouver quelles sont les s à tester. L'intervalle possible pour s va de 0 à R_i^{LO}. En analysant R_i^s, il faut remarquer que le terme avec $j \in hp(i) \wedge \chi_j = LO$ augmente avec s, alors que le terme avec $hp(i, HI)$ diminue avec s. D'où, R_i^s ne peut augmenter que lorsqu'une tâche CM faiblement critique génère un travail CM, il s'agit donc des points à prendre en compte.

3.4 OCBP

Hypothèses

- $K = 2$
- $D_i \in \mathbb{N}^* \ \forall \ i$

OCBP est un algorithme d'ordonnancement basé sur assignation de priorités. Dans la même veine que *Vestal* et *AMC-max*, cet ordonnanceur cherche à assigner la priorité la plus basse récursivement. Cependant, ici ce sont directement les travaux CM qui sont considérés et non les tâches CM. L'algorithme se base uniquement sur le WCET de la criticité d'une tâche CM pour son assignation, c'est pour ça qu'il s'appelle *Own Criticality Based Priority, OCBP*.

Une première version de cet algorithme a été imaginée pour une instance de travaux CM, dans cette configuration, l'assignation de priorité se fait hors ligne [11]. Il semble alors évident de simplement appliquer cette version de l'algorithme pour l'instance CM de travaux CM générée par un système de tâches CM sporadiques. Cependant, cette approche est impossible. Premièrement, la phase hors-ligne d'assignation des priorités peut être infinie, puisque l'instance de travaux CM générés par le système de tâches CM sporadiques est potentiellement infinie. Ensuite, et surtout, l'algorithme dans sa version sur instance CM requiert d'avoir une spécification précise des travaux CM. Or avec des tâches CM sporadiques, il n'est pas possible de savoir exactement quand les travaux CM seront générés, seule une borne inférieure est disponible.

La suite de cette section présente l'algorithme *OBCP* pour des tâches CM sporadiques présenté dans l'ouvrage [20], mais ne s'attarde pas sur la résolution de ces problèmes énoncés. Pour les découvrir, consultez ce même ouvrage.

L'algorithme fait appel à la notion de période occupée, elle-même basée sur celle d'instant oisif, ci-dessous leur définition.

Définition 3.3 (Instant oisif [2])**.**
$x \in \mathbb{N}$ est un instant oisif de l'ordonnancement d'un système si toutes les requêtes émises strictement avant x ont été complétées, avant ou au temps x.

Définition 3.4 (Période occupée [2])**.**
Une période occupée est un intervalle $[a, b)$, tel que a et b sont deux instants oisifs, le processeur est occupé durant l'intervalle et il n'y a pas d'instant oisif intermédiaire.

Supposons qu'une période occupée commence à un instant t_0, durant l'exécution. À cet instant, il faut assigner une priorité à tous les travaux CM qui pourraient potentiellement arriver dans la période occupée commençant à t_0, comme si tous les travaux CM étaient émis au premier instant permis. Pour le bon fonctionnement de l'algorithme *OCBP*, il faut considérer que tous ces travaux CM sont *générés-au-plus-tôt*. Ce principe est illustré avec l'Exemple 3.2.

Exemple 3.2.
Supposons qu'une tâche CM τ_i peut générer trois travaux CM durant la période occupée commençant à t_0. Au plus tôt, ces travaux CM seront générés au temps t_0, t_0+T_i, t_0+2*T_i respectivement et auront pour échéance $t_0 + D_i, t_0 + T_i + D_i, t_0 + 2 * T_i + D_i$ respectivement. Considérer ces travaux CM comme *générés-au-plus-tôt* signifie que leur temps d'arrivée sera t_0 pour chacun d'entre eux, leurs échéances restent inchangées.

Formellement, $\{J_i\}_{i=1}^{n}$ dénote tous ces travaux CM à t_0 ; $J_i = (t_0, d_i, \chi_i, c_i)$

3.4.1 Assignation des priorités

Il faut assigner les priorités aux travaux CM $\{J_i\}_{i=1}^{n}$ conformément à l'assignation de priorité *OCBP*. Cette assignation de priorité est la fonction $\pi : \{J_i\}_{i=1}^{n} \to 1, 2, ..., n$.

L'assignation *OCBP* fonctionne de la manière suivante :

Déterminer au sein d'une instance CM I, le travail CM J_i le plus faiblement prioritaire : un travail CM J_i peut avoir la priorité la plus faible si

— c'est un travail CM faiblement critique ($\chi_i = LO$), et il y a au moins $c_i(LO)$ temps entre son émission et son échéance, si tous les autres travaux CM J_j ont une priorité plus élevée et sont exécutés pour $c_j(LO)$ unité de temps ; ou

— c'est un travail CM fortement critique ($\chi_i = HI$), et il y a au moins $c_i(HI)$ temps entre son émission et son échéance, si tous les autres travaux CM J_j ont une priorité plus élevée et sont exécutés pour $c_j(HI)$ unité de temps.

En général, il est possible que plusieurs travaux CM soient éligibles pour être le travail CM le plus faiblement prioritaire, il faut alors en choisir arbitrairement un. Cette procédure est répétée pour l'ensemble des travaux CM ; sans ce travail CM avec une priorité récemment assignée, jusqu'à ce que tous les travaux CM aient une priorité assignée ou que le travail CM le plus faiblement prioritaire ne puisse être trouvé. Dans ce dernier cas, l'instance CM I n'est pas ordonnançable avec *OCBP* et donc le système de tâches CM τ non plus.

3.4.2 Ordonnancement durant l'exécution

Durant l'exécution, à tout moment, le travail CM J_i qui a été généré, mais pas encore signalé comme complété avec la valeur $\pi(J_i)$ la plus faible, est sélectionné pour exécution. Ce procédé continue jusqu'à ce que l'un des évènements suivants arrive :

E-1 Un travail CM J_i s'exécute pour plus que $c_i(LO)$ unité de temps sans signaler sa complétion. Ceci implique que le système est maintenant de criticité HI et que les travaux CM de criticité LO ne doivent plus être complétés d'ici leurs échéances. Il s'en suit, selon l'exactitude de l'assignation de priorité *OCBP* à t_0, que les travaux CM de criticité HI qui arriveront durant la période occupée courante sont garantis d'être exécutés jusqu'à leur complétion.

E-2 Selon *OCBP*, le processeur est oisif à un certain instant t, seulement si tous les travaux CM qui ont été générés avant t se sont complétés avant ou au temps t. Si c'est le cas, la période occupée courante est terminée et les priorités qui ont été assignées aux travaux CM qui n'ont pas été émis sont annulées. Il faut alors attendre la prochaine émission d'un travail CM, qui va signaler le début d'une nouvelle période occupée. À ce moment-là, il faut recalculer les priorités de tous les travaux CM qui pourraient arriver durant cette nouvelle période occupée.

E-3 L'exécution d'un travail CM J_x moins prioritaire est préemptée à cause de la génération d'un travail CM de plus haute priorité J_y ($\pi(J_y) < \pi(J_x)$), à l'instant t_1. Il faut alors recalculer la fonction de priorité π à ce moment précis. Les travaux CM ayant une priorité plus faible que celle de J_x garderont leur priorité actuelle.

Réassignation des priorités : pour tout i, soit Δ_i le nombre de temps d'exécution, dont le travail CM J_i a bénéficié durant l'intervalle $[t_0, t_1)$. Soit $c_i'(LO) \stackrel{def}{=} c_i(LO) - \Delta_i$, $c_i'(HI) \stackrel{def}{=} c_i(HI) - \Delta_i$ et $c_i' \stackrel{def}{=} [c_i'(LO), c_i'(HI)]$; ceux-ci dénotent les WCETs restants pour J_i.

Il est possible de considérer la charge de travail restante à exécuter durant la période occupée comme l'instance

$$\{J'_i = (t_1, d_i, \chi_i, c'_i)\}_{i=1}^{n} \tag{3.10}$$

Supposons que $(n - n')$ travaux CM ont signalé leurs complétions, durant $[t_0, t_1)$: ce sont les travaux CM J_k avec $\pi(J_k) < \pi(J_x)$, car sinon, J_x aurait été exécuté avant eux. Soit $\{J'_i\}_{i=1}^{n'}$ les travaux CM restants. L'assignation de priorité

$$\pi' : \{J'_i\}_{i=1}^{n'} \to 1, 2, ...n'$$

est obtenue depuis π comme suit :

1. Tous les travaux CM J_k satisfaisant $\pi(J_k) < \pi(J_x)$, qui n'ont pas encore complété leur exécution, ont les priorités $\pi'(J_k)$ recalculées par *OCBP* sous l'hypothèse de *génération-au-plus-tôt*, qu'ils sont tous générés à l'instant t_1
2. Pour tous les travaux CM J_k satisfaisants $\pi(J_k) \geq \pi(J_x), \pi'(J_k) \leftarrow \pi(J_k) - (n - n')$

L'algorithme est totalement présenté.

3.4.3 Test d'ordonnancement

Il existe une condition suffisante d'ordonnançabilité pour *OCBP* venant de l'article [21], cette sous-section la présente.

Le test d'ordonnancement se base sur la charge de travail. Dans l'ordonnancement temps réel classique, il s'agit du maximum, sur tous les intervalles possibles, du temps d'exécution requis cumulé par le système de tâches CM, normalisé par l'intervalle [21]. Il s'agit donc d'une borne inférieure sur la portion de la capacité d'exécution requise par le système de tâches CM pour respecter toutes les échéances.

Parallèlement à ce concept, il est possible de définir la charge de travail pour un système de tâches CM à chaque niveau de criticité.

Définition 3.5 (Charge ℓ-critique)**.**
La charge ℓ-critique d'un système de tâches CM τ est définie par

$$\Gamma_\ell(\tau) \stackrel{def}{=} \max_{0 \leq t_1 \leq t_2} \left\{ \sum_{\forall J_i : \chi_i \geq \ell \wedge t_1 \leq r_i \wedge d_i \leq t_2} c_i(\ell)/(t_2 - t_1) \right\}$$

Pour toute criticité ℓ, $\Gamma_\ell(\tau)$ peut être calculée avec des techniques pour déterminer les charges de travail d'un système de tâches traditionnelles [22].

Intuitivement, $\Gamma_\ell(\tau)$ représente une borne inférieure sur la portion de capacité d'exécution requise par ce système de tâches CM, avec laquelle elle peut respecter toutes ses échéances sujettes seulement à la certification au niveau de criticité ℓ. Pour exécuter correctement un système de tâches CM sporadiques, une condition nécessaire est que cette portion de capacité d'exécution requise ne dépasse pas 1 (avec un processeur à vitesse unitaire).

Dans cette section, *OCBP* est présenté dans le contexte d'un système à criticité double uniquement. Cependant, cet algorithme peut être étendu pour gérer un système de tâches CM sporadiques avec plus de deux niveaux de criticité. Il est alors possible d'utiliser les connaissances venant de [23] pour obtenir le test d'ordonnancement CM suivant :

$$\forall \ell \in [1, K] : \Gamma_\ell(\tau) \leq LoadBound(K) \tag{3.11}$$

Où $\Gamma_\ell(\tau)$ est la charge au niveau de criticité ℓ, et $LoadBound(K)$ est une fonction avec respect du nombre total de niveau criticité K du système, qui est calculée récursivement :

$$LoadBound(1) = 1$$
$$LoadBound(K) = \frac{2}{1 + \sqrt{4\frac{1}{LoadBound(K-1)^2} + 1}}$$

Pour le cas spécial où $K = 2$, une condition sur la charge plus précise est disponible [24] :

$$(\Gamma_{HI}(\tau))^2 + \Gamma_{LO}(\tau) \leq 1 \tag{3.12}$$

Voilà qui offre un test d'ordonnançabilité pour *OCBP*.

3.4.4 Taille de la plus grande période occupée

Lors de l'ordonnancement, il est nécessaire de connaître la taille de la plus grande période occupée pour assigner les priorités aux travaux CM qui y figurent. À nouveau, l'article [11] propose une borne supérieure pour celle-ci.

La taille de la plus grande période occupée est divisée en deux. Premièrement, x_1 est l'intervalle commençant au début de la plus grande période occupée jusqu'au moment où un travail CM dépasse son WCET de criticité LO, s'il y en a un. Ensuite, x_2 est l'intervalle qui commence à la fin de x_1 et se termine à la fin de la plus grande période occupée.

Soit $D_{max} \stackrel{def}{=} max_i D_i$. Tous les travaux CM exécutés pendant $[0, x_1)$ ont leur temps de génération et leur échéance durant l'intervalle $[0, x_1 + D_{max}]$, d'où

$$x_1 \leq \Gamma_{LO}(\tau)(D_{max} + x_1)$$

$$x_1(1 - \Gamma_{LO}(\tau)) \leq \Gamma_{LO} \times D_{max}$$

$$x_1 \leq \frac{\Gamma_{LO}(\tau)}{1 - \Gamma_{LO}(\tau)} \times D_{max} \tag{3.13}$$

Vu que tous les travaux CM exécutés durant $[x_1, x_1+x_2)$ ont leur temps de génération et leur échéance durant l'intervalle $[x_1, x_1 + x_2 + D_{max}]$, on en déduit que

$$x_2 \leq \Gamma_{HI}(\tau)(D_{max} + x_1 + x_2)$$

$$x_2(1 - \Gamma_{HI}(\tau)) \leq \Gamma_{HI}(\tau) \times (D_{max} + x_1)$$

$$x_2 \leq \frac{\Gamma_{HI}(\tau)}{1 - \Gamma_{HI}(\tau)} \times (D_{max} + x_1)$$

$$x_2 \leq \frac{\Gamma_{HI}(\tau)}{1 - \Gamma_{HI}(\tau)} \times (D_{max} + \frac{\Gamma_{LO}(\tau)}{1 - \Gamma_{LO}(\tau)} \times D_{max})$$

$$x_2 \leq \frac{\Gamma_{HI}(\tau)}{(1 - \Gamma_{HI}(\tau))(1 - \Gamma_{LO}(\tau))} \times D_{max} \tag{3.14}$$

La taille de la plus grande période occupée est donc plafonnée par $x_1 + x_2$. Une fois que la plus grande période occupée est bornée, il facile d'obtenir les travaux CM qui peuvent s'y générer.

3.5 PLRS et LPA

Hypothèses

- $K \in \mathbb{N}^*$
- $D_i \in \mathbb{N}^* \; \forall \; i$

PLRS, *Priority List Reuse Scheduling* est une extension d'*OCBP*, mais considère les instants où un travail CM plus prioritaire que celui exécuté actuellement est généré de manière plus abstraite, pour alléger le temps de calcul. *LPA*, *Lazy Priority Adjustment* est une extension de *PLRS*, qui utilise notamment une taille maximale de période occupée plus restreinte. Ces deux algorithmes étant fort proches de l'initial, cette sous-section ne fera que présenter leur condition suffisante d'ordonnancement.

PLRS commence par faire une assignation de priorité initiale et la modifiera en ligne. S'il est possible de faire cette assignation pour un système de tâches CM, il est ordonnançable CM avec *PLRS* [21]. Cette assignation est en fait la même que celle pour *OCBP*. Guan *et al.* proposent d'utiliser cette formulation : chaque tâche CM τ_k a un nombre δ_k représentant le nombre de travaux CM qu'elle générera durant la période occupée au maximum, qui n'ont pas encore de priorité assignée. Initialement, δ_k vaut donc le nombre de travaux CM que la tâche CM τ_k générera durant la période occupée au maximum. L'algorithme commence par décider quel travail CM ayant l'index le plus élevé peut recevoir la priorité la plus faible. Le travail CM $J_k^{\delta_k}$ est éligible à recevoir la priorité la plus faible si

$$\sum_{\tau_j \in \tau} (\delta_j \times C_j(\chi_k)) \leq (\delta_k - 1) \times T_k + D_k \tag{3.15}$$

La partie gauche de l'inéquation représente la charge de travail restante des travaux CM présents dans l'instance CM composée des travaux CM générés par les tâches CM présentes dans τ durant la période occupée maximale, si le système est de criticité χ_k. La partie de droite de l'inéquation est la distance minimale entre l'échéance absolue de $J_k^{\delta_k}$ et le début la période occupée considérée. Si plusieurs travaux CM peuvent recevoir la priorité la plus faible, il faut en choisir un arbitrairement. Une fois ce travail CM le plus faiblement prioritaire désigné, il faut décrémenter δ_k d'une unité pour exclure ce travail CM lors des prochaines étapes. La période occupée maximale est la même que celle dans *OCBP*.

Par la suite, l'algorithme itère jusqu'à ce que tous les travaux CM aient une priorité. Si ce n'est pas possible, alors le système de tâches CM n'est pas ordonnançable avec *PLRS*. Attention, bien que si un système de tâches CM passe cette assignation signifie qu'il est ordonnançable avec *PLRS*, suivre ses priorités en tant que telles ne fonctionnera pas forcément. En effet, l'algorithme modifiera les priorités en ligne, cette partie est consultable en [21].

LPA , en fidèle successeur d'*OCBP*, commence par faire une assignation de priorité hors ligne. Il s'agit de la même que *PLRS*. À nouveau, si un système de tâches CM passe cette assignation, alors il est ordonnançable avec *LPA* [25]. Ici, Gu *et al.* proposent une nouvelle façon de calculer la taille de la période maximale.

Gu *et al.* définissent la charge ℓ-critique comme la borne supérieure de la charge totale des tâches CM dont la criticité n'est pas supérieure à ℓ, dans toutes les périodes occupées, où la criticité ne dépasse pas ℓ. D'où $\Gamma_K(\tau)$ est la borne supérieure pour la taille des périodes occupées pour le système de tâches CM τ. Et ils posent $\Gamma_0(\tau) = 0$.

Ensuite, ils prouvent le théorème suivant.

Théorème 3.1 ([25])**.**
Étant donné un système de tâches CM sporadiques τ et la charge $\ell-1$-critique $\Gamma_{\ell-1}$, on a :

$$\phi_\ell = \frac{\Gamma_{\ell-1}(\tau) + \sum_{\chi_i \geq \ell} C_i(\ell)}{1 - \sum_{\chi_i \geq \ell} \frac{C_i(\ell)}{T_i}} \tag{3.16}$$

$$\Gamma_\ell(\tau) = \Gamma_{\ell-1}(\tau) + \sum_{\chi_i = \ell} C_i(\ell) \times \left(1 + \left\lfloor \frac{\phi_\ell}{T_i} \right\rfloor\right) \tag{3.17}$$

Depuis ce théorème, Gu *et al.* proposent l'algorithme suivant pour calculer la charge et ensuite réutilisent la manière de créer la période occupée d'*OCBP*, en utilisant leur nouvelle façon de calculer la charge.

ALGORITHME 3.1: Calcul de la charge ℓ-critique

```
1  ComputeGamma(ℓ)
2    if ℓ = 0 then
3      return 0
4    endif
5    Γ_ℓ(τ) ←ComputeGamma(ℓ − 1)
6    γ_ℓ ← (Γ_ℓ(τ) + Σ_{χ_i≥ℓ} C_i(ℓ)) / (1 − Σ_{χ_i≥ℓ} C_i(ℓ)/T_i)
7
8    return Γ_{ℓ−1}(τ) + Σ_{χ_i=ℓ} C_i(ℓ) × (1 + ⌊γ_ℓ/T_i⌋)
```

Avec l'Algorithme 3.1, on peut obtenir $\Gamma_K(\tau)$, il s'agit de ComputeGamme(K).

3.6 EDF-VD

Hypothèses

- $K = 2$
- $D_i = T_i \; \forall \; i$

Baruah *et al.* proposent un algorithme[10] qui est une variante de *EDF*, appelé *EDF* à échéance virtuelle, *EDF-VD*, pour *Virtual Deadlines*. Pour la compréhension de l'algorithme, l'explication qui suit considère un système de tâches CM avec une criticité double.

Il faut distinguer deux cas :

— Cas 1. $U_1(1) + U_2(2) \leq 1$: appliquer *EDF* avec les échéances des tâches CM non modifiées. Dès que le système passe au niveau 2, abandonner les tâches CM de niveau 1.

— Cas 2. $U_1(1) + \frac{U_2(1)}{1 - U_2(2)} \leq 1$ et le cas 1 n'est pas satisfait : $\lambda = \frac{U_2(1)}{1 - U_1(1)}$. Lorsque le système est en criticité 1 : pour toutes tâches CM τ_i avec $\chi_i = 2, \widehat{p}_i \stackrel{def}{=} \lambda p_i$. Redéfinir les échéances des tâches CM telles que $\chi_i = 2$ en y ajoutant $\widehat{p}_i$ au temps des générations de chacun de leurs travaux. Ne pas modifier les échéances des tâches CM avec $\chi_i = 1$ et appliquer *EDF*. Lorsque le système atteint la criticité de niveau 2, abandonner les tâches CM de criticité de niveau 1 et réinitialiser les échéances des tâches CM encore considérées et appliquer *EDF* sur ces échéances initiales.

Baruah *et al.* donnent ensuite une condition suffisante d'ordonnançabilité :

Théorème 3.2 ([10])**.**
Si τ satisfait

$$U_1(1) + min\left(U_2(2), \frac{U_2(1)}{1 - U_2(2)}\right) \leq 1$$

Alors τ est ordonnançable avec EDF-VD.

EDF-VD a été généralisé pour des systèmes de tâches CM avec un niveau de criticité plus élevé que 2, l'extension est consultable dans l'ouvrage [10].

3.7 Greedy

Hypothèses

— $K = 2$

— $D_i \leq T_i \; \forall \; i$

Ekberg *et al.* ont proposé un algorithme sur l'analyse de la demande des tâches CM [26], cette section le présente.

Une approche connue pour l'analyse de l'ordonnancement en temps réel classique est d'utiliser une fonction de borne de demande [27]. Une fonction de borne de demande capture la demande d'exécution maximale d'une tâche durant un intervalle d'une taille donnée.

Définition 3.6 (Fonction de borne de demande [26])**.**
Une fonction de borne de demande $dbf(\tau_i, \Delta)$ donne une borne supérieure pour la demande d'exécution maximale possible d'une tâche τ_i dans tous les intervalles de temps d'une taille Δ, où la demande est calculée comme le temps d'exécution requis total des travaux avec leur fenêtre d'ordonnancement ($[r_i, d_i)$) complètement dans l'intervalle.

Un concept similaire est la fonction de borne d'approvisionnement $sbf(\Delta)$ qui limite inférieurement la quantité de temps d'exécution fournie par le processeur dans tous les intervalles de taille Δ. Par exemple, sur un processeur à vitesse unitaire, $sbf(\Delta) = \Delta$.

Ce qui rend les fonctions de borne de demande et d'approvisionnement intéressantes, c'est la proposition suivante :

Proposition 3.1.
Un système de tâches τ (sans criticité mixte) est ordonnançable par EDF sur monoprocesseur avec la fonction de borne inférieure sbf si :

$$\forall \Delta \geq 0 : \sum_{\tau_i \in \tau} dbf(\tau_i, \Delta) \leq sbf(\Delta)$$

Ekberg *et al.* étendent l'idée de la fonction de borne de demande à la criticité mixte, avec double criticité. Pour toute tâche CM, ils construisent deux fonctions de borne de demande, dbf_{LO} et dbf_{HI}, pour les modes de criticité bas et haut respectivement. La Proposition 3.1 est étendue :

Proposition 3.2.
Un système de tâches CM sporadiques τ est ordonnançable par EDF sur un processeur avec la fonction de borne inférieure sbf_{LO} dans le mode à criticité basse et sbf_{HI} dans le mode à criticité haute si les deux conditions suivantes sont satisfaites :

$$\textbf{\textit{Condition A :}} \quad \forall \Delta \geq 0 : \sum_{\tau_i \in \tau} dbf_{LO}(\tau_i, \Delta) \leq sbf_{LO}(\Delta)$$
$$\textbf{\textit{Condition B :}} \quad \forall \Delta \geq 0 : \sum_{\tau_i \in \tau} dbf_{HI}(\tau_i, \Delta) \leq sbf_{HI}(\Delta)$$

Les conditions A et B s'assurent de l'ordonnançabilité du système de tâches CM en basse et haute criticité. Bien que ces deux modes puissent être analysés séparément avec les conditions ci-dessus, la demande en haute criticité dépend de ce qui se passe en basse criticité.

Pour trouver ces fonctions de borne de demande, on suppose, sans perte de généralité, que sbf_{LO} est de vitesse unitaire au maximum.

Dans le cas de la criticité basse, l'exercice est simple. Dans ce mode, les tâches CM τ_i se comportent comme des tâches sporadiques normales, et tous leurs travaux CM finissent, au pire, après $C_i(LO)$ unités de temps. Il est donc possible d'utiliser des méthodes standards pour calculer les fonctions de borne de demande pour tâches sporadiques [27].

Avec dbf_{HI}, c'est plus compliqué, car il faut considérer les travaux CM de criticité haute qui ont été actifs durant le changement de criticité.

Définition 3.7 (Travaux transférés [26])**.**
Un travail CM émis par une tâche CM à haute criticité qui est actif (généré, mais non terminé) au moment du changement de criticité est appelé un travail transféré.

3.7.1 Analyse de la demande des travaux transférés

En haute criticité, il faut finir l'exécution restante des travaux transférés avant leur échéance. La demande de ces travaux transférés doit donc être prise en considération dans chaque dbf_{HI} des tâches CM de haute criticité. Conceptuellement, lors de l'analyse de l'ordonnançabilité en haute criticité, on peut considérer les travaux transférés comme des travaux CM ayant été générés lors du changement de criticité. Cependant, la fenêtre d'ordonnancement devient l'intervalle entre le changement de criticité et l'échéance du travail CM, et peut donc être plus courte que d'autres travaux CM générés par la même tâche CM. Parce que ces travaux CM ont pu déjà être exécutés pour quelques unités de temps, le temps restant de calcul peut être inférieur à celui des autres travaux CM générés par la même tâche CM.

Pour borner la demande en haute criticité (condition B), on suppose que la demande en basse criticité est approvisionnée, sinon le système de tâches CM ne peut être ordonnançable. Il s'agit donc de démontrer $A \wedge (A \rightarrow B)$, ce qui permettra d'affirmer $A \wedge B$.

Analysons les informations disponibles sur les travaux transférés. Au moment du passage vers la criticité supérieure, un travail transféré d'une tâche CM à haute criticité τ_i a $n \geq 0$ unités de temps restantes avant son échéance. La fenêtre d'ordonnancement restante du travail CM est de taille n. Puisque le travail CM aurait respecté son échéance en basse criticité, si le changement n'avait pas eu lieu, il peut y avoir au maximum n unité de temps restante de son WCET de bas niveau $C_i(LO)$ au moment du changement.

Le travail CM aura donc été exécuté durant au moins $C_i(LO)-n$ unités de temps avant le changement de criticité. Comme le système est maintenant en haute criticité, le travail CM doit accomplir jusqu'à $C_i(HI)$ unité de temps au total. Après le changement de criticité, le temps restant d'exécution total pour un travail transféré est au plus $C_i(HI)-(C_I(LO)-n)$. Malheureusement, quand n devient petit, la demande est de plus en plus difficile à respecter et mène à $dbf_{HI}(\tau_i,0) = C_i(HI) - C_I(LO)$ dans le cas extrême. Clairement, une telle fonction de borne de demande ne peut satisfaire la condition B.

3.7.2 Ajustement de la demande des travaux transférés

Le problème ci-dessus vient du fait que *EDF* peut exécuter des travaux CM de haute criticité tard lorsque le système est en basse criticité. Quand le passage à la haute criticité est déclenché, un travail transféré peut avoir une petite fenêtre d'ordonnancement dans laquelle il doit terminer sa demande de haute criticité.
Pour augmenter la taille de sa fenêtre d'ordonnancement restante, Ekberg *et al.* proposent de séparer les échéances relatives en basse et haute criticité. Pour une tâche CM τ_i, *EDF* utilisera alors l'échéance relative $D_i(LO)$ ou $D_i(HI)$, de sorte que si un travail CM est émis au temps t, la priorité assignée par *EDF* est basée sur la valeur de $t+D_i(LO)$ pendant la basse criticité et $t + D_i(HI)$ durant la haute criticité.

Diminuer l'échéance d'une tâche CM ne pose pas de problèmes, car respecter une échéance plus stricte que l'initiale garantit que cette dernière est respectée aussi. En diminuant $D_i(LO)$, il est possible de gagner du temps supplémentaire dans la fenêtre d'ordonnancement des travaux transférés au prix d'empirer la demande en basse criticité. Il faut donc $D_i(LO) = D_i$ si $\chi_i = LO$ et $D_i(LO) \leq D_i(HI) = D_i$ si $\chi_i = HI$. Aussi, il est nécessaire que $C_i(LO) \leq D_i(LO)$, comme pour l'échéance initiale. $D_i(LO)$ n'est pas une vraie échéance dans le sens où son non-respect n'entraîne pas un échec d'ordonnancement. Cependant, elle est appelée échéance, car elle permet de construire dbf_{LO} et elle est utilisée par *EDF* en basse criticité comme s'il s'agissait de l'échéance relative.

Lemme 3.3 (La demande des travaux transférés [26])**.**
En supposant qu'EDF utilise les échéances $D_i(LO)$ et $D_i(HI)$ avec $D_i(LO) \leq D_i(HI) = D_i$ pour une tâche CM τ_i à haute criticité, et qu'il est garanti que la demande en basse criticité est fournie (en utilisant $D_i(LO)$) alors, si le changement de criticité se passe lorsqu'un travail CM de τ_i a n unités de temps avant son échéance réelle, on sait que :

1. *Si $n < D_i(HI) - D_i(LO)$, le travail CM a été terminé avant le passage à la haute criticité.*

2. *Si $n \geq D_i(HI) - D_i(LO)$, le travail CM peut être un travail transféré, et, au moins, $max\,(0, C_i(LO) - n + D_i(HI) - D_i(LO))$ unités de temps de son exécution ont été finis avant le passage en haute criticité.*

La preuve du Lemme 3.3 est disponible dans l'article dont il est issu. Pour la suite, il faut montrer comment définir $dbf_{LO}(\tau_i, \Delta)$ et $dbf_{HI}(\tau_i, \Delta)$ pour $D_i(LO)$ donné.

