UNIVERSITE BRETAGNE LOIRE MATHSTIC

UNIVERSITÉ DE NANTES

THÈSE DE DOCTORAT DE

UNIVERSITÉ DE NANTES
COMUE UNIVERSITÉ BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par
« Khaoula BOUKIR »

« Mise en œuvre de politiques d'ordonnancement temps réel multiprocesseur prouvée »

Thèse présentée et soutenue à « Nantes », le « 16/12/2020 »
Unité de recherche : Laboratoire des sciences du numérique de Nantes

Rapporteurs avant soutenance :

Claire Pagetti — Ingénieur recherche, ONERA
Emmanuel Grolleau — Professeur des universités, ISAE-ENSMA

Composition du Jury :

Présidente :	Isabelle Puaut	Professeur des universités, Université de Rennes 1
Examinateurs :	Olivier Henri Roux	Professeur des universités, Centrale Nantes
	Pascal Richard	Professeur des universités, Université de Poitiers
	Pierre-Emmanuel Hladik	Maître de conférences, INSA de Toulouse
Dir. de thèse :	Jean-Luc Béchennec	Chargé de recherche, CNRS
Co-dir. de thèse :	Anne-Marie Déplanche	Maître de conférences, Université de Nantes

Á mes parents, dont l'amour et le soutien ne ternit jamais.

Á mon petit frère, qui aura à me supporter de nouveau.

Á Anass, à qui aucun mot ne peut décrire ma gratitude pour ses encouragements.

Remerciements

Ce travail de thèse s'est déroulé au sein de l'équipe « Systèmes temps réel » du Laboratoire des Sciences du Numérique de Nantes. Je souhaite ainsi exprimer ma reconnaissance envers toute personne ayant contribué directement ou indirectement à l'issue de ce travail.

Mes premiers remerciements s'adressent à mes deux encadrants Anne-Marie Déplanche et Jean-Luc Béchennec pour leur soutien scientifique, pédagogique et moral. Je leur suis reconnaissante plus particulièrement pour leur grande disponibilité et leur investissement qui m'ont été d'une grande aide. Je les remercie également pour leurs ouverture d'esprit, sympathie et nos longues discussions passionnantes pour changer le monde. J'ai beaucoup appris grâce à eux sur le plan personnel et je garderai que de bons souvenirs.

Je voudrais adresser mes remerciements à Mme Claire Pagetti (Ingénieur de Recherche à l'ONERA) et Mr Emmanuel Grolleau (Professeur à l'ISAE-ENSMA) pour avoir accepté la charge de rapporteurs qui exige certainement un investissement en temps important. Je remercie également Mme Isabelle Puaut (Professeur à l'Université de Rennes 1), Mr Pascal Richard (Professeur à l'Université de Poitiers), Mr Pierre-Emmanuel Hladik (Maître de conférences à l'INSA Toulouse) et Mr Olivier-Henri Roux (Professeur à Centrale Nantes) d'avoir accepté d'évaluer cette thèse.

Mes remerciements s'adressent également aux membres de l'équipe « Systèmes temps réel » du LS2N pour leur accueil, leur aide et leur sympathie. J'ai trouvé dans cette équipe une ambiance conviviale qui me fut précieuse.

Je tiens à remercier aussi mes amis et collègues qui m'ont soutenu durant ces années de thèse.

Enfin, je souhaite exprimer ma gratitude à mes parents Latifa et Mohammed pour leur amour et soutien inconditionnels, à mon frère Abder sur qui je sais que je peux toujours compter et à Anass qui est toujours là pour me pousser à donner le meilleur de moi-même.

Table des matières

Table des figures

CHAPITRE 1

Introduction générale

1.1 Contexte et motivations

1.1.1 Contexte

Les applications informatiques embarquées ont envahi l'environnement industriel ainsi que la vie quotidienne. Leurs domaines d'application sont variés : les transports urbains, l'aviation civile ou militaire, la téléphonie mobile, etc. Les besoins, en terme de vitesse de traitement, de robustesse, de réactivité, d'autonomie et de sûreté de fonctionnement de ces applications sont de plus en plus accrus. Pour répondre à ces besoins, les recherches scientifiques se multiplient autour des trois grands domaines permettant le développement de telles applications :

- l'électronique : études autour des processeurs (microcontrôleurs), de leur consommation, des circuits électroniques dédiés à l'informatique embarquée, etc.
- l'automatique : étude des lois de régulation, de la régulation numérique, etc.
- l'informatique : étude des méthodologies de développement, des systèmes d'exploitation ou exécutifs embarqués, des langages applicatifs, des méthodes de test et de validation, etc.

Le présent travail est résolument orienté vers le domaine de l'informatique. Plus précisément, celui dit des « *systèmes temps réel* », cadre des applications informatiques embarquées.

Qu'est ce qu'un système temps réel : les systèmes temps réel sont de plus en plus présents dans diverses secteurs d'activité tels que l'industrie de production (*e.g.* les usines et les centrales nucléaires), le transport (*e.g.* les systèmes de pilotage embarqués dans l'aéronautique ou l'automobile), etc. Tel que considéré dans ce manuscrit, un système

temps réel est un système de contrôle-commande. Il désigne un système informatique qui se distingue par son aptitude à contrôler un environnement auquel il est connecté, appelé « procédé », en accord avec ses dynamiques et donc dans le respect des contraintes temporelles qu'il impose. Pour ce faire, le système temps réel doit être capable de réagir à la vitesse de l'évolution du procédé contrôlé. De manière schématique, il récupère des données sur l'état du procédé par le biais des capteurs afin de les traiter. Ensuite, suivant le résultat de traitement, il commande le procédé par émission de commandes appropriées via des actionneurs. La dénomination temps réel exprime deux propriétés :

- *temps* qui désigne que "*the correctness of the system depends not only on the logical results of computation, but also on the time at which the results are produced*" tel que défini par John A. Stankovic en 1988 [Sta88].
- *réel* qui désigne la réactivité du système face à l'évolution du procédé [But11].

La conception d'un tel système doit ainsi prendre en compte des contraintes temporelles et veiller à ce qu'elles soient respectées. Il doit donc être prévisible (*predictable*), ce qui signifie que son comportement doit être prévu vis-à-vis des exigences temporelles attendues. Outre ces contraintes, il en existe d'autres, que nous ne traitons pas dans le présent manuscrit, pouvant caractériser un système temps réel, notamment *l'embarquabilité* (capacité de calcul, place mémoire, énergie, etc.) et la *criticité* (fiabilité, sécurité, etc.).

Un système temps réel peut être qualifié de strict ou dur (*hard*), lorsque le non-respect de ses contraintes temporelles n'est pas toléré étant donné qu'il peut conduire à des situations critiques. De l'autre côté, un système temps réel dit souple (*soft*) désigne un système dans lequel le non-respect de ses contraintes peut être toléré jusqu'à une certaine limite.

Architecture d'un système temps réel : un tel système se compose d'une architecture logicielle, soit un ensemble de programmes qui lui permet d'agir sur le procédé, et d'une architecture matérielle permettant l'exécution de ces programmes (cf. Fig. 1.1).

1. *L'architecture logicielle :* cette architecture est constituée typiquement de programmes applicatifs dans une couche dite « *haut niveau* ». Ces programmes assurent les fonctions de contrôle du système temps réel telles que le traitement d'une alarme, l'exécution d'un calcul, etc. Ils sont structurés sous forme de plusieurs entités de base que l'on appelle des *tâches*. Dans de nombreuses situations, l'architecture logicielle intègre également un système d'exploitation temps réel (*RTOS : Real-Time Operating System*), interface permettant de faire le lien entre les programmes applicatifs et l'architecture matérielle du système de contrôle. Il offre aussi des service permettant de contrôler notamment l'exécution, la synchronisation et la coopération entre les tâches. Il est également appelé *exécutif temps réel*.
   Les systèmes d'exploitation doivent répondre aux exigences et complexité croissantes des applications temps réel dont ils sont support, notamment en terme de respect des contraintes temporelles. Afin de garantir le respect de ces exigences, de nombreux

RTOS possèdent un composant qui se charge d'organiser l'exécution des programmes et l'accès aux ressources de calcul tout en respectant les contraintes temporelles. Il s'agit de l'*ordonnanceur*.

2. *L'architecture matérielle :* l'architecture matérielle du système temps réel correspond à l'ensemble des ressources matérielles nécessaires pour l'exécution des programmes de l'architecture logicielle. Cela inclut donc les processeurs (CPU) [1], mais peut également s'étendre à la mémoire, les réseaux, les dispositifs d'entrées/sorties, etc. En fonction du nombre de processeurs et de l'utilisation éventuelle d'un réseau, trois catégories d'architectures matérielles peuvent être distinguées :

   - **architecture monoprocesseur :** cette architecture se compose d'un unique processeur. Tous les programmes du système se partagent ce processeur pour leur exécution.
   - **architecture multiprocesseur/multicœur :** cette architecture est dotée de m processeurs ($m > 1$) sur lesquels peuvent s'exécuter les programmes du système de manière parallèle. Dans ce cas, les processeurs partagent une mémoire commune à laquelle les processeurs accèdent en utilisant un même bus. Il peuvent également partager une horloge commune et un ordonnanceur commun. Il convient de noter qu'un processeur peut avoir plusieurs cœurs fonctionnant simultanément. Dans ce cas, l'architecture est qualifiée multicœur ;
   - **Architecture distribuée :** dans ce type d'architecture, le système temps réel est doté de plusieurs nœuds liés les uns aux autres par l'intermédiaire d'un réseau. L'architecture de chaque nœud peut être de type monoprocesseur ou multiprocesseur. Ce type d'architecture n'est pas utilisé dans le présent travail.

1.1.2 Constat et motivation

Offre riche en matière de politiques d'ordonnancement : avec le progrès technologique, il y a une tendance croissante vers l'utilisation de plateformes constituées de plusieurs ressources de traitement de type multiprocesseur ou encore multicœur afin de combler le besoin, toujours croissant, de capacités de calcul. Ce passage d'architecture monoprocesseur à des architecture multiprocesseur/multicœur a engendré un nombre accru d'études scientifiques en matière d'ordonnancement temps réel multiprocesseur. Ainsi, une pléthore de politiques d'ordonnancement est proposée [CHD15]. Nombre d'entre elles offrent même l'optimalité et permettant, en théorie, une exploitation plus efficace des ressources processeur en plus d'une meilleure gestion en cas de surcharge [DB11]. Les premières recherche en matière d'ordonnancement multiprocesseur ont consisté à adapter les politiques d'ordonnancement monoprocesseur aux architectures multiprocesseurs selon deux catégories. La première consiste à partitionner les tâches entre les processeur de manière à retrouver plusieurs problèmes monoprocesseurs indépendants.

[1] Unités centrales de traitement (*CPU : Central Processing Units*) : ce sont des ressources permettant à des programmes informatiques d'exécuter des instructions machine.

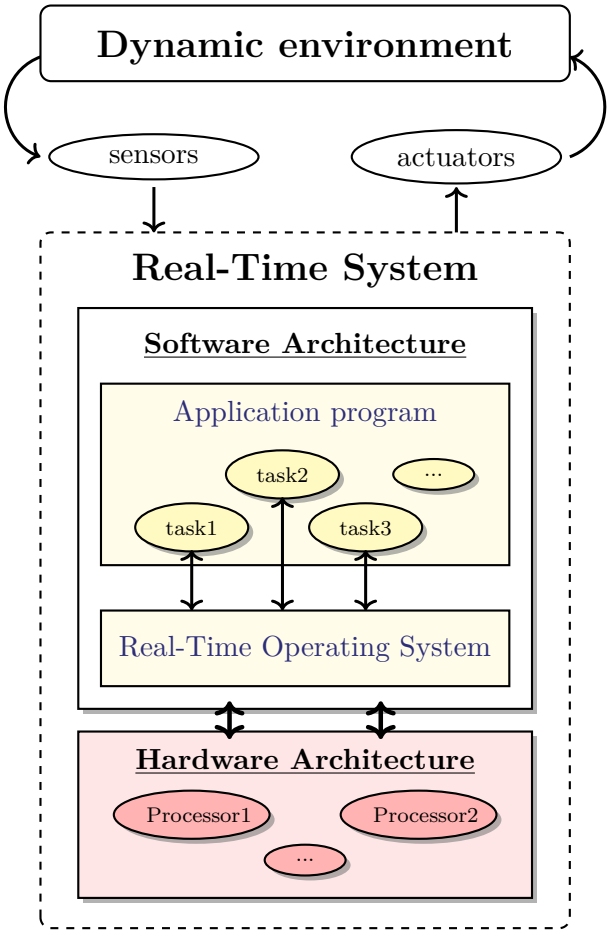


FIGURE 1.1: Architecture d'un système temps réel

La deuxième propose une généralisation des politiques monoprocesseur vers le multiprocesseur qui s'appuie sur une vision globale du système, mais qui présente des limites en termes d'utilisation optimale des processeurs (e.g. G-EDF, G-RM, etc.). Dans les années 90s, de nouvelles approches introduisant l'équité et permettant d'atteindre l'optimalité (e.g. les approches PFair) ont été proposées. Ces dernières ont également donné naissance à d'autres travaux de recherche permettant de contourner le problème du nombre important de préemptions migrations qu'elles engendrent (e.g. RUN, U-EDF, etc.). Dans leur synthèse sur l'ordonnancement temps réel multiprocesseur [CHD15], Cheramy *et al.* identifient plus que cinquante politiques d'ordonnancement multiprocesseur. Une liste qu'ils jugent non exhaustive compte tenu du nombre important des variantes de politiques proposées.

Offre limitée en matière d'implémentation : la mise en œuvre des politiques d'ordonnancement, et en particulier les plus innovantes, au sein d'une plateforme réelle

ne fait pas l'objet de grande attention. En effet, la majorité des systèmes d'exploitation temps réel couramment utilisés, repose encore sur des ordonnanceurs à priorités fixes pour les tâches et/ou des approches ancrées sur le partitionnement dans le cas multiprocesseur. Ce décalage entre l'offre académique en matière de politiques d'ordonnancement et les travaux liés à leurs d'implémentation est dû à la distance entre leurs spécifications très théoriques et le contexte réel au sein duquel elles doivent être réalisées. Cette distance rend difficile le travail de mise en œuvre d'ordonnanceur au sein d'un système d'exploitation temps réel ce qui constitue un frein face à la migration des OS temps réel vers les nouvelles politiques proposées. Ceci soulève même des interrogations sur la faisabilité et/ou les performances effectives des politiques implémentées.

En effet, dans la littérature scientifique, la globalité des politiques d'ordonnancement proposées sont décrites de manière abstraite et algorithmique, avec des hypothèses souvent simplificatrices. Leur description est souvent basée sur un ensemble de N tâches devant s'exécuter sur m processeurs selon les consignes d'un ordonnanceur qui se déclenche lorsque certains événements se produisent. En revanche, le travail d'implémentation d'un ordonnanceur se ramène à un ensemble de programmes qui doivent manipuler des variables, des structures de données, des composants et/ou des fonctions du système d'exploitation. Ces programmes sont tous imbriqués dans le code source du système et interagissent au sein de lui. Dans la théorie, l'algorithme d'ordonnancement suppose qu'à l'arrivée d'un événement d'ordonnancement, il suffit de calculer la séquence en fonction des données de tâches disponibles. Dans la réalité, un tel scénario requiert la gestion des interruptions provoquant ces événements et les programmes réalisant l'ordonnancement sur la base des structures de données des tâches qu'il faut tenir à jour.

Motivation : cette thèse est donc motivée par le constat qu'il y a de nombreuses offres de politiques d'ordonnancement temps réel dans la littérature et très peu d'études visant leur mise en œuvre dans un contexte réel. Notre travail de recherche s'inscrit ainsi dans une démarche à long terme, dont l'objectif est d'étudier de manière approfondie « *l'implémentabilité* » des politiques d'ordonnancement multiprocesseur au sein d'un véritable système d'exploitation temps réel.

1.2 Objectifs

Objectif 1. Implémentation d'une politique d'ordonnancement temps réel global : nous souhaitons étudier de manière générale la mise en œuvre de politiques d'ordonnancement global au sein d'un véritable système d'exploitation temps réel. Pour cela, nous retenons l'exécutif temps réel *Trampoline* [BBFT06] afin de l'étendre et d'y intégrer l'ordonnancement global en multiprocesseur. Nous commençons avec une implémentation de la politique G-EDF mais avec l'intention de pouvoir intégrer d'autres politiques d'ordonnancement global.

Objectif 2. Élaboration d'une démarche de vérification des implémentations

des ordonnanceurs : tel que nous l'avons souligné précédemment, la mise en œuvre d'une politique au sein d'un RTOS est fortement contrainte par le cadre logiciel et matériel qu'il impose. De plus, Trampoline, l'OS retenu pour notre étude, est de type *kernel-based* ; cela signifie que les fonctions relatives à l'ordonnancement sont imbriquées dans le code du noyau de l'OS. Le travail de la mise en œuvre d'un ordonnanceur au niveau du noyau est ainsi rendu difficile et nécessite une parfaite maîtrise du code de l'OS. Un tel travail est potentiellement sujet aux erreurs, notamment car le code OS est écrit en langage C ou Assembleur avec une utilisation intensive de macros et de pointeurs. L'ordonnanceur implémenté pourrait avoir un comportement imprévisible et il convient donc de s'assurer que l'implémentation d'une nouvelle politique d'ordonnancement au sein du RTOS est correcte. Celle-ci doit en effet toujours produire le comportement attendu et conforme aux spécifications fournies par la littérature.

En effet, si nous souhaitons que les résultats théoriques en matières d'ordonnancement se concrétisent au sein des RTOS, les implémentations des politiques d'ordonnancement doivent être soutenues par des moyens de vérification qui fournissent une preuve de leur correction. La vérification manuelle ne peut être une option compte tenu de la complexité et des contraintes d'implémentation. Afin de répondre à cette problématique, nous nous fixons comme objectif d'accompagner la mise en œuvre d'une politique d'ordonnancement global par un travail de vérification permettant de s'assurer que l'implémentation en question est conforme aux attentes.

Utilisation des méthodes formelles : dans le domaine de vérification des systèmes d'exploitation, plusieurs approches existent. De manière générale, la première démarche pour vérifier le bon fonctionnement d'un programme est de le tester, en l'exécutant sous certains cas de test significatifs, et identifier les éventuelles erreurs d'exécution. Le principe étant de stimuler le programme en lui appliquant des valeurs d'entrées et analyser le résultat de son exécution en termes de sorties. Le test logiciel permet le plus souvent de détecter des erreurs liées à la gestion de la mémoire ou des débordements de variables, etc [MBTS04, Mat13]. Toutefois, la confiance que l'on peut accorder à un système ayant passé avec succès une campagne de tests ne peut être totale puisque sa limite est :

« *Program testing can be used to show the presence of bugs, but never to show their absence* » (Edsger Dijkstra[2], 1970).

Le test permet la détection des erreurs pour les cas de test spécifiés, mais il ne garantit pas l'inexistence d'erreurs pour d'autres cas de test ou dans d'autres conditions. D'autant plus quand il s'agit de vérifier des programmes complexes faisant partie du code d'une implémentation au sein d'un système d'exploitation temps réel

[2]**Edsger Wybe Dijkstra** (11/05/1930 - 06/08/2002) était un informaticien et mathématicien néerlandais et un des premiers pionniers dans de nombreux domaines de recherche en informatique. Il reçoit le prix Turing en 1972 pour ses contributions sur la science et l'art des langages de programmation

Une grande famille d'approches de vérification qui a démontré son efficacité existe. Il s'agit des méthodes formelles [Bje05]. Elles désignent des approches qui sont fondées sur des concepts mathématiques et permettent de prouver mathématiquement qu'un système satisfait à sa spécification, au lieu d'observer les traces d'exécution et chasser les erreurs. La vérification est conduite en fournissant des preuves formelles sur des propriétés exprimant la correction du système étudié. Ces approches ont l'avantage de gérer l'aspect exhaustif, qui fait défaut au test logiciel. Deux catégories appartenant à cette approche de vérification se distinguent :

- des méthodes déductives basées sur la preuve de théorèmes (*theorem proving*) [Coo71] : permettant de partir d'un ensemble de formules et d'une propriété, exprimées sous forme d'un théorème, de prouver que le système à vérifier respecte la propriété, grâce à un certain nombre d'axiomes et de règles d'inférence. De telles approches ne sont pas automatisables et exigent le guidage humain pour déduire les preuves.
- des méthodes automatiques basées sur la vérification de modèles (*model-checking*) [CES83] : permettant de parcourir algorithmiquement l'espace d'états complet d'un modèle spécifiant le système à vérifier, afin de valider une certaine propriété. L'avantage de ces approches est la possibilité d'automatiser le parcours de l'espace d'états et de vérifier les propriétés. Ainsi, la procédure de vérification est entièrement automatisable. Cependant, le nombre d'états à parcourir devient souvent très grand, voire même infini dans certains cas, ce qui fait le principal désavantage du model checking.

L'utilisation de ces approches dans plusieurs travaux de recherche a démontré leur intérêt et leur capacité d'identifier les ambiguïtés et les inconsistances qui n'auraient pas été trouvées par les méthodes de test conventionnelles [BS93]. Ainsi, nous nous intéressons dans cette thèse à l'emploi de ces méthodes, notamment le model-checking, afin de définir une démarche de vérification des implémentations d'ordonnanceurs.

1.3 Contribution scientifique

Le travail mené afin de satisfaire les objectifs établis dans la section 1.2 nous conduit à identifier deux contributions majeures. La première contribution consiste à faire migrer le système d'exploitation temps réel Trampoline, qui est conforme à OSEK/VDX et AUTOSAR et caractérisé par un ordonnancement partitionné à priorités fixes vers l'ordonnancement global. Ceci en y implémentant la politique G-EDF. Cette migration consiste donc à modifier les composants et/ou les fonctions de l'OS, et à en rajouter d'autres, afin qu'il supporte l'ordonnancement global dynamique.

La seconde contribution, qui est la contribution majeure de notre travail, est la proposition d'une démarche de vérification générique pour les implémentations d'ordonnanceurs. En s'appuyant sur la contribution précédente, pour l'implémentation de G-EDF dans l'OS étudié, nous appliquons une démarche de vérification dont le but est de vérifier s'il y a une conformité fonctionnelle entre le comportement d'un ordonnanceur, du point de vue spécification algorithmique, et celui de l'ordonnanceur implémenté. Nous

appliquons la même démarche ensuite sur un modèle modélisant une implémentation d'EDF-US[ξ] au sein de Trampoline.

La vérification des implémentations est effectuée en examinant des propriétés comportementales de l'ordonnanceur, déduites sur la base d'une analyse exhaustive de la politique publiée. Ces propriétés, qui sont exprimées dans un langage de description littéraire, doivent être ramenées vers le contexte de l'implémentation de manière à tenir compte de toutes les contraintes et du fonctionnement de l'OS cible. Ainsi, des exigences dépendantes de l'implémentation sont identifiées pour correspondre aux propriétés dépendantes des politiques. Ce sont ces exigences qui sont examinées dans le cadre de notre démarche pour vérifier le comportement de l'ordonnanceur implémenté (cf. Fig 1.2).

Notre démarche de vérification est basée sur le model-checking. Ainsi, elle est menée sur deux phases :

- **spécification de l'implémentation :** cette phase consiste à élaborer un modèle décrivant de manière fidèle le fonctionnement de l'implémentation à vérifier en modélisant les différentes interactions et opérations de l'ordonnanceur au sein de l'OS. Nous nous inspirons dans cette phase d'une thèse réalisée au sein de la même équipe par Toussaint TIGORI [BRT18] qui a proposé un modèle complet de Trampoline conçu en utilisant l'outil UPPAAL [BLL+96]. Ce modèle regroupe toutes les fonctions et les services de l'OS qui ont été ramenés à une combinaison d'automates finis étendus et de fonctions UPPAAL écrites dans une syntaxe similaire au langage C. Notre travail a consisté à adapter le modèle initial de l'OS pour supporter l'ordonnancement global et y intégrer les modèles de l'ordonnanceur et des autres composants du noyau qui contribuent à la décision d'ordonnancement.
- **vérification de l'implémentation :** en matière de vérification, l'objectif est d'examiner la satisfaction des propriétés telles que énoncées dans la littérature scientifique pour la politique implémentée. Pour cela, nous les traduisons en exigences décrivant le comportement attendu de l'implémentation. Ces exigences sont ensuite formalisées sous forme de propriétés pouvant être injectées dans le model-checker UPPAAL afin de statuer sur leur satisfaction ou générer, dans le cas contraire, un contre exemple. La conduite de la vérification nécessite la stimulation du modèle de l'implémentation par des événements d'ordonnancement. Pour cela, nous élaborons un mécanisme permettant de générer de manière indéterministe des événements d'ordonnancement stimulant les modèles des implémentations à vérifier.

1.4 Plan du manuscrit

La thèse est structurée en quatre parties comme suit :

- **partie I :** cette partie présente le contexte général de cette thèse. Elle introduit dans le chapitre 2 les notions de base de l'ordonnancement temps réel en dressant une vue d'ensemble des politiques d'ordonnancement temps réel multiprocesseur publiées dans la littérature. Elle présente brièvement, à travers le chapitre 3, le fonctionnement au

sein des systèmes d'exploitation temps réel, et se termine par le chapitre 4 qui discute le principe des méthodes de vérification formelles.

- **partie II :** cette partie est dédiée à l'implémentation d'un ordonnanceur global au sein de Trampoline. Elle débute par le chapitre 5 dédié à présentation de l'architecture et le fonctionnement de l'OS. Ensuite les travaux menés dans le cadre de l'implémentation de G-EDF sont présentés dans le chapitre 6.
- **partie III :** cette partie précise le principe de la modélisation d'une implémentation d'ordonnanceur. Les notions de base de modélisation sont discutées au début dans le chapitre 7. Ensuite, le modèle pré-existant de l'OS est exposé à la fin de ce chapitre. La modélisation de l'implémentation G-EDF réalisée ainsi qu'un modèle décrivant une implémentation d'EDF-US[ξ] en Trampoline sont détaillés à la fin de cette partie dans le chapitre 8.
- **partie IV :** cette partie est dédiée à l'approche de vérification menée pour les implémentations d'ordonnanceur. Le principe de cette approche est fourni en début de la partie, dans le chapitre 9, avec une instanciation sur une implémentation de G-EDF. Ensuite, des générateurs indéterministes d'événements d'ordonnancement sont proposés à la fin de la partie, dans le chapitre 10, avec un instanciation sur le modèle élaboré pour une implémentation d'EDF-US[ξ].

Enfin, ce mémoire se termine par une conclusion et une présentation des perspectives dégagées à l'issue de nos travaux.

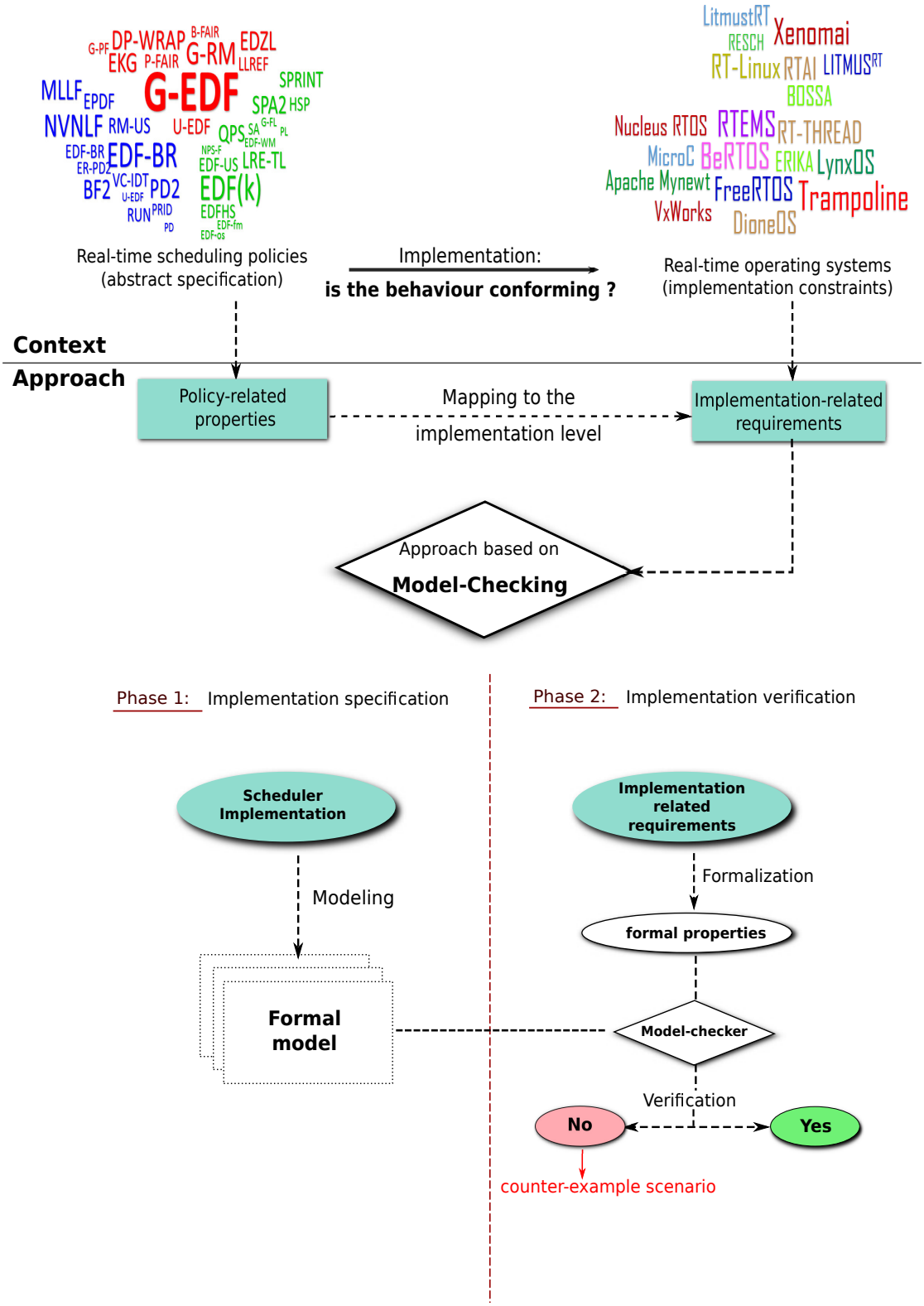


FIGURE 1.2: Démarche de vérification des implémentations

Première partie

Contexte général

Dans cette partie, nous présentons, sous forme de trois chapitres, le contexte général de notre travail de recherche. Le Chapitre 2 introduit les notions de base en matière d'ordonnancement temps réel et fournit un panorama des principales politiques d'ordonnancement temps réel multiprocesseur publiées dans la littérature. Le chapitre 3 est consacré aux systèmes d'exploitation temps réel. Le principe de fonctionnement de ces derniers est examiné avec un accent particulier sur les travaux d'implémentation d'ordonnanceurs. Le Chapitre 4 porte sur les méthodes de vérification formelle en exposant les principes de base des deux familles de vérification formelle : le *theorem proving* et le *model-checking*. Les travaux de recherche utilisant ces méthodes dans la vérification des systèmes d'exploitation sont discutés à la fin de ce chapitre.

CHAPITRE 2

Vue d'ensemble sur l'ordonnancement temps réel

2.1 Modélisation d'un système temps réel et notations

2.1.1 Modélisation de l'architecture logicielle

En matière d'ordonnancement temps réel, le modèle le plus fréquemment utilisé pour les systèmes temps réel est le modèle canonique initialement proposé par Liu et Layland [LL73]. Dans ce modèle, l'architecture logicielle du système temps réel est abstraite par un ensemble de N tâches *(tasks)* noté $\tau = \{\tau_1, \tau_2, ..., \tau_N\}$. Si une tâche est récurrente, elle donne lieu à un ensemble infini de travaux *(jobs)* notés $\tau_i = \{\tau_{i,1}, \tau_{i,2}, ...\}$, tel que $\tau_{i,j}$ correspond au j-ème travail de la tâche τ_i.

L'activation d'une tâche engendre la création d'un travail. Ainsi, la manière dont les travaux sont générés par une tâche permet de faire la distinction entre trois types de tâche :

- *tâches périodiques :* elles sont activées de manière régulière avec une période fixe (notée T_i) qui désigne la durée séparant deux activations successives de travaux ;
- *tâches sporadiques :* elles sont activées de manière irrégulière mais avec une propriété sur la durée minimale séparant deux activations successives de travaux ;
- *tâches apériodiques :* elles sont activées irrégulièrement et utilisées en général pour la gestion des alarmes et des états d'exception.

Nous nous limitons, dans la suite de cette présentation aux tâches périodiques. Selon le modèle de Liu et Layland (cf. Fig. 2.1), chaque tâche périodique τ_i est modélisée par le quadruplet (r_i, C_i, D_i, T_i) où :

- r_i : correspond à la date d'activation de la tâche, il s'agit également de la date d'activation du premier travail de la tâche ;

- C_i : correspond à la durée nécessaire pour l'exécution du code de la tâche. Dans la littérature, deux durées d'exécution sont définies : (i) la durée d'exécution dans le pire cas *(WCET : Worst-Case Execution Time)*, qui correspond à la durée maximale d'exécution ; (ii) la durée d'exécution dans le meilleur cas *(BCET : Best-Case Execution Time)*, qui désigne une durée minimale d'exécution. Le plus souvent, c'est la durée d'exécution maximale qui est considérée dans les recherches en matières d'ordonnancement temps réel.
- D_i : correspond à l'échéance relative ou délai critique de la tâche. Un travail $\tau_{i,j}$ s'activant à une date t doit finir son exécution au maximum à la date $d_{i,j} = t + D_i$. Cette date désigne l'échéance absolue du travail en question ;
- T_i : correspond à la période d'activation de la tâche. Elle permet de déduire les dates d'activation des travaux. $r_{i,j}$ désigne la date d'activation du travail $\tau_{i,j}$.