3.7.3 Formulation des fonctions de borne de demande

Il est déjà établi que lorsque le système est en basse criticité, toutes les tâches CM τ_i comportent comme des tâches sporadiques classiques, avec les paramètres $C_i(LO), D_i(LO), T_i$. La fonction de borne de demande serrée est

$$dbf_{LO}(\tau_i, \Delta) \stackrel{def}{=} max\left(0, \left(\left\lfloor \frac{\Delta - D_i(LO)}{T_i} \right\rfloor + 1\right) \times C_i(LO)\right) \tag{3.18}$$

La fonction de borne de demande pour les tâches CM en haute criticité, $dbf_{HI}(\tau_i, \Delta)$, doit borner supérieurement la demande d'exécution maximale des travaux CM de τ_i avec une fenêtre d'ordonnancement dans un des intervalles de taille Δ. Il peut donc y avoir un travail transféré. Par le Lemme 3.3, on sait que la fenêtre restante d'ordonnancement d'un travail transféré de τ_i est au moins de taille $D_i(HI) - D_i(LO)$ unités de temps. L'intervalle de temps $D_i(HI) - D_i(LO)$ est donc l'intervalle minimal dans lequel une fenêtre d'ordonnancement d'un travail CM de τ_i peut rentrer. Plus généralement, la taille du plus petit intervalle dans lequel peuvent rentrer k travaux CM de τ_i est $(D_i(HI) - D_i(LO)) + (k-1) * T_i$. La demande d'exécution de τ_i durant un intervalle de taille Δ est donc bornée par :

$$full(\tau_i, \Delta) \stackrel{def}{=} max\left(0, \left(\left\lfloor \frac{\Delta - (D_i(HI) - D_i(LO))}{T_i} \right\rfloor + 1\right) \times C_i(HI)\right)$$

La fonction $full(\tau_i, \Delta)$ ne tient pas compte qu'un travail transféré ait pu être exécuté pour quelques unités de temps durant la basse criticité. Il est possible de vérifier si tous les travaux CM qui contribuent à la demande d'exécution $full(\tau_i, \Delta)$ peuvent insérer leur fenêtre d'ordonnancement dans un intervalle de taille Δ sans considérer un éventuel travail transféré. Pour ce faire, il faut soustraire le temps d'exécution effectué en basse criticité à $full(\tau_i, \Delta)$.

Pour un intervalle de taille Δ, il peut y avoir au plus $n = \Delta \ mod \ T_i$ unités de temps restantes pour le premier travail CM, potentiellement un travail transféré. Si $n \geq D_i(HI)$, il y a assez de place pour la fenêtre d'ordonnancement d'un travail CM complet et rien ne peut être soustrait de $full(\tau_i, \Delta)$.

Si $n < D_i(HI) - D_i(LO)$, alors les travaux CM contribuant à $full(\tau_i, \Delta)$ peuvent placer leur période entière dans l'intervalle, donc rien n'est à soustraire. Sinon, on utilise le Lemme 3.3 pour quantifier le temps de calcul à soustraire :

$$done(\tau_i, \Delta) \stackrel{def}{=} \begin{cases} max(0, C_i(LO) - n + D_i(HI) - D_i(LO)) & \text{si } D_i(HI) > n \\ & \text{et } n \geq D_i(HI) - D_i(LO) \\ 0 & \text{sinon.} \end{cases}$$

avec $n = \Delta \; mod \; T_i$.

Les deux termes peuvent maintenant être combinés pour obtenir $dbf(\tau_i, \Delta)$:

$$dbf_{HI}(\tau_i, \Delta) \stackrel{def}{=} full(\tau_i, \Delta) - done(\tau_i, \Delta) \tag{3.19}$$

3.7.4 Réglage efficient des échéances relatives

À présent, il faut fixer $D_i(LO)$ pour les tâches CM. Il n'est pas envisageable de tester toutes les combinaisons de leurs valeurs, car, bien que borné, cet ensemble est exponentiel. Ekberg *et al.* proposent une heuristique pour régler ces échéances relatives.

Leur algorithme se base sur le Lemme 3.4 qui suit :

Lemme 3.4 ([26])**.**
Si deux tâches CM τ_i et τ_j de hautes criticités sont identiques, sauf que $D_i(LO) = D_j(LO) - \delta$ pour $\delta \in \mathbb{Z}$ alors :

$$dbf_{LO}(\tau_i, \Delta) = dbf_{LO}(\tau_j, \Delta + \delta)$$
$$dbf_{HI}(\tau_i, \Delta) = dbf_{HI}(\tau_j, \Delta - \delta)$$

Ce dernier signifie qu'en diminuant $D_i(LO)$ de δ, il est permis de bouger $dbf_{HI}(\tau_i, \Delta)$ par δ pas vers la gauche au prix de bouger $dbf_{LO}(\tau_i, \Delta)$ par δ pas vers la droite. En d'autres termes, il s'agit de déplacer dbf_{LO} et dbf_{HI} en espérant trouver une configuration où la demande totale du système de tâches CM est fournie en basse et haute criticité.

L'algorithme complet est disponible dans [26], mais il reste une question à régler ? Quelle taille maximale pour Δ ? Ekberg *et al.* suggèrent l'utilisation de la méthode présentée en [27], une méthode qui fonctionne en ordonnancement classique. Ils proposent alors une transformation du système de tâches CM vers un système de tâches classique. Cependant, cette méthode ne peut être utilisée que quand $U_\tau(LO) < 1$ et $U_\tau(HI) < 1$.

3.8 LWLF

Hypothèses

— $K \in \mathbb{N}^*$

— $D_i \in \mathbb{N}^* \; \forall \; i$

Dans la littérature, en ordonnancement temps réel, il existe encore un type d'algorithme qui n'a pas encore été utilisé pour l'ordonnancement en criticité mixte. Il s'agit de *LLF, Least Laxity First*.

LLF est un algorithme d'ordonnancement DP. Il assigne aux travaux une priorité inversement proportionnelle à leur laxité.

Définition 3.8 (Laxité[2])**.**
La laxité d'un travail est le temps maximal durant lequel il peut se permettre d'être oisif sans dépasser son échéance.

Pour la criticité mixte, on définit la notion de pire laxité.

Définition 3.9 (Pire laxité)**.**
La pire laxité d'un travail CM est le temps maximal durant lequel il peut se permettre d'être oisif sans dépasser son échéance, pour son pire WCET.

LWLF, Least Worst Laxity First, ordonnance donc les travaux CM en leur attribuant un une priorité inversement proportionnelle à leur pire laxité.

Chapitre 4

Sémantique sous forme d'automate

Ce chapitre est une présentation de la réduction du problème d'ordonnancement d'un système de tâches CM vers le problème de l'accessibilité dans un automate.

En premier lieu, l'automate construit s'occupera de tâches CM périodiques. La deuxième section gère les tâches CM sporadiques. Pour terminer, la relation de simulation de tâches oisives pour un automate basé sur un système de tâches CM sporadiques est définie et prouvée comme exacte.

La partie sur les tâches CM périodiques est plus simple que celle sur les tâches CM sporadiques. Cette première section est donc une bonne entrée en la matière.

Chacune des constructions d'automates, pour tâches CM périodiques et sporadiques, se suffit. C'est-à-dire qu'il n'est pas nécessaire de consulter l'une pour comprendre l'autre. Ces automates répondant à deux problèmes différents, il semble pertinent de ne pas les lier.

En revanche, la relation de simulation présentée dans la dernière section de ce chapitre se base sur l'automate construit pour tâches CM sporadiques et ne peut être utilisée pour celui avec les tâches CM périodiques comme fondation.

4.1 Système de tâches CM périodiques

Le but de cette section est de modéliser les différentes exécutions d'un système de tâches CM périodiques, ordonnancé par un algorithme donné, sous la forme d'un automate fini.

Il s'agit, dans premier temps, de définir les états qui composent l'automate et puis, de déterminer les transitions entre ceux-ci.

Pour représenter un système de tâches CM en exécution, il faut qu'il soit possible d'y retrouver les caractéristiques des tâches CM et le scénario. Pour ce faire, on établit l'état du système comme reprenant ces caractéristiques. Pour le scénario dans lequel on se trouve, il suffit d'y inclure le niveau de criticité lors de l'exécution. Il faut ensuite pouvoir définir où les tâches CM en sont dans leur exécution et quand elles doivent être terminées au plus tard.

Définition 4.1 (État du système)**.**
Soit $\tau = \tau_1, \tau_2, \tau_3...$ un système de tâches CM périodiques, l'état du système de τ est le tuple $S \stackrel{def}{=} \langle at_S, rct_S, crit_S \rangle$ avec

— at_S, une fonction représentant le temps d'arrivée du travail CM courant d'une tâche CM : $\tau \rightarrow \mathbb{N},\ at_S(\tau_i) \leq R_{max}$ avec $R_{max} \stackrel{def}{=} max(max_i\ (T_i),\ max_i\ (O_i))$

— rct_S, le temps de calcul restant du travail CM généré par une tâche CM : $\tau \rightarrow \mathbb{N},\ 0 \leq rct_S(\tau_i) \leq C_{max}$ avec $C_{max} \stackrel{def}{=} max_{i,j}\ C_i(j)$

— $crit_S$, le niveau de criticité actuel du scénario, $\in \mathbb{N},\ 1 \leq crit_S \leq K$

Cette définition reprend correctement les caractéristiques dont l'état du système à besoin. Le temps d'arrivée des tâches CM permet de savoir quand la tâche CM a émis un travail CM et donc d'en déduire son échéance. Le temps restant de calcul permet de savoir où la tâche CM en est dans son exécution. Enfin, le niveau de criticité du scénario est représenté tel quel.

On définira tous les états du système possibles, selon la définition ci-dessus, en partant d'un système de tâches CM τ par $States(\tau)$.

L'automate sera composé de ces états du système comme états. On remarque que at_S n'est pas borné inférieurement dans la définition, l'automate semble alors infini. Il est possible de finir l'ensemble des états du système et ce sera démontré par la suite.

En regardant la définition, il est possible d'avoir des états du système qui ne sont pas cohérents avec l'ordonnancement d'un système de tâches CM. Par exemple, il est possible que le temps restant de calcul d'une tâche CM de criticité inférieure à celle du scénario soit positif. Il s'agirait d'une contradiction avec la définition de l'ordonnancement en criticité mixte. Les transitions de l'automate seront telles que ce genre d'état ne sera jamais atteint.

Avant de pouvoir définir les transitions de l'automate, il faut établir des notions auxiliaires sur lesquelles elles se baseront.

La première de celles-ci est l'ensemble des tâches CM actives, il s'agit des tâches CM qui souhaitent prendre possession du processeur. De plus, l'ensemble des tâches CM abandonnées de par la criticité du scénario est défini.

Définition 4.2 (Tâches CM actives et abandonnées)**.**
Une tâche CM est dite abandonnée lorsque sa criticité est inférieure à celle de l'état du système. De par la définition de l'ordonnancement en criticité mixte, ces tâches CM ne font plus partie de celles à ordonnancer. Les tâches CM abandonnées sont représentées dans le système comme venant de soumettre un travail CM ne nécessitant aucune ressource.

$$Discarded(S) \stackrel{def}{=} \{\tau_i \mid at_S(\tau_i) = 0 \ \wedge \ rct_S(\tau_i) = 0\}$$

Une tâche CM est active dans l'état S si elle a émis un travail CM dans celui-ci et n'a pas été abandonnée.

$$Active(S) \stackrel{def}{=} \{\tau_i \mid at_S(\tau_i) < 0 \ \vee \ (at_S(\tau_i) = 0 \ \wedge \ rct_S(\tau_i) > 0)\}$$

Si une tâche CM a généré un travail CM et que celui-ci est complété, la définition telle quelle qualifierait toujours ces tâches CM d'actives. Cependant, le système de transition sera construit de sorte que, lorsqu'une tâche CM a terminé un travail CM, son temps d'arrivée deviendra celui du prochain travail CM.
Deux cas seront possibles, soit le temps d'arrivée est dans le futur et donc la tâche CM ne sera plus active, soit le temps d'arrivée est immédiat, c'est-à-dire que le travail CM s'est terminé juste avant la soumission d'un nouveau travail CM et donc la tâche CM restera active.

En ordonnancement en criticité mixte, les tâches CM signalent leur complétion. Ce type d'ordonnancement fait l'hypothèse que les tâches CM n'excèdent pas leur pire WCET estimé. Une tâche CM qui aurait été exécutée pour temps égal à son pire WCET se signalera toujours comme complétée. De cette observation, en découle l'ensemble des tâches implicitement terminées.

Définition 4.3 (Tâches CM implicitement terminées)**.**
Une tâche CM qui aurait été exécutée pour un temps égal à son pire WCET sera implicitement terminée. Il faut donc s'assurer que la tâche CM n'a plus de temps de calcul restant et que l'estimation qui a été utilisée pour son temps restant de calcul est équivalente à la pire d'entre elles.

$$ImplicitelyDone(S) \stackrel{def}{=} \{\tau_i \mid rct_S = 0 \ \wedge \ at_S(\tau_i) < 0 \ \wedge \ C_i(crit_S) = C_i(\chi_i)\}$$

Les états du système ne sont pas tous des états qui seront réellement possibles durant l'ordonnancement d'un système de tâches CM. Par exemple, il n'y a aucune contrainte sur la criticité de l'état du système. En revanche, il est possible de détecter qu'un état doit passer au niveau de criticité supérieure. C'est le but de la notion de criticité réelle d'un état du système.

Définition 4.4 (Criticité réelle d'un état)**.**
Pour détecter que l'état doit passer à la criticité suivante, il faut qu'une tâche CM ait exécuté totalement son WCET pour la criticité actuelle sans se terminer. Il faut vérifier si une tâche CM est toujours active bien qu'elle n'ait plus de temps de calcul restant.

On définit la criticité réelle d'un état S comme celle de l'état, si aucune tâche CM n'a été exécutée complètement sans avoir signalé sa complétion ; celle de l'état incrémentée d'un sinon.

$$Critical_S \stackrel{def}{=} \begin{cases} crit_S + 1 & \text{si } \exists\ \tau_i \in Active(S) \text{ t.q. } rct_S(\tau_i) = 0 \\ crit_S & \text{sinon.} \end{cases}$$

Il faut rappeler que le but de ce modélisme est de déterminer l'ordonnançabilité CM d'un système de tâches CM. Il s'agit donc de s'assurer qu'aucune tâche CM ne manque son échéance. La laxité d'une tâche CM permet de savoir durant combien d'unité de temps la tâche CM peut être oisive avant de manquer son échéance. À tout moment, si la laxité d'une tâche CM est négative, fatalement, elle ne parviendra pas à être exécutée pour son temps restant de calcul total.

Définition 4.5 (Laxité)**.**
La laxité d'une tâche CM τ_i d'un état S est :

$$Laxity_S(\tau_i) \stackrel{def}{=} at_S(\tau_i) + D_i - rct_S(\tau_i)$$

Pour obtenir la laxité d'une tâche CM, il faut commencer par acquérir le nombre d'unités de temps restantes avant l'échéance de la tâche CM. Ensuite il faut y soustraire le temps restant de calcul maximal de la tâche CM.

Cette définition de la laxité est valide pour l'ordonnancement CM, mais aussi pour le classique. En mixité critique, il est possible d'aller plus loin. En effet, la laxité ici tient compte du temps restant de calcul maximal pour le niveau de criticité actuel, mais il faut aussi, à tout moment, pouvoir exécuter les tâches CM actives pour leur pire WCET. C'est à partir de ce constat que l'idée de pire laxité a émergé.

Définition 4.6 (Pire laxité)**.**
La pire laxité d'une tâche CM τ_i dans l'état S est :

$$WorstLaxity_S(\tau_i) \stackrel{def}{=} at_S(\tau_i) + D_i - (rct_S(\tau_i) + (C_i(K) - C_i(crit_S))$$

Pour obtenir cette pire laxité, il faut reprendre la définition de la laxité et ajouter au temps restant de calcul, le temps restant de calcul qui viendrait s'ajouter dans le pire scénario.

La pire laxité a été introduite pour repérer un manquement d'échéance. Le but de l'automate construit est de déterminer si un système de tâches CM périodiques est ordonnançable. D'où, dès qu'un état où la pire laxité d'une des tâches CM est négative, l'ensemble des tâches CM n'est pas ordonnançable avec l'algorithme en question.

Il est donc possible de définir les états erronés de l'automate comme étant ceux où une tâche CM manquera son échéance.

Définition 4.7 (États erronés)**.**
Un état S est erroné si l'une des pires laxités des tâches CM est négative, on obtient alors l'ensemble :

$$Fail_\tau \stackrel{def}{=} \{S \mid \exists\, \tau_i \in \tau : WorstLaxity_S(\tau_i) < 0\}$$

Grâce à ces états erronés, nous pouvons à présent borner inférieurement at_S. En effet, puisque, dès que la pire laxité d'un état est négative : il est erroné, il n'est pas nécessaire d'inclure dans l'automate les états où $\exists\, \tau_i \in \tau\ tq\ WorstLaxity_S(\tau_i) < -1$.

Dès lors :

$$at_S(\tau_i) + D_{max} - (rct_S(\tau_i) + (C_{max}(K) - C_{max}(1)) \geq -1$$
$$at_S(\tau_i) \geq +rct_S(\tau_i) + C_{max}(K) - (D_{max} + C_{max}(1) + 1) \geq -(D_{max} + C_{max}(1) + 1)$$
$$-(D_{max} + C_{max}(1) + 1) \leq at_S(\tau_i) \leq R_{max}$$

Avec $D_{max} \stackrel{def}{=} max_i\ D_i$ et $R_{max} \stackrel{def}{=} max(max_i\ (T_i),\ max_i\ (O_i))$ pour tout $\tau_i \in \tau$.

L'automate qui se construit ici est basé sur un système de tâches CM, mais aussi sur un ordonnanceur. Il est donc nécessaire de choisir un tel ordonnanceur, c'est-à-dire une fonction sur un état du système vers une tâche CM à ordonnancer.

Définition 4.8 (Ordonnanceur)**.**
Un *ordonnanceur* monoprocesseur pour τ est une fonction $Run : States(\tau) \rightarrow 2^{\tau}$ telle que $Run(S) \subseteq Active(S)$ et $0 \leq |Run(S)| \leq 1$. De plus :

— Run est *work-conserving*, si et seulement si pour S : $|Run(S)| = min\{1, |Active(S)|\}$

— Run est *memoryless*, si et seulement si pour S_1, S_2 avec $Active(S_1) = Active(S_2)$: $\forall\ \tau_i \in Active(S_1) : (at_{S_1} = at_{S_2} \ \wedge\ rct_{S_1} = rct_{S_2} \ \wedge\ crit_{S_1} = crit_{S_2})$ implique $Run(S_1) = Run(S_2)$

L'ordonnanceur *work-conserving* ordonnancera toujours une tâche s'il y a des tâches actives dans l'état du système. Un ordonnanceur *memoryless* signifie qu'il se base uniquement sur l'état du système pour choisir la tâche la plus prioritaire. C'est-à-dire qu'il ne va pas prêter attention à ce qu'il s'est passé avant.

Il est maintenant possible de définir les transitions de l'automate. L'automate aura une transition depuis un état du système vers tous ceux qu'il peut générer. Il peut y en avoir plusieurs, car une tâche CM peut signaler ou non sa complétion.

Pour définir cet ensemble de transition, l'automate utilisera des transitions intermédiaires pour représenter différentes actions au cours d'une unité de temps.

Durant une unité de temps, une tâche CM est exécutée, ensuite elle signale ou non sa complétion et enfin, s'il y a lieu, la criticité de l'état du système peut passer au niveau supérieur.

La première transition établie est la transition d'exécution. Cette transition permet à une tâche CM de profiter de la ressource partagée pendant un coup d'horloge.

Définition 4.9 (Transition d'exécution)**.**
Soit $S = \langle at_S, rct_S, crit_S \rangle$ un état du système et Run un *ordonnanceur* pour τ. On dit que l'état du système $S^+ = \langle at_S^+, rct_S^+, crit_S^+ \rangle$ est un *successeur-exécuté* de S avec Run, noté $S \xrightarrow{Run} S^+$, si et seulement si :

— Pour tout $\tau_i \in Run(S) : rct_S^+(\tau_i) = rct_S(\tau_i) - 1$

— Pour tout $\tau_i \notin Run(S) : rct_S^+(\tau_i) = rct_S(\tau_i)$

— Pour tout $\tau_i \in \tau$:

$$at_S^+(\tau_i) = \begin{cases} 0 & \text{si } \chi_i < crit_S \\ at_S(\tau_i) - 1 & \text{si } \chi_i \geq crit_S \end{cases}$$

— $crit_S^+ = crit_S$

La tâche CM ordonnancée verra son temps restant d'exécution décrémentée d'une unité et le temps d'arrivée de toutes les tâches CM sera lui aussi décrémenté d'une unité, sauf pour les tâches CM qui ont été abandonnées. Cette transition ne modifie pas la criticité de l'état du système.

Après avoir été exécutée, une tâche CM peut signaler sa complétion. La transition de terminaison se charge de représenter cette modification dans l'état du système.

Définition 4.10 (Transition de terminaison)**.**
Soit $S = \langle at_S, rct_S, crit_S \rangle$ un état du système et $\tau' \subseteq Active(S)$ un ensemble de tâches CM actives pouvant signaler leur complétion. On dit que l'état du système $S' = \langle at'_S, rct'_S, crit'_S \rangle$ est un *successeur-τ'-terminé* de S, noté $S \xrightarrow{\tau'} S'$, si et seulement si :

- Pour tout $\tau_i \in \tau' \cup ImplicitelyDone(S)$:
  - $rct_{S'}(\tau_i) = C_i(crit_S)$
  - $at_{S'}(\tau_i) = at_S(\tau_i) + T_i$
- Pour tout $\tau_i \notin \tau' \cup ImplicitelyDone(S)$:
  - $rct_{S'}(\tau_i) = rct_S(\tau_i)$
  - $at_{S'}(\tau_i) = at_S(\tau_i)$
- $crit_{S'} = crit_S$

Cette transition se base sur un ensemble de tâches CM actives qui vont signaler leur complétion. Cet ensemble peut être vide, dans ce cas aucune tâche CM ne se termine explicitement. En plus de ces tâches CM, les tâches CM qui sont implicitement terminées seront elles aussi complétées. Comme il s'agit d'ensembles, une tâche CM peut faire partie des deux groupes et leur union ne la considérera qu'une seule fois.

Pour les tâches CM qui ne se terminent pas, rien ne change. Pour les tâches CM se complétant le temps restant de calcul est mis au WCET de la tâche CM pour le niveau de criticité de l'état. Il s'agit du temps de calcul dont la tâche CM aura besoin lors de sa prochaine arrivée. Le temps d'arrivée de la tâche CM est incrémenté de sa période, s'il devient positif alors la tâche CM ne sera pas active directement.

Enfin, il est nécessaire de fixer la transition critique. Celle-ci ajuste le niveau de criticité si une tâche CM n'a pas signalé sa complétion, bien qu'elle ait utilisé tout son temps de calcul prévu pour le niveau actuel.

Définition 4.11 (Transition critique)**.**
Soit $S = \langle at_S, rct_S, crit_S \rangle$ un état du système. On dit que l'état du système $S^C = \langle at_S^C, rct_S^C, crit_S^C \rangle$ est un *successeur-critique* de S, noté $S \xrightarrow{C} S^C$, si et seulement si :

— $crit_S^C = Critical_S$

— Pour tout $\tau_i \in \tau$:

$$rct_S^C(\tau_i) = \begin{cases} rct_S + c_i(Critical_S) - c_i(crit_S) & \text{si } X_i \geq Critical_S \\ 0 & \text{sinon.} \end{cases}$$

$$at_S^C(\tau_i) = \begin{cases} at_S & \text{si } X_i \geq Critical_S \\ 0 & \text{sinon.} \end{cases}$$

Cette transition commence par mettre à jour la criticité du système. Ensuite, elle augmente les temps restants d'exécution des tâches CM non abandonnées, s'il y a effectivement un changement de criticité. Pour ce faire, il faut ajouter le temps d'exécution établi pour le niveau supérieur, mais ne pas réallouer le temps qui a déjà été utilisé par la tâche CM.

Le temps d'arrivée n'est pas modifié, sauf si la criticité de l'état dépasse celle de certaines tâches CM. Alors, leur temps d'arrivée et leur temps d'exécution restant sont mis à zéro et elles rejoignent donc bien l'ensemble $Discarded(S^C)$. Si la criticité réelle d'un état est identique à sa criticité, cette transition ne fera aucune modification.

Il est maintenant possible de définir l'automate simulant l'ordonnancement CM de tâche CM périodique.

Définition 4.12.
Étant donné un système de tâches CM périodiques τ et un ordonnanceur Run, l'automate $\overline{A}(\tau, Run)$ est le tuple (V, E, S_0, F) où :

— $V = States(\tau)$

— $(S_1, S_2) \in E$, si et seulement s'il existe les états intermédiaires S' et $S'' \in States(\tau)$ et $\tau' \subseteq Run(S_1)$ tel que : $S_1 \xrightarrow{Run} S' \xrightarrow{\tau'} S'' \xrightarrow{C} S_2$

— $S_0 = \langle at_{S_0}, rct_{S_0}, crit_{S_0} \rangle$:

  — $at_{S_0} = O_i \ \forall \ i$

  — $rct_{S_0} = C_i(1) \ \forall \ i$

  — $crit_{S_0} = 1$

— $F = Fail_\tau$

Les états sont tous les états du système qu'il est possible de générer depuis le système de tâches CM avec une pire laxité minimum de -1.

Les transitions relient un état vers ceux dont il peut être le prédécesseur selon les trois transitions intermédiaires, les transitions d'exécution, de terminaison et critique en série pour tous les sous-ensembles τ' possibles. On remarque que les tâches CM pouvant signaler leur complétion sont celles qui ont été ordonnancées. En effet, il faut qu'une tâche CM soit exécutée pour pouvoir signaler qu'elle a terminé son travail CM. Il s'agit des tâches CM ordonnancées dans l'état initial et non dans l'état intermédiaire, après exécution.

Pour l'état initial, les temps d'arrivées sont les décalages des tâches CM. Le temps restant de calcul est le WCET pour le niveau de criticité le plus bas. Enfin, la criticité de l'état du système est la criticité initiale.

Finalement, les états erronés sont ceux qui ont une pire laxité négative.

Exemple 4.1.
Cet exemple présentera la construction partielle d'un automate défini sur un système de tâches CM périodiques.

i	O	T	D	C	χ
0	0	3	3	[2, 3]	2

TABLEAU 4.1 – le système de tâches CM utilisé pour l'Exemple 4.1.

Considérons le système de tâches CM τ présenté en Tableau 4.1. Cet ensemble contient une seule tâche CM avec échéance implicite et une criticité double.

Cet exemple présente la construction d'un automate et illustre les états intermédiaires et les transitions.

Dans la Figure 4.1, les états du système sont représentés par les nœuds. Le tuple inscrit dans le nœud est le couple $(at_S(\tau_0), rct_S(\tau_0))$ et le nombre le suivant est $crit_S$.
Les flèches continues représentent une transition intermédiaire d'un état vers un autre. Les flèches rouges représentent la transition d'exécution. Les flèches noires sont les transitions de terminaison. Enfin, les flèches bleues sont les transitions critiques.

Le nœud initial est celui avec la flèche orpheline pointée sur lui. Cet état du système correspond bien à S_0 tel qu'établit en Définition 4.12, $at_{S_0}(\tau_0) = O_0,\ rct_{S_0}(\tau_0) = C_0(1),\ crit_{S_0} = 1$.