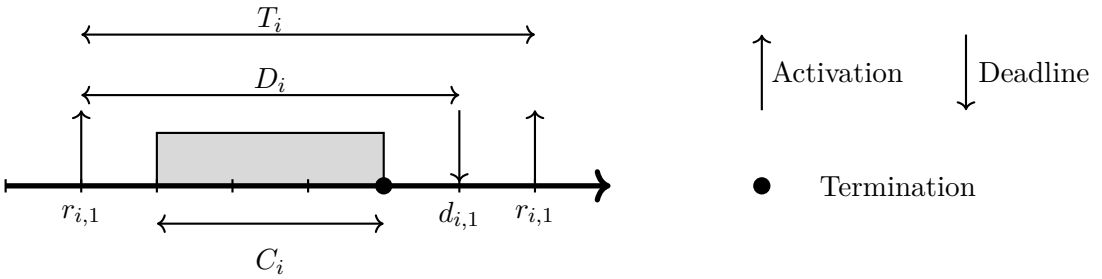


FIGURE 2.1: Modèle canonique d'une tâche périodique

À partir de ce quadruplet, d'autres paramètres caractérisant les tâches peuvent être déduits, nous les présentons ainsi :

- le taux d'utilisation d'une tâche τ_i : désigne la portion de la capacité de traitement qu'une tâche consomme sur un processeur pour son exécution. Il correspond au rapport entre sa durée d'exécution et sa période :

$$U_i \stackrel{def}{=} \frac{C_i}{T_i}$$

- le taux d'utilisation total d'un système de tâches : correspond à la somme des taux d'utilisation des tâches qui le composent :

$$U_{sum} \stackrel{def}{=} \sum_{i=1}^{N} U_i$$

- le taux d'utilisation maximal d'un système de tâches : correspond au plus grand taux d'utilisation des tâches qui le composent :

$$U_{max} \stackrel{def}{=} max\{u_1, ..., u_N\}$$

- la laxité d'un travail $\tau_{i,j}$ à l'instant t : correspond au temps maximum pendant lequel le travail peut retarder son exécution sans qu'il dépasse son échéance :

$$L_{i,j}(t) \stackrel{def}{=} d_{i,j} - t - C_{i,j}(t)$$

Sachant que t désigne la date courante et $C_{i,j}(t)$la durée d'exécution restante au travail $\tau_{i,j}$ à l'instant t.

En fonction de l'échéance relative D_i d'une tâche τ_i, trois types de tâches peuvent être distingués :

- tâche à échéance implicite (*implicit deadline*) : lorsque $D_i = T_i$;
- tâche à échéance contrainte (*constrained deadline*) : lorsque $D_i < T_i$;
- tâche à échéance arbitraire (*arbitrary deadline*) : lorsque l'échéance relative est supérieure à la période de la tâche $D_i > T_i$.

Un système de tâches est composé de tâches *synchrones* quand elles sont toutes activées au même instant : $\forall i, j \in \{1, ..., N\}$ tel que $i \neq j$, nous avons $r_i = r_j$. Dans le cas contraire, le système est qualifié d'*asynchrone*.

2.1.2 Modélisation de l'architecture matérielle

L'ensemble des tâches doit s'exécuter sur un ou plusieurs processeurs, qui modélisent l'architecture matérielle du système temps réel. Cet ensemble est noté $\mathbb{P} = \{P_1, P_2, ..., P_m\}$ tel que $m \in \mathbb{N}^*$. Dans le cas d'une architecture matérielle multiprocesseur, trois types d'architecture peuvent être distingués :

- processeurs homogènes : les capacités de calcul des processeurs qui composent l'architecture sont identiques ;
- processeurs uniformes : chaque processeur k possède une capacité de calcul noté s_k et peut exécuter $s \times t$ unités de temps d'une tâche sur un intervalle de t unités temporelles ;
- processeurs hétérogènes : chaque couple tâche-processeur (τ_i, P_j) a une capacité de calcul notée $c_{i,j}$.

2.1.3 Ordonnancement des tâches

Ordonnancer un système de tâches temps réel revient à définir dans quel ordre et pour quelle durée il faut exécuter les travaux de ces tâches sur chaque processeur de la plateforme, de manière à ce qu'elles ne dépassent pas leurs échéances si le système le permet. Deux catégories d'ordonnancement peuvent se définir, selon l'architecture matérielle du système :

- **Ordonnancement monoprocesseur :** dans ce cas, l'exécution des tâches se fait sur un seul processeur. Le problème d'ordonnancement est d'une seule dimension et revient à déterminer une organisation temporelle de l'exécution des travaux sur le processeur (choisir quelle tâche il faut exécuter à chaque instant).

- **Ordonnancement multiprocesseur :** la plateforme d'exécution étant une architecture multiprocesseur, l'ordonnanceur doit assurer une organisation spatiale (sur quel processeur un travail est exécuté) et une organisation temporelle sur chaque processeur. De ce fait, le problème d'ordonnancement a deux dimensions. Dans ce type d'ordonnancement, les tâches peuvent être amenées à migrer entre processeurs, c'est-à-dire commencer à s'exécuter sur un processeur puis être interrompues à un certain moment pour reprendre leur exécution sur un autre processeur.

Les décisions prises par l'ordonnanceur sont le résultat d'algorithmes d'ordonnancement. Deux grandes classes d'algorithmes existent : (i) des algorithmes **hors ligne** qui construisent une séquence d'exécution des travaux avant le démarrage du système et qui sera répétée indéfiniment ; (ii) des algorithmes **en ligne** pour lesquels l'ordonnancement des tâches se décide au fur et à mesure que ces dernières s'exécutent. De manière classique, ces algorithmes prennent leurs décisions en s'appuyant sur des priorités qu'ils associent aux travaux. Il est possible de classer la gestion de ces priorités en trois catégories [CFH+04] :

- des algorithmes à priorité fixe pour les tâches : pour lesquels les travaux d'une même tâche ont tous la même priorité qui est celle de la tâche.
- des algorithmes à priorité fixe pour les travaux : ce qui signifie que la priorité d'un travail ne change pas durant son exécution. Mais deux travaux d'une même tâche peuvent avoir des priorités différentes.
- des algorithmes à priorité dynamique pour les travaux : pour lesquels les priorités des travaux peuvent évoluer durant leur exécution.

Ordonnanceur préemptif : un ordonnanceur est dit *préemptif* s'il peut interrompre l'exécution d'un travail au profit d'un autre plus prioritaire, d'en mémoriser l'état afin de le relancer plus tard. Cette opération s'appelle une préemption.

La notion de préemption introduit également, dans le contexte des systèmes multiprocesseur, la notion de la migration. Celle-ci consiste à préempter et reprendre son exécution sur un autre processeur. Certains ordonnanceurs permettent aux travaux successifs d'une tâche d'être exécutés sur différents processeurs. Tandis que d'autres ordonnanceurs sont en mesure de préempter un travail et le relancer sur un processeur différent. Ainsi, selon Anderson et. al [ABD05], trois types de migration peuvent être distinguées :

- aucune migration n'est autorisée.
- migration restreinte aux frontières des travaux.
- migration libre des travaux pendant leur exécution.

Ordonnanceur conservatif *(work-conserving)* : un ordonnanceur est qualifié de *conservatif*, s'il ne permet pas de laisser un travail en attente alors qu'un processeur est disponible pour l'exécuter. Ainsi, il ne permet pas de laisser un processeur dans un état oisif (*idle*) tant qu'il y a des travaux pour être exécutés.

2.2 Notion d'ordonnançabilité et d'optimalité

L'étude d'ordonnançabilité consiste à déterminer si un ensemble de tâches est ordonnançable par un algorithme d'ordonnancement sur une architecture matérielle spécifiée. Elle s'appuie sur les définitions suivantes :

> **Définition 2.1.** *Un système S $(\tau + \mathbb{P})$ est fiablement ordonnancé par un algorithme d'ordonnancement A, si et seulement si la séquence d'ordonnancement produite par A est valide (ie. toutes les tâches de τ respectent leurs contraintes temporelles).*

> **Définition 2.2.** *un système S $(\tau + \mathbb{P})$ est ordonnançable, si et seulement s'il existe un algorithme A qui l'ordonnance fiablement.*

Les études d'ordonnançabilité reposent, pour les plus simples, sur les taux d'utilisation des tâches, le taux d'utilisation maximal et le taux d'utilisation global d'un ensemble de tâches. Elles permettent ainsi de déduire des conditions nécessaires (Définition 2.3) et/ou des conditions suffisantes (Définition 2.4) d'ordonnançabilité.

> **Définition 2.3.** *Lorsqu'un système S ne respecte pas une condition nécessaire d'ordonnançabilité d'un algorithme d'ordonnancement A, alors il n'est pas ordonnançable par cet algorithme. En revanche, si un système S respecte une condition nécessaire, alors cela ne veut pas dire que A ordonnancera correctement S.*

> **Définition 2.4.** *Lorsqu'un système S respecte une condition suffisante d'ordonnançabilité d'un algorithme d'ordonnancement A, alors il est ordonnançable par cet algorithme. En revanche, si un système S ne respecte pas une condition suffisante, alors cela ne veut pas dire que A n'est pas en mesure d'ordonnancer correctement S.*

En s'appuyant sur la notion d'ordonnançabilité, un algorithme d'ordonnancement peut être qualifié ou non d'optimal selon la définition suivante :

> **Définition 2.5.** *Un algorithme d'ordonnancement A est dit optimal pour une classe de tâches, une architecture matérielle et parmi une classe de politiques d'ordonnancement, si et seulement si, il peut ordonnancer fiablement tout système S ordonnançable par une politique de cette classe.*

2.3 Classement des politiques d'ordonnancement multiprocesseur

L'allocation spatiale et temporelle des tâches pour leur exécution, ainsi que leur capacité à migrer ou non, donne lieu à trois catégories de politiques d'ordonnancement qui sont discutées dans ce qui suit.

Ordonnancement par partitionnement : ce type d'ordonnancement consiste à partitionner l'ensemble des tâches à exécuter en m sous-ensembles, où m désigne le nombre de processeurs de la plateforme. Ainsi, chaque sous-ensemble se voit affecté à un processeur unique. Les tâches attribuées à un même processeur sont ordonnancées selon un ordonnancement monoprocesseur (cf. Fig. 2.2). Aucune tâche n'est autorisée à migrer vers un autre processeur. Le partitionnement des tâche doit être effectué hors ligne, ce qui nécessite de connaître au préalable l'ensemble des tâches. Ce principe est similaire à celui du *BinPacking* qui consiste à ranger un nombre d'éléments (tâches) caractérisés par leur taille dans un nombre déterminé de sacs ou boites (processeurs). La capacité des processeurs est déterminée par les conditions d'ordonnançabilité monoprocesseur et en se basant sur les caractéristiques des tâches, notamment leur facteur d'utilisation. L'affectation des tâches est alors ramenée à une solution qui utilise un nombre de sacs inférieur ou égal au nombre de processeurs. Cette affectation est déterminée grâce à des heuristiques dont les plus classiques sont :

- *First-Fit :* une tâche est affectée au premier processeur trouvé tel que son ordonnanceur local peut l'ordonnancer avec les tâches déjà affectées ;
- *Next-Fit :* une tâche est affectée au premier processeur trouvé suivant celui dans lequel la tâche précédente a été placée ;
- *Best-Fit :* une tâche est affectée au processeur ayant la plus petite capacité disponible et permettant d'ordonnancer la tâche ;
- *Worst-Fit :* une tâche est affectée au processeur ayant la plus grande capacité disponible et permettant d'ordonnancer la tâche ;

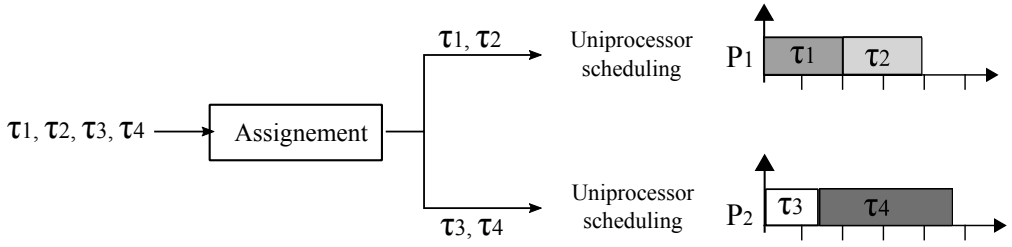


FIGURE 2.2: Schéma de principe de l'ordonnancement par partitionnement.

Ordonnancement global : le principe pour ce type d'ordonnancement est de n'appliquer qu'un seul ordonnancement pour tout le système (ensemble de tâches + processeurs) et de ne pas restreindre l'exécution d'une tâche sur un unique processeur. Ainsi, une seule liste de travaux à exécuter est organisée. De manière schématique, l'ordonnanceur décide, selon une règle de priorité, l'attribution des m travaux les plus prioritaires aux m processeurs du système (cf. Fig. 2.3). Ces travaux sont autorisés à migrer d'un processeur à un autre au cours de leur exécution.

Ordonnancement hybride : il s'agit d'une combinaison entre l'ordonnancement partitionné et l'ordonnancement global. Ceci donne lieu à deux approches possibles :

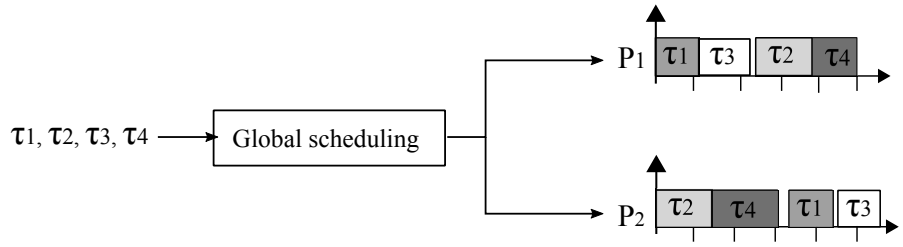


FIGURE 2.3: Schéma de principe de l'ordonnancement global.

- *l'ordonnancement semi partitionné :* le principe consiste à partitionner au maximum les tâches à exécuter jusqu'à ce que le système ne soit plus partitionnable. Les tâches qui ne peuvent pas s'exécuter entièrement sur un processeur sont ainsi autorisées à migrer sur d'autres processeurs selon deux niveaux de contrôle de migration :
  - *restricted migration :* dans le cas où la migration n'est possible qu'entre les travaux. Ce qui signifie qu'un travail est exécuté entièrement sur un processeur.
  - *portioned migration :* dans ce cas un travail est autorisé à migrer pendant son exécution. Pour cela, chaque tâche est découpée en portions fixes, chaque portion est affectée séquentiellement à un processeur.
- *le clustering :* dans cette catégorie, ce ne sont pas les tâches qui sont limitées à migrer mais plutôt les processeurs sur lesquels les tâches peuvent migrer. Ainsi, des regroupements de processeurs appelés *clusters* sont formés. Les tâches sont partitionnées entre ces clusters et au sein de chaque cluster, l'ordonnancement global est appliqué.

2.4 Politiques d'ordonnancement temps réel multiprocesseur

Les premiers travaux de recherche en matière de politiques d'ordonnancement multiprocesseur se sont portés sur l'ordonnancement par partitionnement en raison de sa simplicité et du fait qu'il reprend les résultats existants sur l'ordonnancement monoprocesseur. L'ordonnancement global a été mis à l'écart pendant longtemps notamment à cause des difficultés pour prouver l'ordonnançabilité d'un système même s'il satisfait les conditions d'ordonnançabilité en monoprocesseur [DL78]. Toutefois, un regain d'intérêt a été noté pour cette catégorie d'ordonnancement vers les années 90s avec l'apparition des approches globales dites *équitables* pour lesquelles l'optimalité a été prouvée. Les politiques d'ordonnancement global souffrant de certains défauts, d'autres études se portant vers l'ordonnancement hybride ont donné naissance à des politiques permettant de mieux contrôler la migration des tâches. À travers cette section, nous exposons brièvement les principales politiques d'ordonnancement temps réel multiprocesseur. Notre objectif est d'introduire les notions de base des approches existantes pour l'ordonnancement multiprocesseur et non pas de fournir un état de l'art complet. Ainsi, cette présentation étant brève, un lecteur intéressé par la découverte des autres politiques de la même catégorie

est invité à à se reporter à l'article [CHD15].

2.4.1 Généralisation des algorithmes monoprocesseurs

Cette approche consiste à généraliser une politique d'ordonnancement monoprocesseur au cas multiprocesseur en exécutant au plus les m travaux les plus prioritaires selon cette politique sur les m processeurs. Les politiques qui sont généralisées au cas multiprocesseur sont préfixées par « G- »pour la mention Global. Deux techniques de généralisation d'algorithmes monoprocesseurs sont ainsi envisageables : une généralisation verticale et une généralisation horizontale [NBN+12].

1. *Généralisation Verticale :* la généralisation d'un algorithme selon cette approche se fait en ordonnançant les travaux selon l'algorithme en question en appliquant la règle suivante : les m travaux (au plus) les plus prioritaires sont alloués aux m processeurs. Informellement, cette allocation est qualifiée de verticale pour la raison suivante : à chaque instant t, dans la représentation graphique d'une séquence d'ordonnancement, les m travaux les plus prioritaires sont assignés verticalement, « du haut vers le bas », aux processeurs (cf. Fig. 2.4). Dans ce qui suit, nous présentons quelques algorithmes d'ordonnancement global de ce type.