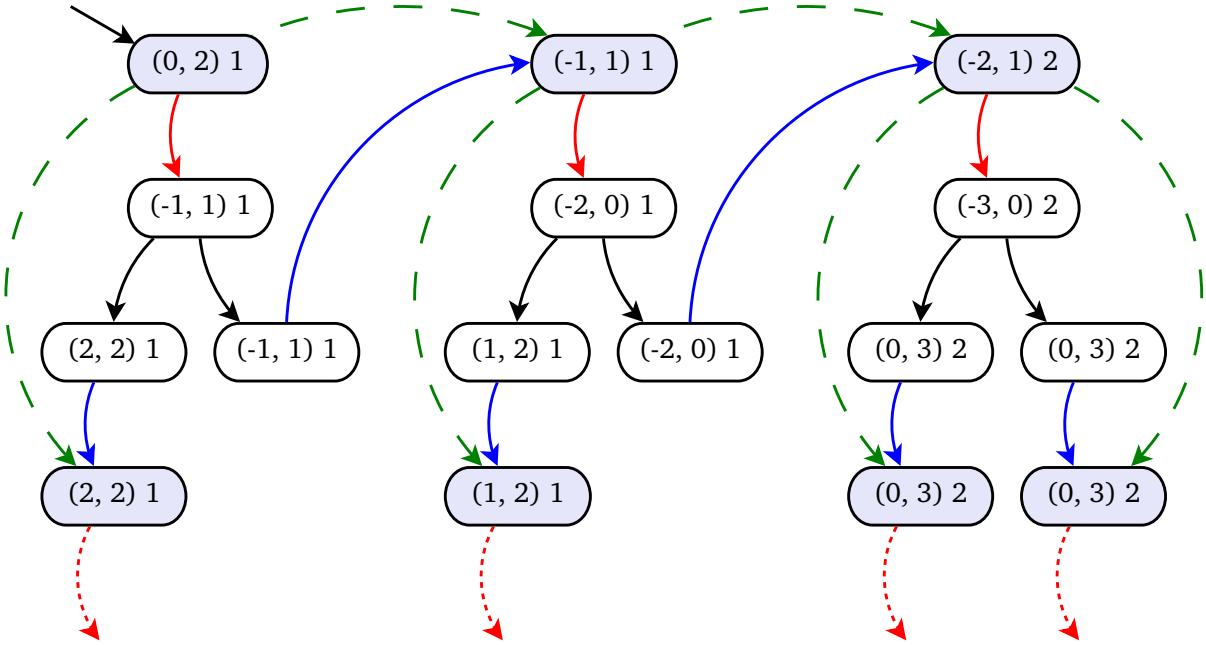

FIGURE 4.1 – Construction partielle de l'automate basé sur τ en Tableau 4.1 avec et Run = *EDF-VD*

Depuis l'état initial, s'en suit une transition d'exécution où l'unique tâche CM τ_0 est ordonnancée. La tâche CM τ_0 est bel et bien active, car $at_{S_0}(\tau_0) = 0 \ \wedge \ rct_{S_0}(\tau_0) > 0$ par Définition 4.2. Comme attendu par Définition 4.9, le temps restant de calcul et le temps d'arrivée sont décrémentés d'une unité. Ensuite, il y a deux transitions de terminaison possibles. Soit la tâche CM τ_0 signale sa complétion, soit elle ne le fait pas. Si elle la signale, il faut suivre la flèche de gauche. Dans cet état nouvellement atteint, le temps restant de calcul est remis à son maximum pour la criticité du système et le temps d'arrivée est placé dans le futur, en corrélation avec la période de la tâche CM. En revanche, si la tâche CM ne signale pas sa complétion, l'état est alors identique au précédent. Il s'en suit la transition critique, depuis ces deux derniers états, elle ne change rien. En effet, aucun de ces états ne déclenche une criticité plus élevée, car aucune tâche CM n'a été exécutée jusqu'au bout de son WCET sans signaler sa complétion. Une fois cette série de trois transitions passées, il faut relier les états entre eux, c'est ce que représente la flèche discontinue verte.

Pour l'état $\langle(2, 2)1\rangle$, le temps d'arrivée de τ_0 va décrémenter, de par la transition d'exécution. Les autres transitions ne modifieront rien. Finalement, le temps d'arrivée de la tâche CM τ_0 arrivera jusqu'à zéro et il s'agira alors de l'état initial.

L'état $\langle(-1, 1)1\rangle$ va se comporter comme l'état initial, sauf lors de la transition critique. En effet, si la tâche CM τ_0 ne signale pas sa complétion, elle déclenchera un passage à criticité supérieure.

Si τ_0 s'est effectivement terminé, l'état du système se comportera de la même manière que si elle s'était signalée complétée lors de son exécution précédente sauf que le prochain temps d'arrivée sera déjà décrémenté d'une unité de plus.

Si τ_0 n'est s'est pas déclarée comme finie, alors il est nécessaire d'indiquer la criticité du système comme étant supérieure et d'ajouter le temps de calcul nécessaire en plus. C'est ainsi que l'état $\langle(-2,1)2\rangle$ est atteint. Celui-ci va alors ordonnancer τ_0 et son temps restant de calcul sera alors nul. Il s'en suit la transition de terminaison. Dans les deux cas, la tâche CM se termine. Dans cette configuration, $ImplicitelyDone(S) = \{\tau_0\}$ par Définition 4.3 et $\tau' \subseteq \{\tau_0\}$ et donc, dans tous les cas, $\tau' \cup ImplicitelyDone(S) = \{\tau_0\}$.

Le reste de l'automate n'est pas présenté ici, mais il devra être construit, c'est ce que représentent les flèches rouges en pointillé. Au final, il faut remarquer que dans l'automate, seuls les états avec un fond gris sont atteints, et sont reliés par les transitions discontinues vertes.

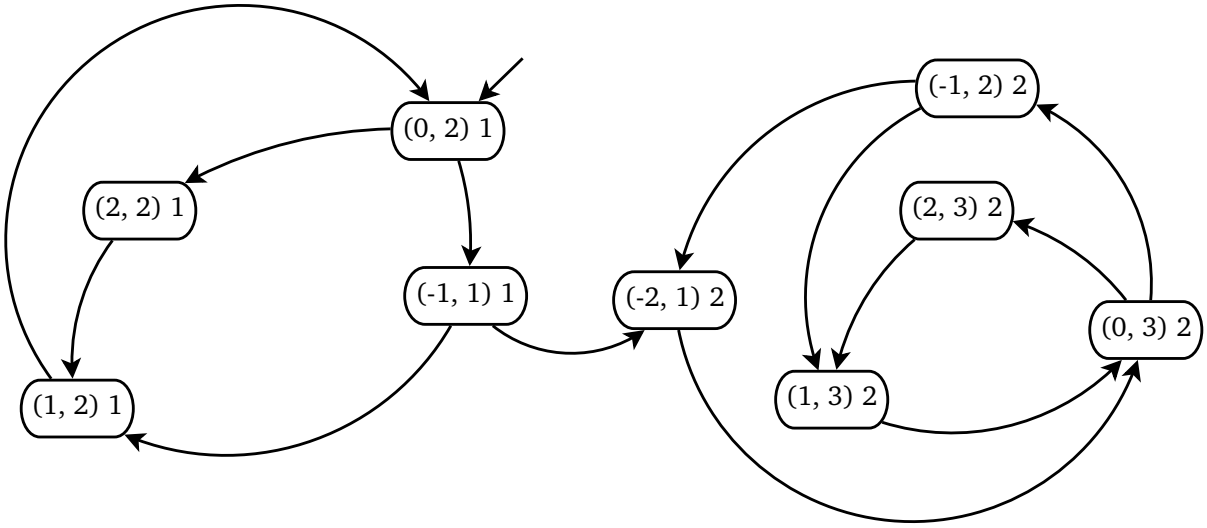

FIGURE 4.2 – Automate basé sur τ en Tableau 4.1 avec et Run = *EDF-VD*

La Figure 4.2 présente l'automate complètement construit, sans les états intermédiaires. La scission entre les deux niveaux de criticité est flagrante.

Maintenant que le principe de l'automate a été illustré clairement, il possible de prendre pour exemple un nouveau système de tâches CM.

Exemple 4.2.
Cet exemple illustre le comportement de tâches CM avec des criticités différentes et non ordonnançables. La figure Figure 4.3 construit partiellement l'automate basé sur le

i	O	T	D	C	χ
0	0	3	3	[2, 3]	2
1	0	3	3	[2, 2]	1

TABLEAU 4.2 – le système de tâches CM utilisé pour l'Exemple 4.2.

système de tâches CM en Tableau 4.2. Dans celui-ci, les nœuds comptent deux couples (at_S, rct_S), le premier est pour la tâche CM τ_0 et le second pour τ_1.

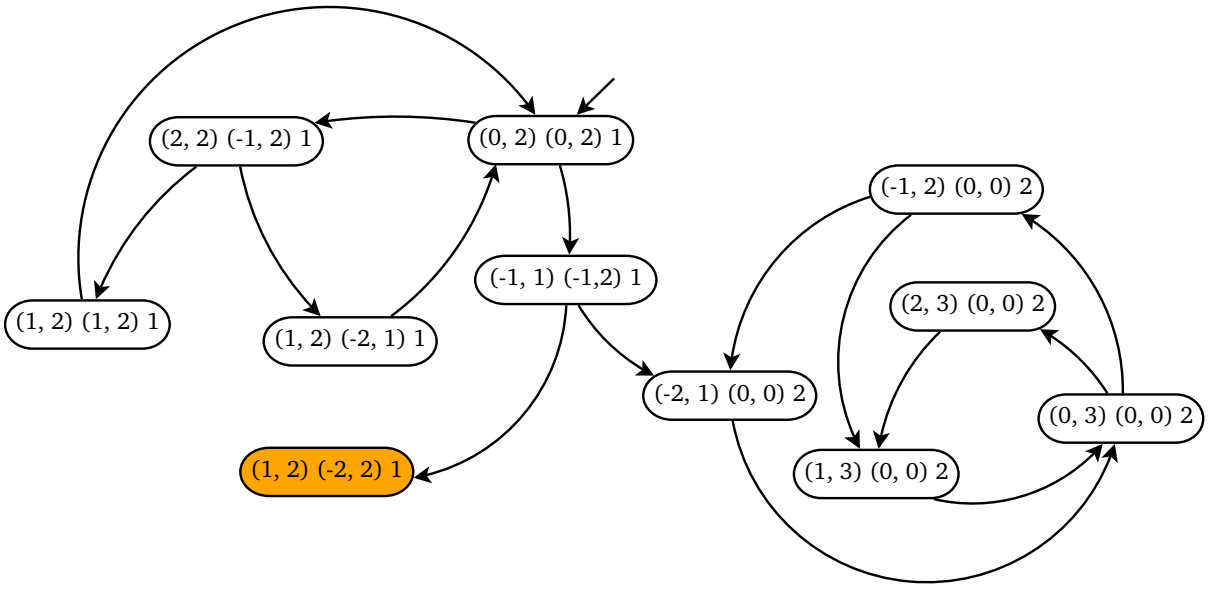

FIGURE 4.3 – Automate basé sur τ en Tableau 4.2 avec et Run = *EDF-VD*

Dans ce nouvel automate, deux nouveaux comportements sont présents.

Premièrement, quand l'automate est en criticité supérieure, la tâche CM τ_1 ayant une criticité inférieure à celle du système n'est plus considérée. En effet, τ_1 se retrouve dans l'ensemble $Discarded(S)$.

Ensuite, il y a un état erroné, représenté en orange dans le graphe. Il est erroné, car l'une des pires laxités de ses tâches CM est négative. Il s'agit de la tâche CM τ_1, $WorstLaxity_{S_{Fail}}(\tau_1) = at_S(\tau_1) + D_1 - (rct_S(\tau_1) + (C_1(K) - C_1(crit_{S_{Fail}})) = -2 + 3 - (2 + 2 - 2) = -1$. le système de tâches CM τ n'est effectivement pas ordonnançable. C'est détectable en regardant la faisabilité CM de τ, $U_{\tau,1}(1) + U_{\tau,2}(1) = 4/3 > 1$.

La création d'un automate pour représenter l'ordonnancement CM de tâches CM périodiques est maintenant connue. Ce modèle est intéressant et est une bonne entrée en la matière. Cependant, supposer que les tâches CM sont périodiques est une forte abstraction de la réalité. C'est pourquoi la section suivante propose un modélisme pour l'ordonnancement d'un système de tâches CM sporadiques.

4.2 Système de tâches CM sporadiques

Cette section présente la modélisation de l'ordonnancement de tâches CM sporadiques.

Il faut commencer par la représentation des états du système, qui seront les états de l'automate. Dans ce nouveau modèle, une tâche CM est caractérisée par son temps de restant de calcul, le temps d'arrivée minimum de sa prochaine émission et sa complétion. Le système est aussi défini par sa criticité.

Définition 4.13 (État du système)**.**
Soit $\tau = \tau_1, \tau_2, \tau_3...$ un système de tâches CM sporadiques, l'état du système de τ est le tuple $S \stackrel{def}{=} \langle nat_S, rct_S, done_S, crit_S \rangle$ avec

- nat_S, une fonction représentant le temps d'arrivée minimum du prochain travail CM d'une tâche CM : $\tau \to \mathbb{N},\ nat_S(\tau_i) \leq R_{max}$ avec $R_{max} \stackrel{def}{=} max(max_i\,(T_i),\ max_i\,(O_i))$
- rct_S, le temps de calcul restant du travail CM généré par une tâche CM : $\tau \to \mathbb{N},\ 0 \leq rct_S(\tau_i) \leq C_{max}$ avec $C_{max} \stackrel{def}{=} max_{i,j}\ C_i(j)$
- $done_S$, la complétion d'un travail CM : $\tau \to \{True, False\}$
- $crit_S$, le niveau de criticité actuel du scénario, $\in \mathbb{N},\ 1 \leq crit_S \leq K$

Le temps de calcul restant est borné entre 0 et le maximum des pires WCETs des tâches CM. La criticité du système est comprise entre 1 et K, la criticité du système de tâches CM à ordonnancer. Les complétions des tâches CM sont des booléens. Le prochain temps d'arrivée est borné supérieurement, par la période maximale dans le système de tâches CM. Comme nat n'a pas de plancher, l'automate semble infini. Cependant, comme l'objectif est de détecter la présence d'états qui font échouer un ordonnancement CM, il n'est pas nécessaire d'explorer tous ces états. Cette observation capitale sera prouvée par la suite.

On définira tous les états du système possibles, selon la définition ci-dessus, en partant d'un système de tâches CM τ par $States(\tau)$.

Il aussi apparent que des états seront contradictoires. Par exemple, il est possible d'avoir une tâche CM avec un temps restant de calcul positif, alors qu'elle est indiquée comme complétée. Ce genre d'états ne seront pas atteints depuis l'état initial et les transitions de l'automate.

Les états de l'automate étant définis, la prochaine étape est de fixer ses transitions. Pour ce faire, il est nécessaire de définir certaines notions auxiliaires sur lesquelles les transitions seront basées.

En premier lieu, il est souhaitable de savoir quelles sont les tâches CM actives dans l'état du système. Il s'agit des tâches CM qui ont besoin de temps de calcul.

Définition 4.14 (Tâches CM actives)**.**
Une tâche CM est active dans l'état S si elle ne s'est pas signalée comme complétée dans celui-ci.

$$Active(S) \stackrel{def}{=} \{\tau_i \mid done_S(\tau_i) = False\}$$

Ici les tâches CM actives sont simplement celles qui sont définies comme telles dans l'état de par leur attribut $done$. Il n'est pas possible de considérer simplement le temps de calcul restant, et dire qu'il doit être positif pour qu'une tâche CM soit active. En effet, ce n'est pas parce que le temps restant de calcul d'une tâche CM est nul qu'elle est terminée. La tâche CM doit signaler sa complétion et ce temps restant de calcul peut être nul pour la criticité actuelle, mais pas pour la suivante.

On profite de cette définition pour fixer l'ensemble des tâches CM abandonnées, les tâches CM qui ne sont plus considérées, car elles ont une criticité inférieure à celle du scénario actuel :

$$Discarded(S) \stackrel{def}{=} \{\tau_i \mid \chi_i < crit_s\}$$

Et on fixe aussi l'ensemble des tâches CM oisives :

$$Idle(S) \stackrel{def}{=} \{\tau_i \mid done_S(\tau_i) = True \ \wedge \ \chi_i \geq crit_s\}$$

Il s'agit simplement des tâches CM non abandonnées et non actives.

Bien que les tâches CM se terminent sur signal, elles ne dépassent jamais leur pire WCET. C'est-à-dire qu'une tâche CM qui a été exécutée pour un temps égal à sa pire estimation se signalera toujours comme étant finie.

Définition 4.15 (Tâches CM implicitement terminées)**.**
Une tâche CM active qui aurait été exécutée pour un temps égal à son pire WCET sera implicitement terminée. Il faut donc s'assurer que la tâche CM est active et non abandonnée, qu'elle n'a plus de temps de calcul restant et que l'estimation qui a été utilisée pour son temps restant de calcul est équivalente à la pire d'entre elles.

$$ImplicitelyDone(S) \stackrel{def}{=} \{\tau_i \mid rct_S(\tau_i) = 0 \ \wedge \ done_S(\tau_i) = False \ \wedge \ C_i(crit_S) = C_i(\chi_i)\}$$

L'automate se base sur des tâches CM sporadiques, il sera donc utile de savoir quelles tâches CM peuvent soumettre un travail CM dans un certain état du système.

Définition 4.16 (Tâches CM éligibles à la soumission d'un travail CM)**.**
Une tâche CM est éligible pour soumettre un travail CM dans un état S si elle n'est pas active, son prochain temps d'arrivée est nul ou négatif et elle n'a pas été abandonnée à cause de sa criticité.

$$Eligibile(S) \stackrel{def}{=} \{\tau_i \mid rct_S(\tau_i) = 0 \ \wedge \ nat_S(\tau_i) \leq 0 \ \wedge \ done_S(\tau_i) \ \wedge \ crit_S \geq \chi_i\}$$

Les états du système ne sont pas tous des états qui seront réellement possibles durant l'ordonnancement d'un système de tâches CM. Par exemple il n'y a aucune contrainte sur la criticité de l'état du système. En revanche, il est possible de détecter qu'un état doit passer au niveau de criticité supérieure. C'est le but la notion de criticité réelle d'un état du système.

Définition 4.17 (Criticité réelle d'un état)**.**
Pour détecter un passage à criticité supérieure, il faut qu'une tâche CM ait été exécutée complètement pour son WCET actuel sans avoir signalé sa complétion.

La criticité réelle d'un état S est celle de l'état si aucune tâche CM n'a été exécutée complètement sans avoir signalé sa complétion, celle de l'état incrémentée d'un sinon.

$$Critical_S \stackrel{def}{=} \begin{cases} crit_S + 1 & \text{si } \exists\ \tau_i \in Active(S) \text{ t.q. } rct_S(\tau_i) = 0 \\ crit_S & \text{sinon.} \end{cases}$$

Il faut rappeler que le but de ce modélisme est de déterminer l'ordonnançabilité CM d'un système de tâches CM. Il s'agit donc de s'assurer qu'aucune tâche CM ne manque son échéance. La laxité d'une tâche permet de savoir durant combien d'unités de temps la tâche peut être oisive avant de manquer son échéance.

À tout moment, si la laxité d'une tâche est négative, fatalement, elle ne parviendra pas à être exécutée pour son temps restant de calcul total.

Définition 4.18 (Laxité)**.**
La laxité d'une tâche CM τ_i dans l'état S est :

$$Laxity_S(\tau_i) \stackrel{def}{=} nat_S(\tau_i) - T_i + D_i - rct_S(\tau_i)$$

Pour obtenir la laxité d'une tâche CM dans l'état du système, il faut, en premier lieu, obtenir le nombre d'unités de temps qu'il reste avant son échéance. Pour ce faire, il est nécessaire de savoir quand la tâche CM a été émise, ce qui s'obtient par la différence entre le prochain temps d'arrivée et la période de la tâche CM. Par la suite, il suffit d'y ajouter l'échéance relative de la tâche CM, pour finalement en soustraire le temps de calcul restant.

Cette définition de la laxité est valide pour l'ordonnancement CM, mais aussi pour le classique. En mixité critique, il est possible d'aller plus loin. En effet, la laxité ici tient compte du temps restant de calcul maximal pour le niveau de criticité actuel, mais il faut aussi à tout moment pouvoir exécuter les tâches CM actives pour leur pire WCET. C'est à partir de ce constat que l'idée de pire laxité a émergé.

Définition 4.19 (Pire laxité)**.**
La pire laxité d'une tâche CM τ_i dans l'état S est :

$$WorstLaxity_S(\tau_i) \stackrel{def}{=} nat_S(\tau_i) - T_i + D_i - (rct_S(\tau_i) + (C_i(K) - C_i(crit_S))$$

La définition reprend celle de la laxité classique, sauf qu'elle ajoute au temps restant de calcul, celui qui pourrait s'y additionner en cas de criticité supérieure.

Comme il est maintenant possible de savoir quand un ordonnancement CM n'est pas possible selon un ordonnanceur, et que le but est justement de déterminer si c'est le cas, la définition des états erronés se construit facilement.

Définition 4.20 (États erronés)**.**
Un état S est erroné si l'une des pires laxités des tâches CM est négative, on obtient alors l'ensemble :

$$Fail_\tau \stackrel{def}{=} \{S \mid \exists\, \tau_i \in \tau : WorstLaxity_S(\tau_i) < 0\}$$

Grâce à ces états erronés, il est à présent possible de borner inférieurement nat_S. En effet, puisque dès que la pire laxité d'un état est négative, il est erroné. Dans ce cas, il n'est pas nécessaire d'inclure dans l'automate les états où $\exists\, \tau_i \in \tau$ t.q. $WorstLaxity_S(\tau_i) < -1$.

Dès lors :

$$nat_S(\tau_i) - T_{min} + D_{max} - (rct_S(\tau_i) + (C_{min}(K) - C_{max}(1)) \geq -1$$
$$nat_S(\tau_i) \geq (rct_S(\tau_i) + (C_{min}(K) - C_{max}(1))) - (D_{max} + 1) + T_{min} \geq$$
$$T_{min} - (D_{max} + 1) + (C_{min}(K) - C_{max}(1))$$
$$T_{min} - (D_{max} + 1) + (C_{min}(K) - C_{max}(1)) \leq nat_S(\tau_i) \leq R_{max}$$

avec $D_{max} \stackrel{def}{=} max_i\ D_i$, $C(1)_{max} \stackrel{def}{=} max_i\ C_i(1)$, $C(K)_{min} \stackrel{def}{=} min_i\ C_i(K)$, $T_{min} \stackrel{def}{=} min_i\ (T_i)$ et $R_{max} \stackrel{def}{=} max(max_i\ (T_i), max_i\ (O_i))$

L'automate qui se construit ici est basé sur un système de tâches CM, mais aussi sur un ordonnanceur. Il est donc nécessaire de choisir un tel ordonnanceur, c'est-à-dire une fonction sur un état du système vers une tâche CM à ordonnancer.

Définition 4.21 (Ordonnanceur)**.**
Un *ordonnanceur* monoprocesseur pour τ est une fonction $Run : State(\tau) \rightarrow 2^\tau$ telle que $Run(S) \subseteq Active(S)$ et $0 \leq |Run(S)| \leq 1$.
De plus :

- Run est *work-conserving*, si et seulement si pour S : $|Run(S)| = min\{1, |Active(S)|\}$
- Run est *memoryless*, si et seulement si pour S_1, S_2 avec $Active(S_1) = Active(S_2)$: $\forall\, \tau_i \in Active(S_1) : (nat_{S_1} = nat_{S_2}\ \wedge\ rct_{S_1} = rct_{S_2}\ \wedge\ done_{S_1} = done_{S_2}\ \wedge\ crit_{S_1} = crit_{S_2})$ implique $Run(S_1) = Run(S_2)$

L'ordonnanceur *work-conserving* ordonnancera toujours une tâche s'il y a des tâches actives dans l'état du système. Un ordonnanceur *memoryless* signifie qu'il se base uniquement sur l'état du système pour choisir la tâche la plus prioritaire. C'est-à-dire qu'il ne va pas prêter attention à ce qu'il s'est passé avant.

Il est maintenant possible de définir les transitions de l'automate. L'automate aura une transition depuis un état du système vers tous ceux qu'il peut générer. Il peut y en avoir plusieurs, car une tâche CM active peut signaler ou non sa complétion et une tâche CM éligible à la soumission d'un travail CM peut générer ou non un travail CM.

Pour définir cet ensemble de transitions, l'automate utilisera des transitions intermédiaires pour représenter différentes actions au cours d'une unité de temps.

Durant une unité de temps, une tâche CM est exécutée, ensuite elle signale on non sa complétion et, s'il y a lieu, la criticité de l'état du système peut passer au niveau supérieur. Enfin, les tâches CM qui sont éligibles à la soumission d'un travail CM peuvent décider d'émettre un travail CM. La première transition établie est la transition d'exécution. Cette transition permet à une tâche CM de profiter de la ressource partagée pendant un coup d'horloge

Définition 4.22 (Transition d'exécution)**.**
Soit $S = \langle nat_S, rct_S, done_S, crit_S \rangle$ un état du système et Run un *ordonnanceur* pour τ. On dit que l'état du système $S^+ = \langle nat_S^+, rct_S^+, done_S^+, crit_S^+ \rangle$ est un *successeur-exécuté* de S avec Run, noté $S \xrightarrow{Run} S^+$, si et seulement si :

— Pour tout $\tau_i \in Run(S) : rct_S^+(\tau_i) = rct_S(\tau_i) - 1$

— Pour tout $\tau_i \notin Run(S) : rct_S^+(\tau_i) = rct_S(\tau_i)$

— Pour tout $\tau_i \in \tau$:

$$nat_S^+(\tau_i) = \begin{cases} max(nat_S(\tau_i) - 1, 0) & si\ \tau_i \notin Active(S) \\ nat_S(\tau_i) - 1 & si\ \tau_i \in Active(S) \end{cases}$$

— $done_S^+ = done_S$

— $crit_S^+ = crit_S$

La tâche CM ordonnancée verra son temps restant d'exécution décrémentée d'une unité.

Les tâches CM actives verront leur prochain temps d'arrivée décrémenté d'une unité. Les tâches CM oisives verront, elles aussi, leur prochain temps d'arrivée décrémenté, mais si celui-ci devient négatif, il est fixé à 0. En effet, une fois que le temps d'arrivée d'une tâche CM oisive est nul, elle est éligible à la soumission d'un travail CM et donc il n'est pas utile de continuer à le décrémenter. Ici il n'est pas nécessaire de dissocier les tâches CM abandonnées et celles toujours en course. C'est le cas, car les tâches CM abandonnées seront marquées comme terminées.

Cette transition ne modifie pas la criticité de l'état du système ni la complétion des tâches CM.

Après avoir été exécutée, une tâche CM peut signaler sa complétion. La transition de terminaison se charge de représenter cette modification dans l'état du système.

Définition 4.23 (Transition de terminaison)**.**
Soit $S = \langle nat_S, rct_S, done_S, crit_S \rangle$ un état du système et $\tau^T \subseteq Active(S)$ un ensemble de tâches CM actives pouvant signaler leur complétion. On dit que l'état du système $S^T = \langle nat_S^T, rct_S^T, done_S^T, crit_S^T \rangle$ est un *successeur-τ^T-terminé* de S, noté $S \xrightarrow{\tau^T} S^T$, si et seulement si :

- Pour tout $\tau_i \in \tau^T \cup ImplicitelyDone(S)$:
  - $rct_S^T(\tau_i) = 0$
  - $done_S^T(\tau_i) = True$
- Pour tout $\tau_i \notin \tau^T \cup ImplicitelyDone(S)$:
  - $rct_S^T(\tau_i) = rct_S(\tau_i)$
  - $done_S^T(\tau_i) = done_S(\tau_i)$
- $nat_S^T = nat_S$
- $crit_S^T = crit_S$

Cette transition se base sur un ensemble de tâches CM actives qui vont signaler leurs complétions. Cet ensemble peut être vide, dans ce cas aucune tâche CM ne se termine explicitement. En plus de ces tâches CM, les tâches CM qui sont implicitement terminées seront elles aussi complétées. Comme il s'agit d'ensembles, une tâche CM peut faire partie des deux groupes et leur union ne la considérera qu'une seule fois.

Pour les tâches CM qui ne se terminent pas, rien ne change. Pour les tâches CM se complétant, elles sont signalées comme telles. De plus, leur temps restant de calcul est fixé à zéro. Le prochain temps d'arrivée reste le même.
Dans les deux cas, la transition de complétion ne modifie pas la criticité de l'état.

Ensuite, il est nécessaire de fixer la transition critique. Celle-ci ajuste le niveau de criticité, si une tâche CM n'a pas signalé sa complétion, bien qu'elle ait utilisé tout son temps de calcul prévu au niveau actuel.

Définition 4.24 (Transition critique)**.**
Soit $S = \langle nat_S, rct_S, done_S, crit_S \rangle$ un état du système. On dit que l'état du système $S^C = \langle nat_S^C, rct_S^C, done_S^C, crit_S^C \rangle$ est un *successeur-critique* de S, noté $S \xrightarrow{C} S^C$, si et seulement si :

— $crit_S^C = Critical_S$

— Pour tout $\tau_i \in \tau$:

$$rct_S^C(\tau_i) = \begin{cases} rct_S(\tau_i) + c_i(Critical_S) - c_i(crit_S) & si\ X_i \geq Critical_S \wedge \tau_i \in Active(S) \\ rct_S(\tau_i) & si\ X_i \geq Critical_S \wedge \tau_i \notin Active(S) \\ 0 & \text{sinon.} \end{cases}$$

$$nat_S^C(\tau_i) = \begin{cases} nat_S(\tau_i) & si\ X_i \geq Critical_S \\ 0 & \text{sinon.} \end{cases}$$

$$done_S^C(\tau_i) = \begin{cases} done_S(\tau_i) & si\ X_i \geq Critical_S \\ True & \text{sinon.} \end{cases}$$

Cette transition commence par mettre à jour la criticité du système. Ensuite, elle augmente les temps restants d'exécution des tâches CM actives s'il y a effectivement un changement de criticité. Pour ce faire, il faut ajouter le temps d'exécution établi pour le niveau supérieur, mais ne pas réallouer le temps qui a déjà été utilisé par la tâche CM.

Les prochains temps d'arrivées et les complétions des tâches CM ne sont pas modifiés, sauf si la criticité dépasse celle de certaines tâches CM. Alors, leurs prochains temps d'arrivées et leur temps d'exécution restant sont mis à zéro ; de plus, elles sont signalées comme terminées.

Si la criticité réelle d'un état est identique à sa criticité, cette transition ne fera aucune modification.

Pour terminer, les tâches CM éligibles à la soumission d'un travail CM peuvent en émettre un. C'est la transition de requête qui représente ces générations de travaux CM.