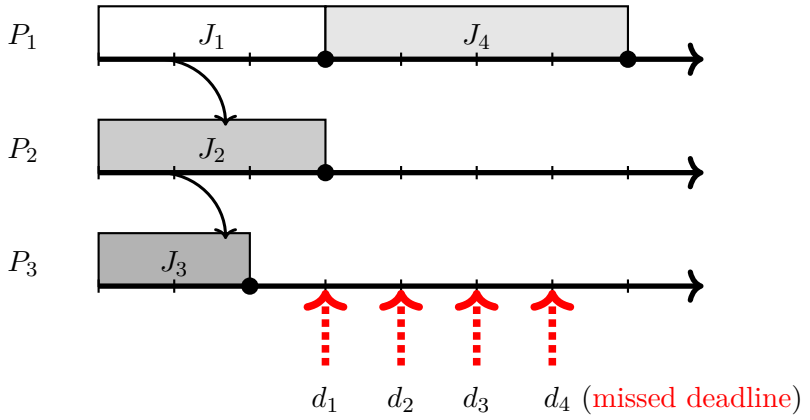


FIGURE 2.4: Séquence d'ordonnancement selon une généralisation verticale d'EDF : considérons l'exemple d'un système de quatre travaux de tâches à échéances implicites $J_i = (C_i, d_i)$ et trois processeurs P_j tel que $J_1 = (3, 3)$, $J_2 = (3, 4)$, $J_3 = (2, 5)$, $J_4 = (4, 6)$. En appliquant EDF nous obtenons $J_1 >> J_2 >> J_3 >> J_4$ avec « >> » correspond à « plus prioritaire que ». La généralisation verticale de cet algorithme en multiprocesseur consiste à assigner verticalement les travaux les plus prioritaires sur les processeurs de la plateforme. Ainsi, J_1, J_2 et J_3 sont attribués respectivement aux processeurs P_1, P_2 et P_3. Ensuite J_4 est attribué à P_1 où l'on peut constater que l'échéance de son travail est dépassée.

politiques G-RM et G-EDF : c'est une généralisation des algorithmes d'ordonnancement monoprocesseur RM (*Rate Monotonic*) et EDF (*Earliest Deadline First*) respectivement. RM est un algorithme à priorité fixe pour les tâches. La priorité de

ces tâches dépend de leur période : plus elle est petite et plus la tâche est prioritaire. Quand à EDF, c'est un algorithme à priorité fixe pour les travaux. Son principe consiste à attribuer la priorité la plus haute au travail dont l'échéance absolue est la plus proche.
La construction d'un ordonnancement global selon la généralisation verticale de ces algorithmes ne permet pas d'atteindre l'optimalité étant donné que les travaux les plus prioritaires sont sélectionnés sans anticiper l'impact de leur attribution sur les futures exécutions des autres travaux. Dhall et Liu ont prouvé qu'un ensemble de tâche dont le taux d'utilisation total est proche de 1 ne peut être correctement ordonnancé en appliquant globalement les politiques RM et EDF [DL78]. Une illustration de ce problème est fournie dans la figure 2.4 pour une séquence d'ordonnancement construite selon une généralisation verticale de G-EDF. Ainsi, des études visant à déterminer des conditions suffisantes d'ordonnançabilité pour G-RM [ABJ01, BG03] et G-EDF [SB02, Bar04] ont été proposées. Ces conditions dépendent principalement des taux d'utilisation total et maximal des systèmes de tâches. Toutefois, elles ne permettent pas une utilisation totale des processeurs. Sur la base de ces études, de nouvelles politiques ont vu le jour. Ci-dessous, nous présentons quelques unes.

politiques RM-US[ξ] et EDF-US[ξ] : elles sont respectivement deux extensions pour les politiques G-RM et G-EDF. Partant des études d'ordonnançabilité de ces dernières, RM-US[ξ] [ABJ01] et EDF-US[ξ] [SB02] proposent de considérer un seuil au taux d'utilisation pour déterminer la priorité des tâches. Ainsi, le choix d'attribution d'une priorité à une tâche τ_i est décidé en fonction de son taux d'utilisation U_i et d'un seuil déterminé ξ selon les deux règles qui suivent :

- si $U_i > \xi$ alors la tâche τ_i est considérée comme tâche *lourde* et se voit attribuer la priorité maximale. Les conflits entre les tâches lourdes sont résolus de manière arbitraire.
- si $U_i \leq \xi$ alors la tâche τ_i est considérée comme tâche *légère* et dans ce cas sa priorité est évaluée selon le principe de la politique RM pour RM-US[ξ] ou selon EDF pour EDF-US[ξ].

2. *Généralisation horizontale :* ce type de généralisation permet une assignation séquentielle des travaux sur les processeurs de la plateforme. Cette assignation est effectuée de manière horizontale en commençant par le premier processeur tout en veillant à ce que les tâches respectent leurs échéances. Cette technique consiste à maximiser la charge sur chaque processeur avant de pouvoir commencer l'allocation sur le suivant, d'où l'appellation horizontale (cf. Fig 2.5). Une des politiques inspirées par cette généralisation est la politique U-EDF [NBN+12] dont le principe est présenté dans la suite.

Politique U-EDF *(Unfair Earliest Deadline First)* : c'est une politique d'ordonnancement multiprocesseur démontrée optimale pour des tâches sporadiques à échéances implicites. La séquence d'ordonnancement selon la politique est construite

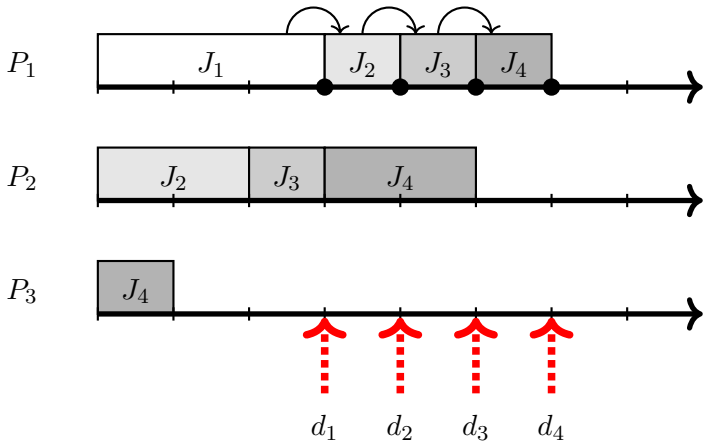


FIGURE 2.5: Exemple d'une séquence d'ordonnancement selon une généralisation horizontale d'EDF : en reprenant le même exemple de la figure 2.4, les travaux J_1 à J_4 sont attribués de manière séquentielle sur le premier processeur jusqu'à ce que ce dernier ne puisse plus assurer leur ordonnancement sans qu'une échéance ne soit dépassée. Ensuite, la même opération est répétée sur le deuxième processeur, ainsi de suite. Dans ce cas de figure, l'exécution du travail J_4 est attribuée aux trois processeurs de manière à ne pas avoir le parallélisme d'exécution d'un même travail.

en ligne en deux phases :

- phase de pré-allocation : cette phase consiste à attribuer les travaux aux processeurs en adoptant la généralisation horizontale. Le temps consacré pour l'exécution d'une tâche sur un processeur (que les auteurs appellent *budget*) comprend une dotation temporelle pour son travail courant et une réservation dépendante de son taux d'utilisation pour ses travaux futurs. La réservation permet de garantir l'exécution des travaux futurs avant leur échéance.
- phase d'ordonnancement : cette phase consiste à ordonnancer les travaux selon EDF-D en fonction des budgets calculés dans la phase précédente. EDF-D est une variante d'EDF qui permet d'attribuer des tâches dites *éligibles* sur les processeurs de la plateforme. Une tâche est considérée éligible sur un processeur, si et seulement si elle a une dotation d'exécution sur ce processeur, et si elle ne s'exécute pas parallèlement sur un autre processeur. Notons également que ces tâches éligibles sont exécutées sur les processeurs dans un ordre croissant des échéances absolues.

Le processus d'ordonnancement selon U-EDF est effectué à la réception de quatre événements : (i) l'activation d'un travail ; (ii) la terminaison d'un travail ; (iii) l'occurrence d'une date d'échéance ; (iv) la consommation d'une dotation temporelle. Le calcul du budget d'une tâche à un instant donné nécessite la prise en compte des budgets restants (non encore consommés) des autres tâches et des autres événements se produisant sur les autres processeurs. Ainsi, ces calculs restent compliqués et exigent en outre la considération des travaux en cours et des travaux futurs. Ceci représente l'inconvénient majeur de cette politique. Dans la mesures où ces calculs ne sont pas

faciles à résumer, un lecteur intéressé par les détails est invité à découvrir l'article des auteurs [NBN+12].

2.4.2 Politiques globales équitables

Cette famille de politiques introduit la notion d'*équité* étant donné que l'exécution des travaux se fait de manière équitable et proportionnelle aux taux d'exécution des tâches [BCPV96]. Des résultats intéressants en terme d'optimalité ont été publiés pour ces politiques.

Approche PFair *(Proportionate Fair)* : l'exécution des tâches selon un algorithme PFair suit un taux régulier et de manière rapprochée à l'équité. La notion d'équité revient à exécuter les tâches pendant une durée qui est proportionnelle à leur taux d'exécution. Autrement dit, sur un intervalle de temps $[0, t[$, une tâche τ_i reçoit exactement $U_i \times t$ unités de temps processeur pour son exécution. Cette exécution est qualifiée également de *idéale* ou *fluide*.

Afin de se rapprocher d'une exécution fluide, un algorithme de l'approche PFair discrétise l'espace temporel en intervalles uniformes appelés *slots* : $[t, t+1[$ tel que $t \in \mathbb{N}$ et définit une fonction $S : \tau \times \mathbb{N} \to \{0, 1\}$ qui renvoie 1 quand une tâche τ_i est exécutée sur un processeur pendant le slot t et 0 sinon. Une fonction de décalage entre une exécution fluide de la tâche et son exécution réelle résultante de l'ordonnancement est introduite : $lag(\tau_i, t) = U_i \times t - \sum_{i=0}^{t-1} S(\tau_i, t)$. Ainsi, afin d'assurer une exécution rapprochée d'une exécution fluide pour les tâches, la valeur absolue de cet écart est maintenue au dessous d'un quantum de temps ($|lag(\tau_i, t)| < 1$). La figure 2.6 illustre un exemple d'exécution PFair pour une tâche périodique à échéance implicite avec $\tau_i(C_i = 3, T_i = 6)$.

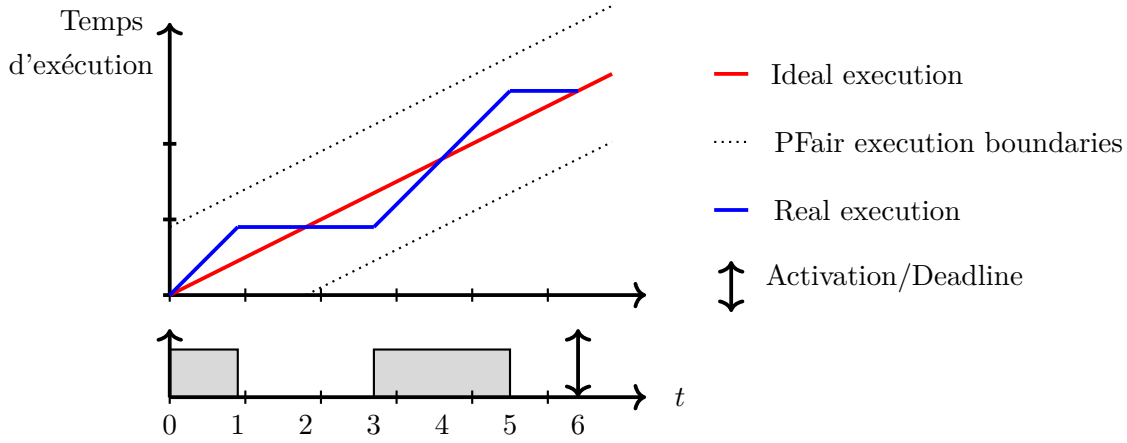


FIGURE 2.6: Exécution Pfair d'une tâche.

Parmi les algorithmes d'ordonnancement de l'approche PFair, nous citons PF (*Proportionate Fair*) [BCPV96], EPDF (*Earliest Pseudo-Deadline First*) [AS00b], PD (*Pseudo-*

Deadline) [BCPV96], PD² (*Pseudo-Deadline²*) [AS99a]. Les algorithmes de cette approche ont été démontrés optimaux pour des tâches périodiques, synchrones et à échéances implicites [BCPV96]. Toutefois, cette optimalité est à nuancer en pratique, en raison des surcoûts d'exécution engendrés par les ré-ordonnancements effectués sur chaque slot et par le nombre élevé des préemptions pouvant être introduit mais qui est négligé en théorie.

Approche DP-Fair *(Deadline Partitioning Fair)* : c'est une famille d'algorithmes qui a été proposée dans le but de remédier au nombre important de découpages engendrés par les approches PFair dans un intervalle temporel pour se rapprocher d'une exécution fluide. Dans leur article « *How much fairness is necessary ?* » Zhu *et al.* [ZMM03] expliquent que pour parvenir à l'optimalité, il n'est pas forcément nécessaire de se rapprocher autant d'une exécution idéale. Il suffit qu'à chaque échéance, aucune tâche ne soit en retard vis-à-vis de son exécution idéale. Ainsi, le temps est partitionné en intervalles délimités par les dates d'échéance des tâches du système. La fonctionnement d'ordonnancement selon cette approche se déroule en deux phases principales. Une première phase consiste à calculer pour chaque tâche des dotations temporelles pour son exécution sur chaque intervalle. Le calcul de cette dotation dépend également de son taux d'utilisation. La deuxième phase, appelée phase de distribution temporelle, consiste à répartir pour chaque tâche et sur chaque intervalle, les dotations temporelles qui lui sont allouées sur les m processeurs. Plusieurs politiques de la famille DPFair existent. Elles se distinguent principalement dans la manière selon laquelle les dotations temporelles sont calculées et distribuées sur les processeurs et les intervalles. Les plus connues sont la politique BFair (*Boundary* Fair) [NSG+14] en temps discret et DP-WRAP (*Deadline Partitioning-Wrap*) [LFS+10], LLREF [CRJ06] et LRE-TL [FN09] en temps continu.

2.4.3 Politiques hybrides

Algorithme RUN *Reduction to UNiprocessor* : c'est un algorithme qui a été démontré optimal pour l'ordonnancement des tâches périodiques à échéances implicites en multiprocesseur [RLM+11]. Il permet d'obtenir l'optimalité tout en réduisant les nombres de préemptions et migrations engendrés par les approches précédentes visant l'équité. Ceci en introduisant une version d'équité proportionnelle dite *faible*. Le déroulement de l'ordonnancement se fait en deux phases en suivant une approche similaire au *clustering* :

- une première phase effectuée hors-ligne qui consiste à construire un arbre contenant les tâches du système dans les feuilles et des *serveurs* dans des nœuds. Les serveurs correspondent à des tâches virtuelles qui regroupent plusieurs tâches dont la somme de leurs taux d'utilisation ne dépasse pas un. Ces serveurs sont attribués à des groupes de processeurs virtuels de telle manière à ce qu'une tâche ne puisse pas migrer entre deux groupes différents de processeurs.
- la deuxième phase de l'algorithme est effectuée en-ligne. C'est une phase d'ordonnancement dans laquelle des budgets sont affectés au serveurs. Ils sont proportionnels

à leurs taux d'utilisation. Ensuite, ces serveurs sont exécutés en parcourant l'arbre depuis la racine vers les feuilles et en suivant des règles basées sur EDF.

EDF-fm (*EDF-Fixed/Migrating*) et EKG (*EDF with task splitting and K processors in a Group*) : d'autres politiques d'ordonnancement existent et qui adoptent une approche hybride dans le calcul de la décision d'ordonnancement. Dans la catégorie du semi-partitionné, nous citons EDF-fm [ABD05] et EKG [AT06] qui sont tous les deux basés sur la politique EDF. Leur principe consiste à attribuer statiquement les tâches aux processeurs en utilisant une affectation de type Next-Fit. Quand une tâche ne peut pas s'exécuter entièrement sur un processeur, elle est découpée en deux parties. Ainsi, un système pourrait avoir au maximum $m - 1$ tâches migrantes. Ce nombre est réduit encore plus dans la politique EKG en proposant un regroupements de K processeurs en dehors duquel il est interdit pour une tâche de migrer.

2.5 Conclusion

Dans ce chapitre, nous avons présenté un aperçu sur l'ordonnancement temps réel avec un focus sur l'ordonnancement multiprocesseur. Ainsi, le chapitre a commencé par présenter le modèle d'un système temps réel du point de vue de la théorie d'ordonnancement. Ensuite nous avons discuté les principales approches de politiques d'ordonnancement multiprocesseur publiées dans la littérature.

Avec les limites d'optimalité relevées pour les politiques généralisant les algorithmes monoprocesseurs au cas multiprocesseur, plusieurs études théoriques se sont attachées soit aux tests d'ordonnançabilité pour celles-ci, soit à la proposition d'autres approches de politiques visant l'optimalité. Ceci a donné naissance à un nombre accru de politiques d'ordonnancement multiprocesseur dans les trois catégories : partitionné, global et hybride. Certaines politiques d'ordonnancement global, notamment celles basées sur la notion d'équité, permettent d'atteindre l'optimalité, mais au prix d'un nombre important de préemptions et migrations. Ainsi, des politiques d'ordonnancement hybrides ont été proposées pour remédier à ce problème en essayant de limiter les migrations seulement pour certaines tâches ou sur un regroupement de processeurs. Toutefois, les études liées à ces politiques, que ce soit pour l'ordonnancement global ou hybride, demeurent théoriques et négligent généralement les surcoût à l'exécution dûs aux ré-ordonnancements et aux migrations. En outre, en dehors des études expérimentales de l'évaluation des performances ou des travaux d'analyse d'ordonnançabilité pour ces politiques, peu d'efforts visant leur mise en œuvre sur une plateforme réelle ont été notés.

Nous fournissons dans le tableau 2.1 une liste non exhaustive des principaux algorithmes multiprocesseurs connus à ce jour, en corrélation avec l'état de l'art de [CHD15].