Définition 4.25 (Transition de requête)**.**
Soit $S = \langle nat_S, rct_S, done_S, crit_S \rangle$ un état du système et $\tau^R \subseteq Eligible(S)$ un ensemble de tâches CM éligibles. On dit que l'état du système $S^R = \langle nat_S^R, rct_S^R, done_S^R, crit_S^R \rangle$ est un *successeur-τ^R-requête* de S, noté $S \xrightarrow{\tau^R} S^R$, si et seulement :

- Pour tout $\tau_i \in \tau^R$:
  - $nat_S(\tau_i) + T_i \leq nat_S^R(\tau_i) \leq T_i$
  - $rct_S^R(\tau_i) = C_i(crit_S)$
  - $done_S^R(\tau_i) = False$
- Pour tout $\tau_i \notin \tau^R$:
  - $nat_S^R(\tau_i) = nat_S(\tau_i)$
  - $rct_S^R(\tau_i) = rct_S(\tau_i)$
  - $done_S^R(\tau_i) = done_S(\tau_i)$
- $crit_S^R = crit_S$

Cette transition se base sur un ensemble de tâches CM qui vont générer un travail CM. Cet ensemble est un sous-ensemble de tâches CM éligibles à la soumission d'un travail CM. La transition ne modifie que les tâches CM qui souhaitent émettre un travail CM. Celles-ci sont signalées comme non complétées, leur temps restant de calcul est placé à leur WCET pour la criticité de l'état du système. Enfin, leur prochain temps d'arrivée est compris entre l'addition du prochain temps d'arrivée du travail CM généré ici et de la période de la tâche CM, et juste la période de la tâche CM.

Il faut se rappeler que pour qu'une tâche CM soit éligible à la soumission d'un travail CM, elle doit avoir un prochain temps d'arrivée nul ou négatif. Si le temps d'arrivée est nul, cela signifie que la tâche CM a terminé de compléter son travail CM précédent avant la génération du suivant. Dans ce cas, il n'y a qu'un seul successeur. En revanche, si le travail CM précédent s'est terminé après le temps d'arrivée minimal du prochain travail CM, alors il est nécessaire de simuler que ce travail CM soit arrivé au premier instant où il pouvait être émis ainsi que les suivants, en générant un voisin à l'état en question pour chaque temps possible.

Pour un état S et $\tau^R \subseteq Eligible(S)$ un ensemble, on note l'ensemble de ses *successeurs-τ^R-requête* S^{τ^R}.

Il est maintenant possible de définir l'automate simulant l'ordonnancement CM de tâches CM sporadiques.

Définition 4.26.
Étant donné un système de tâches CM sporadiques τ et un ordonnanceur Run, l'automate $\overline{A}(\tau, Run)$ est le tuple (V, E, S_0, F) où :

- $V = States(\tau)$
- $(S_1, S_2) \in E$, si et seulement s'il existe les états intermédiaires S^+, S^T et $S^C \in States(\tau)$ et $\tau^T \subseteq Run(S_1), \tau^R \subseteq Eligilbe(S^C)$ tel que :
  $S_1 \xrightarrow{Run} S^+ \xrightarrow{\tau^T} S^T \xrightarrow{C} S^C \xrightarrow{\tau^R} S_2$
- $S_0 = (nat_{S_0}, rct_{S_0}, done_{S_0}, crit_{S_0})$:
  - $nat_{S_0}(\tau_i) = O_i \; \forall \; i$
  - $rct_{S_0}(\tau_i) = 0 \; \forall \; i$
  - $done_{S_0}(\tau_i) = True \; \forall \; i$
  - $crit_{S_0} = 1$
- $F = Fail_\tau$

Les états sont tous les états du système qu'il est possible de générer.

Les transitions relient un état vers ceux dont il peut être le prédécesseur selon les quatre transitions intermédiaires, les transitions d'exécution, de terminaison, critique et de requête en série pour tous les sous-ensembles τ^T et τ^R possibles. On remarque que les tâches CM pouvant signaler leur complétion sont celles qui ont été ordonnancées. En effet, il faut qu'une tâche CM soit exécutée pour pouvoir signaler qu'elle a terminé son travail CM. Il s'agit des tâches CM ordonnancées dans l'état initial et non dans l'état intermédiaire, après exécution.

Pour l'état initial, les prochains temps d'arrivées sont les décalages des tâches CM. Les temps restants de calcul sont nuls, car il faut qu'une tâche CM fasse une requête pour générer un travail CM et réserver du temps de calcul pour lui. Il en va de même pour la complétion des tâches CM signalée comme vraie. Enfin, la criticité de l'état du système est la criticité initiale.

Finalement, les états erronés sont ceux qui ont une pire laxité négative.

Exemple 4.3.
Cet exemple présentera la construction partielle d'un automate défini sur un système de tâches CM sporadiques. Considérons l'ensemble tâche CM τ présenter en Tableau 4.3. Cet ensemble contient une seule tâche CM avec échéance sous requête et une criticité double.

i	O	T	D	C	χ
0	0	3	3	[2, 3]	2

TABLEAU 4.3 – le système de tâches CM utilisé pour l'Exemple 4.3.

Cet exemple présente la construction d'un automate et illustre les états intermédiaires et les transitions.

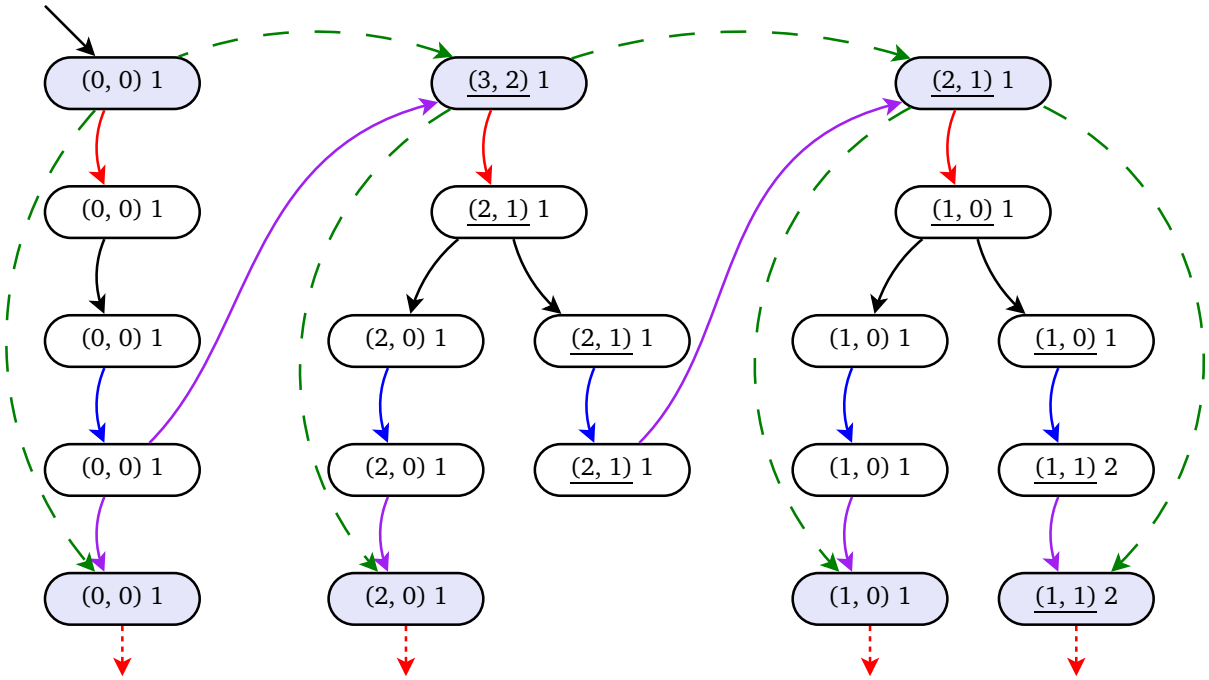

FIGURE 4.4 – Création des transitions de l'automate basé sur τ en Tableau 4.3 avec et Run = *EDF-VD*

Dans la Figure 4.4, les états du système sont représentés par les nœuds. Le tuple inscrit dans le nœud est le couple $(nat_S(\tau_0), rct_S(\tau_0))$ et le nombre le suivant est $crit_S$. De plus, le couple $(nat_S(\tau_0), rct_S(\tau_0))$ est souligné si la tâche CM τ_0 n'est pas finie, autrement il n'est pas souligné.

Les flèches continues représentent une transition intermédiaire d'un état vers un autre. Les flèches rouges représentent la transition d'exécution. Les flèches noires sont les transitions de terminaison. Les flèches bleues sont les transitions critiques. Enfin, les flèches mauves sont les transitions de requête.

Le nœud initial est celui avec la flèche orpheline pointée sur lui. Cet état du système correspond bien à S_0 tel qu'établi en Définition 4.26, $nat_{S_0}(\tau_0) = O_0,\ rct_{S_0}(\tau_0) = 0,\ crit_{S_0} = 1, done_{S_0}(\tau_0) = True$.

Depuis l'état initial, rien ne se passe jusqu'à la transition de requête. Ce comportement est normal, car, à ce moment-là, aucune tâche CM n'est active. Lors de la transition de requête, la tâche CM τ_0 est éligible. Deux configurations sont alors possibles, soit la tâche CM ne souhaite pas générer un travail CM et, dans ce cas, l'état du système reste identique, soit elle émet un travail CM. Si la tâche CM a généré un travail CM, le nœud mène à l'état $\langle\underline{(3,2)}1\rangle$, respectant bien la définition de la transition de requête. Le temps restant de calcul est fixé par le WCET de la tâche CM pour la criticité de l'état, le prochain temps d'arrivée est la période, et la tâche CM est active.

Une fois cette série de quatre transitions passée, il faut relier les états entre eux, c'est ce que représentent les flèches discontinues vertes.

Depuis l'état où la tâche CM a soumis son premier travail CM, il s'en suit une transition d'exécution où l'unique tâche CM τ_0 est ordonnancée. La tâche CM τ_0 est bel et bien active, car $done(\tau_0) = False$ par Définition 4.14. Comme attendu par Définition 4.22, le temps restant de calcul et le temps d'arrivée sont décrémentés d'une unité. Ensuite, il y a deux transitions de terminaison possible. Soit la tâche CM τ_0 signale sa complétion, soit elle ne le fait pas. Si elle la signale, il faut suivre la flèche de gauche. Dans cet état nouvellement atteint, le temps restant de calcul est remis à zéro et le prochain temps d'arrivée reste identique. En revanche, si la tâche CM ne signale pas sa complétion, l'état est alors identique au précédent. Il s'en suit la transition critique, depuis ces deux derniers états, elle ne change rien. En effet, aucun de ces états ne déclenche une criticité plus élevée, car aucune tâche CM n'a été exécutée jusqu'au bout de son WCET sans signaler sa complétion. Enfin, les deux états doivent passer par la transition de requête, mais aucune tâche CM n'est éligible et donc elles ne modifient pas les deux descendants.

Pour l'état $\langle(2,0)1\rangle$, le prochain temps d'arrivée de τ_0 va être décrémenté, de par la transition d'exécution. Les autres transitions ne modifieront rien. Finalement, le prochain temps d'arrivée de la tâche CM τ_0 arrivera jusqu'à zéro et il s'agira alors de l'état initial.

L'état $\langle\underline{(2,1)}1\rangle$ va se comporter comme l'état $\langle\underline{(3,2)}1\rangle$, sauf lors de la transition critique. En effet, si la tâche CM τ_0 ne signale pas sa complétion, elle déclenchera un passage à criticité supérieure.

Si τ_0 s'est effectivement terminée, l'état du système se comportera de la même manière que si elle s'était signalée complètee lors de son exécution précédente, sauf que le prochain temps d'arrivée sera déjà décrémenté d'une unité de plus.

Il faut remarquer que les deux *successeurs-terminés* mènent à des états où le couple $nat_S(\tau_0), rct_S(\tau_0)$ est identique. C'est la caractéristique $done_S(\tau_0)$ qui permet de savoir si la tâche CM s'est complétée.

Si τ_0 n'est s'est pas déclarée comme finie, alors il est nécessaire d'indiquer la criticité du système comme étant supérieure et d'ajouter le temps de calcul requis en plus. C'est ainsi que l'état $\langle \underline{(1,1)}2 \rangle$ est atteint.

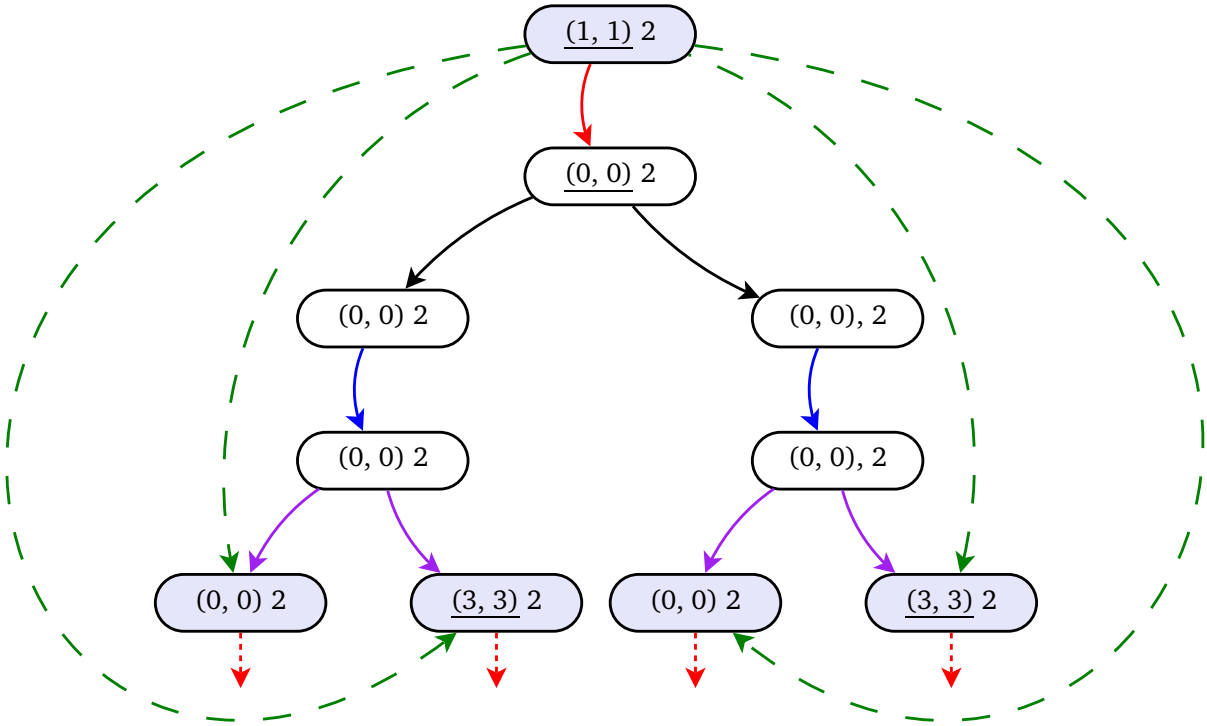

FIGURE 4.5 – Suite de la création des transitions de l'automate basé sur τ en Tableau 4.3 avec et Run = *EDF-VD*

La génération des successeurs de $\langle \underline{(1,1)}2 \rangle$ est illustrée en Figure 4.5.

Celui-ci va alors ordonnancer τ_0 et son temps restant de calcul sera nul. S'en suit la transition de terminaison. Dans les deux cas, la tâche CM se termine. Dans cette configuration, $ImplicitelyDone(S) = \{\tau_0\}$ par Définition 4.15 et $\tau^T \subseteq \{\tau_0\}$ et donc, dans tous les cas, $\tau^T \cup ImplicitelyDone(S) = \{\tau_0\}$. De ces deux états identiques, sortent deux transitions de requête. En effet, dans ces états, τ_0 est éligible : sa criticité la maintient en course et son prochain temps d'arrivée est nul. L'état mène alors soit à un état identique où la tâche CM est oisive, soit dans un état où la tâche CM reste active avec un temps de calcul valant son WCET pour la criticité de l'état, et son prochain temps d'arrivée est sa période.

Le reste de l'automate n'est pas présenté ici, mais il devra être construit, c'est ce qui est représenté par les flèches rouges en pointillé.

Au final, il faut remarquer que dans l'automate, seuls les états avec un fond gris sont atteints, et sont reliés par les transitions discontinues vertes. De plus, certains états des doublons, dans l'automate il s'agit du même état, le choix a été fait de les illustrer séparément pour rendre l'exemple plus clair.

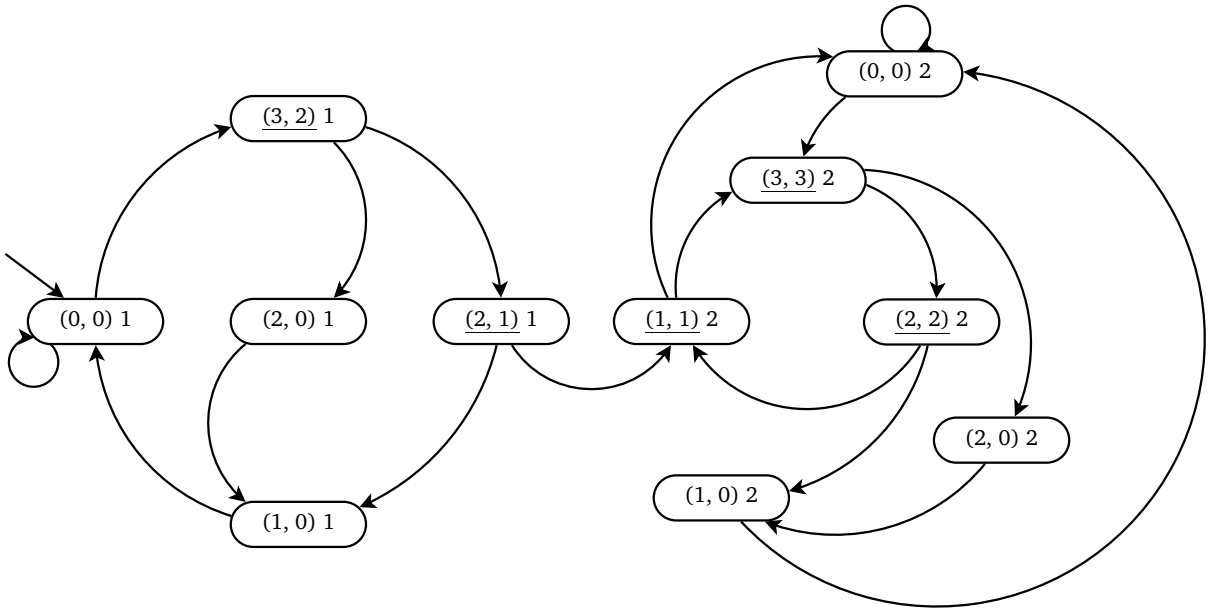

FIGURE 4.6 – Automate basé sur τ en Tableau 4.3 avec et Run = *EDF-VD*

La Figure 4.6 présente l'automate complètement construit, sans les états intermédiaires. La scission entre les deux niveaux de criticité est flagrante. Le sporadisme de l'automate est représenté grâce aux états $\langle(0,0)1\rangle$ et $\langle(0,0)2\rangle$. En effet, l'automate peut boucler sur ceux-ci avant qu'une tâche CM ne génère un travail CM, après quoi, il arrivera aux états $\langle(3,2)1\rangle$ et $\langle(3,3)2\rangle$ respectivement.

Maintenant que le principe de l'automate a été illustré clairement, il possible de prendre pour exemple un nouveau système de tâches CM.

Exemple 4.4.
Cet exemple illustre le comportement d'un système de tâches CM sporadiques non ordonnançable avec des criticités différentes. La figure Figure 4.7 construit l'automate

i	O	T	D	C	χ
0	0	2	3	[1, 2]	2
0	0	1	1	[1, 1]	1

TABLEAU 4.4 – le système de tâches CM utilisé pour l'Exemple 4.4.

basé sur le système de tâches CM en Tableau 4.4. Dans celui-ci, les nœuds comptent deux couples $(nat_S(\tau_i), rct_S(\tau_i))$, le premier est pour la tâche CM τ_0 et le second pour τ_1.

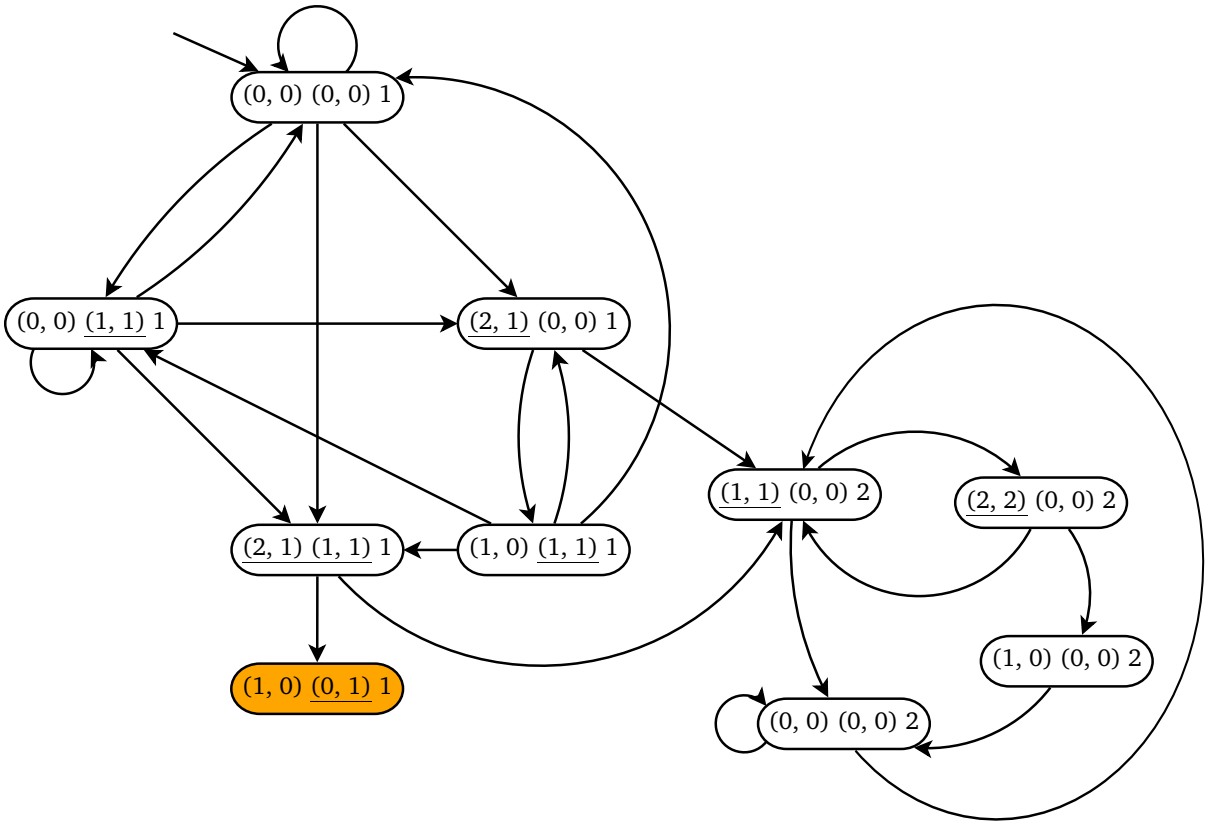

FIGURE 4.7 – Automate basé sur τ en Tableau 4.4 avec et Run = *EDF-VD*

Dans ce second exemple, il est clair que le sporadisme des tâches CM génère beaucoup de comportements différents possibles. Le nœud initial a quatre voisins, dont lui-même. Chacun de ceux-ci représente une combinaison de générations de travaux CM différente. Lorsque la tâche CM τ_0 émet un travail CM, elle est ordonnancée et puis peut mener à la criticité supérieure si elle ne se complète pas. Dans ce cas, la tâche CM τ_1 est abandonnée et ne peut plus émettre de travaux CM. Retour à la criticité de niveau un, lorsque les deux tâches CM émettent un travail CM, alors l'ordonnancement mène à un état erroné (en orange sur le graphe). Cet état a effectivement une pire laxité négative, $WorstLaxity_{S_{Fail}} = nat_{S_{Fail}}(\tau_1) - T_1 + D_1 - (rct_{S_{Fail}}(\tau_1) + C_1(2) - C_1(1) = 0 - 1 + 1 - (1 + 1 - 1) = -1 < 0$. le système de tâches CM τ n'est pas ordonnançable CM. C'est détectable en regardant la faisabilité CM de τ, $U_{\tau,1}(1) + U_{\tau,2}(1) = 3/2 > 1$.

Comme présenté dans l'état de l'art, il peut exister des relations de simulation qui permettent de réduire le temps d'exploration dans un automate. La section suivante présente une relation de simulation pour l'automate basée sur l'ordonnancement CM de tâches CM sporadiques. La simulation jouera sur l'oisiveté des tâches CM.

4.3 Simulation de tâches oisives

Dans cette section, la relation de simulation de tâches oisives $\succeq_{idle}$ est définie. Cette relation de simulation suit l'idée que Geeraets, Goossens et Lindström ont présentée en [14]. La relation se base sur l'inspection de caractéristiques de l'état d'un système, c'est-à-dire $cirt, done, rct$ et nat.

Définition 4.27 (Simulation de tâches oisives)**.**
Soit τ un système de tâches CM sporadiques. Le préordre de tâches oisives $\succeq_{idle} \subseteq States(\tau) \times States(\tau)$ est tel que pour tout $S_1, S_2 : S_2 \succeq_{idle} S_1$, si et seulement si :

— $crit_{S_2} = crit_{S_1}$

— $done_{S_2} = done_{S_1}$

— $rct_{S_2} = rct_{S_1}$

— Pour tout τ_i tel que $done_{S_1}(\tau_i) = True : nat_{S_2}(\tau_i) \leq nat_{S_1}(\tau_i)$

— Pour tout τ_i tel que $done_{S_1}(\tau_i) = False : nat_{S_2}(\tau_i) = nat_{S_1}(\tau_i)$

Cette relation est transitive et réflexive, il s'agit donc bien d'un préordre. La relation définit aussi un ordre partiel sur $Active(S)$, car celle-ci est antisymétrique. Remarquons que $Active(S_1) = Active(S_2)$ puisque $done_{S_1} = done_{S_2}$.

La relation annonce qu'un état du système en simule un autre s'il a la même criticité, que les tâches CM actives sont identiques et que pour les tâches CM oisives, le prochain temps d'arrivée est égal ou plus proche.

La relation étant maintenant définie, il va falloir prouver que ce préordre est effectivement une relation de simulation. Il va falloir démontrer que si un état en simule un autre, alors tous les successeurs de l'état simulés devront être simulés par des successeurs de l'état simulant. Il va donc falloir s'assurer que ces simulations sont présentes pour chacune des transitions intermédiaires. Pour ce faire, la démonstration s'appuiera sur des lemmes qui vont être définis maintenant.

Le premier lemme qui sera utile pose l'égalité de la tâche CM ordonnancée lors d'une simulation entre deux états. Cette information sera importante lors de phase de la démonstration sur la transition d'exécution et de complétion.

Lemme 4.1.
Si S_1 et S_2 sont des états tels que $S_2 \succeq_{idle} S_1$ et Run un ordonnanceur sans mémoire alors $Run(S_2) = Run(S_1)$.

Démonstration. Par la Définition 4.27 : $S_2 \succeq_{idle} S_1$ implique que $Active(S_2) = Active(S_1)$. Soit τ_i une tâche CM faisant partie de $Active(S_1)$, d'où $done_{S_1}(\tau_i) = False$. Dans ce cas, et puisque $S_2 \succeq_{idle} S_1$, on en conclut que $done_{S_2}(\tau_i) = done_{S_1}(\tau_i)$, $nat_{S_2}(\tau_i) = nat_{S_1}(\tau_i)$, $rct_{S_2}(\tau_i) = rct_{S_1}(\tau_i)$ et que $crit_{S_2} = crit_{S_1}$ pour tout $\tau_i \in Active(S_1)$. Donc, comme Run est sans mémoire par hypothèse et qu'il s'agit d'un sous-ensemble des tâches CM actives, $Run(S_1) = Run(S_2)$. □

Ensuite, il est nécessaire que les tâches CM implicitement terminées soient les mêmes dans le couple d'états simulant, simulé. En effet, cette égalité doit être présente sinon la transition de complétion ne pourrait pas générer des états qui se simulent.

Lemme 4.2.
Si S_1 et S_2 sont des états tels que $S_2 \succeq_{idle} S_1$ alors $ImplicitelyDone(S_2) = ImplicitelyDone(S_1)$.

Démonstration. Par la Définition 4.27 : $S_2 \succeq_{idle} S_1$ implique que $done_{S_2}(\tau_i) = done_{S_1}(\tau_i)$, $rct_{S_2}(\tau_i) = rct_{S_1}(\tau_i)$ et que $crit_{S_2} = crit_{S_1}$ pour tout $\tau_i \in \tau$. D'où $ImplicitelyDone(S_2) = ImplicitelyDone(S_1)$. □

Il faut aussi que si un état génère un passage à criticité supérieure, l'état qui le simule le fasse aussi. C'est pour cette raison qu'il est obligatoire que la criticité réelle de l'état simulant soit identique à celle de l'état simulé.

Lemme 4.3.
Si S_1 et S_2 sont des états tels que $S_2 \succeq_{idle} S_1$ alors $Critical_{S_2} = Critical_{S_1}$.

Démonstration. Par la Définition 4.27 : $S_1 \succeq_{idle} S_2$ implique que $Active(S_2) = Active(S_1)$, $crit_{S_2} = crit_{S_1}$ et que $rct_{S_2}(\tau_i) = rct_{S_1}(\tau_i)$ pour tout $\tau_i \in \tau$. D'où $Critical_{S_2} = Critical_{S_1}$. □

Pour terminer, il faut prouver que les tâches CM éligibles à la soumission d'un travail CM simulant englobent celles de l'état simulé. Vu que les prochains temps d'arrivées sont en avance dans l'état simulant, il peut y avoir des tâches CM éligibles en plus, mais pas en moins. Comme la transition de requête se base sur un sous-ensemble de tâches CM éligible à la soumission d'un travail CM, pour l'état simulant, il sera toujours possible d'avoir les mêmes tâches CM qui génèrent un travail CM que dans l'état simulé.

Lemme 4.4.
Si S_1 et S_2 sont deux états tels que $S_2 \succeq_{idle} S_1$ alors $Eligible(S_2) \supseteq Eligible(S_1)$.

Démonstration. Prouvons que toute tâche CM $\tau_i \in Eligible(S_1) \rightarrow \tau_i \in Eligible(S_2)$. Puisque $\tau_i \in Eligible(S_1)$, on a $done_{S_1}(\tau_i) = True$, $crit_{S_1} \geq \chi_i$, et $nat_{S_1}(\tau_i) \leq 0$ par définition de $Eligible$. Donc, puisque $S_2 \succeq_{idle} S_1$, $done_{S_2}(\tau_i) = done_{S_1}(\tau_i) = True$, $crit_{S_2} = crit_{S_1} \geq \chi_i$ et $nat_{S_2}(\tau_i) \leq nat_{S_1}(\tau_i) \leq 0$ par définition de $\succeq_{idle}$. D'où $\tau_i \in Eligible(S_2)$. □

Pour comprendre l'objectif de la définition et du lemme suivants il faut se rappeler que quand nat est négatif, il est nécessaire de représenter les cas où la tâche CM a émis un travail CM à chaque temps possible. Le modèle attend d'abord que le travail CM courant soit terminé et puis il va générer un travail CM pour chaque prochain temps d'arrivée compris dans l'intervalle $[nat_S(\tau_i) + T_i, T_i]$. Comme un état simulant peut avoir un prochain temps d'arrivée inférieur, il va falloir prouver qu'il générera un état où les prochains temps d'arrivées des tâches CM fraîchement générées sont identiques à ceux de l'état simulé. Identiques, car, à ce moment-là, il s'agira de tâches CM actives.

Définition 4.28.
Soit S_1 et S_2 deux états tels que $S_2 \succeq_{idle} S_1$ et $\tau^R \subseteq Eligible(S_1)$. Alors, pour tout $\overline{S}_1 \in S_1^{\tau^R}$, un état $\overline{S}_2 \in S_2^{\tau^R}$ est appelé $analogue\ \tau^R - oisif$ de $\overline{S}_1$, si et seulement si :

$$\forall \tau_i \in \tau^R : nat_{\overline{S}_2}(\tau_i) = nat_{\overline{S}_1}(\tau_i)$$

Lemme 4.5.
Soit S_1 et S_2 deux états tels que $S_2 \succeq_{idle} S_1$ et $\tau^R \subseteq Eligible(S_1)$ un sous-ensemble de tâches CM. Alors, pour tout $\overline{S}_1 \in S_1^{\tau^R}$ il y a un $analogue\ \tau^R - oisif$ $\overline{S}_2 \in S_2^{\tau^R}$ de $\overline{S}_1$.

Démonstration. La Définition 4.25 et la Définition 4.27 posent que pour tout $\tau_i \in \tau^R \subseteq Eligible(S_1)$ on a $nat_{S_2}(\tau_i) \leq nat_{S_1}(\tau_i) \leq 0$, $rct_{S_2}(\tau_i) = rct_{S_1}(\tau_i) = 0$, $done_{S_2}(\tau_i) = done_{S_1}(\tau_i) = True$ et que $crit_{S_2} = crit_{S_1}$. Par définition, tout $\overline{S}_1 \in S_1^{\tau^R}$ satisfait :

$$\forall \tau_i \in \tau^R : nat_{\overline{S}_1}(\tau_i) \in I_1^i = [nat_{S_1}(\tau_i) + T_i, T_i]$$

De manière similaire, par la Définition 4.25, pour tout vecteur $(k_1, k_2, ..., k_n)$(où $n = |\tau^R|$) tel que pour tout $1 \leq i \leq n : k_i \in I_2^i = [nat_{S_2}(\tau_i) + T_i, T_i]$, il existe $\overline{S}_2 \in S_2^{\tau^R}$ avec $nat_{\overline{S}_2}(\tau_i) = k_i$ pour tout $1 \leq i \leq n$. C'est-à-dire qu'à chaque fois qu'un vecteur $(k_1, k_2, ..., k_n)$ de valeurs dans $I_2^1 \times I_2^2 \times ... \times I_2^n$ est fixé, il est possible de trouver un état $\overline{S}_2 \in S_2^{\tau^R}$ tel que le nat de chaque tâche CM τ_i est exactement k_i. Cependant, comme $nat_{S_2}(\tau_i) \leq nat_{S_1}(\tau_i)$ pour tout $1 \leq i \leq n$ on en conclut que $nat_{S_1}(\tau_i) \in I_1^i \subseteq I_2^i$ pour tout $1 \leq i \leq n$. Dès lors, le vecteur $nat_{\overline{S}_1}(\tau_1), nat_{\overline{S}_1}(\tau_2), ..., nat_{\overline{S}_1}(\tau_n)$ est dans $I_2^1 \times I_2^2 \times ... \times I_2^n$. D'où il existe $\overline{S}_2 \in S_2^{\tau^R}$ tel que pour tout $1 \leq i \leq n : nat_{\overline{S}_2}(\tau_i) = nat_{\overline{S}_1}(\tau_i)$ □

Prouvons maintenant que $\succeq_{idle}$ est bien une relation de simulation lorsqu'on considère un ordonnanceur sans mémoire.

Théorème 4.1.
Soit τ un système de tâches CM sporadique et Run un ordonnanceur sans mémoire (déterminé) pour τ sur un processeur. Alors, $\succeq_{idle}$ est une relation de simulation pour $\overline{A}(\tau, Run)$.

Démonstration. Soit S_1, S_1' et S_2 trois états dans $States(\tau)$ tel que $(S_1, S_1') \in E$ et $S_2 \succeq_{idle} S_1$, montrons qu'il existe $S_2' \in States(\tau)$ avec $(S_2, S_2') \in E$ et $S_2' \succeq_{idle} S_1'$.

Puisque $(S_1, S_1') \in E$, il existe S_1^+, S_1^T et $S_1^C \in States(\tau)$ et $\tau^T \subseteq Run(S_1), \tau^R \subseteq Eligilbe(S_1^C)$ tel que : $S_1 \xrightarrow{Run} S_1^+ \xrightarrow{\tau^T} S_1^T \xrightarrow{C} S_1^C \xrightarrow{\tau^R} S_1^R = S_1'$.

Par la Définition 4.27 et le Lemme 4.1, $S_2 \succeq_{idle} S_1$ implique que $Active(S_2) = Active(S_1)$ et $Run(S_2) = Run(S_1)$. Soit S_1^+ l'état unique tel que $S_1 \xrightarrow{Run} S_1^+$, montrons qu'il existe S_2^+ tel que $S_2 \xrightarrow{Run} S_2^+$ et que $S_2^+ \succeq_{idle} S_1^+$.

1. Pour tout $\tau_i \in \tau : done^+_{S_1}(\tau_i) = done_{S_1}(\tau_i), done^+_{S_2}(\tau_i) = done_{S_2}(\tau_i)$. Puisque $S_2 \succeq_{idle} S_1$, on a aussi que $done_{S_2}(\tau_i) = done_{S_1}(\tau_i)$
   On en conclut que $done^+_{S_2} = done^+_{S_1}$.
2. $crit^+_{S_1} = crit_{S_1}, crit^+_{S_2} = crit_{S_2}$. Puisque $S_2 \succeq_{idle} S_1$, on a aussi que $crit_{S_2} = crit_{S_1}$
   On en conclut que $crit^+_{S_2} = crit^+_{S_1}$.
3. Pour tout $\tau_i \in Run(S_1) = Run(S_2) : rct^+_{S_1}(\tau_i) = rct_{S_1}(\tau_i) - 1, rct^+_{S_2}(\tau_i) = rct_{S_2}(\tau_i) - 1$. Puisque $S_2 \succeq_{idle} S_1$, on sait que $rct_{S_2} = rct_{S_1}$. D'où $rct^+_{S_2}(\tau_i) = rct^+_{S_1}(\tau_i)$.
   Pour une tâche CM $\tau_i \notin Run(S_1) = Run(S_2) : rct^+_{S_1}(\tau_i) = rct_{S_1}(\tau_i), rct^+_{S_2}(\tau_i) = rct_{S_2}(\tau_i)$. D'où $rct^+_{S_2}(\tau_i) = rct^+_{S_1}(\tau_i)$.
   On en conclut alors que $rct^+_{S_2} = rct^+_{S_1}$.
4. Soit τ_i une tâche CM telle que $done^+_{S_1}(\tau_i) = True$, on sait que $done_{S_2}(\tau_i) = done_{S_1}(\tau_i) = True$ par l'item 1. Dès lors, $nat^+_{S_1} = max(nat_{S_1}(\tau_i) - 1, 0)$ et $nat^+_{S_2} = max(nat_{S_2}(\tau_i) - 1, 0)$. Cependant, comme $S_2 \succeq_{idle} S_1$ et $done_{S_1}(\tau_i) = True$, on sait que $nat_{S_2}(\tau_i) \leq nat_{S_1}(\tau_i)$.
   On en conclut que $nat^+_{S_2}(\tau_i) \leq nat^+_{S_1}(\tau_i)$.
5. Soit τ_i une tâche CM telle que $done^+_{S_1}(\tau_i) = False$, on sait que $done_{S_2}(\tau_i) = done_{S_1}(\tau_i) = False$ par l'item 1. Dès lors $nat^+_{S_1}(\tau_i) = nat_{S_1}(\tau_i) - 1$ et $nat^+_{S_2}(\tau_i) = nat_{S_2}(\tau_i) - 1$. Cependant, comme $S_2 \succeq_{idle} S_1$ et $done_{S_1}(\tau_i) = False$, on sait que $nat_{S_2}(\tau_i) = nat_{S_1}(\tau_i)$. On en conclut que $nat^+_{S_2}(\tau_i) = nat^+_{S_1}(\tau_i)$.
   On en conclut que $nat^+_{S_2}(\tau_i) = nat^+_{S_1}(\tau_i)$.

Par la Définition 4.22. Il est maintenant établi que $S_2^+ \succeq_{idle} S_1^+$.

Par le Lemme 4.1 et le Lemme 4.2, nous obtenons que $Run(S_1) = Run(S_2)$ et que $ImplicitelyDone(S_1^+) = ImplicitelyDone(S_2^+)$.
Prouvons maintenant que, pour tout $\tau^T \subseteq Run(S_1) \cup ImplicitelyDone(S_1^+)$ menant à $S_1^+ \xrightarrow{\tau^T} S_1^T$, il existe un S_2^T tel que $S_2^+ \xrightarrow{\tau^T} S_2^T$ et $S_2^T \succeq_{idle} S_1^T$.

1. Par la Définition 4.23 on a $crit_{S_1}^T = crit_{S_1}^+$ et $crit_{S_2}^T = crit_{S_2}^+$. Et puisque $S_2^+ \succeq_{idle} S_1^+$ on a $crit_{S_2}^+ = crit_{S_1}^+$.
   On en conclut que $crit_{S_2}^T = crit_{S_1}^T$.
2. Par la Définition 4.23, pour tout $\tau_i \in Run(S_1) \cup ImplicitelyDone(S_1^+)$: $done_{S_1}^T(\tau_i) = True = done_{S_2}^T(\tau_i)$.
   Pour tout $\tau_i \notin Run(S_1) \cup ImplicitelyDone(S_1^+)$ on a $done_{S_1}^T(\tau_i) = done_{S_1}^+(\tau_i)$ et $done_{S_2}^T(\tau_i) = done_{S_2}^+(\tau_i)$. Et puisque $S_2^+ \succeq_{idle} S_1^+$ on a $done_{S_2}^+(\tau_i) = done_{S_1}^+(\tau_i)$. D'où $done_{S_2}^T(\tau_i) = done_{S_1}^T(\tau_i)$.
   On en conclut que $done_{S_2}^T = done_{S_1}^T$.
3. Par la Définition 4.23, pour tout $\tau_i \in Run(S_1) \cup ImplicitelyDone(S_1^+)$: $rct_{S_1}^T(\tau_i) = 0 = rct_{S_2}^T(\tau_i)$.
   Pour tout $\tau_i \notin Run(S_1) \cup ImplicitelyDone(S_1^+)$ on a $rct_{S_1}^T(\tau_i) = rct_{S_1}^+(\tau_i)$ et $rct_{S_2}^T(\tau_i) = rct_{S_2}^+(\tau_i)$. Et puisque $S_2^+ \succeq_{idle} S_1^+$ on a $rct_{S_2}^+(\tau_i) = rct_{S_1}^+(\tau_i)$. D'où $rct_{S_2}^T(\tau_i) = rct_{S_1}^T(\tau_i)$.
   On en conclut que $rct_{S_2}^T = rct_{S_1}^T$.
4. Par la Définition 4.23, pour tout $\tau_i \in \tau$:

$$nat_{S_1}^T(\tau_i) = nat_{S_1}^+(\tau_i) \; et \; nat_{S_2}^T(\tau_i) = nat_{S_2}^+(\tau_i) \tag{4.1}$$

   (a) pour tout $\tau_i \in \tau$ avec $done_{S_1}^T = False$ on sait que $done_{S_2}^+ = done_{S_1}^+ = False$, car cette transition n'active jamais une tâche CM, voir item 2. Comme $S_2^+ \succeq_{idle} S_1^+$ et $done_{S_1}^+ = False$ on a $nat_{S_2}^+(\tau_i) = nat_{S_1}^+(\tau_i)$.
   Par l'Équation 4.1, on en conclut que $nat_{S_2}^T(\tau_i) = nat_{S_1}^T(\tau_i)$.

   (b) pour tout $\tau_i \in \tau$ avec $done_{S_1}^T = True$

   — Soit $done_{S_2}^+ = done_{S_1}^+ = False$ et donc $nat_{S_2}^+(\tau_i) = nat_{S_1}^+(\tau_i)$, car $S_2^+ \succeq_{idle} S_1^+$, d'où $nat_{S_2}^T(\tau_i) \leq nat_{S_1}^T(\tau_i)$ par l'Équation 4.1.

   — Soit $done_{S_2}^+ = done_{S_1}^+ = True$ et donc $nat_{S_2}^+(\tau_i) \leq nat_{S_1}^+(\tau_i)$, car $S_2^+ \succeq_{idle} S_1^+$, d'où $nat_{S_2}^T(\tau_i) \leq nat_{S_1}^T(\tau_i)$ par l'Équation 4.1.

Il est maintenant établi que $S_2^T \succeq_{idle} S_1^T$.

Par le Lemme 4.3, nous obtenons que $Critical_{S_2^T} = Critical_{S_1^T}$.

Prouvons maintenant que pour l'unique état S_1^C tel que $S_1^T \xrightarrow{C} S_1^C$, il existe S_2^C tel que $S_2^T \xrightarrow{C} S_2^C$ et $S_2^C \succeq_{idle} S_1^C$.

1. Par la Définition 4.24 on a $crit_{S_1}^C = Critical_{S_1^T}$ et $crit_{S_2}^C = Critical_{S_2^T}$. Et puisque $S_2^+ \succeq_{idle} S_1^+$ on a $Critical_{S_1^T} = Critical_{S_2^T}$.
   On en conclut que $crit_{S_2}^C = crit_{S_1}^C$.
2. Par la Définition 4.24, pour toute tâche CM $\tau_i \in \tau$ avec $\chi_i < Critical_{S_1^T}$:
   (a) $done_{S_1}^C(\tau_i) = True = done_{S_2}^C(\tau_i)$
   (b) $rct_{S_1}^C(\tau_i) = 0 = rct_{S_2}^C(\tau_i)$
   (c) $nat_{S_1}^C(\tau_i) = 0 = nat_{S_2}^C(\tau_i)$
3. Par la Définition 4.24, pour toute tâche CM $\tau_i \in \tau$ avec $\chi_i \geq Critical_{S_1^T}$:
   (a) $done_{S_1}^C(\tau_i) = done_{S_1}^T(\tau_i)$ et $done_{S_2}^C(\tau_i) = done_{S_2}^T(\tau_i)$. Et puisque $S_2^T \succeq_{idle} S_1^T$ on a $done_{S_2}^T(\tau_i) = done_{S_1}^T(\tau_i)$.
   On en conclut que $done_{S_2}^C(\tau_i) = done_{S_1}^C(\tau_i)$.
   (b) $rct_{S_1}^C(\tau_i) = rct_{S_1}^T(\tau_i) + C_i(Critical_{S_1^T}) - C_i(crit_{S_1}^T)$ et $rct_{S_2}^C(\tau_i) = rct_{S_2}^T(\tau_i) + C_i(Critical_{S_2^T}) - C_i(crit_{S_2}^T)$. Et puisque $S_2^T \succeq_{idle} S_1^T$ on a $rct_{S_2}^T(\tau_i) = rct_{S_1}^T(\tau_i)$, $crit_{S_2}^T = crit_{S_1}^T$ et $Critical_{S_2^T} = Critical_{S_1^T}$.
   On en conclut que $rct_{S_2}^C(\tau_i) = rct_{S_1}^C(\tau_i)$.
   (c) Par la Définition 4.24, pour tout $\tau_i \in \tau$:

$$nat_{S_1}^C(\tau_i) = nat_{S_1}^T(\tau_i) \; et \; nat_{S_2}^C(\tau_i) = nat_{S_2}^T(\tau_i) \tag{4.2}$$

   — Pour tout τ_i avec $done_{S_1}^C = True$, on sait que $done_{S_2}^T(\tau_i) = done_{S_1}^T(\tau_i) = True$ par l'item (a). Puisque $S_2^T \succeq_{idle} S_1^T$ on a $nat_{S_2}^T(\tau_i) \leq nat_{S_1}^T(\tau_i)$.
   On en conclut que $nat_{S_2}^C(\tau_i) \leq nat_{S_1}^C(\tau_i)$ par l'Équation 4.2.
   — Pour tout τ_i avec $done_{S_1}^C = False$, on sait que $done_{S_2}^T(\tau_i) = done_{S_1}^T(\tau_i) = False$ par l'item (a). Puisque $S_2^T \succeq_{idle} S_1^T$ on a $nat_{S_2}^T(\tau_i) = nat_{S_1}^T(\tau_i)$.
   On en conclut que $nat_{S_2}^C(\tau_i) = nat_{S_1}^C(\tau_i)$ par l'Équation 4.2.

Il est maintenant établi que $S_2^C \succeq_{idle} S_1^C$.

Rappelons que comme $(S_1, S_1') \in E$, il existe S_1^+, S_1^T et $S_1^C \in States(\tau)$ et $\tau^T \subseteq Run(S_1), \tau^R \subseteq Eligilbe(S_1^C)$ tel que : $S_1 \xrightarrow{Run} S_1^+ \xrightarrow{\tau^T} S_1^T \xrightarrow{C} S_1^C \xrightarrow{\tau^R} S_1^R = S_1'$ par la Définition 4.26. Soit S_2^R, un $analogue\ \tau^R - oisif$ de S_1^R (qui existe de par le Lemme 4.5), prouvons que $S_2^R \succeq_{idle} S_1^R$.

1. Pour tout $\tau_i \in \tau^R : rct_{S_1}^R(\tau_i) = C_i(\chi_i) = rct_{S_2}^R(\tau_i)$.
   Pour tout $\tau_i \notin \tau^R : rct_{S_1}^R(\tau_i) = rct_{S_1}^C(\tau_i), rct_{S_2}^R(\tau_i) = rct_{S_2}^C(\tau_i)$, et comme $S_2^C \succeq_{idle} S_1^C : rct_{S_2}^C(\tau_i) = rct_{S_1}^C(\tau_i)$.
   Donc $rct_{S_2}^R = rct_{S_1}^R$.
2. Pour tout $\tau_i \in \tau^R : done_{S_1}^R(\tau_i) = False = done_{S_2}^R(\tau_i)$.
   Pour tout $\tau_i \notin \tau^R : done_{S_1}^R(\tau_i) = done_{S_1}^C(\tau_i), done_{S_2}^R(\tau_i) = done_{S_2}^C(\tau_i)$, et comme $S_2^C \succeq_{idle} S_1^C : done_{S_2}^C(\tau_i) = done_{S_1}^C(\tau_i)$.
   Donc $done_{S_2}^R = done_{S_1}^R$.
3. $crit_{S_1}^R = crit_{S_1}^C, crit_{S_2}^R = crit_{S_2}^C$. Comme $S_2^C \succeq_{idle} S_1^C : crit_{S_2}^C = crit_{S_1}^C$.
   Donc $crit_{S_2}^R = crit_{S_1}^R$.
4. Pour tout $\tau_i \in \tau^R : nat_{S_1}^R(\tau_i) = nat_{S_2}^R(\tau_i)$ puisque S_2^R est un $analogue\ \tau^R - oisif$ de S_1^R.
5. Pour tout $\tau_i \notin \tau^R$:

$$nat_{S_1}^R(\tau_i) = nat_{S_1}^C(\tau_i)\ et\ nat_{S_2}^R(\tau_i) = nat_{S_2}^C(\tau_i) \tag{4.3}$$

   et

$$done_{S_1}^R(\tau_i) = done_{S_1}^C(\tau_i)\ et\ done_{S_2}^R(\tau_i) = done_{S_2}^C(\tau_i) \tag{4.4}$$

Par la Définition 4.25, on considère deux autres cas :

(a) Si $done_{S_1}^R(\tau_i) = True$ alors $done_{S_2}^R(\tau_i) = True$ par l'item 2. Cependant, par l'Équation 4.4, $done_{S_1}^C(\tau_i) = True$. Donc, comme $S_2^C \succeq_{idle} S_1^C : nat_{S_2}^C \leq nat_{S_1}^C$, par la Définition 4.27.
   D'où $nat_{S_2}^R(\tau_i) \leq nat_{S_1}^R(\tau_i)$ par l'Équation 4.3.

(b) Si $done_{S_1}^R(\tau_i) = False$ alors $done_{S_2}^R(\tau_i) = False$ par l'item 2. Cependant, par l'Équation 4.4, $done_{S_1}^C(\tau_i) = False$. Donc, comme $S_2^C \succeq_{idle} S_1^C : nat_{S_2}^C = nat_{S_1}^C$, par la Définition 4.27.
   D'où $nat_{S_2}^R(\tau_i) = nat_{S_1}^R(\tau_i)$ par l'Équation 4.3.

Il est maintenant établi que $S_2^R \succeq_{idle} S_1^R$ ce qui prouve bien que tous les états intermédiaires nécessaires à la présence de l'arc (S_1, S_1') sont bien simulés par ceux générés par le lien $(S_2, S_2)'$, si et seulement $S_2 \succeq_{idle} S_1$.

Pour terminer la démonstration, il reste à prouver que si $S_2 \succeq_{idle} S_1$ et $S_1 \in Fail_\tau$ alors $S_2 \in Fail_\tau$ aussi. Soit τ_i tel que $WorstLaxity_{S_1}(\tau_i) < 0$.

Donc $WorstLaxity_{S_1}(\tau_i) = nat_{S_1}(\tau_i) - T_i + D_i - (rct_{S_1}(\tau_i) + (C_i(K) - C_i(crit_{S_1}))$

Puisque $S_2 \succeq_{idle} S_1 : crit_{S_2}(\tau_i) = crit_{S_1}(\tau_i), rct_{S_2}(\tau_i) = rct_{S_1}(\tau_i), nat_{S_2}(\tau_i) \leq nat_{S_1}(\tau_i)$. D'où $WorstLaxityS_2(\tau_i) = nat_{S_2}(\tau_i) - T_i + D_i - (rct_{S_2}(\tau_i) + (C_i(K) - C_i(crit_{S_2})) \leq WorstLaxityS_1(\tau_i) < 0$ et donc $S_2 \in Fail_\tau$.

□

Exemple 4.5.
En regardant l'Exemple 4.4 de la section précédente, on s'aperçoit que certains états sont simulés par d'autres selon $\succeq_{idle}$. L'exemple se base sur le système de tâches CM présenté en Exemple 4.4. Analysons les simulations présentes.

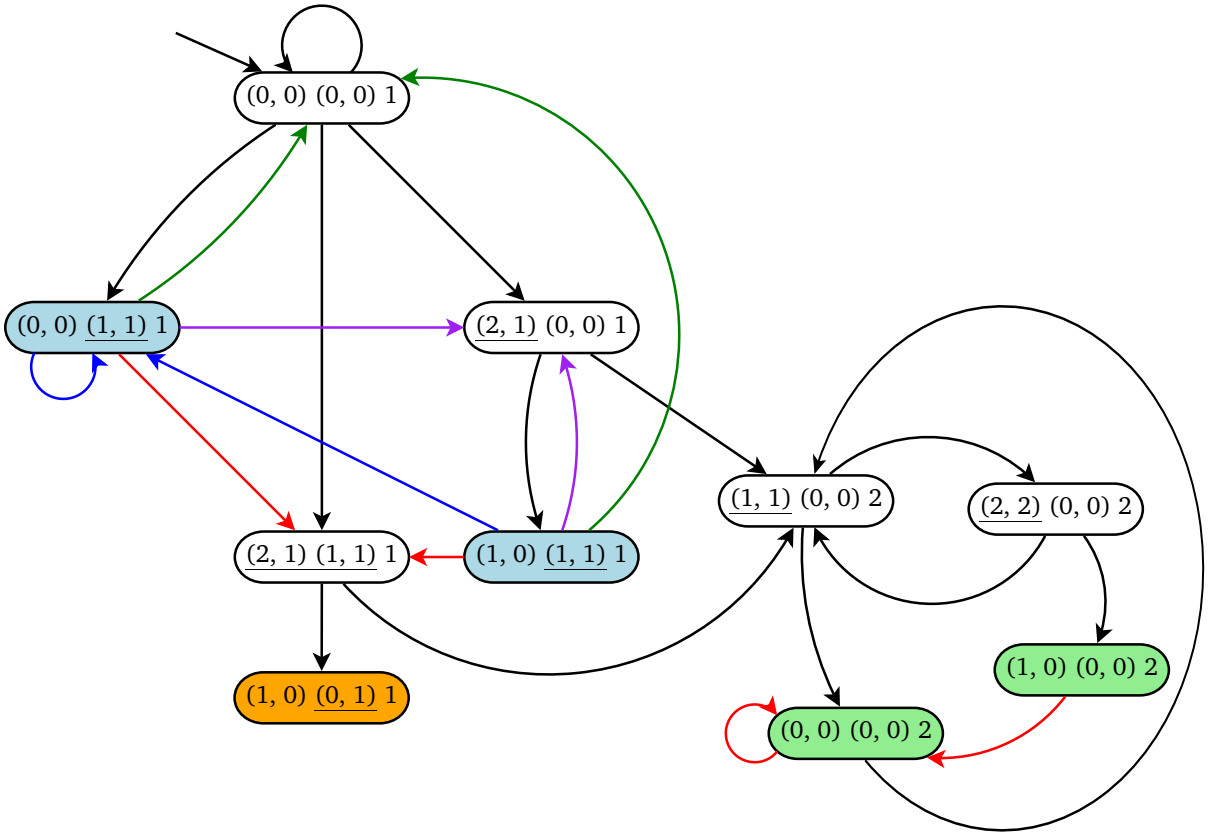

FIGURE 4.8 – Automate basé sur τ en Tableau 4.4 avec et Run = *EDF-VD*, analyse des simulations

Dans la Figure 4.8, le nœud en orange est un état erroné, et les nœuds colorés de la même la couleur sont un couple d'états simulé simulant.

Les transitions sortantes des nœuds qui se simulent sont colorées de manière à représenter que chaque état atteignable, depuis le nœud simulé, l'est aussi depuis le nœud simulant. Comme c'est le cas pour la paire d'états bleue, $\langle(1,0)\underline{(1,1)}1\rangle$ et $\langle(0,0)\underline{(1,1)}1\rangle$. Selon la relation de simulation de tâches oisives, le premier est bien simulé par le deuxième. Donc, si le nœud $\langle(0,0)\underline{(1,1)}1\rangle$ est visité, il ne sera pas nécessaire d'explorer $\langle(1,0)\underline{(1,1)}1\rangle$. En revanche, l'inverse est incorrect.

Le principe est le même pour la paire d'états verte, dans le niveau de criticité supérieur. C'est bien l'état $\langle(0,0)(0,0)2\rangle$ qui simule $\langle(1,0)(0,0)2\rangle$. En effet, il possède les mêmes transitions et une en plus.

4.3.1 Autres relations de simulations infructueuses

Simulation de pire scénario

On pourrait être tenté de construire une relation de simulation basée sur le fait que l'état simulant soit «pire» que l'état simulé.

Basée sur *nat* On voudrait alors que deux états soient identiques, sauf pour le nat des tâches CM actives. L'état simulant devrait avoir des nat plus petits pour les tâches CM actives. Dans cet état du système, les échéances des tâches CM actives seraient alors plus rapprochées que dans l'état simulé. L'état simulant serait alors plus «dur» à ordonnancer.

Malheureusement, il n'est pas possible de jouer sur cette relation, car elle modifierait le comportement de l'ordonnanceur. En effet, il ne serait pas garanti que les tâches CM ordonnancées dans les états simulés et simulant soient identiques.
Cette modification pose un problème, car les états qui viennent d'être exécutés peuvent signaler leur complétion. Les états résultant de ces transitions de terminaisons ne seront alors plus un couple simulé-simulant.

On pourrait utiliser cette relation de simulation uniquement lorsque les deux états respectent la condition précédente, mais en plus, l'ordonnanceur devrait choisir la même tâche CM à exécuter pour des états qui se simulent.
Pour ce faire, il faudrait alors s'assurer que les transitions respectent l'égalité du choix de la tâche CM fait par l'ordonnanceur. Or, les transitions ne donnent pas d'informations sur le comportement de l'ordonnanceur.
Il faudrait rendre la preuve propre à un ordonnanceur en particulier ou à un certain type d'ordonnanceurs. Le test passerait alors à côté d'un grand nombre d'algorithmes d'ordonnancement.