Scheduling policy	Reference
Partitioned schedulers	
P-RM (Partitioned Rate Monotonic)	
P-FP (Partitioned Fixed Priority)	
P-DM (Partitioned Deadline Monotonic)	
P-EDF (Partitioned Earliest Deadline First)	
P-FENP (Partitioned First Execution Non Preemptive)	[CSMC18]
Global schedulers generalizing single processor policies	
RM-US (Rate Monotonic with Utilization Separation)	Andersson *et al.*[ABJ01]
EDF-US[ξ] (Earliest Deadline First with Utilization Separation)	Srinivasan, Baruah[SB02]
SM-US (Slack Monotonic with Utilization Separation)	Andersson[And08]
DM-DS (Deadline Monotonic with Density Separation)	Bertogna *et al.*[BCL05]
PriD/EDF$^{(k)}$ (Priority-Driven / Earliest Deadline First $^{(k)}$)	Goossens *et al.*[GFB03]
fpEDF (fixed-priority Earliest Deadline First)	Baruah[Bar04]
Tp-TkC (Fixed Priority with adaptiveTkC)	Andersson, Jonsson[AJ00]
GFL (Global-Fair Lateness)	Erickson, Anderson[EAW14]
EDZL (Earliest Deadline Zero Laxity)	S. Lee[Lee94]
EDCL (Earliest Deadline Critical Laxity)	Kato et Yamasaki[KY08b]
FPZL, FPCL, FPSL (Fixed Priority Zero Laxity)	Davis et Kato[DK12]
GLLF (Global Least Laxity First)	Mok[Mok83]
GMLLF (Global Modified Least Laxity First)	S-H Oh et Yang[OY98]
U-EDF (Unfair-EDF)	*Nelissen et al.*[NBN$^+$12]
PFair global schedulers	
EPDF (Earliest Pseudo-Deadline First)	Andersson, Srinivasan[AS00b]
PF (Proportionate Fair)	Baruah *et al.*[BCPV96]
PD (Pseudo-Deadline)	Baruah *et al.*[BGP95]
PD^2 (Pseudo-Deadline^2)	Andersson, Srinivasan[AS99a]
ER-PD^2 (Early-Release Fair Pseudo Deadline^2)	Andersson, Srinivasan[AS00a]
PL (Pseudo-Laxity)	Kim, Cho[KC11]
DPFair global schedulers	
LLREF (Largest Local Remaining execution time First)	Cho *et al.*[CRJ06]
LRE-TL (Local Remaining Execution-Time and Local plane)	Funk, Nanadur[FN09]
DP-WRAP (Deadline Partitioning-Wrap)	Levin *et al.*[LFS$^+$10]
BF (Boundary Fair)	Zhu *et al.*[ZMM03]
BF^2 (Boundary Fair^2)	Nelissen *et al.*[NSG$^+$14]
SA (Scheduling Algorithm)	Khema, Shyamasundar[KS97]
Hybrid schedulers	
EDF-fm (Earliest Deadline First-fixed or migrating)	Anderson *et al.*[ABD05]
EKG (EDF with task splitting and k processors in a Group)	Anderson *et al.* [AT06]
Ehd2-SIP ou EDDHP	Kato *et al.* [KY07]
EDF-RRJM (EDF-Round Robin Job Migration)	George *et al.* [GCS11]
EDHS (Earliest Deadline and Highest-priority Split)	Kato et Yamasaki [KY08a]
DM with Priority Migration	Kato *et al.* [KY09]
Carousel-EDF	Sousa *et al.* [SSTB13]
RUN	Regnier *et al.* [RLM$^+$11]
Quasi-Partitioning Scheduler	Massa *et al.* [MLR$^+$14]

TABLE 2.1: Exemples de politiques d'ordonnancement multiprocesseur.

CHAPITRE 3

Systèmes d'Exploitation Temps Réel

3.1 Qu'est ce qu'un Système d'exploitation temps réel (SETR) ?

Un système d'exploitation temps réel (*RTOS : Real-Time Operating System*) est un support d'exécution offrant une couche d'abstraction dite « *bas niveau* » entre les programmes applicatifs d'un système temps réel et la couche matérielle d'exécution. Ainsi, il doit assurer une exécution prévisible qui doit se conformer aux contraintes temporelles du système temps réel auquel il appartient. Il dispose pour cela d'un « ordonnanceur temps réel » lui permettant d'organiser l'exécution de ces programmes selon les règles fixées par la politique d'ordonnancement qu'il implémente. Une présentation détaillée du principe de l'ordonnancement temps réel a été fournie dans le chapitre 2.

Dans la littérature, le terme « système d'exploitation temps réel » est un terme générique désignant une plateforme logicielle supportant l'exécution temps réel. Dans le domaine des systèmes embarqués, qui est le nôtre, le terme *exécutif temps réel* est également employé pour désigner une plateforme d'exécution utilisée pour ce type de systèmes. Généralement, un exécutif temps réel se compose le plus souvent d'un noyau temps réel qui constitue le cœur de l'exécutif. Il doit répondre à un certain nombre d'exigences afin de pouvoir supporter la particularité du temps réel. Ainsi, il doit pouvoir fournir des services d'ordonnancement temps réel, de communication et synchronisation entre les tâches, des routines pour la gestion des ressources et des primitives de communication. Ce sont des « fonctions de base »qui sont déclenchées le plus souvent par des appels système ou par l'écoulement de temps.

3.1.1 Modes d'exécution dans un SETR

Dans un système d'exploitation temps réel, des modes d'exécution sont définis avec des permissions d'accès aux ressources de l'architecture matérielle (mémoire, périphériques, instructions, etc.). Ces permissions permettent de protéger ces ressources et éviter la corruption du système (intentionnellement ou non) en fournissant aux services de l'application et aux fonctions du noyau, une interface uniforme pour l'accès. Plusieurs modes d'exécution peuvent être définis, les plus souvent utilisés sont :

- **mode utilisateur** : c'est un mode d'exécution des programmes d'application dans lequel des restrictions d'accès aux ressources sont appliquées et certaines instructions ne peuvent pas y être exécutées.
- **mode noyau (super-utilisateur)** : dans ce mode, les fonctions ont un accès total aux ressources de la machine et toutes les instructions peuvent être exécutées. Le basculement vers ce mode se fait soit suite à la réception d'une interruption matérielle (e.g. une interruption reçue d'un compteur activant une tâche périodiquement), ou bien suite à la réception d'une interruption logicielle générée par un processeur exécutant une instruction.

3.1.2 Implémentation d'un ordonnanceur au sein d'un SETR

La particularité des systèmes temps réel réside dans leur utilisation d'un composant d'ordonnancement temps réel. La mise en œuvre d'un tel composant au sein de l'exécutif peut être réalisée selon trois approches principales [LRSF04] :

- *Implémentation basée sur le noyau (kernel-based) :* dans cette approche, le code de l'ordonnanceur est complètement immergé et dépendant du code du noyau, ce qui permet une utilisation complète et efficace des services du système d'exploitation par l'ordonnanceur. Cependant, le développement de l'ordonnanceur selon cette approche n'est pas une tâche facile pour les développeurs. Ainsi, à chaque fois qu'il est question d'implémenter un nouvel ordonnanceur, il faut identifier les parties du noyau qui doivent être modifiées et faire les modifications à plusieurs endroits. Ceci exige une expertise considérable en programmation bas niveau. Parmi les exécutifs temps réel qui adoptent une telle approche, on trouve RT-Linux [rtl], RED-Linux [WL99], Trampoline [BBFT06] ;
- *Implémentation basée sur un intergiciel (middlware-based) :* cette approche consiste à séparer le code de l'ordonnanceur de celui du noyau à travers un *intergiciel*. De manière schématique, un intergiciel est une couche intermédiaire permettant de faire communiquer plusieurs composants d'un système. Dans une telle implémentation, au lieu d'avoir le code de l'ordonnanceur réparti dans le code du noyau et s'interfaçant directement avec les fonctions de celui-ci, l'ordonnanceur est considéré comme une boite noire ayant son code séparé. L'ordonnancement des tâches étant déterminé sur la base des données échangées entre l'ordonnanceur et le noyau, un mécanisme de communication entre les deux est assuré grâce à des fonctions et des bibliothèques dédiées fournies par l'intergiciel implémentant toutes les fonctionnalités requises pour

cette communication. Ainsi, l'échange des données entre le noyau et l'ordonnanceur se fait en invoquant des fonctions de l'intergiciel tout en faisant une abstraction du code des deux communicants. Cette approche d'implémentation permet de faciliter la modification de l'ordonnanceur et la portabilité de son code sans pour autant altérer le code du noyau. CORBA [WDG+97] et ExSched [ÅNKR12] sont deux exemples d'exécutif temps réel suivant cette approche ;

- *Implémentation haut niveau (high-level based) :* il s'agit d'une approche de niveau supérieur, qui permet de décrire le comportement de l'ordonnanceur dans une structure de langage dédié haut niveau (*DSL : Domain-Specific Language*). Le principe consiste à élaborer un langage dédié à l'ordonnanceur fournissant au programmeur des domaines de variables, des déclarations et des abstractions spécifiques au fonctionnement de l'ordonnanceur au sein de l'OS en question. À partir de ce langage, une spécification exhaustive des paramètres et des objets que l'ordonnanceur utilise (les tâches, les événements, etc.) est élaborée. Cette spécification permet ainsi de générer le code exécutable de l'ordonnanceur qui peut ensuite être intégré dans le système d'exploitation temps réel. Les travaux menés autour de BOSSA [MLD05] et de son langage dédié s'inscrivent dans cette démarche.

L'implémentation d'un ordonnanceur au sein d'un exécutif temps réel doit tenir en compte des conditions d'appel aux service d'ordonnancement. En principe, l'ordonnanceur est censé se déclencher à la suite de la réception d'un événement nécessitant un ré-ordonnancement au niveau du RTOS. Ceci donnent lieu à deux modes de déclenchement de l'ordonnanceur possibles :

- **approche conduite par le temps (*tick-driven*) :** également appelée approche synchrone. Dans cette approche, le procédé d'observation des événements de ré-ordonnancement est mis en place à des instants discrets de manière périodique. Ainsi, s'il est question d'un ré-ordonnancement, l'ordonnanceur est appelé seulement à ces dates. Les occurrences des événements de ré-ordonnancement sont perçues comme étant simultanées et se produisant à l'instant d'appel de l'ordonnanceur (cf. Fig. 3.1).

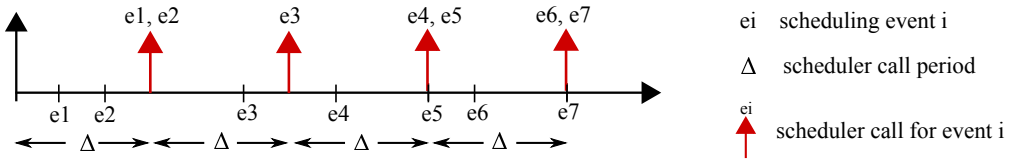


FIGURE 3.1: Les instants d'appel de l'ordonnanceur selon l'approche conduite par le temps.

- **approche conduite par les événements (*event-driven*) :** également appelée approche asynchrone. Elle consiste à observer en permanence les occurrences des événements pouvant conduire à un ré-ordonnancement. La réaction à ces événements conduit *instantanément* à l'appel de l'ordonnanceur (cf. Fig. 3.2).

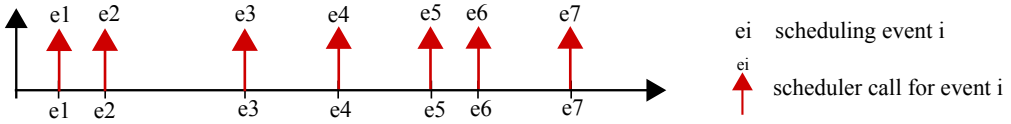


FIGURE 3.2: Les instants d'appel de l'ordonnanceur selon l'approche conduite par les événements.

3.1.3 Classification des SETR

Une grande part des systèmes d'exploitation temps réel est développée et utilisée dans le domaine des systèmes embarqués, allant des exécutifs à usage spécifique pour les micro-contrôleurs jusqu'aux exécutifs généraux à extensions temps réel. L'offre est large que ce soit dans le domaine industriel ou bien dans la recherche académique. Nous discutons dans cette section la classification de ces systèmes qui peut être faite selon trois catégories :

1. **Exécutifs temps réel dédiés aux systèmes embarqués :** appelés *Purpose-Built Real-Time Operating System* en anglais, cette catégorie est corrélée à la définition classique d'un système d'exploitation temps réel. Ce type d'exécutifs est utilisé pour les systèmes embarqués dans lesquels une exécution rapide et hautement prévisible doit être garantie. Ils sont architecturés et implémentés spécifiquement pour l'hébergement des applications temps réel.
2. **Exécutifs temps réel Unix :** les applications temps réel mises en œuvre dans les systèmes Unix sont supportées par deux types de systèmes :
   - **les systèmes POSIX [Gal95] :** la norme POSIX introduit une standardisation respectant les spécifications temps réel de base (e.g. l'ordonnancement temps réel) tout en supportant les fonctionnalités de la programmation standard des systèmes Unix (les processus, les threads, les systèmes de fichiers, etc.).
   - **les systèmes Linux temps réel [Bar97] :** pour ce qui est de l'approche Linux temps réel, son apparition est plus ou moins liée à la notoriété de Linux en tant que source ouverte. Parmi les systèmes se basant sur cette approche, le projet RTAI (*Real-Time Application Interface*) [MBD+00] du département d'Ingénierie Aérospatiale de l'Ecole Polytechnique de Milan qui s'inspire de RT-Linux (*Real-Time Linux*) [rtl]. Les deux systèmes se basent sur un noyau Linux.
3. **Exécutifs génériques avec des extensions temps réel :** au lieu de développer un exécutif temps réel en partant de zéro, plusieurs projets proposent plutôt de se baser sur un système d'exploitation générique (*GPOS : General-Purpose Operating System*) et d'y intégrer des fonctionnalités permettant d'avoir une version temps réel. À présent, la base des systèmes génériques majoritairement utilisée est Linux.

3.2 Travaux d'implémentation d'ordonnanceur temps réel

En corrélation avec la classification des systèmes d'exploitation temps réel présentée en 3.1.3, nous proposons dans cette section un panorama des principaux travaux de re-

cherche en matière d'implémentation des politiques d'ordonnancement temps réel. Cette présentation a pour objectif de donner un contexte de notre travail d'implémentation en Trampoline. Compte tenu de l'étendue du marché des SETRs, nous nous limitons à des exemples représentatifs des systèmes d'exploitation avec une offre multiprocesseur/multi-cœur.

Cas des exécutifs embarqués : la majorité des exécutifs de cette catégorie se base sur des ordonnanceurs à priorité fixe pour les tâches afin de répondre aux contraintes de prévisibilité. De la même manière, dans les architecture multiprocesseur, le type d'ordonnancement prédominant est l'ordonnancement partitionné à priorité fixe pour les tâches. C'est le cas de certains systèmes commerciaux tels que ThreadX [Lam09], Nucleus RTOS [Gra09], ou bien OSE [Ene]. La même observation peut être faite pour les systèmes dédiés à la recherche académique. Nous citons dans cette catégories Spring OS [SR89] ou Fiasco [PSD11]. Toutefois, des projets de recherche commencent à s'engager dans l'implémentation d'autres ordonnanceurs. À titre d'exemple, il convient de noter EPOS [FW08] qui est un framework multi-plateforme pour les systèmes embarqués, qui a été étendu pour pouvoir supporter des ordonnanceurs à priorité fixe pour les tâches et les travaux en ordonnancement partitionné, global et en clustering. Similairement dans l'industrie, ERIKA Enterprise RTOS [Srl] qui est un système d'exploitation dédié aux petits micro-contrôleurs et conforme au standard AUTOSAR (cf § 5.2). Il intègre un ordonnanceur partitionné basé sur EDF. Nucleus RTOS [Gra09] est aussi une offre industrielle pour les systèmes embarqués qui implémente un ordonnanceur partitionné basé sur une priorité fixe pour les tâches.

Notons que dans cette catégorie, nous trouvons également Trampoline, le système d'exploitation sur lequel notre étude est menée. Son implémentation actuelle offre un ordonnanceur à priorité fixe pour les tâches avec préemption possible. En multicœur, l'ordonnancement est simplement partitionné.

Cas des systèmes POSIX : en ce qui concerne les systèmes adoptant la norme POSIX, QNX Neutrino [PN03] et VxWorks [Riv03] sont les SETRs les plus connus. Ils fournissent un support multiprocesseur se basant sur un ordonnancement partitionné à priorité fixe pour les tâches avec préemption et en favorisant les affinités sur les processeurs. Ces affinités indiquent sur quel(s) processeur(s) la tâche peut être exécutée et permettent ainsi de limiter les migrations, voire les préemptions. Il convient de souligner que les deux systèmes d'exploitation sont populaires dans le domaine des systèmes temps réel. En revanche, ils ne conviennent pas aux systèmes dont les ressources sont fortement limitées et qui nécessitent une certification.

Cas des systèmes d'exploitation génériques étendus au temps réel (focus sur les systèmes Linux) : dans cette catégorie, les développements les plus récents et notables en matière d'ordonnancement concernent la politique EDF en multiprocesseur implémentée dans *SCHED_DEADLINE* [FTC+09]. C'est une classe de Linux qui intègre la politique EDF en partitionné, global et clustering. En outre, elle utilise l'API

des groupes de contrôle (*cgroups*) pour prendre en charge nativement les plateformes multicœurs.

Nous trouvons également LITMUS$^{\mathrm{RT}}$ [CLB+06] qui est une extension native majeure de Linux utilisée dans plusieurs travaux de recherche avec une communauté active. Il est spécialement conçu pour l'évaluation des algorithmes d'ordonnancement temps réel multiprocesseur. Il permet de les implémenter sous forme de plugin en utilisant l'infrastructure de Linux. Plusieurs travaux ont été menés en utilisant LITMUS^{RT} afin d'étudier le comportement des politiques d'ordonnancement sur une architecture réelle. Ainsi, LITMUS^{RT} intègre les politiques Partitioned-EDF [Bar13], Global-EDF, Partitioned-FP [LL73] et Pseudo-Deadline^2 [AS99b]. D'autres politiques ont été expérimentées sur cette plateforme, mais qui n'ont pas été intégrées. C'est le cas par exemple des politiques semi-partitionnées EDF-fm [ABD05], NPS-F [BA11] et RUN [RLM+11].