Enfin, on pourrait essayer de limiter l'inégalité entre les nat seulement à la tâche CM ordonnancée. On devrait alors s'assurer que le choix des tâches CM à ordonnancer dans la paire d'états simulé et simulant soit identique. Cette prémisse ne serait pas dénuée de sens, car elle signifierait que : si, dans un état initial, une tâche CM est la plus prioritaire, alors dans un état où cette tâche CM aurait une échéance plus proche, elle serait aussi définie comme étant la plus prioritaire.
Le problème est identique au précédent, les transitions ne garantiraient pas l'égalité du choix de l'ordonnanceur.

Basé sur *rct* L'idée serait qu'un état en simule un autre s'il est identique, sauf pour les tâches CM actives, où leur rct pourrait être plus conséquent. Dans cette configuration, il resterait plus de temps de calcul restant aux tâches CM actives, rendant l'état moins laxiste.

La modélisation du système est basée sur le fait que les tâches CM se terminent lorsqu'elles le signalent, ou alors qu'elles ont été exécutées pour leur pire WCET. Entretemps, il peut y avoir des passages à niveau de criticité supérieurs lorsqu'une tâche CM a été exécutée complètement pour son WCET actuel, mais ne s'est pas signalée comme terminée.
Pour détecter qu'une tâche CM déclenche une criticité supérieure, il faut justement vérifier que son temps restant de calcul est nul et qu'elle est toujours active. Donc, si cette relation de simulation était en place, un état déclenchant un passage à criticité supérieure pourrait être simulé par un état qui ne le ferait pas, car le temps restant de calcul peut y être supérieur. D'où, cette relation de simulation n'est pas utilisable. La relation pourrait être restreinte aux tâches CM qui n'ont pas un temps restant de calcul nul. Immanquablement, la simulation ne serait pas respectée lors de la transition d'exécution.

Simulation de scénario avancé

Pour terminer, on pourrait développer une relation qui simulerait un état moins avancé. C'est-à-dire qu'un état simulerait les états qui ont des coups d'horloge de retard sur lui. Pour ce faire, on souhaiterait que deux états soient identiques, sauf pour le temps restant de calcul et le temps d'arrivée prochain, mais les deux états devraient avoir la même laxité pour les tâches CM actives.

Outre les problèmes déjà énoncés plus tôt, les candidats à cette relation de simulation seraient les états qui se succèdent d'une ou plusieurs transitions (totales) sans signalement de complétion ni émission de travail CM. Dans ce cas, il faudrait toujours visiter un état simulé pour obtenir son simulant en tant que successeur. La relation ne permettrait donc pas de restreindre le nombre d'états à visiter. En revanche, elle diminuerait la taille requise pour stocker les états visités.

Chapitre 5

Résultats

Ce chapitre présente les résultats du mémoire.

Tous les tests d'ordonnancement présentés au chapitre consacré à ceux-ci ont été implémentés. L'exploration d'automate généré pour vérifier l'ordonnançabilité de tâches CM périodiques et sporadiques a été aussi implémentée. Enfin, une version de cette exploration pour les tâches CM sporadiques avec antichaîne basée sur la relation de simulation tâches oisives est aussi implémentée.

Les résultats sont divisés en deux catégories. Premièrement, il y a l'analyse de la complexité du test d'ordonnancement en criticité mixte par exploration. Ensuite, une comparaison de la performance des différents algorithmes d'ordonnancement est proposée. Avant de passer à ces résultats, la méthodologie des tests est donnée.

5.1 Méthodologie

5.1.1 Génération de systèmes de tâches CM

La génération d'un système de tâches CM suit les idées proposées en [26] et [14].

Un système de tâches CM à double criticité est généré en commençant avec un ensemble vide $\tau = \emptyset$ dans lequel des tâches CM aléatoires sont ajoutées. La génération des tâches CM est contrôlée par quatre paramètres :

— la probabilité P_{HI} que la tâche CM soit fortement critique

— le ratio maximum R_{HI} entre le temps d'exécution pour forte criticité et faible criticité

— et la période maximum T^{MAX}

Pour les tests visant à analyser la complexité de l'exploration d'un automate généré par un problème d'ordonnancement en criticité mixte, les temps d'exécution en faible criticité suivent une distribution exponentielle de moyenne $0.35T_i$. Dans ce cas, chaque nouvelle tâche CM τ_i est alors générée comme ceci :

— $\chi_i = HI$ avec une probabilité P_{HI}, $\chi_i = LO$ sinon

— $C_i(LO)$ suit une distribution exponentielle de moyenne $0.35T_i$

— T_i est généré selon la distribution uniforme sur $\{C_i(LO), C_i(LO)+1, ..., T^{MAX}\}$

— $C_i(HI)$ est généré selon la distribution uniforme sur $\{C_i(LO), C_i(LO)+1, ..., min(T_i, R_{HI} * C_i(LO)\}$ si $\chi_i = HI$. $C_i(HI) = C_i(LO)$ sinon.

— $D_i = T_i$, car les tests considèrent des échéances implicites.

Pour ce test, on définira un nombre de tâches CM à générer.

Pour les systèmes de tâches CM ayant pour but d'analyser la performance d'un algorithme d'ordonnancement, un nouveau paramètre est défini : C_{LO}^{MAX}, le temps d'exécution maximum pour faible criticité.
La génération d'une tâche CM τ_i est identique à celle ci-dessus, sauf que :

— $C_i(LO)$ est généré selon la distribution uniforme sur $\{1, 2, ..., C_{LO}^{MAX}\}$

Pour évaluer le taux d'ordonnançabilité des algorithmes, chaque système de tâches CM est généré avec une utilisation moyenne objective $U^{*'}$. Cette utilisation est la moyenne des utilisations pour chacun des niveaux de criticité.

Comme il est compliqué d'obtenir une utilisation exacte en travaillant avec des valeurs entières, on autorise l'utilisation moyenne du système de tâches CM à se trouver entre $U_{min}^{*'} = U^{*'} - 0.005$ et $U_{max}^{*'} = U^{*\prime} + 0.005$.

Tant que $U_\tau^* < U_{min}^{*'}$, on continue d'ajouter des tâches CM dans l'ensemble. Si $U_\tau^* > U_{max}^{*'}$ alors on vide l'ensemble et on en régénère un. Si un système de tâches CM satisfait $U_{min}^{*'} \leq U_\tau^* \leq U_{min}^{*'}$, celui-ci est fini sauf si $\chi_i = \chi_j \; \forall \; \tau_i, \tau_j \in \tau$: t.q $\tau_i \neq \tau_j$ ou $U_\tau(HI) > 1$ ou $U_\tau(LO) > 1$. Aussi, il faut qu'il y ait au moins une tâche CM de criticité élevée dont le pire WCET de haute criticité est strictement supérieur à celui du niveau bas.

Chacun de ces ensembles est aussi généré avec un nombre de tâches CM souhaité ; une fois l'ensemble généré, s'il n'a pas la taille désirée, on le vide et on recommence le processus.

5.1.2 Implémentation et exécution

Tous les algorithmes ont été implémentés en python 3. Les tests ont été réalisés sur le supercalculateur de l'ULB, *HYDRA* [1].

5.2 Analyse de la complexité

Ici, les ensembles générés de tâches CM générées ont pour paramètre $P_{HI} = 0.5, T^{MAX} = 30, R_{HI} = 2$ et le temps d'exécution en criticité basse suit une distribution exponentielle de moyenne $0.35T_i$.

1500 systèmes de tâches CM ont été générés avec 2, 3 ou 4 tâches CM par ensemble (500 ensembles pour chacun de ces nombres). Ces ensembles ont ensuite été explorés en tant que systèmes de tâches CM périodiques, sporadiques, avec et sans antichaîne. Les résultats qui suivent analysent la complexité de ces explorations. Lorsqu'on parle d'états explorés, ceux-ci comprennent les états intermédiaires, même s'ils ne sont pas vraiment visités.

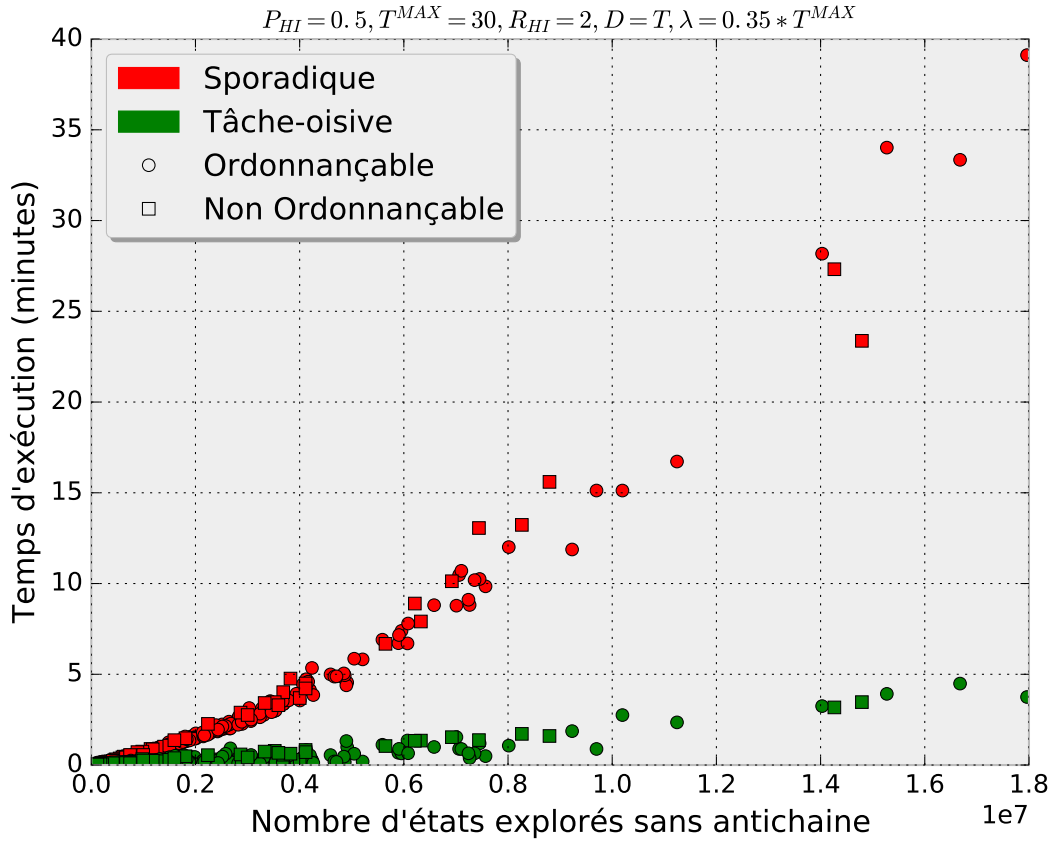

FIGURE 5.1 – Analyse de la performance de la relation de simulation

1. https://cc.ulb.ac.be/hpc/hydra.php

Sporadique : avec antichaîne vs sans antichaîne : la Figure 5.1 compare le temps d'exécution de l'exploration d'un automate généré depuis un système de tâches CM sporadiques jusqu'à arrêt. La comparaison est entre l'exploration avec et sans antichaîne.
Chacun des points sur le graphique est un système de tâches CM différent, ceux-ci ont le temps d'exécution qu'il a fallu pour l'explorer, et le nombre d'états explorés sans antichaîne. Les points rouges sont les résultats sans antichaîne, et les verts sont ceux avec antichaîne. De plus, chacun de ces points est rond si le système de tâches CM est effectivement ordonnançable, sinon il est carré.
Ces résultats présentent une nette amélioration du temps d'exécution avec une antichaîne.

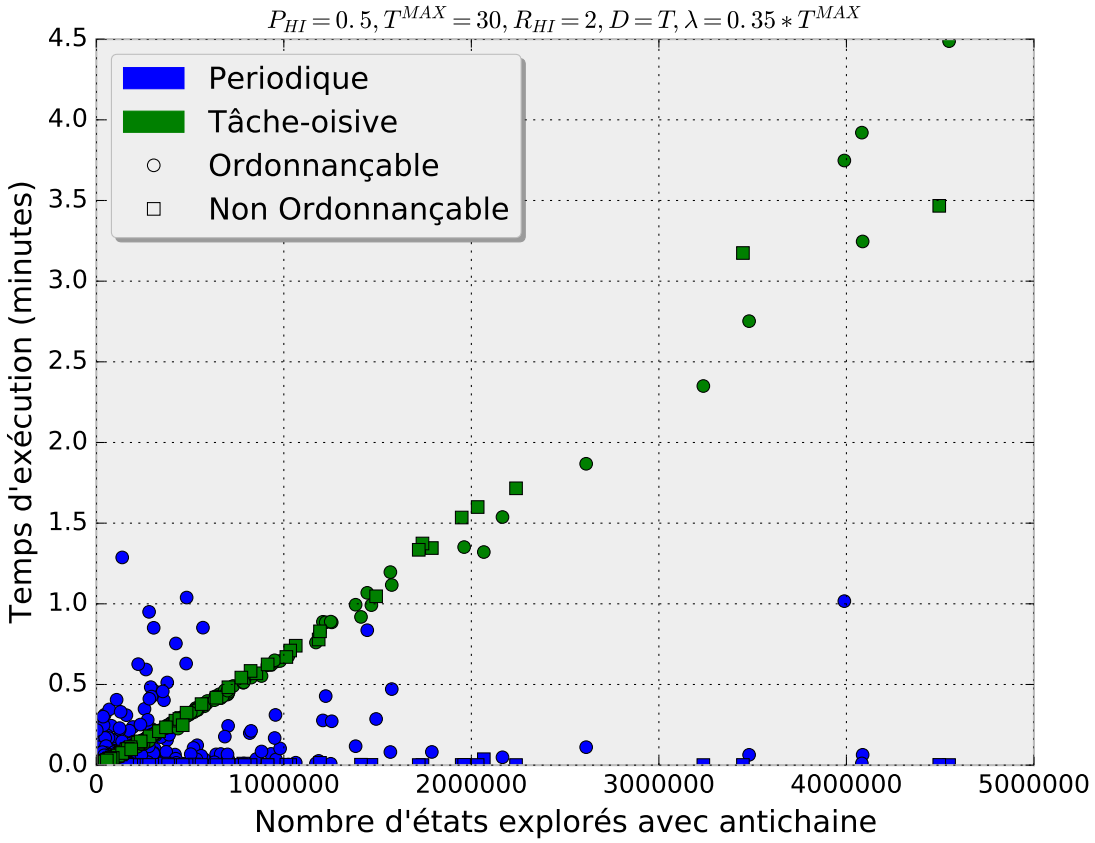

FIGURE 5.2 – Analyse de la performance de la relation de simulation

Périodique vs sporadique avec antichaîne : la Figure 5.2 compare le temps d'exécution du test d'ordonnançabilité par exploration basé sur tâches CM périodiques et sporadiques avec antichaîne.
Ici les points sont tracés sur un graphique du temps d'exécution en fonction du nombre d'états explorés avec antichaîne.
Le temps d'exécution de l'exploration pour tâches CM périodiques est parfois supérieur

pour les ensembles où l'exploration pour tâches CM sporadiques avec antichaîne prend peu de temps. Ce comportement vient du fait que les antichaînes permettent une représentation efficiente des états et donc l'algorithme peut passer moins de temps à vérifier si un état a déjà été exploré, parce qu'il y en a moins. Mais dès que le système de tâches CM génère un scénario plus compliqué à explorer, il est clair que les tâches CM périodiques génèrent beaucoup moins de cas que les tâches CM sporadiques.

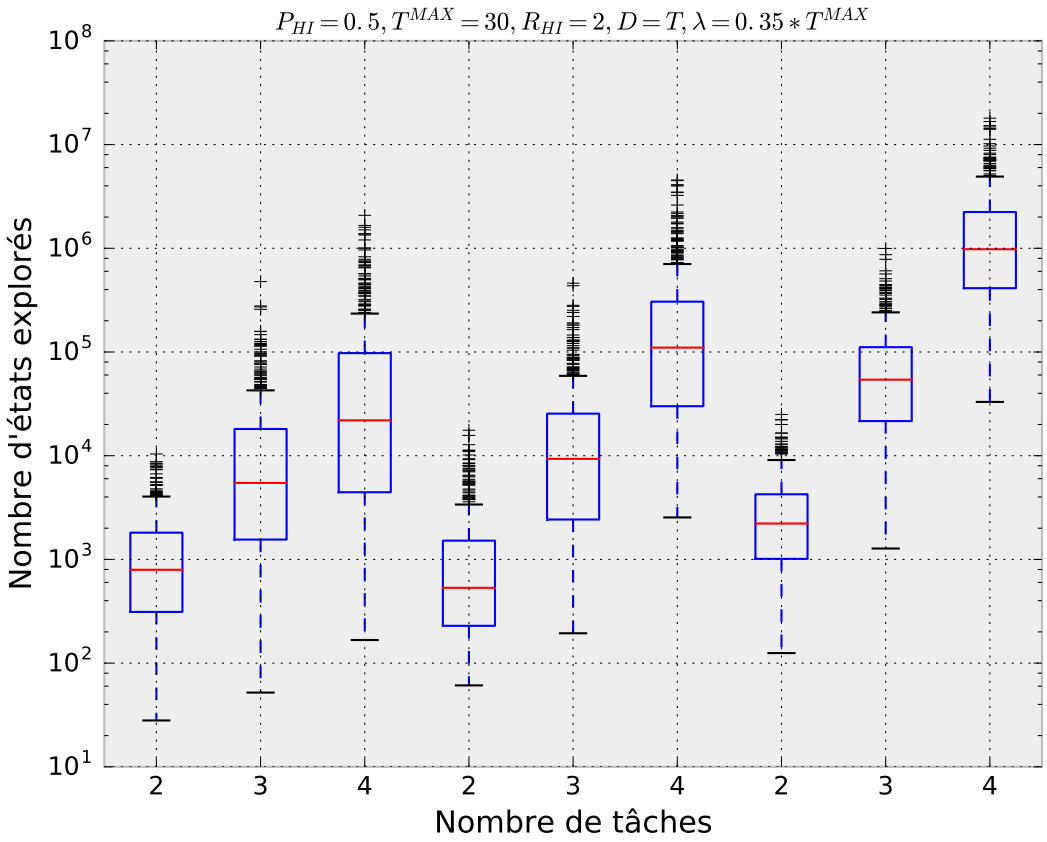

FIGURE 5.3 – Taille de l'automate en fonction du nombre de tâches CM

Complexité en espace : sur la Figure 5.3 on peut lire le nombre d'états visité pour chacun des types d'exploration, pour chaque nombre de tâches CM testé. Il est à noter que l'échelle pour le nombre d'états exploré est logarithmique.
Les explorations sont dans l'ordre, périodique, sporadique sans antichaîne, sporadique avec antichaîne.
La première observation est que le nombre d'états visité est exponentiel en fonction du nombre de tâches CM présentes dans le système de tâches CM. Pour chacune des tailles des systèmes de tâches CM, l'exploration sporadique avec antichaîne est inférieure à celle sans. L'exploration sporadique avec antichaîne est aussi plus gourmande que la périodique, sauf pour une taille de deux. Cependant, on remarque que la taille de l'au-

tomate pour tâches CM sporadiques augmente plus rapidement que celle pour tâches CM périodiques.

Ces analyses permettent de visualiser le gain de temps et d'espace net entre l'exploration sporadique avec et sans antichaîne. Comme soupçonné, l'exploration pour tâche CM périodique est bien moins complexe que celle pour tâche CM sporadique même si les antichaînes permettent de réduire cet espace.

Dans la suite des résultats, lorsqu'on parlera d'exploration pour tâche CM sporadique, il s'agira toujours de celle avec antichaîne. En effet, celle-ci est moins complexe et mène au même résultat.

5.3 Analyse de l'ordonnancement

Dans cette section, une comparaison de la performance des algorithmes d'ordonnancement pour tâches CM sporadiques est présentée. Dans un premier temps, les différentes versions des algorithmes basés sur le même principe sont comparées. Ensuite, une comparaison générale des algorithmes est analysée.

Pour chacun de ces tests, le taux d'ordonnançabilité d'un algorithme est tracé en fonction de l'utilisation du système de tâches CM. Chaque point est le résultat de 2000 simulations, le taux d'ordonnançabilité est alors le nombre de systèmes de tâches CM sporadiques ordonnançables divisé par le nombre total de systèmes de tâches CM testés pour cette utilisation, c'est-à-dire 2000. 41 utilisations sont tracées, il s'agit de $0.4 + (x/40) * 0.6 \; pour \; 0 \leq x \leq 40, \; x \in \mathbb{N}$. L'utilisation ne va donc pas en dessous de 40%, d'autres analyses ont montré que le taux d'ordonnançabilité est de 100% dans de telles conditions [26].

Les systèmes de tâches CM sont générés comme expliqué dans la première section de ce chapitre. Les paramètres de la génération sont $P_{HI} = 0.5, T^{MAX} = 30, R_{HI} = 2, C_{LO}^{MAX} = 15$. Le nombre de tâches CM par ensemble est de 4.

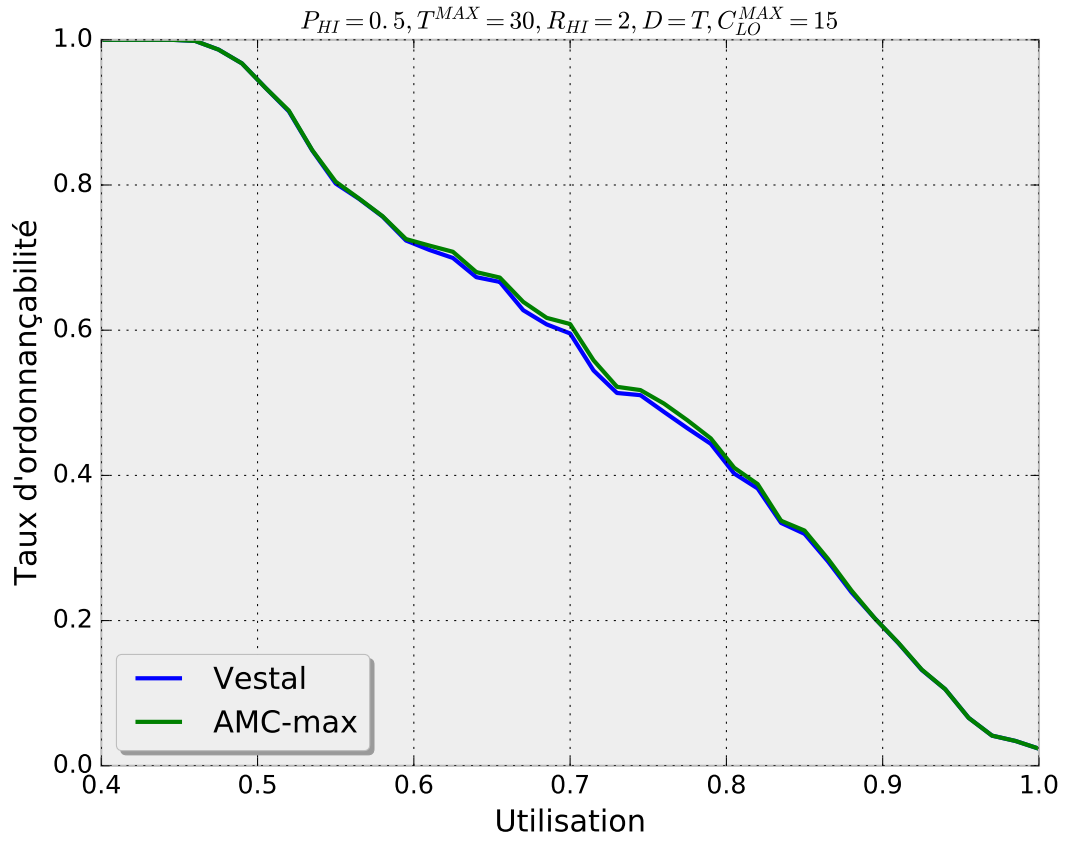

FIGURE 5.4 – Ordonnançabilité de Vestal et AMC-max

Vestal vs AMC-max : la Figure 5.4 compare l'ordonnançabilité de *Vestal* et *AMC-max*, basée sur leur assignation de priorité initiale présentée en section 3.3.
Ici l'algorithme *AMC-max* borne supérieurement les résultats de *Vestal*. *AMC-max* ayant une condition pour vérifier qu'une tâche CM peut recevoir une certaine priorité plus laxiste que celle de *Vestal*, ce résultat est celui attendu.

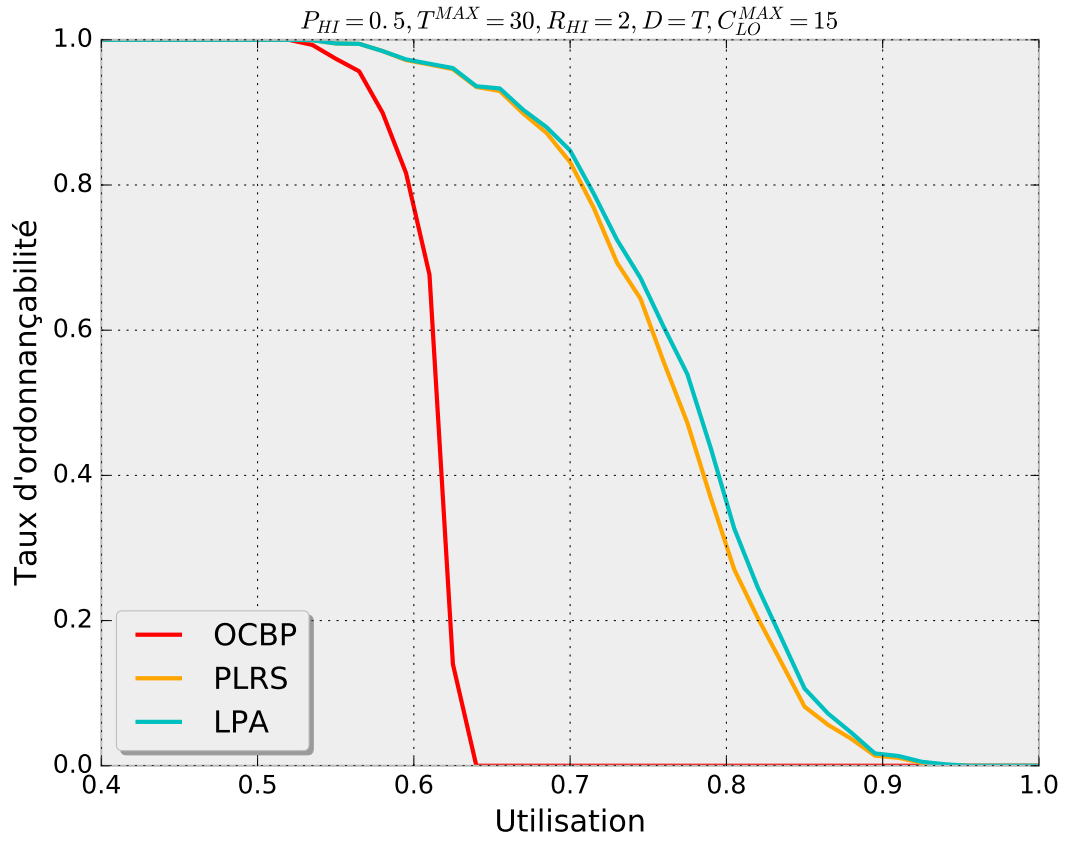

FIGURE 5.5 – Ordonnançabilité de OCBP et extensions

OCBP et extensions : la Figure 5.5 compare les performances de *OCBP* et de ses extensions *PLRS* et *LPA*.
Le test d'*OCBP* basé sur la charge d'un système de tâches CM est clairement plus pessimiste que celui des deux autres, basés sur une assignation de priorité initiale présentée en section 3.5.

En comparant l'ordonnançabilité de *PLRS* et *LPA*, *LPA* offre un résultat légèrement meilleur. L'inverse serait étonnant, car *LPA* a la même assignation de priorité que *PLRS*, sauf que *LPA* borne le plus grand intervalle occupé avec plus de précision que *PLRS*, menant à moins d'échecs d'assignation.

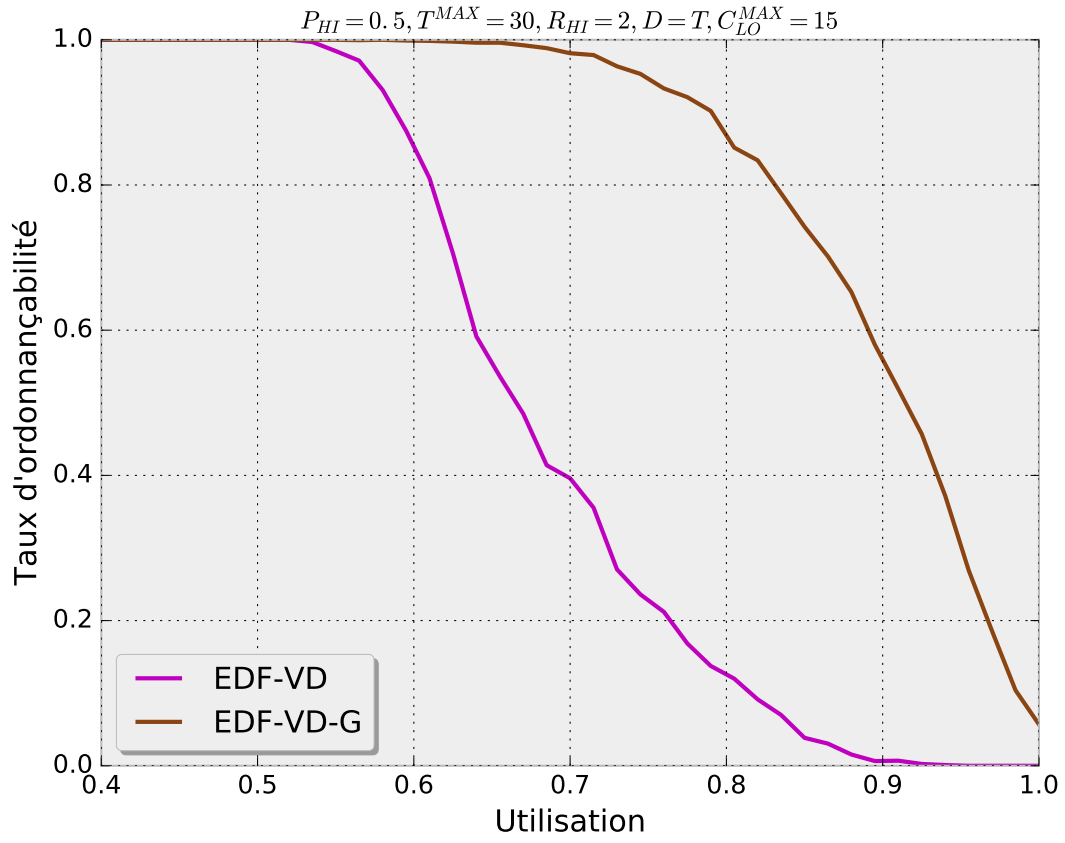

FIGURE 5.6 – Ordonnançabilité de EDF-VD

EDF-VD : la Figure 5.6 compare le test d'ordonnancement préalable de *EDF-VD* basé sur l'utilisation à différents niveaux de criticité et l'ordonnançabilité réelle de l'algorithme, définie grâce à la réduction de cette question vers l'exploration d'un automate. Lorsque *EDF-VD* est utilisé pour l'exploration d'un automate, le cas 2 (*cf.* section 3.6) est utilisé même si la condition n'est pas respectée.

La différence entre le test d'ordonnançabilité préalable et l'ordonnançabilité réelle de *EDF-VD* est flagrante. L'ordonnançabilité de *EDF-VD* est bien plus grande que celle qui était connue jusqu'à présent.

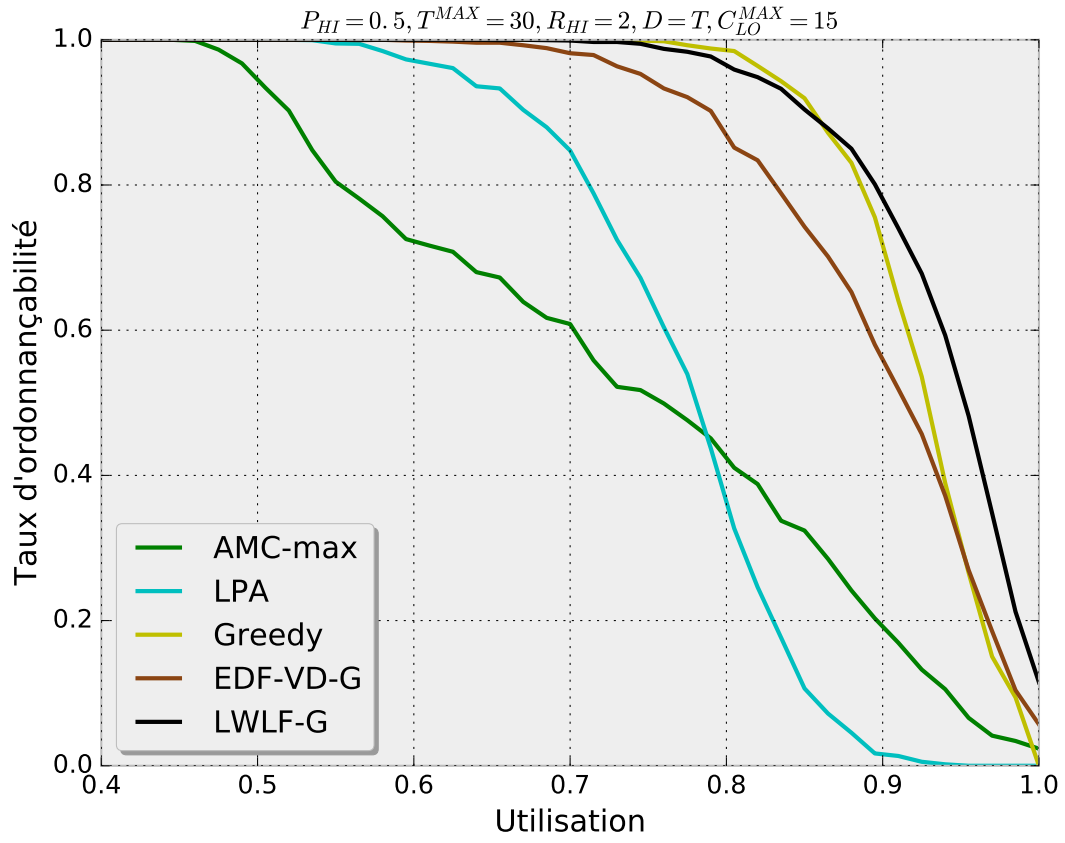

FIGURE 5.7 – Comparaison des algorithmes

Comparaison générale : la Figure 5.7 compare les différents algorithmes entre eux. Les versions les plus optimisées, et les meilleurs tests sont utilisés. Pour l'algorithme *Greedy*, le taux d'ordonnançabilité à 100% est placé arbitrairement à 0, car le calcul utilisé pour déterminer Δ requiert que l'utilisation soit inférieure à 1.

AMC-max donne le plus mauvais résultat jusqu'à une utilisation à 80% où *LPA* prend alors la relève. *LPA* base son test notamment sur le calcul de la plus grande période occupée et celle-ci est calculée à partir de l'utilisation du système de tâches CM. Lorsque cette utilisation est égale à 1, la plus grande période occupée est alors infinie, et cet algorithme est inutilisable. Ensuite vient *EDF-VD*, qui borne les deux précédents algorithmes. Puis *Greedy* et *LWLF* s'entremêlent. L'ordonnançabilité de *LWLF* est calculée grâce à la réduction de l'ordonnançabilité vers l'accessibilité dans un automate. *Greedy* et *LWLF* ont un taux d'ordonnançabilité identique jusqu'à 75%, ensuite *Greedy* domine jusqu'à environ 85%, où *LWLF* reprend le dessus.

Chapitre 6

Conclusion

Ce mémoire a commencé par définir l'ordonnancement en criticité mixte, par expliquer son importance et celle de son test. En deuxième lieu, il a été proposé de réduire ce problème vers celui de l'accessibilité dans un automate. Les notions nécessaires à la compréhension de ce second problème ont été données. Ensuite, la notion d'antichaîne et de relation de simulation ont été expliquées, ainsi que l'amélioration qu'elles apportent à l'accessibilité dans un automate.

La seconde partie du travail regroupait les différents algorithmes d'ordonnancement d'un système de tâches CM sporadiques, en donnant leur fonctionnement général et leur test d'ordonnançabilité. En plus de ces algorithmes venant de la littérature, l'algorithme original *LWLF* a été proposé.

La troisième partie du mémoire a décrit comment réduire l'ordonnançabilité CM vers l'accessibilité dans un automate, apportant le test d'ordonnançabilité CM exact pour un algorithme. Deux problèmes ont été réduits, celui concernant les systèmes de tâches CM périodiques et celui pour les systèmes de tâches CM sporadiques. Enfin, cette partie a proposé la relation de simulation de tâches oisives pour les systèmes de tâches CM sporadiques.

Tout ce travail a permis de comparer les taux d'ordonnançabilité CM de différents algorithmes, mais avant d'en tirer des conclusions, il faut déjà apprécier l'outil en lui-même, et le gain que les antichaînes lui apportent.

En effet, l'ajout d'une antichaîne à l'automate a permis de gagner un temps considérable durant l'exploration. De plus, la relation de simulation de tâches oisives semble pouvoir être exportée pour différents types d'ordonnancement lors de l'utilisation de tâches CM sporadiques, comme c'est le cas pour l'ordonnancement classique sur multiprocesseur [14].

De plus, la création de ce test d'ordonnançabilité CM exacte démocratise le problème dont il est issu. L'ordonnançabilité en criticité mixte selon un algorithme pourrait alors devenir lui-même un problème vers lequel un autre est réduit.

Enfin, cette réduction de problème est un canal entre deux domaines des sciences informatiques. Utiliser des outils interdisciplinaires est toujours souhaitable, car cela permet de résoudre plus de problèmes et cette démarche est une réussite pour ce travail.

Lors de la comparaison de l'ordonnançabilité des algorithmes entre eux, on s'aperçoit que les trois meilleurs sont *Greedy*, *LWLF* et *EDF-VD*.
Greedy donne une très bonne ordonnançabilité, c'est intéressant, car il est basé sur un test en temps polynomial. En revanche, l'algorithme *LWLF* est plus puissant que *Greedy* pour des systèmes de tâche CM à haute utilisation.

Ce dernier résultat est très intéressant, car il montre qu'avec l'outil créé dans ce mémoire, il sera possible de tester toute une série d'algorithmes issue de l'imagination, sans pour autant directement devoir trouver une condition suffisante d'ordonnançabilité théorique. Ce qui a été le cas pour *LWLF* et cet algorithme, basé sur une simple intuition, donne l'un des meilleurs résultats.

Travaux ultérieurs Pour continuer dans le développement du travail effectué dans ce mémoire, une première étape serait de développer une implémentation de la réduction proposée en un langage plus optimisé, tel que *c++*. Cette approche n'a pas été suivie ici, car l'approche utilisée a fonctionné sur essais et erreurs, par recherche.
Un gain dans la complexité en temps permettrait de faire des analyses plus poussées du taux d'ordonnançabilité d'algorithmes, telles que faire varier les différents paramètres de création de systèmes de tâches CM (T^{MAX}, R_{HI} ...), tester avec des systèmes de tâches CM plus grands, avec plus de niveaux de criticités, avec des échéances arbitraires.

Un futur travail pourrait aussi explorer la création d'autres relations de simulations. Des pistes infructueuses sont données à la fin du chapitre présentant la relation de simulation de tâches oisives.

Ce mémoire a proposé l'algorithme *LWLF* qui a donné de bons résultats. Continuer à explorer cet algorithme permettrait peut-être de trouver une condition suffisante d'ordonnançabilité à complexité polynomiale par exemple.

En possession d'un tel test d'ordonnançabilité exact, il est possible d'imaginer une série de nouveaux algorithmes et de tester leur potentiel rapidement. Cet outil peut alors aider la conception d'un hypothétique algorithme optimal pour l'ordonnancement CM.

Enfin, pour obtenir un test d'ordonnançabilité CM indépendant d'un algorithme d'ordonnancement, il faudrait envisager de créer un jeu d'accessibilité basé sur l'automate présenté dans ce mémoire. L'agent aurait le pouvoir sur la tâche CM que l'ordonnanceur sélectionne, le reste ferait partie de l'environnement. Il s'agirait alors de voir s'il est possible de trouver une stratégie gagnante et si oui, alors le système de tâches CM sur lequel est basé l'automate est ordonnançable CM.

Annexe A

Implémentation en Python 3

A.1 Modèles de donnés

A.1.1 Tâche CM

```
class Task:
    def __init__(self, O, T, D, X, C):
        self.O = O
        self.T = T
        self.D = D
        self.X = X
        self.C = tuple(C)

    def getUtilisation(self, l):
        return self.C[l-1]/self.T

    def getWorstC(self):
        return self.C[self.X-1]

    def __repr__(self):
        return "Task("+str(self.O)+", "+str(self.T)+", "+str
 ↪  (self.D)+", "+str(self.X)+", "+str(self.C)+")"

    def generateJobForItv(self, itv):
        nb = 0
        while itv >= 0:
            itv -= self.T
            nb += 1
```

```
23          return nb
24
25      def __hash__(self):
26          return hash((self.O, self.T, self.D, self.X, self.C
    ↪ ))
27
28      def __eq__(self, other):
29          return self.O == other.O and self.T == other.T and
    ↪ self.D == other.D and self.X == other.X and self.C ==
    ↪ other.C
```

A.1.2 Système de Tâches CM

```
1  from Task import *
2
3  class TaskSet:
4      def __init__(self, tasks=[]):
5          self.tasks = tasks
6
7      def addTask(self, task):
8          self.tasks.append(task)
9
10     def clear(self):
11         self.tasks = list()
12
13     def getMaxO(self):
14         res = 0
15         for task in self.tasks:
16             if task.O > res:
17                 res = task.O
18         return res
19
20     def getMaxT(self):
21         res = 0
22         for task in self.tasks:
23             if task.T > res:
24                 res = task.T
25         return res
```

```
26
27      def getMaxC(self):
28          res = 0
29          for task in self.tasks:
30              currentMax =  max(task.C)
31              if currentMax > res:
32                  res = currentMax
33          return res
34
35      def getK(self):
36          res = 0
37          for task in self.tasks:
38              if task.X > res:
39                  res = task.X
40          return res
41
42      def getSize(self):
43          return len(self.tasks)
44
45      def getTask(self, i):
46          return self.tasks[i]
47
48      def getT(self):
49          res = []
50          for t in self.tasks:
51              res.append(t.T)
52          return res
53
54      def getO(self):
55          res = []
56          for t in self.tasks:
57              res.append(t.O)
58          return res
59
60      def getD(self):
61          res = []
62          for t in self.tasks:
63              res.append(t.D)
```

```
64          return res
65
66      def getX(self):
67          res = []
68          for t in self.tasks:
69              res.append(t.X)
70          return res
71
72      def getC(self):
73          res = []
74          for t in self.tasks:
75              res.append(t.C)
76          return res
77
78      def getMaxCs(self):
79          res = []
80          for task in self.tasks:
81              res.append(max(task.C))
82          return res
83
84      def getUtilisationOfLevelAtLevel(self, K, l):
85          ut = 0.0
86          for i in range(self.getSize()):
87              if self.getTask(i).X == K:
88                  ut += self.getTask(i).getUtilisation(l)
89          return ut
90
91      def getUtilisationOfLevel(self, K):
92          ut = 0.0
93          for i in range(self.getSize()):
94              if self.getTask(i).X >= K:
95                  ut += self.getTask(i).getUtilisation(K)
96          return ut
97
98      def getAverageUtilisation(self):
99          if self.getSize() == 0:
100             return 0
101         res = 0.0
```

```
102             for i in range(1,self.getK()+1):
103                 res += self.getUtilisationOfLevel(i)
104             return res/self.getK()
105 
106         def __repr__(self):
107             return "TaskSet("+str(self.tasks)+")"
108 
109         def __getitem__(self,index):
110             return self.getTask(index)
111 
112         def __hash__(self):
113             return hash(tuple(self.tasks))
114 
115         def copy(self):
116             return TaskSet(self.tasks)
117 
118         def __eq__(self, other):
119             return self.tasks == other.tasks
```

A.2 Test d'ordonnançabilité CM

A.2.1 Vestal

```
1  from Task import *
2  from TaskSet import *
3  from math import ceil
4 
5  class Vestal:
6      def __init__(self, ts):
7          self.ts = ts
8          self.prio = [0]*ts.getSize()
9 
10     def getWorstResponseTimeStep(self, i, r):
11         res = 0
12         for j in range(self.ts.getSize()):
13             if True:
14                 if self.prio[j] <= self.prio[i]:
```

```
                    interference = (ceil(r/self.ts.getTask(j
↪ ).T))*self.ts.getTask(j).C[self.ts.getTask(i).X-1]
                    res += interference
        return res

    def testPriority(self, i, p):
        initialPriority = self.prio[i]
        self.prio[i] = p
        current = self.ts.getTask(i).getWorstC()
        new = None
        while (True):
            new = self.getWorstResponseTimeStep(i, current)
            if new > self.ts.getTask(i).D:
                self.prio[i]=initialPriority
                return False
            if current == new:
                return True
            current = new

    def assign(self):
        lowestPriority = self.ts.getSize()-1
        notAssigned = list(range(self.ts.getSize()))
        while lowestPriority >= 0:
            found = False
            assigned = None
            for i in notAssigned:
                isEligible = self.testPriority(i,
↪ lowestPriority)
                if isEligible:
                    assigned = i
                    found = True
                    break
            if found:
                notAssigned.remove(assigned)
                lowestPriority -= 1
            else:
                break

```

```
51          if lowestPriority < 0:
52              return True
53          else:
54              return False
55
56      def test(self):
57          return self.assign()
```

A.2.2 AMC-max

```
1  from Task import *
2  from TaskSet import *
3  from math import ceil, floor
4
5  class AMCmax:
6      def __init__(self, ts):
7          self.ts = ts
8          self.prio = [0 for i in range(ts.getSize())]
9
10     def getWorstResponseTimeStep(self, i, r, l):
11         res = self.ts.getTask(i).getWorstC()
12         for j in range(self.ts.getSize()):
13             if self.ts.getTask(j).X >= l:
14                 if self.prio[j] < self.prio[i]:
15                     interference = (ceil(r/self.ts.getTask(j
   ↪  ).T))*self.ts.getTask(j).C[l-1]
16                     res += interference
17         return res
18
19     def getLoInterference(self, i, s):
20         res = 0
21         for j in range(self.ts.getSize()):
22             if self.ts.getTask(j).X == 1:
23                 if self.prio[j] < self.prio[i]:
24                     interference = ((floor(s/self.ts.getTask
   ↪  (j).T))+1)*self.ts.getTask(j).C[1-1]
25                     res += interference
26         return res
```

```
27
28      def getMaxGeneration(self, k, t, s):
29          res1 = ceil(t/self.ts.getTask(k).T)
30          res2 = ceil((t-s-(self.ts.getTask(k).T - self.ts.
    ↪ getTask(k).D))/self.ts.getTask(k).T)+1
31          return min(res1, res2)
32
33      def getHiInterference(self, i, t, s):
34          res = 0
35          for j in range(self.ts.getSize()):
36              if self.ts.getTask(j).X >= 2:
37                  if self.prio[j] < self.prio[i]:
38                      Mstk = self.getMaxGeneration(j, t, s)
39                      interference = Mstk*self.ts.getTask(j).C
    ↪ [2-1]
40                      interference += (ceil(t/self.ts.getTask(
    ↪ j).T) - Mstk)*self.ts.getTask(j).C[1-1]
41                      res += interference
42          return res
43
44      def getResponseTimeSwitchStep(self, i, r, s):
45          res = self.ts.getTask(i).C[2-1]
46          res += self.getLoInterference(i, s)
47          res += self.getHiInterference(i, r, s)
48          return res
49
50      def isALoGenerationPoint(self, s):
51          for i in range(self.ts.getSize()):
52              task = self.ts.getTask(i)
53              if task.X == 1:
54                  if s%task.T == 0:
55                      return True
56          return False
57
58      def testPriority(self, i, p):
59          initialPriority = self.prio[i]
60          self.prio[i] = p
61          current = 0
```

```
62          new = None
63
64          while (True):
65              new = self.getWorstResponseTimeStep(i, current,
    ↪ 1)
66
67              if new > self.ts.getTask(i).D:
68                  self.prio[i]=initialPriority
69                  return False
70              if current == new:
71                  break
72              current = new
73
74          if self.ts.getTask(i).X == 1 :
75              return True
76          riLO = current
77
78          current = 0
79          new = None
80          while (True):
81              new = self.getWorstResponseTimeStep(i, current,
    ↪ 2)
82              if new > self.ts.getTask(i).D:
83                  self.prio[i]=initialPriority
84                  return False
85              if current == new:
86                  break
87              current = new
88
89          for s in range(riLO):
90              if self.isALoGenerationPoint(s):
91                  current = 0
92                  new = None
93                  while (True):
94                      new = self.getResponseTimeSwitchStep(i,
    ↪ current, s)
95                      if new > self.ts.getTask(i).D:
96                          self.prio[i]=initialPriority
```

```
                            return False
                        if current == new:
                            break
                        current = new
            return True

    def assign(self):
        lowestPriority = self.ts.getSize()-1
        notAssigned = list(range(self.ts.getSize()))

        while lowestPriority > 0:
            found = False
            assigned = None
            for i in notAssigned:
                isEligible = self.testPriority(i,
↪ lowestPriority)
                if isEligible:
                    #priority already assigned
                    assigned = i
                    found = True
                    break
            if found:
                notAssigned.remove(assigned)
                lowestPriority -= 1
            else:
                break

        if lowestPriority == 0:
            return True
        else:
            return False

    def test(self):
        return self.assign()
```

A.2.3 OCBP

```
import math
```

```
2  from TaskSet import *
3  from functools import reduce
4
5  def gcd(numbers):
6
   ↪  """Return the greatest common divisor of the given integers"""
7      from fractions import gcd
8      return reduce(gcd, numbers)
9
10 def lcm(numbers):
11     """Return lowest common multiple."""
12     def lcm(a, b):
13         return (a * b) // gcd((a, b))
14     return reduce(lcm, numbers, 1)
15
16
17 class OCBP:
18     def __init__(self, ts):
19         self.tasks = ts
20
21     def getLoad(self, X):
22         pass
23
24     def getDemandBound(self, task, t, X):
25         return max(0, (math.floor((t-self.tasks[task].D)/
   ↪  self.tasks[task].T)+1)*self.tasks[task].C[X-1])
26
27     def getSumDBF(self, t, X):
28         res = 0
29         for task in range(self.tasks.getSize()):
30             if self.tasks[task].X >= X:
31                 res += self.getDemandBound(task, t, X)
32         res /= t
33         return res
34
35     def getHyperPeriod(self):
36         return lcm(self.tasks.getT())
37
```

```
38      def getNextLoadTry(self, counter):
39          instant = 0
40          task = None
41          for i in range(self.tasks.getSize()):
42              current = self.tasks[i].D + counter[i]*self.
    ↪  tasks[i].T
43              if current < instant or task == None:
44                  instant = current
45                  task = i
46          return instant, task
47
48      def getExactLoad(self, X):
49          HP = self.getHyperPeriod()
50          counter = [0]*self.tasks.getSize()
51          instant = 0
52          load = 0
53          while (instant < HP):
54              instant, task = self.getNextLoadTry(counter)
55              counter[task] += 1
56              currentLoad = self.getSumDBF(instant, X)
57              if currentLoad > load:
58                  load = currentLoad
59          return load
60
61      def test(self):
62          return self.getExactLoad(2)**2 + self.getExactLoad(1
    ↪  ) <= 1
```

A.2.4 PLRS

```
1  import math
2  from functools import reduce
3
4  def gcd(numbers):
5
    ↪  """Return the greatest common divisor of the given integers"""
6      from fractions import gcd
7      return reduce(gcd, numbers)
```

```
8
9  def lcm(numbers):
10     """Return lowest common multiple."""
11     def lcm(a, b):
12         return (a * b) // gcd((a, b))
13     return reduce(lcm, numbers, 1)
14
15
16 class PLRS:
17     def __init__(self, ts):
18         self.tasks = ts
19         self.instance = []
20         self.nbJobs =  [0]*ts.getSize()
21
22     def getLoad(self, X):
23         pass
24
25     def getDemandBound(self, task, t, X):
26         return max(0, (math.floor((t-self.tasks[task].D)/
   ↪ self.tasks[task].T)+1)*self.tasks[task].C[X-1])
27
28     def getSumDBF(self, t, X):
29         res = 0
30         for task in range(self.tasks.getSize()):
31             if self.tasks[task].X >= X:
32                 res += self.getDemandBound(task, t, X)
33         res /= t
34         return res
35
36     def getHyperPeriod(self):
37         return lcm(self.tasks.getT())
38
39     def getNextLoadTry(self, counter):
40         instant = 0
41         task = None
42         for i in range(self.tasks.getSize()):
43             current = self.tasks[i].D + counter[i]*self.
   ↪ tasks[i].T
```

```
44              if current < instant or task == None:
45                  instant = current
46                  task = i
47          return instant, task
48
49      def getExactLoad(self, X):
50          HP = self.getHyperPeriod()
51          counter = [0]*self.tasks.getSize()
52          instant = 0
53          load = 0
54          while (instant < HP):
55              instant, task = self.getNextLoadTry(counter)
56              counter[task] += 1
57              currentLoad = self.getSumDBF(instant, X)
58              if currentLoad > load:
59                  load = currentLoad
60          return load
61
62      def getBusyPeriod(self):
63          dmax = max(self.tasks.getD())
64          exactLoad1 = self.getExactLoad(1)
65          exactLoad2 = self.getExactLoad(2)
66
67          if 1-exactLoad1 < 0.00000001 or 1-exactLoad2 <
    ↪ 0.00000001 :
68              return "Fail"
69
70          x1 = (exactLoad1/(1-exactLoad1))*dmax
71          x2 = (exactLoad2/((1-exactLoad1)*(1-exactLoad2)))*
    ↪ dmax
72          return x1+x2
73
74      def testPriority(self, i, p):
75          res = 0
76          for j in range(self.tasks.getSize()):
77              task = self.tasks[j]
78              res += self.nbJobs[j]*task.C[self.tasks[self.
    ↪ instance[i]].X-1]
```

```
79          rhs = (self.nbJobs[self.instance[i]]-1)*self.tasks[
    ↪ self.instance[i]].T+self.tasks[self.instance[i]].D
80          return res <= rhs
81
82      def assign(self):
83          lowestPriority = len(self.instance)-1
84          notAssigned = list(range(len(self.instance)))
85          while lowestPriority >= 0:
86              found = False
87              assigned = None
88              for i in notAssigned:
89                  isEligible = self.testPriority(i,
    ↪ lowestPriority)
90                  if isEligible:
91                      self.nbJobs[self.instance[i]] -= 1
92                      assigned = i
93                      found = True
94                      break
95              if found:
96                  notAssigned.remove(assigned)
97                  lowestPriority -= 1
98              else:
99                  break
100
101         if lowestPriority < 0:
102             return True
103         else:
104             return False
105
106     def initAssign(self):
107         BP = self.getBusyPeriod()
108         if BP == "Fail":
109             return False
110         for i in range(self.tasks.getSize()):
111             task = self.tasks[i]
112             nbjobsPerTask = task.generateJobForItv(BP)
113             for j in range(nbjobsPerTask):
114                 self.instance.append(i)
```

```
115                 self.nbJobs[i] = nbjobsPerTask
116             return(self.assign())
117 
118         def test(self):
119             return self.initAssign()
```

A.2.5 LPA

```
1  import math
2  from functools import reduce
3  
4  def gcd(numbers):
5  
   ↪  """Return the greatest common divisor of the given integers"""
6      from fractions import gcd
7      return reduce(gcd, numbers)
8  
9  def lcm(numbers):
10     """Return lowest common multiple."""
11     def lcm(a, b):
12         return (a * b) // gcd((a, b))
13     return reduce(lcm, numbers, 1)
14 
15 class LPA:
16     def __init__(self, ts):
17         self.tasks = ts
18         self.instance = []
19         self.nbJobs =  [0]*ts.getSize()
20         self.Ld = [0]*(ts.getK()+1)
21 
22     def getBusyPeriod(self):
23         res = self.computeGamma(self.tasks.getK())
24         return res
25 
26     def testPriority(self, i, p):
27         res = 0
28         for j in range(self.tasks.getSize()):
29             task = self.tasks[j]
```

```
30              res += self.nbJobs[j]*task.C[self.tasks[self.
    ↪ instance[i]].X-1]
31          rhs = (self.nbJobs[self.instance[i]]-1)*self.tasks[
    ↪ self.instance[i]].T+self.tasks[self.instance[i]].D
32          return res <= rhs
33
34      def assign(self):
35          lowestPriority = len(self.instance)-1
36          notAssigned = list(range(len(self.instance)))
37          while lowestPriority >= 0:
38              found = False
39              assigned = None
40              for i in notAssigned:
41                  isEligible = self.testPriority(i,
    ↪ lowestPriority)
42                  if isEligible:
43                      #priority already assigned
44                      self.nbJobs[self.instance[i]] -= 1
45                      assigned = i
46                      found = True
47                      break
48              if found:
49                  notAssigned.remove(assigned)
50                  lowestPriority -= 1
51              else:
52                  break
53
54          if lowestPriority < 0:
55              return True
56          else:
57              return False
58
59      def initAssign(self):
60          BP = self.getBusyPeriod()
61          if BP == "Fail":
62              return False
63          for i in range(self.tasks.getSize()):
64              task = self.tasks[i]
```

```
65              nbjobsPerTask = task.generateJobForItv(BP)
66              for j in range(nbjobsPerTask):
67                  self.instance.append(i)
68              self.nbJobs[i] = nbjobsPerTask
69          return(self.assign())
70
71
72      def computeGamma(self, X):
73          if X == 0:
74              return 0.0
75
76          ld = self.computeGamma(X-1)
77          if ld == ("Fail"):
78              return "Fail"
79
80          sumCHigherX = 0.0
81          sumConTHigherX = 0.0
82          for i in range(self.tasks.getSize()):
83              if self.tasks[i].X >= X:
84                  sumCHigherX += self.tasks[i].C[X-1]
85                  sumConTHigherX += (self.tasks[i].C[X-1]*1.0)
    ↪ /self.tasks[i].T
86
87          gamma = (ld+sumCHigherX)
88          if 1-sumConTHigherX <0.00000001:
89              return ("Fail")
90          gamma /= (1.0-sumConTHigherX)
91
92          returnSum = 0.0
93          for i in range(self.tasks.getSize()):
94              if self.tasks[i].X == X:
95                  returnSum += self.tasks[i].C[X-1]*(1+math.
    ↪ floor(gamma/self.tasks[i].T))
96          res = self.Ld[X-1] + returnSum
97          self.Ld[X] = res
98          return res
99
100     def test(self):
```

```
101             return self.initAssign()
```

A.2.6 EDF-VD

```
1  class EDFVD:
2      def __init__(self, ts):
3          self.ts = ts
4
5      def test(self):
6          res = self.ts.getUtilisationOfLevelAtLevel(1,1)
7          if self.ts.getUtilisationOfLevelAtLevel(2,2) < 1:
8              res += min(self.ts.getUtilisationOfLevelAtLevel(
   ↪  2,2), self.ts.getUtilisationOfLevelAtLevel(2,1)/(1-self.
   ↪  ts.getUtilisationOfLevelAtLevel(2,2)))
9          else:
10             res += self.ts.getUtilisationOfLevelAtLevel(2,2)
11         return res <= 1
```