D'autres exécutifs qui étendent Linux en temps réel existent et implémentent des politiques d'ordonnancement à priorité fixe pour les tâches. Nous citons LynxOS [SB90], Xenomai [Ger04], RT-Linux (*Real-Time Linux*) [rtl] et son successeur RTAI (*Real-Time Application Interface*) [MBD+00].

3.3 Conclusion

Dans ce chapitre, nous avons fourni un aperçu sur les travaux en matière d'implémentation des ordonnanceurs au sein des systèmes d'exploitation temps réel. La plupart des RTOSs, sauf quelques exceptions notables (*e.g.* LITMUS^{RT}), reposent sur des ordonnanceurs à priorité fixe pour les tâches et sur l'ordonnancement partitionné lorsque l'offre multiprocesseur est supportée. Ceci motive notre intérêt pour l'implémentation des ordonnanceurs globaux de type G-EDF et EDF-US[ξ]. Le tableau 3.1 résume, dans un récapitulatif non exhaustif compte tenu du nombre accru des SETRs publiés, les ordonnanceurs implémentés au sein des SETRs les plus connus dans le monde industriel et académique. Ce tableau est organisé selon la classification introduite dans ce chapitre.

RTOS	Implemented scheduler	References
	Purpose-built RTOS	
ThreadX	partitioned static priority and round robin schedulers	[Lam05]
Nucleus RTOS	partitioned static priority and round robin schedulers	[Gra09]
OSE	partitioned static priority and round robin schedulers	[Ene]
ERIKA Entreprise	partitioned and global schedulers : static priority and EDF	[Srl]
RTEMS	partitioned static priority, round robin, RM and EDF schedulers	[BS14]
SpringOS	partitioned and global static priority	[SR89]
Fiasco	partitioned static priority and round robin	[PSD11]
EOPS	partitioned and clustered schedulers : static priority, round robin and First Come First Serve	[FW08]
Trampoline	partitioned static priority	[BBFT06]
FreeRTOS	partitioned static priority scheduler	[B$^+$08]
	Posix like systems	
VxWorks	partitioned schedulers : static priority, round robin and RM	[Riv03]
QNX Neutrino	partitioned FIFO and round robin scheduler	[PN03]
S.Ha.R.K	scheduling algorithms testbed : partitioned RM, EDF, Round Robin, and others	[GAGB01, AB00]
	Extended GPOS	
LITMUS^{RT}	shceduling algorithms testbed : partitioned and global schedulers : FP and EDF ; semi-partitioned EDF-fm ; PD^2 ; NPS-F, RUN and others	[CLB$^+$06]
RT-Linux	partitioned FIFO, RM and EDF scheduler	[rtl]
Xenomai	partitioned static priority and round robin scheduler	[Ger04]
RED-Linux	partitioned RM and EDF schedulers	[WL99]

TABLE 3.1: Liste non exhaustive des systèmes d'exploitation temps réel avec une offre multiprocesseur/multi-cœur.

CHAPITRE 4

Méthodes de vérification formelle

4.1 Introduction

Dans notre démarche de vérification, nous avons opté pour l'emploi des méthodes formelles. Ainsi, à travers ce chapitre, nous examinons les méthodes de vérification existantes dans la littérature en se focalisant sur deux grandes familles de méthodes formelles : la vérification par preuves et la vérification des modèles. Nous fournissons également une vue d'ensemble des travaux utilisant ces méthodes pour la vérification des systèmes en général et des implémentations d'ordonnanceur en particulier.

4.2 Méthodes basées sur la preuve de théorèmes

Le principe des méthodes basées sur la preuve des théorèmes consiste à spécifier le système à vérifier sous forme de théorèmes mathématiques de base appelés *axiomes* et à en déduire des propriétés comportementales du système en utilisant un ensemble de règles d'inférence. Les axiomes sont des propositions évidentes par elles-mêmes et ne nécessitant pas de démonstrations. Le processus de déduction correspond à la construction d'une preuve de théorèmes et est réalisé dans une logique choisie, le plus souvent, en faisant appel à un logiciel démonstrateur. Ce dernier prend en entrée des axiomes fournis par l'utilisateur et tente ensuite de construire : soit une preuve de théorème en générant les étapes de preuves intermédiaires ; soit d'infirmer les axiomes énoncés. Une interaction avec le démonstrateur est souvent requise afin de guider la preuve en choisissant les règles de déduction à appliquer. Deux types de logique peuvent être employés par les démonstrateurs pour construire les preuves : la logique du premier ordre *(First Order Logic)* et la logique d'ordre supérieur *(High Order Logic)* issues toutes les deux de la logique des prédicats [Kow74] dont nous fournissons un aperçu du principe dans ce qui suit.

4.2.1 Qu'est ce qu'un prédicat en logique ?

De manière générale, un prédicat désigne une phrase en langage naturel exprimant une propriété d'une ou plusieurs variables et/ou constantes du système à formaliser. Par exemple, la phrase « *la tâche* τ_0 *est prête* » est un prédicat où l'expression « *est prête* » désigne une propriété de « *la tâche* τ_0 » qui est une constante. Un prédicat peut également désigner une relation lorsqu'il concerne plusieurs variables ou constantes. Par exemple, la phrase « *la tâche* τ_0 *est plus prioritaire que la tâche* τ_1 » exprime une relation entre deux constantes τ_0 et τ_1.

4.2.2 Expression d'une propriété dans la logique des prédicats

Les propriétés en logique de prédicats doivent être exprimées sous forme de formules logiques en utilisant ce qui est appelé dans la littérature des *objets* à savoir :

- les prédicats ;
- les termes et les atomes : les termes $t_1, ..., t_n$ correspondent à des variables ou constantes et à des fonctions sur ces variables. Quant aux atomes, ils désignent des prédicats exprimés sur des termes : si $t_1, ..., t_n$ sont des termes et P est un prédicat, alors $P(t_1, ..., t_n)$ est un atome ;
- les opérateurs logiques : ils sont utilisés pour établir des liaisons entre les prédicats. Les opérateurs les plus utilisés sont : (i) la négation $\neg$; (ii) la conjonction $\wedge$; (iii) la disjonction $\vee$; (iv) l'implication $\Longrightarrow$; (v) l'équivalence $\Longleftrightarrow$;
- les quantificateurs : ce sont des notations concises appliquées aux termes et atomes permettant d'indiquer leur portée dans la formule logique. Traditionnellement deux quantificateurs existent : (i) le quantificateur universel dénoté par $\forall$; (ii) le quantificateur existentiel dénoté $\exists$.

Exemple 4.1. La formalisation de l'expression « *Il y a une tâche plus prioritaire que les autres* » peut être faite avec la formule logique : $\exists x \forall y \, (T(x) \wedge T(y) \wedge Prio(x, y))$, en notant que :

- x et y sont des termes, $T(x)$ et $Prio(x, y)$ sont des atomes.
- $T(x)$: exprime que x est une tâche.
- $Prio(x, y)$: exprime que x est plus prioritaire que y.

4.2.3 Aspect axiomatique de la logique des prédicats

Cet aspect correspond à la manière avec laquelle un système est spécifié pour être vérifié dans la logique des prédicats. En effet, la spécification du dit système est faite sous forme d'un système formel dans lequel les déductions réalisées conduisent à des théorèmes. Ces déductions nécessitent le recours à des axiomes qui désignent une évidence en soi et sont utilisés dans la logique pour représenter une vérité première (qui ne nécessite pas de démonstration). La double négation représente un exemple d'évidence : $\neg\neg A \Leftrightarrow A$.

4.2.4 Démonstration d'une propriété

Informellement, la démarche consiste à considérer des propriétés valides (les axiomes) et à tenter de mettre en évidence des propriétés sous-jacentes (des conclusions) en utilisant des règles de déduction. Ce processus peut être réalisé en plusieurs étapes pour déduire à chaque fois de nouvelles conclusions en se basant sur celles déduites préalablement. Un exemple de règle de déduction qui est très régulièrement utilisée en déduction naturelle est **modus ponens**. De manière formelle, elle affirme qu'une propriété q peut être déduite des axiomes (ou des propriétés qui en dérivent) $p_1, ..., p_n$ si et seulement si q est une conséquence logique de $p_1 \wedge ... \wedge p_n$. Autrement dit, si et seulement si $(p_1 \wedge ... \wedge p_n) \implies q$ est également valide. Ce processus s'écrit sous la forme suivante :

$$\frac{(p_1 \wedge ... \wedge p_n) \implies q \qquad p_1, ..., p_n}{q}$$

Exemple 4.2. Considérons le raisonnement suivant :

1. Si x est une tâche prête et y est un processeur libre alors x s'exécute sur y ;
2. x est une tâche prête $(T(x))$;
3. y est un processeur libre $(P(y))$;
4. conclusion : si (1), (2) et (3) sont valides, alors « x s'exécute sur y $(EXEC(x, y))$ » en est une déduction logique et s'écrit sous la forme :

$$\frac{\mathbf{1.}\ (T(x) \wedge P(y)) \implies EXEC(x, y) \qquad \mathbf{2.}\ T(x),\ \ \mathbf{3.}\ P(y)}{\mathbf{4.}\ EXEC(x, y)}$$

4.2.5 Logique du premier ordre

Il s'agit de la logique des prédicats [Fre72] inventée par Gottlob Frege[1]. C'est une logique permettant de faire intervenir la notion de prédicats sur des variables appartenant à un seul domaine non vide (e.g. entiers naturels) afin d'exprimer des fonctions et des relations sur ce domaine en utilisant des règles telles que celle présentée ci-dessus. La spécificité de cette logique, qui constitue également sa limite, c'est qu'elle ne permet pas de porter les quantifications sur les fonctions et les prédicats. Seules les variables peuvent être quantifiées (e.g. $\forall x P(x)$).

[1] **Friedrich Ludwig Gottlob Frege** (08/11/1848 – 26/07/1925) est un mathématicien, logicien et philosophe allemand, créateur de la logique moderne et plus précisément du calcul des prédicats ou de la logique du premier ordre.

4.2.6 Logique d'ordre supérieur

Dans cette logique, les fonctions et les prédicats sont considérés au même titre que les variables. Pour cela, les quantifications peuvent porter également sur ces deux objets et cette écriture $\exists f \forall x P(f(x))$ est ainsi possible. Cette logique permet également d'utiliser des variables appartenant à plusieurs domaines (entiers, réels, listes, etc.). Ainsi, à l'opposé de la logique du premier ordre, cette logique est souvent plus employée dans la vérification des systèmes informatiques puisqu'ils manipulent des données et des variables de plusieurs types.

4.2.7 Approches par preuves de théorèmes dans la vérification des systèmes

Plusieurs approches existent dans le domaine de la vérification des systèmes informatiques. La différence entre eux réside dans le degré d'automatisation du processus de construction de preuve. Deux catégories principales peuvent être distinguées :

- **la preuve de théorèmes interactive (assistée) [HUW14] :** cette approche consiste à utiliser des prouveurs de type assistant de preuve qui permettent de construire des preuves en s'appuyant fortement sur le guidage de l'utilisateur. Ce dernier coopère avec le prouveur en spécifiant la structure globale de la preuve sous formes de théorèmes, tandis que l'outil les utilise pour remplir les étapes de preuves qui peuvent être faites automatiquement et vérifier leur validité. Il est important de noter que c'est la vérification des preuves qui est automatique et non pas leur spécification. Ainsi, cette approche nécessite une intervention humaine en boucle pour dériver des théorèmes non triviaux et les réinjecter dans l'outil. Parmi les outils existants les plus connus, nous notons Coq[DFH+91] et Isabelle/HOL [NPW02].
- **la preuve de théorèmes automatique [Fit12] :** elle consiste à mener les preuves sous des prouveurs automatiques qui permettent de vérifier la validité des théorèmes sans avoir besoin d'une intervention d'utilisateur. La preuve formelle est entièrement construite par l'outil en se basant sur une description du système à vérifier, un ensemble d'axiomes et de règles d'inférence.

4.2.8 Avantages et limites de l'approche

Les techniques de preuves de théorèmes sont utilisées pour la vérification des systèmes d'exploitation temps réel dans la mesure où elles peuvent agir avec des espaces d'états infinis et peuvent valider des propriétés pour des valeurs de variables arbitraires. Toutefois, leur principal inconvénient réside dans la lenteur du processus de construction d'une preuve, même en utilisant les prouveurs automatiques. Ce processus exige un niveau assez élevé d'expertise de la part de l'utilisateur et les théorèmes et preuves introduits ou déduits risquent ainsi d'être erronés et peuvent être lourds et difficiles à comprendre. Ces inconvénients constituent ainsi un obstacle à l'adoption et à l'utilisation répandue des preuves de théorème quand il s'agit de vérifier des systèmes complexes et faisant intervenir différents types de variables.

4.3 Méthodes de vérification basées sur des modèles

Le *model-checking* est une approche de vérification formelle automatisée permettant de vérifier que le comportement d'un système réel est conforme à sa spécification. Cette spécification exprime ce que le système doit effectuer et représente certaines exigences relatives à son comportement attendu. Pour cela, une abstraction formelle du système à vérifier est construite sous forme d'un modèle. Cette abstraction doit être la plus proche possible du système d'un point de vue comportemental. La vérification est ensuite conduite sur ce modèle afin d'examiner la conformité de son comportement avec les exigences de la spécification. Ainsi, une approche de vérification par model-checking est conduite sur deux phases principales illustrée dans la figure 4.1 :

- **phase de modélisation du système :** dans laquelle le comportement global du système à vérifier est abstrait sous forme d'un *modèle formel*. Il est élaboré le plus souvent avec des machines à états telles que les automates et les réseaux de Petri [Pet77]. La modélisation peut également être effectuée via des algèbres de processus [Hoa78, M$^+$80]. Durant cette phase, les exigences sont également exprimées de manière formelle, soit sous la forme de machines à états, soit sous la forme de propriétés en utilisant des formules de logique temporelle (cf. 4.3.1).
- **phase de vérification des exigences de la spécification :** dans cette phase, une exploration exhaustive de l'espace des états du système est effectuée à l'aide d'un outil de vérification (*model-checker*). Ce dernier prend en entrée le modèle du système et une propriété à vérifier. Il explore ensuite l'ensemble des exécutions possibles du système à partir de son état initial. L'exploration de l'espace d'états est effectuée en entier à moins qu'une violation de la propriété ne soit détectée ou que l'espace d'états dépasse la mémoire disponible sur la machine. Dans le cas où la propriété violée est une propriété de sûreté (cf. § 4.3.2), un contre-exemple est généré automatiquement sous forme d'une trace d'exécution à partir de l'état initial jusqu'à l'état violant la propriété.

4.3.1 La logique temporelle

La logique temporelle [GHRF94] est une logique formelle dite *modale*. Elle étend les logiques classiques de premier ordre et d'ordre supérieur avec des modalités permettant d'exprimer des propositions en fournissant des informations sur le temps. Ainsi, la validité de l'énoncé des propositions dépend également des états du système pendant son évolution dans le temps. Cela signifie qu'une proposition qui est vraie à un certain moment peut devenir fausse dans le futur, et vice-versa. Pour cela, cette logique intègre des opérateurs temporels et des opérateurs permettant d'explorer les chemins d'exécution en plus des opérateurs logiques. Deux logiques temporelles principales sont plus fréquemment utilisées : la logique temporelle linéaire *Linear Temporal Logic* [Pnu77] et la logique temporelle arborescente *(Computation Tree Logic)* [CE81].

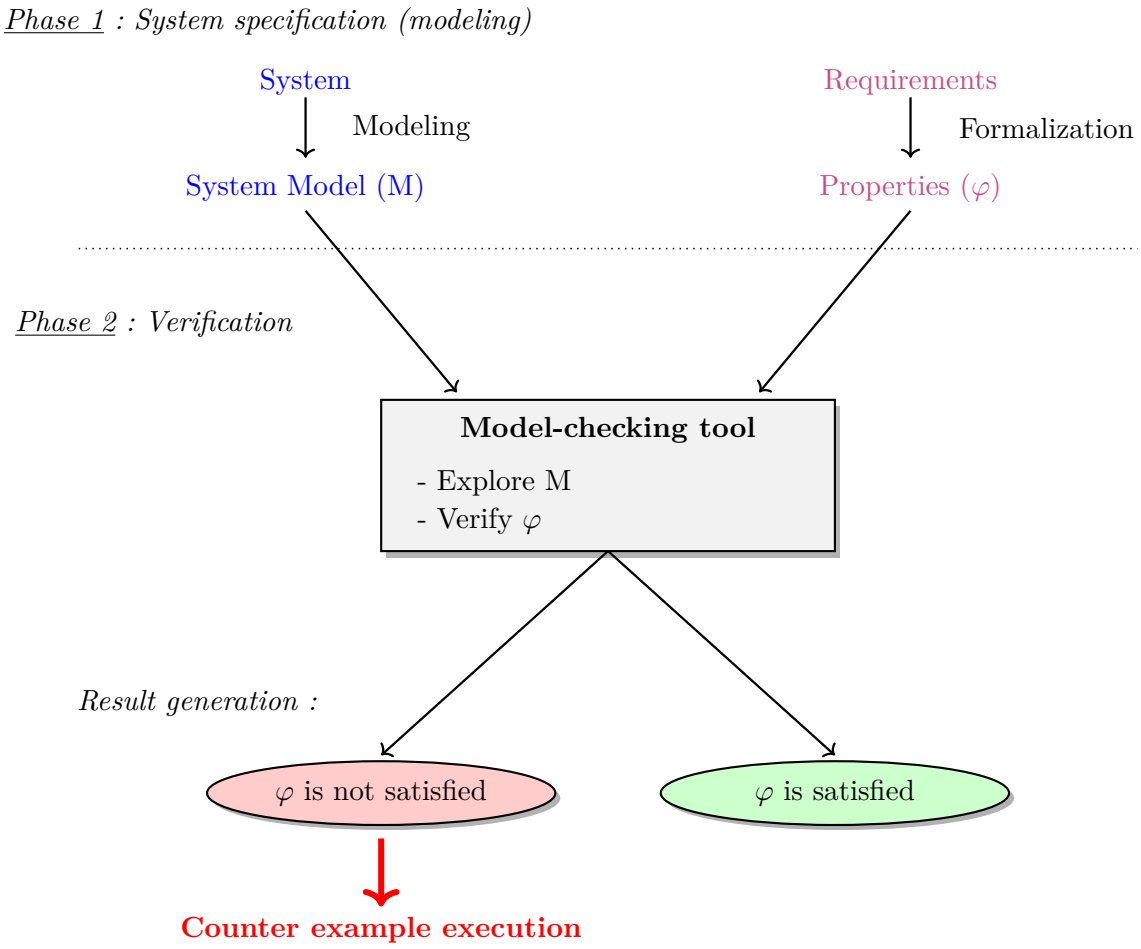


FIGURE 4.1: Processus de vérification par model-checking d'une propriété de sûreté.

Logique temporelle linéaire (LTL) : c'est une logique temporelle issue de la famille de logiques CTL* [EH83] qui permet d'exprimer des propriétés en prenant en considération l'évolution du temps. Elle permet l'exploration d'un ensemble de chemins d'exécution de manière linéaire. Un chemin désigne une séquence infinie d'états d'un système. La satisfaction d'une propriété est vérifiée de manière indépendante pour chaque chemin sans possibilité d'explorer un autre pendant la vérification. D'où la linéarité de la logique. Les propriétés à vérifier sont exprimées à l'aide de formules LTL en utilisant la sémantique décrite dans la définition 4.1. Un exemple d'une formule LTL exprimée avec cette sémantique est fourni dans la figure 4.2(a).

Définition 4.1. *(Formule LTL)*
Soit un ensemble fini AP de variables $p \in AP$ propositionnelles (des propositions

atomiques pouvant être remplacées par vrai *ou* faux*). Une formule LTL ϕ peut avoir plusieurs valeurs s'écrivant avec la syntaxe :*

$$\varphi \ ::= \ p \mid \neg \varphi \mid \varphi \vee \varphi \mid X\varphi \mid \varphi U \varphi$$

Tel que $\neg$ et $\vee$ sont des opérateurs logiques et X et U sont des opérateurs temporels modaux avec :

- **X** *: tel que* $\mathbf{X}\varphi$ *désigne que φ est vrai pour l'état suivant (neXt) sur le chemin subséquent à l'état courant.*
- **U** *: tel que* $\varphi\mathbf{U}\psi$ *désigne que φ est toujours satisfaite jusqu'à ce que (Until) ψ le soit aussi sur le chemin subséquent à l'état courant.*

Propriété 4.1. *À partir de la définition des opérateurs précédents, trois autres opérateurs modaux peuvent dériver :*

- $\mathbf{F}\varphi = \text{true } \mathbf{U}\varphi$ *: tel que* $\mathbf{F}\varphi$ *désigne que φ est satisfaite éventuellement (Finally) au moins dans un état sur le chemin subséquent à l'état courant.*
- $\mathbf{G}\varphi = \neg\, \mathbf{F}\neg\varphi$ *: tel que* $\mathbf{G}\varphi$ *désigne que p est satisfaite pour tous les états (Globally) sur le chemin subséquent à l'état courant.*
- $\varphi\mathbf{R}\psi = \neg(\neg\varphi\mathbf{U}\neg\psi)$ *: tel que* $\varphi\mathbf{R}\psi$ *désigne que ψ est toujours satisfaite tant que φ ne l'est pas (Release).*

Logique temporelle arborescente (CTL) : contrairement à la logique linéaire, CTL est une logique arborescente qui permet d'exprimer des propriétés portant sur les différents chemins d'exécution possibles à partir de l'état courant. À l'inverse de la logique LTL, l'exploration des chemins dans le cadre de la logique CTL est faite de manière arborescente. Ceci revient au fait qu'elle considère, que le futur est non déterminé. Autrement dit, tous les états futurs potentiels d'un état donné doivent être considérés même s'ils appartiennent à des chemins d'exécution différents. Ainsi, les états explorés appartenant à ces chemins forment des arbres d'exécution interdépendants. Les propriétés à vérifier sont exprimées à l'aide de formules CTL en utilisant la sémantique décrite dans la définition 4.2. Un exemple d'une formule CTL exprimée avec cette sémantique est fourni dans la figure 4.2(b).

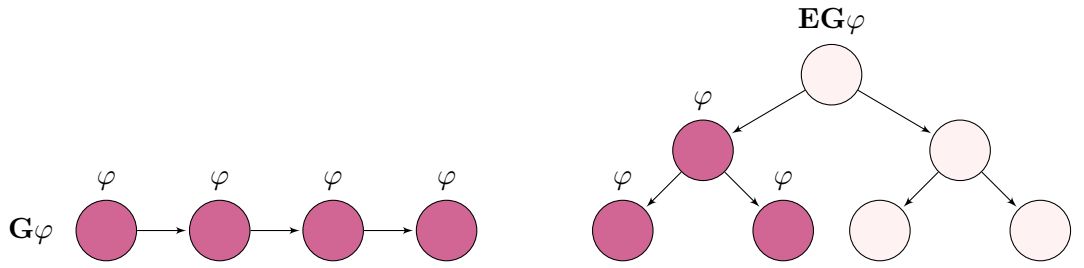
Définition 4.2. *(Formule CTL) Soit un ensemble fini AP de propositions atomiques $p \in AP$. Une formule CTL ϕ peut avoir plusieurs valeurs s'écrivant avec la syntaxe :*

$$\varphi \ ::= \ p \mid \neg \varphi \mid \varphi \vee \varphi \mid EX\varphi \mid AX\varphi \mid E\varphi U \varphi \mid A\varphi U \varphi$$

En plus des modalités temporelles U et X, la logique CTL permet d'exprimer des formules avec des quantificateurs de chemins :

***A**φ (All) : permet d'exprimer qu'une propriété φ est satisfaite sur tous les chemins issus de l'état courant.*

- ***E**φ (Exists) : permet d'exprimer qu'il existe au moins un chemin à partir de l'état courant qui permet de satisfaire la propriété φ.*



(a) Vérification d'un propriété LTL : la propriété φ est satisfaite sur tous les états du chemin d'exécution.

(b) Vérification d'une propriété CTL : il existe un chemin à partir de l'état courant pour lequel φ est toujours satisfaite.

FIGURE 4.2: Différence entre la logique temporelle et la logique arborescente

4.3.2 Les propriétés de correction

De manière générale, les propriétés de correction d'un système se décomposent, le plus fréquemment, en deux catégories [Lam77] : les propriétés de sûreté et les propriétés de vivacité. Ces deux catégories de propriétés peuvent également être ramenées à un problème d'accessibilité en cherchant les états accessibles du système permettant de les satisfaire.

la sûreté (safety) : informellement, cette propriété permet d'énoncer que « *une situation non désirée ne se produira jamais* ». Elle est formulée en CTL avec la formule : $AG\neg\varphi$, et en LTL avec la formule : $G\neg\varphi$ étant donné que le quantificateur de chemin est implicite (φ correspond à la situation non désirée). Ainsi, elle peut être vérifiée par un test d'accessibilité d'un état dans lequel φ n'est pas satisfaite. La limite de cette propriété, c'est qu'elle ne permet pas de prendre en compte l'évolution de l'exécution du système. En effet, s'il n'y a pas d'évolution, le système ne risque pas de se retrouver dans une mauvaise situation.

Exemple 4.3. propriété de sûreté :

- *« Une situation de blocage (deadlock) ne se produit jamais ».*

la vivacité (liveness) : cette propriété permet d'affirmer, informellement, qu"« *une situation désirée finit forcément par se produire* ». Ceci permet de tenir compte du cas où le système à vérifier avance dans son exécution. Cette propriété peut être formulée en CTL avec la formule : $AF\varphi$, et en LTL avec la formule : $F\varphi$ (φ correspond à la situation désirée).

Exemple 4.4. propriété de vivacité :

- *« L'ordonnanceur finit par être appelé ».*

l'accessibilité ou atteignabilité (reachability) : cette propriété énonce informellement que « *une situation désirée est atteinte* ». Ceci revient à vérifier s'il existe un chemin commençant de l'état initial et menant vers un état *state* vérifiant la situation donnée en utilisant la formule CTL : $EF(state)$. La logique linéaire ne permet pas de la formuler étant donné qu'il n'est pas possible de parcourir les chemins de manière arborescente.

Exemple 4.5. propriété d'atteignabilité :

- *« Il existe un chemin pour lequel une tâche prête sera toujours exécutée ».*

4.3.3 Avantages et limites de l'approche

Contrairement à la vérification déductive, la vérification des modèles consiste à chercher un état du système qui satisfait ou non une propriété, au lieu de le démontrer par des théorèmes. L'avantage principal de cette dernière approche est son caractère automatique : le processus de la vérification d'une propriété ne nécessite pas une intervention de la part de l'utilisateur du moment où la propriété en question est formulée. En outre, si une propriété n'est pas vérifiée, la trace d'exécution de contre-exemple générée permet facilement d'identifier l'état ou les états du système à l'origine de la violation la propriété (si l'espace d'états ne dépasse pas la capacité de la machine sur laquelle la vérification est menée). Toutefois, cette technique présente un inconvénient majeur : l'explosion combinatoire des états du système dans le processus de vérification dûe à l'exploration exhaustive de l'espace d'états. La taille de cet espace peut excéder la quantité de mémoire disponible sur la machine sur laquelle la vérification est conduite. Pour faire face à cet obstacle, des études de réduction permettant de limiter les redondances dans l'exploration ont été proposées. Il existe des méthodes qui se basent sur la détection des chemins d'exécution équivalents pour n'en parcourir qu'un seul [God90]. D'autres s'intéressent plus à la détection des ensembles d'états symétriques et en les remplaçant par une classe d'équivalence [CFJ93, Ros98] ou à une représentation compacte des états symboliquement par des diagrammes de décision tel que la technique du model-checking symbolique par BDD (*Binary Decision Diagram*) [BCM$^+$92].

4.4 Application des méthodes formelles dans la vérification des systèmes d'exploitation

L'utilisation des méthodes formelles pour la vérification des implémentations de bas niveau a longtemps été jugée coûteuse en terme de mise en œuvre, voire impossible. Au cours des dernières années, ce point de vue a changé. En effet, la recherche académique s'est de plus en intéressée à la vérification formelle ciblant le code de bas niveau. La

vérification des systèmes d'exploitation et notamment les systèmes d'exploitation temps réel en est un domaine d'application évident, du fait de l'exigence de prédictibilité dont ils font état. Ces méthodes tiennent leur succès du fait qu'elles permettent de tendre vers une garantie élevée de la correction du système à vérifier. De plus, de nos jours, fournir une preuve formelle qu'un système donné est implémenté correctement est devenue une tâche réalisable (quoique fastidieuse) avec l'émergence de nouveaux logiciels dédiés. Ainsi, plusieurs travaux dans ce contexte ont été conduits. La différence entre ces travaux relève de trois aspects principaux :

- le niveau de détail d'abstraction : certains travaux peuvent s'intéresser seulement à la vérification des services de l'OS offerts à l'application. En contrepartie, d'autres travaux peuvent plutôt mettre l'accent sur les interactions matérielles.
- la complexité du problème de vérification : il s'agit ici de l'objectif de la vérification. Certains problèmes peuvent facilement être vérifiés en suivant des méthodes restreintes offrant un degré important d'automatisation. D'autres problèmes nécessitent des méthodes plus complexes et doivent être guidés de manière interactive par l'homme.
- la taille du système à vérifier : la taille du système joue un rôle crucial dans la vérification. Selon la nature du problème résolu par le logiciel, l'effort nécessaire à la vérification n'est pas forcément linéaire, mais parfois exponentiel.

Dans la suite de cette section, nous citons, à titre d'exemples ; certains travaux de modélisation et de vérification conduits sur des systèmes d'exploitation en se basant sur les méthodes formelles. Ces travaux sont sélectionnés soit parce que la démarche de vérification et de modélisation qu'ils proposent est basée sur le model-checking, telle que la nôtre. Ou bien parce qu'ils ont le même objectif que nous en matière de vérification des systèmes d'exploitation notamment les implémentations d'ordonnanceur. Toutefois, nous restons conscients que cette liste n'est pas exhaustive et que d'autres travaux non discutés à ce niveau peuvent exister.

4.4.1 Travaux liés à la modélisation des systèmes

La librairie RTLib : en matière de modélisation, Shan et al. [SGQ16] proposent une librairie de *templates* UPPAAL d'automates temporisés offrant une sémantique unifiée pour la modélisation des systèmes temps réel. Il s'agit d'une modélisation fine et exhaustive du comportement d'une tâche pendant son exécution. Les modèles de la bibliothèque offrent plusieurs schémas d'activation de tâches avec ou sans précédence. Le modèle de l'ordonnanceur est également fourni pour un ordonnancement conduit par les événements, préemptif ou non, partitionné ou global, en considérant les politiques FP et EDF. Des exigences temporelles peuvent être spécifiées dans le but de mesurer les temps de réponse des tâches et/ou le respect de leurs échéances. Cette bibliothèque est proposée dans le cadre d'un projet plus large d'évaluation et comparaison des outils et techniques existants d'analyse d'ordonnançabilité dans les systèmes temps réel.

Modélisation de noyau Linux PREEMPT_RT : Oliveira et al. [dOCdO18] proposent un modèle du noyau de Linux PREEMPT_RT [McK05] à base d'automates pour un système monocœur. Ils suivent une approche modulaire de modélisation qui consiste à diviser le noyau en générateurs et spécifications. Les générateurs sont des événements modélisés sous forme de sous-systèmes indépendants dont chacun a un ensemble de spécifications qui constituent ses interactions avec les autres sous-systèmes. L'objectif de ce modèle est de comprendre les propriétés d'un système complexe comme Linux et s'inscrit dans le cadre d'une étude du comportement temporel de l'exécution de ses threads.

4.4.2 Travaux liés à la vérification des systèmes

Nous présentons dans cette partie certains travaux basés sur des méthodes formelles pour la vérification des systèmes d'exploitation temps réel avec différents objectifs et différents niveaux d'abstraction.

Le projet VFiasco : c'est un projet de collaboration entre l'université TU de Dresde en Allemagne et l'université RU de Nimègue aux Pays-Bas [HT05]. Il s'agit d'un projet de vérification d'une partie du noyau de L4 [Lie96] implémentée en C++. La vérification de Fiasco met particulièrement l'accent sur les propriétés des accès mémoire dans l'OS. Cependant, certaines parties substantielles du noyau, telles que l'ordonnanceur, n'ont pas été vérifiées. Le travail de modélisation et de vérification du code source C++ a été conduit dans le prouveur de théorème PVS [ORR$^+$96].

Le projet Verisoft : c'est un projet qui couvre la vérification d'un système informatique en entier (de la couche matérielle jusqu'aux programmes de l'application) [AHL$^+$08]. Le projet met l'accent sur la vérification du code assembleur et les interactions au niveau de l'architecture matérielle de leur micro-noyau générique (*General-purpose*) VAMOS [DDB08] hautement simplifié (développé en langage propriétaire C0 [Arn10]). Les preuves sont élaborées et vérifiées selon la logique d'Hoare à 75% par la machine en utilisant Isabelle/HOL [NPW02]. Toutefois, l'automatisation complète du processus de la vérification n'est pas un objectif explicite étant donné que la logique de Hoare ne le permet pas.

FreeRTOS : il s'agit d'un système d'exploitation temps réel multitâche pour les systèmes embarqués [B$^+$08]. Une approche de vérification de la correction fonctionnelle de l'ordonnanceur et de la sûreté des accès mémoire dans un système d'exploitation est proposée par Ferreira et al. [FGH$^+$14] en prenant FreeRTOS comme cas d'étude. Le processus de vérification utilise le système `HIP/SLEEK` [CDNQ12]. Les propriétés de l'ordonnanceur qui ont été vérifiées sont liées à un ordonnancement monoprocesseur à base de priorité fixe pour les tâches selon la politique round-robin.

Vérification d'un système d'exploitation d'OSEK/VDX : nous citons dans ce paragraphe deux travaux qui se sont intéressés à la vérification d'un système d'exploita-

tion conforme au standard OSEK/VDX par model-checking :

1. vérification par model-checking en utilisant l'algèbre de processus : il s'agit d'une recherche proposée par Huang et. al [HZZ+11] de modélisation et de vérification au niveau du code d'un système d'exploitation conforme au standard OSEK/VDX. Le système est modélisé en utilisant le calculus CSP [Hoa78] qui est une algèbre de processus permettant de modéliser les interactions au sein d'un système de contrôle. Le modèle de l'OS est ensuite simulé et vérifié dans le model-checker PAT [SLD08]. Les propriétés vérifiées relèvent de la gestion des tâches, la gestion des ressources et l'absence du blocage. Elles sont vérifiées par rapport à une configuration de tâches donnée. Ainsi, la stimulation du modèle construit peut ne pas être suffisante dans la mesure où elle ne permet pas de mettre l'OS face à un nombre pertinent de situations d'exécution possibles.
2. vérification par model-checking en utilisant des automates temporisés : ce travail se focalise sur la vérification des applications multi-tâches d'un système de contrôle temps réel s'exécutant sur un OS temps réel compatible avec le standard OSEK/VDX [WH08]. Le modèle formel du noyau de l'OS est construit de manière exhaustive afin d'obtenir un modèle à granularité fine. Les services de l'OS, les services d'interruption et les tâches sont abstraits par des automates temporisés et partageant des variables globales qui constituent les variables du noyau. La démarche de modélisation proposée est démontrée à l'aide d'un cas d'étude précis. Il s'agit d'un système de contrôle pour des boîtes de vitesse automatisées des véhicules. Ainsi, un certain nombre de propriétés sont formalisées sous UPPAAL [BLL+96] et qui garantissent la sûreté, la vivacité bornée et l'absence de blocage pour ce cas particulier.