A.2.7 Greedy

```
1  from TaskSet import *
2  from Task import *
3  import math
4  from functools import reduce
5
6  def gcd(numbers):
7
   ↪  """Return the greatest common divisor of the given integers"""
8      from fractions import gcd
9      return reduce(gcd, numbers)
10
11 def lcm(numbers):
12     """Return lowest common multiple."""
13     def lcm(a, b):
14         return (a * b) // gcd((a, b))
15     return reduce(lcm, numbers, 1)
16
17 class Greedy:
```

```
18      def __init__(self, ts):
19          self.ts = ts
20          self.Dlo = [0] * ts.getSize()
21
22      def test(self):
23          return self.tuneDeadlines()
24
25      def dbfLO(self, i, delta):
26          res = delta - self.Dlo[i]
27          res /= self.ts[i].T
28          res = math.floor(res)
29          res += 1
30          res *= self.ts[i].C[1-1]
31          return max(0, res)
32
33      def full(self, i, delta):
34          res = delta - (self.ts[i].D-self.Dlo[i])
35          res /= self.ts[i].T
36          res = math.floor(res)
37          res += 1
38          res *= self.ts[i].C[2-1]
39          return max(0, res)
40
41      def done(self, i, delta):
42          n = delta % self.ts[i].T
43          if (self.ts[i].D > n and n >= (self.ts[i].D-self.Dlo
    ↪ [i])):
44              res = self.ts[i].C[1-1] - n + self.ts[i].D -
    ↪ self.Dlo[i]
45              return max(0, res)
46          else:
47              return 0
48
49      def dbfHI(self, i, delta):
50          return self.full(i, delta) - self.done(i, delta)
51
52      def conditionA(self, delta):
53          res = 0
```

```
54          for i in range(self.ts.getSize()):
55              res += self.dbfLO(i, delta)
56          return res <= delta
57
58      def conditionB(self, delta):
59          res = 0
60          for i in range(self.ts.getSize()):
61              if self.ts[i].X == 2:
62                  res += self.dbfHI(i, delta)
63          return res <= delta
64
65      def fLO(self, i):
66          return (self.ts[i].C[1-1], self.Dlo[i], self.ts[i].T
    ↪ )
67
68      def fHI(self, i):
69          return (self.ts[i].C[2-1], self.ts[i].D-self.Dlo[i],
    ↪ self.ts[i].T)
70
71      def getL(self, tasks):
72          c = 0
73          for t in tasks:
74              c += t[0]/t[2]
75          if c > 1:
76              return "Fail"
77
78          Ts = []
79          Ds = []
80          Cs = []
81          for t in tasks:
82              Ts.append(t[2])
83              Ds.append(t[1])
84              Cs.append(t[0])
85
86          P = lcm(Ts)
87          D = max(Ds)
88          M = P+D
89          if c == 1:
```

```
90              T = M
91          else:
92              T = min(M, math.ceil(c/(1-c))* max(map(int.
    ↪ __sub__,Ts, Ds)))
93          return T
94
95      def nbToDlo(self, nb, Ds, base):
96
97          res = list(base)
98          for i in range(len(res)-1):
99              toDiv = reduce(int.__mul__, Ds[i+1:])
100             div = nb // toDiv
101
102             rem = nb % toDiv
103             res[i] += div
104             nb = rem
105
106         res[-1] += nb
107         return res
108
109
110     def getLmax(self):
111         res = 0
112         Ds = self.ts.getD()
113         Ds = list(map(int.__add__, Ds, len(Ds)*[1]))
114         base = len(Ds)*[0]
115         size = len(Ds)*[0]
116
117         for i in range(self.ts.getSize()):
118             if self.ts[i].X == 1:
119                 Ds[i] = 1
120                 base[i] = self.ts[i].D
121             else:
122                 Ds[i] -= self.ts[i].C[1-1]
123                 base[i] = self.ts[i].C[1-1]
124
125         final = reduce(int.__mul__, Ds)
126         current = 0
```

```
127         while current < final:
128             self.Dlo = self.nbToDlo(current, Ds, base)
129 
130             tLO =[]
131             for i in range(self.ts.getSize()):
132                 tLO.append(self.fLO(i))
133             lLO = self.getL(tLO)
134 
135             tHI =[]
136             for i in range(self.ts.getSize()):
137                 if self.ts[i].X == 2:
138                     tHI.append(self.fHI(i))
139             lHI = self.getL(tHI)
140 
141             if lHI == "Fail" or lLO == "Fail":
142                 return "Fail"
143 
144             res = max(res, lHI, lLO)
145             current += 1
146         return res
147 
148     def tuneDeadlines(self):
149         self.Dlo = self.ts.getD()
150 
151         candidates = []
152         for i in range(self.ts.getSize()):
153             if self.ts[i].X == 2 and self.ts[i].D > self.ts[
    ↪ i].C[1-1]:
154                 candidates.append(i)
155 
156         mod = "Fail"
157         Lmax = self.getLmax()
158         if Lmax == "Fail":
159             return False
160 
161         while True:
162             final = True
163             for L in range(Lmax+1):
```

```
164                     if not self.conditionA(L):
165                         if mod == "Fail":
166                             return False
167                         self.Dlo[mod] += 1
168                         try:
169                             candidates.remove(mod)
170                         except:
171                             pass
172                         mod = "Fail"
173                         final = False
174                     elif not self.conditionB(L):
175                         if len(candidates) == 0:
176                             return False
177                         argmax = 0
178                         imax = None
179                         for i in candidates:
180                             current = self.dbfHI(i, L) - self.
    ↪ dbfHI(i, L-1)
181                             if current > argmax or imax == None:
182                                 argmax = current
183                                 imax = i
184                         mod = imax
185                         self.Dlo[mod] -= 1
186                         if self.Dlo[mod] == self.ts[mod].C[1-1]:
187                             candidates.remove(mod)
188                         final = False
189                         break
190                 if final:
191                     return True
```

A.3 Automate périodique

A.3.1 Etat du système

```
1 class SystemState:
2     def __init__(self, at, rct, crit, hashInfo):
3         self.at = at
4         self.rct = rct
```

```
5           self.crit = crit
6           self.hashInfo = hashInfo
7
8       def getActive(self):
9           res = []
10          for i in range(len(self.at)):
11              if self.at[i] < 0 or (self.at[i] == 0 and self.
    ↪  rct[i] > 0):
12                  res.append(i)
13          return res
14
15      def getCrit(self):
16          for it in self.getActive():
17              if self.rct[it] == 0:
18                  return self.crit+1
19          return self.crit
20
21      def getLaxity(self, D):
22          laxity = []
23          for i in range(len(self.at)):
24             laxity.append(self.at[i]-self.rct[i]+D[i])
25          return laxity
26
27      def getWorstLaxity(self, C):
28          D = self.hashInfo[0]
29          T = self.hashInfo[1]
30          laxity = []
31          for i in range(len(self.nat)):
32             laxity.append(self.at[i]-(self.rct[i] +(C[i][-1]-
    ↪  C[i][self.crit-1]))+D[i])
33          return laxity
34
35      def isFail(self, D):
36          laxity = self.getLaxity(D)
37          for t in laxity:
38              if t < 0:
39                  return True
40          return False
```

```
41
42      def getImplicitelyDone(self, X):
43          res = []
44          for i in self.getActive():
45              if self.rct[i] == 0 and self.crit == X[i]:
46                  res.append(i)
47          return res
48
49      def getExecutionTransition(self, run, X):
50          critP = self.crit
51          rctP = list(self.rct)
52          for i in run:
53              rctP[i]-=1
54          atP = list(self.at)
55          for i in range(len(self.at)):
56              if X[i] >= self.crit:
57                  atP[i] -= 1
58              else:
59                  atP[i] = 0
60          return SystemState(atP, rctP, critP, self.hashInfo)
61
62      def getTerminationTransition(self, toTerminate, X, C, T
    ↪ ):
63          critP = self.crit
64          toTerminate = toTerminate.union(self.
    ↪ getImplicitelyDone(X))
65          rctP = list(self.rct)
66          atP = list(self.at)
67          for i in toTerminate:
68              rctP[i] = C[i][self.crit-1]
69              atP[i] += T[i]
70          return SystemState(atP, rctP, critP, self.hashInfo)
71
72      def getCriticalTransition(self, X, C):
73          critP = self.getCrit()
74          rctP = list(self.rct)
75          atP = list(self.at)
76          for i in range(len(self.at)):
```

```
77              if X[i] >= self.getCrit():
78                  rctP[i] += C[i][self.getCrit()-1]-C[i][self.
    ↪ crit-1]
79              else:
80                  rctP[i] = 0
81                  atP[i] = 0
82          return SystemState(atP, rctP, critP, self.hashInfo)
83
84      def getRelativeDeadline(self, D, T, i):
85          return self.at[i]+D
86
87      def isWCSimulation(self, ss):
88          if self.crit != ss.crit or self.at != ss.at:
89              return False
90          for i in range(len(self.rct)):
91              if self.rct[i] == 0:
92                  if ss.rct[i] != 0:
93                      return False
94              else:
95                  if self.rct[i] < ss.rct[i]:
96                      return False
97          return True
98
99      def __repr__(self):
100         return("at :"+str(self.at)+"\n rct :"+str(self.rct)+
    ↪ "\n crit :"+str(self.crit))
101
102     def __eq__(self, other):
103         return (self.at == other.at and self.rct == other.
    ↪ rct and self.crit == other.crit)
104
105     def __hash__(self):
106         #hashinfo = D, T, Cmax, K
107         D = self.hashInfo[0]
108         T = self.hashInfo[1]
109         Cmax = self.hashInfo[2]
110         K = self.hashInfo[3]
111         #-D-1 <= at <= T
```

```
112             #0 <= at+D+1 <= T+D+1
113             #0 <= rct <= Cmax
114             #1 <= crit <= K
115             h = 0
116 
117             h = self.crit
118             factor = K
119             for i in range(len(self.at)):
120                 h += self.rct[i]*factor
121                 factor*=(Cmax[i]+1)
122                 h += (self.at[i]+D[i]+1)*factor
123                 factor*=(T[i]+D[i]+1)
124             return h
```

A.3.2 Exploration

```
1  from Task import *
2  from TaskSet import *
3  from Scheduler import *
4  from itertools import *
5  from PeriodicSystemState import SystemState
6 
7  def powerset(iterable):
8 
   ↪  "powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)"
9      s = list(iterable)
10     return chain.from_iterable(combinations(s, r) for r in
   ↪  range(len(s)+1))
11 
12 EDFVD = 0
13 LWLF = 1
14 
15 class PeriodicGraph:
16     def __init__(self, taskSet, sched = EDFVD):
17         self.taskSet = taskSet
18         self.maxO = taskSet.getMaxO()
19         self.maxT = taskSet.getMaxT()
20         self.maxC = taskSet.getMaxC()
```

```
21          self.K = taskSet.getK()
22          self.size = taskSet.getSize()
23          self.nbVisited = 0
24          self.nbInterVisited = 0
25          self.scheduler = Scheduler(taskSet, sched)
26          self.visited = set()
27
28      def getInitialVertex(self):
29          rct0 = []
30          at0 = []
31          crit0 = 1
32          for i in range(self.size):
33              at0.append(self.taskSet.getTask(i).O)
34              rct0.append(self.taskSet.getTask(i).C[0])
35          hashInfo = (self.taskSet.getD(), self.taskSet.getT
    ↪ (), self.taskSet.getMaxCs(), self.taskSet.getK())
36          v0 = SystemState(at0, rct0, crit0, hashInfo)
37          return v0
38
39      def getNeighbour(self, ss):
40          neighbours = []
41          run = self.scheduler.run(ss)
42
43          ssP = ss.getExecutionTransition(run, self.taskSet.
    ↪ getX())
44          self.nbInterVisited += 1
45
46          ssPPs = []
47          for combination in powerset(run):
48              current = ssP.getTerminationTransition(set(
    ↪ combination), self.taskSet.getX(), self.taskSet.getC(),
    ↪ self.taskSet.getT())
49              ssPPs.append(current)
50              self.nbInterVisited += 1
51
52          for ssPP in ssPPs:
53              ssPPP = ssPP.getCriticalTransition(self.taskSet.
    ↪ getX(), self.taskSet.getC())
```

```
54              neighbours.append(ssPPP)
55              self.nbInterVisited += 1
56
57          return neighbours
58
59      def bfs(self):
60          queue = [self.getInitialVertex()]
61          visited = set()
62          while queue:
63              vertex = queue.pop(0)
64              if not vertex in visited:
65                  visited.add(vertex)
66                  self.nbVisited+=1
67                  if (vertex.isFail(self.taskSet.getD())):
68                      return False, self.nbInterVisited, self.
    ↪ nbVisited, self.nbVisited, self.nbVisited
69                  queue.extend(set(self.getNeighbour(vertex)))
70
71          return True, self.nbInterVisited, self.nbVisited,
    ↪ self.nbVisited, self.nbVisited
```

A.4 Automate sporadique

A.4.1 Etat du système

```
1  from functools import reduce
2  from Scheduler import *
3
4  IDLE = 0
5  NONE = 3
6
7  class SystemState:
8      def __init__(self, nat, rct, done, crit, sim, hashInfo,
   ↪ scheduler=0):
9          self.nat = nat
10         self.rct = rct
11         self.done = done
12         self.crit = crit
```

```
13          self.hashInfo = hashInfo
14          self.size = (len(nat))
15          self.SIM = sim
16          self.scheduler = scheduler
17
18      def setSched(self, run):
19          self.run = run
20          self.runned = [(i in run) for i in range(self.size)]
21          self.hashrun = 0
22          for i in range(len(self.runned)):
23              self.hashrun += self.runned[i]*(2**i)
24
25      def getRun(self):
26          return self.run
27
28      def getActive(self):
29          res = []
30          for i in range(len(self.nat)):
31              if not self.done[i]:
32                  res.append(i)
33          return res
34
35      def getCrit(self):
36          for it in self.getActive():
37              if self.rct[it] == 0:
38                  return self.crit+1
39          return self.crit
40
41      def getLaxity(self):
42          D = self.hashInfo[0]
43          T = self.hashInfo[1]
44          laxity = []
45          for i in range(len(self.nat)):
46             laxity.append(self.nat[i]-self.rct[i]+D[i]-T[i])
47          return laxity
48
49      def getWorstLaxity(self, C):
50          D = self.hashInfo[0]
```

```
51          T = self.hashInfo[1]
52          laxity = []
53          for i in range(len(self.nat)):
54              laxity.append(self.nat[i]-(self.rct[i] +(C[i][-1
    ↪ ]-C[i][self.crit-1]))+D[i]-T[i])
55          return laxity
56
57      def isFail(self, C):
58          laxity = self.getWorstLaxity(C)
59          for t in laxity:
60              if t < 0:
61                  return True
62          return False
63
64      def getImplicitelyDone(self, X, C):
65          res = []
66          for i in self.getActive():
67              if self.rct[i] == 0 and C[i][self.crit-1] == C[i
    ↪ ][-1]:
68                  res.append(i)
69          return res
70
71      def getEligible(self, X):
72          res = []
73          for i in range(len(self.nat)):
74              if self.crit <= X[i] and self.done[i] and self.
    ↪ nat[i] <= 0:
75                  res.append(i)
76          return res
77
78      def getExecutionTransition(self, run, X):
79          critP = self.crit
80          doneP = list(self.done)
81          rctP = list(self.rct)
82          natP = list(self.nat)
83          for i in run:
84              rctP[i]-=1
85          for i in range(len(self.nat)):
```

```
86              if not self.done[i]:
87                  natP[i] -= 1
88              else:
89                  natP[i] = max(0, self.nat[i]-1)
90          return SystemState(natP, rctP, doneP, critP, self.
    ↪ SIM, self.hashInfo, self.scheduler)
91
92      def getTerminationTransition(self, toTerminate, X, C):
93          T = self.hashInfo[1]
94          critP = self.crit
95          toTerminate = toTerminate.union(self.
    ↪ getImplicitelyDone(X, C))
96          rctP = list(self.rct)
97          natP = list(self.nat)
98          doneP = list(self.done)
99          for i in toTerminate:
100             rctP[i] = 0
101             doneP[i] = True
102         return SystemState(natP, rctP, doneP, critP, self.
    ↪ SIM, self.hashInfo, self.scheduler)
103
104     def getCriticalTransition(self, X, C):
105         critP = self.getCrit()
106         rctP = list(self.rct)
107         natP = list(self.nat)
108         doneP = list(self.done)
109         for i in range(len(self.nat)):
110             if X[i] >= self.getCrit():
111                 if not self.done[i]:
112                     rctP[i] += C[i][self.getCrit()-1]-C[i][
    ↪ self.crit-1]
113             else:
114                 rctP[i] = 0
115                 natP[i] = 0
116                 doneP[i] = True
117         return SystemState(natP, rctP, doneP, critP, self.
    ↪ SIM, self.hashInfo, self.scheduler)
118
```

```
119     def getRequestTransition(self, request, C):
120         T = self.hashInfo[1]
121         res = []
122         critP = self.getCrit()
123         rctP = list(self.rct)
124         natP = list(self.nat)
125         doneP = list(self.done)
126         for it in request:
127             rctP[it]=C[it][self.crit-1]
128             doneP[it] = False
129         size = []
130         toCombinate = []
131         nbCombinaison = 1
132         for it in request:
133             toCombinate.append(range(natP[it]+T[it], T[it]+1
    ↪ ))
134             size.append(len(toCombinate[-1]))
135             nbCombinaison *= size[-1]
136         for i in range(nbCombinaison):
137             for j in range(len(request)):
138                 div = 1
139                 for k in range(j+1, len(request)):
140                     div *=size[k]
141                 mod = size[j]
142                 natP[request[j]] = toCombinate[j][(i//div)%
    ↪ mod]
143             res.append(SystemState(list(natP), list(rctP),
    ↪ list(doneP), critP, self.SIM, self.hashInfo, self.
    ↪ scheduler))
144         return res
145 
146     def getRelativeDeadline(self, D, T, i):
147         return self.nat[i]-T+D
148 
149     def getRelevantAttributeIdle(self):
150         return tuple(self.nat[i] for i in range(len(self.nat
    ↪ )) if self.done[i])
151 
```

```
152     def __repr__(self):
153         return('\n'+str(self.nat)+", "+str(self.rct)+", "+
    ↪ str(list(map(bool,self.done)))+", "+str(self.crit))
154 
155     def __eq__(self, other):
156         return (self.nat == other.nat and self.rct == other.
    ↪ rct and self.crit == other.crit and self.done == self.
    ↪ done)
157 
158     def __hash__(self):
159         if self.SIM == IDLE:
160             return self.hashIdle()
161         elif self.SIM == NONE:
162             return self.hashNONE()
163 
164     def hashIdle(self):
165         #hashinfo = D, T, Cmax, K
166         D = self.hashInfo[0]
167         T = self.hashInfo[1]
168         Cmax = self.hashInfo[2]
169         K = self.hashInfo[3]
170         #T-(D+1) <= nat <= T
171         #0 <= nat+D+1-T <= D+1
172         #0 <= rct <= Cmax
173         #1 <= crit <= K
174         #0 <= crit-1 <= K-1
175         #0 <= done <= 1
176         h = 0
177 
178         h = self.crit-1
179         factor = K-1+1
180         for i in range(len(self.nat)):
181             h += (Cmax[i]-self.rct[i])*factor
182             factor*=(Cmax[i]+1)
183             h += ((self.done[i]+1)%2)*(self.nat[i]+D[i]+1-T[
    ↪ i])*factor
184             h += 0*factor
185             factor*=((D[i]+1)+1)
```

```
186             h+= int(self.done[i])*factor
187             factor*=(2)
188         return h
189 
190     def hashNONE(self):
191         #hashinfo = D, T, Cmax, K
192         D = self.hashInfo[0]
193         T = self.hashInfo[1]
194         Cmax = self.hashInfo[2]
195         K = self.hashInfo[3]
196         #T-(D+1) <= nat <= T
197         #0 <= nat+D+1-T <= D+1
198         #0 <= rct <= Cmax
199         #1 <= crit <= K
200         #0 <= crit-1 <= K-1
201         #0 <= done <= 1
202         h = 0
203 
204         h = self.crit-1
205         factor = K-1+1
206         for i in range(len(self.nat)):
207             h += (Cmax[i]-self.rct[i])*factor
208             factor*=(Cmax[i]+1)
209             h += (self.nat[i]+D[i]+1-T[i])*factor
210             factor*=((D[i]+1)+1)
211             h+= int(self.done[i])*factor
212             factor*=(2)
213         return h
```

A.4.2 Exploration

```
1 from SporadicSystemState import SystemState
2 from Task import *
3 from TaskSet import *
4 from Scheduler import *
5 from itertools import *
6 from SporadicMaxSet import MaxSet
7 
```

```
8  IDLE = 0
9  NONE = 3
10
11 EDFVD = 0
12 LWLF = 1
13
14 def powerset(iterable):
15     s = list(iterable)
16     return chain.from_iterable(combinations(s, r) for r in
   ↪  range(len(s)+1))
17
18 class SporadicGraph:
19     def __init__(self, taskSet, sim, sched = EDFVD):
20         self.taskSet = taskSet
21         self.maxO = taskSet.getMaxO()
22         self.maxT = taskSet.getMaxT()
23         self.maxC = taskSet.getMaxC()
24         self.K = taskSet.getK()
25         self.size = taskSet.getSize()
26         self.nbVisitedInter = 0
27         self.nbVisited = 0
28         self.scheduler = Scheduler(taskSet, sched)
29         self.SIM = sim
30
31     def setSim(self, sim):
32         self.SIM = sim
33
34     def getInitialVertex(self):
35         rct0 = tuple([0]*self.size)
36         at0 = tuple(self.taskSet.getO())
37         done0 = tuple([True]*self.size)
38         crit0 = 1
39         hashInfo = (self.taskSet.getD(), self.taskSet.getT
   ↪  (), self.taskSet.getMaxCs(), self.taskSet.getK())
40         v0 = SystemState(at0, rct0, done0, crit0, self.SIM,
   ↪  hashInfo)
41         return v0
42
```

```
43      def getNeighbour(self, ss):
44          neighbours = []
45          run = self.scheduler.run(ss)
46
47          ssP = ss.getExecutionTransition(run, self.taskSet.
    ↪ getX())
48          self.nbVisitedInter+=1
49
50          ssPPs = []
51          for combination in powerset(run):
52              current = ssP.getTerminationTransition(set(
    ↪ combination), self.taskSet.getX(), self.taskSet.getC())
53              ssPPs.append(current)
54              self.nbVisitedInter+=1
55
56          ssPPPs = []
57          for ssPP in ssPPs:
58              ssPPP = ssPP.getCriticalTransition(self.taskSet.
    ↪ getX(), self.taskSet.getC())
59              ssPPPs.append(ssPPP)
60              self.nbVisitedInter+=1
61
62          for ssPPP in ssPPPs:
63              for combination in powerset(ssPPP.getEligible(
    ↪ self.taskSet.getX())):
64                  neighbours+= ssPPP.getRequestTransition(
    ↪ combination, self.taskSet.getC())
65                  self.nbVisitedInter+=1
66
67          return neighbours
68
69      def bfsMax(self):
70          self.nbVisitedInter = 0
71          queue = [self.getInitialVertex()]
72          visited = MaxSet(self.SIM)
73          while queue:
74              vertex = queue.pop(0)
75              isNotInSet = visited.add(vertex)
```

```
76              if isNotInSet :
77                  self.nbVisited += 1
78                  if (vertex.isFail(self.taskSet.getC())):
79                      return False, self.nbVisitedInter, self.
    ↪ nbVisited, visited.size, len(visited.set)
80                  queue.extend(self.getNeighbour(vertex))
81          return True, self.nbVisitedInter, self.nbVisited,
    ↪ visited.size, len(visited.set)
82
83      def test(self):
84          return self.bfsMax()[0]
```

A.4.3 Antichaîne

```
1  from functools import reduce
2
3
4  IDLE = 0
5  NONE = 3
6
7  class MaxSet:
8      def __init__(self, sim):
9          self.set = dict()
10         self.size = 0
11         self.SIM = sim
12
13     def add(self, ss):
14         if self.SIM == IDLE:
15             return self.addIdle(ss)
16         elif self.SIM == NONE:
17             return  self.addNONE(ss)
18
19
20     def addIdle(self, ss):
21         h = hash(ss)
22         new = ss.getRelevantAttributeIdle()
23         try:
24             currents = self.set[h]
```

```
25          except KeyError:
26              self.set[h] = set()
27              self.set[h].add(new)
28              self.size += 1
29              return True
30          else:
31              toReplace = []
32              for current in currents:
33                  resIdle = map(int.__le__, current, new)
34                  CurrentSimuleNew = all(resIdle)
35
36                  if CurrentSimuleNew:
37                      return False
38
39                  resIdle = map(int.__le__, new, current)
40                  newSimuleCurrent = all(resIdle)
41
42                  if newSimuleCurrent:
43                      toReplace.append(current)
44
45              if len(toReplace) > 0:
46                      for toRemove in toReplace:
47                          self.set[h].discard(toRemove)
48                          self.size -= 1
49                      self.set[h].add(new)
50                      self.size += 1
51                      return True
52
53              else:
54                  self.set[h].add(new)
55                  self.size += 1
56                  return True
57
58      def addNONE(self, ss):
59          try:
60              t = self.set[ss]
61          except KeyError:
62              self.set[ss] = 0
```

```
63                 self.size += 1
64                 return True
65             else:
66                 return False
67 
68     def __set__(self):
69         return set(self.set.values())
70 
71     def __iter__(self):
72         return self.set.values().__iter__()
73 
74     def __len__(self):
75         return len(self.set)
```

Annexe B

Bibliographie

[1] Joël Goossens. *INFO-F404, Operating Systems II*. Université Libre de Bruxelles, 2014.

[2] Joël Goossens. *Scheduling of hard real-time periodic systems with various kinds of deadline and offset constraints*. PhD thesis, Université Libre de Bruxelles, 1999.

[3] François Santy. Ordonnancement temps réel monoprocesseur de systèmes à criticité mixte. Master's thesis, Université Libre de Bruxelles, 2012.

[4] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1) :46–61, 1973.

[5] Dionisio De Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Real-Time Systems Symposium, 2009*, pages 291–300. IEEE, 2009.

[6] Sven Nordhoff. Do-178c/ed-12c : the new software standard for the avionic industry : goal, changes and challenges. Technical report, SQS Software Quality System, 2012.

[7] James Barhorst, Todd Belote, Pam Binns, Jon Hoffman, James Paunicka, Prakash Sarathy, John Scoredos, Peter Stanfill, Douglas Stuart, and Russell Urzi. A research agenda for mixed-criticality systems. *Cyber-Physical Systems Week*, 2009.

[8] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Real-Time Systems Symposium, 2007.*, pages 239–243. IEEE, 2007.

[9] Erik Ramsgaard Wognsen, René Rydhof Hansen, and Kim Guldstrand Larsen. *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications : 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II*, chapter Battery-Aware Scheduling of Mixed Criticality Systems, pages 208–222. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[10] Sanjoy K. Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Alberto Marchetti-Spaccamela, Suzanne van der Ster, and Leen Stougie. Mixed-criticality scheduling of sporadic task systems. In *Algorithms - ESA 2011 - 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings*, pages 555–566, 2011.

[11] Sanjoy Baruah, Haohan Li, and Leen Stougie. Towards the design of certifiable mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 13–22. IEEE, 2010.

[12] Theodore P Baker and Michele Cirinei. Brute-force determination of multiprocessor schedulability for sets of sporadic hard-deadline tasks. In *Principles of Distributed Systems*, pages 62–75. Springer, 2007.

[13] Sanjoy Baruah. Pré-publication : Mixed criticality schedulability analysis is highly intractable, 2009.

[14] Gilles Geeraerts, Joël Goossens, and Markus Lindström. Multiprocessor schedulability of arbitrary-deadline sporadic tasks : Complexity and antichain algorithm. *Real-time systems*, 49(2) :171–218, 2013.

[15] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[16] Christel Baier and Katoen Joost-Pieter. *Principles of model checking*. MIT press Cambridge, 2008.

[17] Laurent Doyen and Jean-François Raskin. Antichain algorithms for finite automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 2–22. Springer, 2010.

[18] Neil C Audsley. *Optimal priority assignment and feasibility of static priority tasks with arbitrary start times*. Citeseer, 1991.

[19] Sanjoy K Baruah, Alan Burns, and Robert I Davis. Response-time analysis for mixed criticality systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 34–43. IEEE, 2011.

[20] Haohan Li and Sanjoy Baruah. An algorithm for scheduling certifiable mixed-criticality sporadic task systems. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 183–192. IEEE, 2010.

[21] Nan Guan, Pontus Ekberg, Martin Stigge, and Wang Yi. Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 13–23. IEEE, 2011.

[22] Theodore P. Baker. Algorithms for determining the load of a sporadic task system. Technical report, 2005.

[23] Sanjoy K Baruah, Haohan Li, and Leen Stougie. Mixed-criticality scheduling : Improved resource-augmentation results.

[24] Haohan Li and Sanjoy Baruah. Load-based schedulability analysis of certifiable mixed-criticality systems. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 99–108. ACM, 2010.

[25] Chuancai Gu, Nan Guan, Qingxu Deng, and Wang Yi. Improving ocbp-based scheduling for mixed-criticality sporadic task systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2013 IEEE 19th International Conference on*, pages 247–256. IEEE, 2013.

[26] Pontus Ekberg and Wang Yi. Outstanding paper award : Bounding and shaping the demand of mixed-criticality sporadic tasks. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 135–144. IEEE, 2012.

[27] Sanjoy K Baruah, Aloysius K Mok, and Louis E Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 182–190. IEEE, 1990.

Annexe C

Index