Bossa : c'est un exécutif temps réel développé à l'origine à l'École des Mines de Nantes qui se base sur le noyau Linux [BM01]. Une vérification de l'OS est proposée en [BFLM07] en utilisant l'outil de preuve FMona [BF]. L'objectif est de vérifier que l'OS et la politique d'ordonnancement implémentée se comportent conformément à leur spécification. Cette vérification est assurée en étudiant le comportement du système vis-à-vis de certains événements spécifiques à l'OS (e.g terminaison d'un processus). Ces derniers sont gérés par un composant de l'OS qui définit la réaction de l'ordonnanceur à la réception d'un événement. Ainsi, pour une politique d'ordonnancement, les propriétés vérifiées s'assurent que les gestionnaires d'événements sont bien définis selon la spécification et qu'ils fonctionnent correctement selon le type d'événement reçu au niveau de l'OS.

seL4 : seL4 un micro-noyau dont la correction fonctionnelle a été prouvée en [KEH+09] en utilisant le prouveur de théorème Isabelle/HOL[NPW02]. Une chaîne de vérification complète a été fournie, en allant des exigences habituelles de sûreté de haut niveau jusqu'à celles du code machine exécutable. Ces résultats ont été possibles grâce à la simplicité du micro-noyau. Par exemple, son code noyau n'a que peu de points d'interruption, dont certains sont désactivés. Cela vient aussi du fait qu'il ne s'agit pas d'un système d'exploitation supportant des applications temps réel. Cette recherche a été également accompagnée d'une analyse précise du pire temps d'exécution.

Vérification d'un noyau compatible Ravenscar : l'étude se focalise sur le noyau (*Run-time kernel*) d'un système d'exploitation conforme au profil Ravenscar [Bur99]. Ce dernier est un sous ensemble du langage ADA à la mise en œuvre des systèmes temps réel nécessitant le plus haut niveau de sûreté. La vérification est conduite sur un modèle, élaboré en UPPAAL en utilisant des automates temporisés, avec une analyse d'accessibilité par model-checking. La vérification du noyau n'est pas entièrement menée. Elle se limite à certaines propriétés relatives à la gestion des tâches et à leurs caractéristiques temporelles. Elle est également menée pour une seule application qui est réduite afin de permettre de vérifier seulement les propriétés proposées. Les auteurs notent que cette première vérification pourrait être utilisée dans le cadre d'une étude plus large permettant ainsi de vérifier un système Ravenscar en entier avec tous les éléments du noyau, les services de l'OS ainsi que la couche bas niveau.

Vérification d'un programme concurrent : le programme est un cas de test de Piko/RT [2] qui est un noyau temps réel de type Linux optimisé pour la famille de microcontrôleurs ARM cortex-M.

Le cas de test est une application classique de producteur/consommateur ordonnancée par un ordonnanceur préemptif à priorité fixe. Un modèle existant de cet ordonnanceur est adapté pour être vérifié sous Spin [HG03] par model-checking. Trois principales propriétés sont vérifiées :

- l'absence de famine. C'est à dire qu'à aucun moment un processus ne se met en attente d'une ressource disponible ;
- l'absence de compétition entre processus pour l'acquisition d'une ressource ;
- l'absence d'interblocage des processus. C'est à dire qu'aucun processus ne se trouve en attente d'un verrou non libéré par un autre processus.

4.5 Conclusion

Les travaux d'application des méthodes formelles dans la vérification des systèmes d'exploitation sont de plus en plus nombreux avec des objectifs différents et menés pour plusieurs exécutifs. La liste des travaux présentée dans la section 4.4 ne constitue pas un état de l'art exhaustif, mais permet de présenter ceux qui se rapprochent de notre démarche de vérification. Notre objectif reste majoritairement différent des leurs étant donné que nous cherchons à proposer une démarche de vérification la plus générique qui vise le composant de l'ordonnanceur et qui pourrait éventuellement être appliquées à d'autres implémentations d'ordonnanceurs dans d'autres contextes. Cet objectif ne signifie pas que la correction de l'exécutif que nous utilisons est ignorée. Bien au contraire, notre implémentation est effectuée sur un système d'exploitation dont la vérification a été préalablement effectuée dans le cadre d'un autre travail de recherche [BRT18].

Notre étude ne traite pas l'aspect d'évaluation des politiques d'ordonnancement en terme d'analyse d'ordonnançabilité tel que c'est adressé en [SGQ16] pour la libraire RT-

[2] github.com/PikoRT/pikoRT

Lib, c'est plutôt le caractère fonctionnel de la mise en œuvre d'une politique qui nous intéresse. Dans ce contexte, les travaux autour de FreeRTOS et BOSSA s'inscrivent dans la même perspective que la nôtre. Cependant, les implémentations d'ordonnanceurs qui y sont vérifiées restent autour des politiques à priorité fixe pour les travaux en monoprocesseur. Ainsi, à l'inverse de notre études, les contraintes d'interaction et synchronisation entre processeurs dans le cadre de l'ordonnancement multiprocesseur global ne sont pas traitées. Des contraintes qui pourtant ne peuvent être ignorées si l'on veut encourager la migration des RTOS vers les politiques d'ordonnancement plus sophistiquées. C'est le cas aussi pour PREEMPT_RT, VAMOS (étudié dans le projet VeriSoft), PiKo/RT [HG03].

Parmi les études citées au préalable, certaines ne s'intéressent pas à l'ordonnanceur en particulier ou aux implémentations des politiques d'ordonnancement temps réel. En partie, cela est dû au fait que certains des systèmes d'exploitation, sur lesquels l'étude est faite, ne supportent pas les systèmes temps réel. C'est le cas de L4 et SeL4 . D'autres systèmes d'exploitation se concentrent seulement sur certains aspects de correction, tels que l'absence de blocage dans le système, en conduisant la vérification sur un seul cas d'étude ou pour une seule configuration de tâches. À l'opposé, notre contribution par rapport à ces travaux est d'abord de mener la vérification sur des politiques d'ordonnancement global, mais également de la mener avec une couverture la plus large possible et donc la plus indépendante possible de configurations particulières de tâches.

Deuxième partie

Implémentation d'un ordonnanceur global au sein de Trampoline

Cette partie est consacrée au travail d'implémentation d'un ordonnanceur de type G-EDF au sein de l'exécutif temps réel Trampoline. L'objectif de cette implémentation est, tout d'abord, d'étendre Trampoline, initialement conçu avec de l'ordonnancement partitionné à priorité fixe pour les tâches, afin d'y inclure l'ordonnancement global. Cette étape s'inscrit dans notre perspective visant à étudier l'implémentabilité des politiques d'ordonnancement global dans la mesure où il est nécessaire de se confronter à l'exercice de la réalisation d'une mise en œuvre avant d'envisager le travail de vérification. Ainsi, l'implémentation de G-EDF est effectuée, intégrée au code source de Trampoline et utilisée ultérieurement dans le cadre de notre approche de vérification des implémentations d'ordonnanceur par model-checking.

Ainsi, dans cette partie, nous abordons d'abord la présentation de l'architecture de Trampoline en introduisant les deux standards OSEK/VDX et AUTOSAR auxquels il est conforme. Ensuite nous nous focalisons sur le fonctionnement de l'ordonnanceur au sein du système d'exploitation pour pouvoir aborder la manière selon laquelle nous l'avons adapté à l'ordonnancement global. Ainsi, les choix d'implémentation faits afin d'intégrer la politique G-EDF dans l'OS sont détaillés.

CHAPITRE 5

TRAMPOLINE : UNE IMPLÉMENTATION DES STANDARDS OSEK/VDX ET AUTOSAR

5.1 Le standard OSEK/VDX

OSEK est le sigle pour "*Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug*", ce qui signifie en français : Systèmes ouverts et interfaces correspondantes pour l'électronique des véhicules automobiles. Il s'agit d'un standard qui a été créé en 1993 par un consortium de constructeurs et équipementiers automobiles allemands (BMW, Bosh, DaimlerChrysler, Opel, Siemens, et VW) ainsi qu'un département de l'Université de Karlsruhe. L'objectif était de développer un standard pour une architecture ouverte reliant les divers contrôleurs électroniques d'un véhicule tout en permettant de réduire les coûts récurrents relatifs au développement et l'adaptation des logiciels embarqués sur ces contrôleurs. En 1994, les constructeurs français Renault et PSA qui développaient un projet similaire, VDX (*Vehicle Distributed eXecutive*), rejoignirent le consortium.

L'union a été présentée, en octobre 1995, avec un standard harmonisé OSEK/VDX [G+05]. Depuis, le standard OSEK/VDX a acquis une reconnaissance plus large dans l'industrie de l'automobile, et un comité de pilotage qui comprend plusieurs constructeurs mondiaux a été formé.

5.1.1 Spécifications OSEK/VDX

OSEK/VDX définit un ensemble de spécifications pour le développement d'un système temps réel pour l'automobile. Ainsi, l'architecture spécifiée par OSEK/VDX comprend trois parties majeures :

- **OSEK COM (Communication) :** définit les protocoles de communication qui

permettent de gérer les échanges de messages entre et dans les composants logiciels du système.
- **OSEK NM (Network Manager) :** définit les protocoles de gestion des réseaux permettant la communication dans le système.
- **OSEK OS (Operating System) :** définit un ensemble de services d'un système d'exploitation temps réel pour l'exécution des applications multitâches embarquées (cf. § 5.1.2).

Le standard OSEK/VDX impose que le système d'exploitation temps réel soit statique. Ceci exige que l'application contienne uniquement des éléments qui sont définis avant le démarrage du système et qu'aucun objet ne soit créé dynamiquement. Cette configuration permet d'assurer une faible empreinte mémoire et un très faible surcoût processeur. À cet effet, OSEK/VDX spécifie un langage de configuration permettant aux concepteurs de définir statiquement les applications et les objets qui les composent avant de les compiler. Il s'agit du langage OIL (OSEK Implementation language) [Zah98]. Le processus de définition et d'exécution d'une application sous un système d'exploitation standardisé OSEK/VDX est détaillé dans la section 5.3.3. Notons que cette définition statique d'une application est un facteur important pour les industries où la sûreté de fonctionnement est importante. Elle permet également de réduire la complexité du processus de compilation de l'application et facilite les tests et le débogage du système.

5.1.2 OSEK OS

Un système d'exploitation OSEK est un support d'exécution pour les programmes d'application temps réel. Il permet une exécution temps réel contrôlée de plusieurs processus[1] qui fonctionnent de manière concurrente. Ainsi, il fournit plusieurs objets pour l'utilisation du CPU. Les principaux d'entre eux sont cités ci-dessous :

- **les tâches :** ce sont des entités qui offrent une structure logicielle (Framework) pour l'exécution d'un programme de contrôle (cf. § 5.1.3).
- **les ISRs (Interrupt Service Routine) :** ce sont des routines déclenchées à chaque fois qu'une interruption a lieu. Deux types d'ISR sont proposées par OSEK : (i) les ISRs de catégorie 1 qui sont exécutées directement en utilisant le contexte actif quand elles sont détectées. Ainsi, elles n'utilisent aucun service de l'OS et ne sont pas réellement ordonnancées ; (ii) les ISRs de catégorie 2 qui sont capables de faire des appels aux services du système d'exploitation pouvant alors éventuellement entraîner un ré-ordonnancement.
- **les événements :** ce sont des objets permettant de synchroniser deux tâches en créant un ordre de précédence entre leurs exécutions.
- **les ressources :** elles permettent de contrôler les accès concurrents à des ressources partagées (e.g. périphériques d'entrée/sortie, zone mémoire, etc.). Ce mécanisme est assuré en mettant en place des sections critiques dans les programmes des tâches.

[1] Dans le suite de ce document, un processus est considéré soit comme une tâche, soit comme une ISR2.

- **les compteurs :** un compteur est objet basé sur une horloge système et qui permet de compter jusqu'à une certaine valeur spécifiée et de déclencher en conséquence une alarme (cf. § 5.1.5).
- **les alarmes :** elles sont utilisées pour mettre en place des événements récurrents en association avec des compteurs (cf. § 5.1.5).

Un système d'exploitation OSEK définit trois niveaux de traitement : (i) traitement des interruptions ; (ii) traitement de l'ordonnanceur ; (iii) traitement de tâches. Ces traitements se font selon une priorité : les tâches possèdent une priorité de traitement inférieure à celle l'ordonnanceur, et celui-ci possède une priorité de traitement inférieure à celle des interruptions.

5.1.3 Gestion des tâches

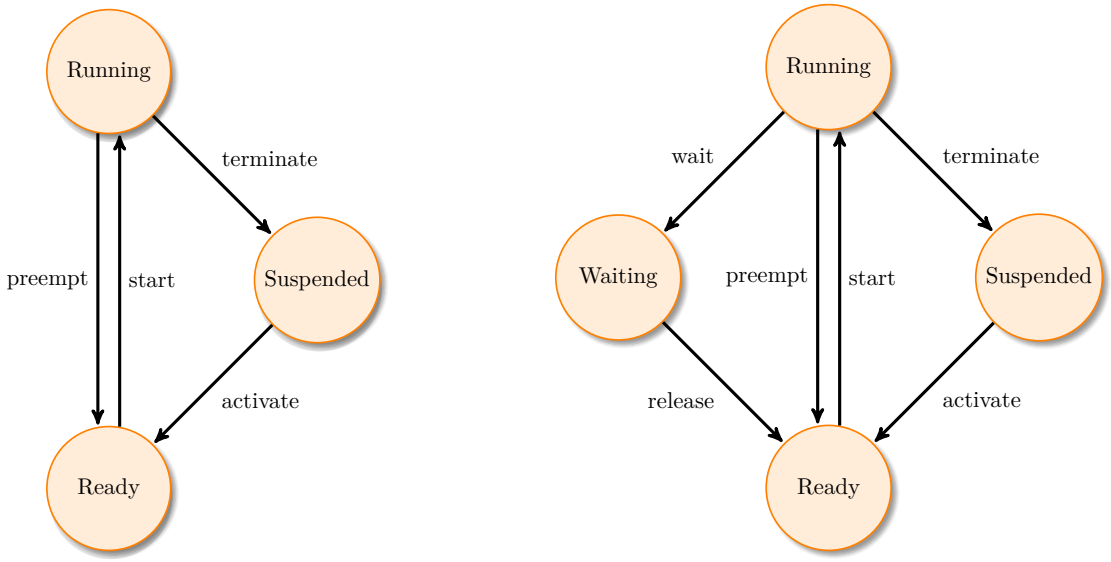
Notion de tâche OSEK : une application temps réel est constituée de plusieurs programmes de contrôle. Ces programmes peuvent être subdivisés en plusieurs parties exécutées en fonction de leurs exigences temps réel et des traitement qu'elles doivent effectuer. Ces parties sont mises en œuvre par le biais de portions de code séquentiel appelées *tâches*. Une tâche peut avoir plusieurs instances d'exécution appelées *travaux*. Le système d'exploitation OSEK permet une exécution simultanée et asynchrone de ces travaux contrôlée par l'ordonnanceur. Le standard OSEK définit deux catégories de tâches :

- *tâches basiques* : ce sont des tâches dont l'exécution n'est pas susceptible d'être bloquée suite à l'attente d'un événement. Ainsi, elles libèrent le CPU seulement quand elles sont préemptées par d'autres tâches, terminées ou lorsqu'une interruption est reçue.
- *tâches étendues* : à l'inverse des tâches basiques, les tâches étendues peuvent libérer le CPU pour se mettre en attente d'un événement.

Les états d'une tâche OSEK : Durant sa durée de vie, une tâche passe par plusieurs états. Les différents états possibles selon le standard OSEK sont :

- *Suspendue (Suspended) :* c'est un état passif dans lequel la tâche est endormie en attendant qu'elle soit activée.
- *Prête (Ready) :* dans cet état, la tâche a été activée ou préemptée et est en attente qu'un processeur soit libéré pour y être exécutée.
- *En exécution (Running) :* dans cet état, la tâche est en train de s'exécuter sur un processeur.
- *En attente (Waiting) :* dans cet état, la tâche est en attente d'un événement. Elle reste bloquée dans cet état jusqu'à ce que l'événement en question soit signalé. Cet état ne concerne que les tâches étendues.

Notons qu'une tâche basique ne peut être que dans les trois premiers états. Tandis qu'une tâche étendue peut passer par les quatre états (cf. Fig. 5.1).



(a) Graphe d'états d'une tâche basique

(b) Graphe d'états d'une tâche étendue

FIGURE 5.1: Les états d'une tâche selon le standard OSEK

Les services de gestion d'une tâche OSEK : la gestion des tâches au sein d'un système d'exploitation OSEK se fait par le biais de certains services de l'OS :

- `ActivateTask` : permet d'activer un nouveau travail de la tâche dont l'identifiant est passé en paramètre. La tâche passe de l'état `SUSPENDED` à l'état `READY` si elle n'a pas un autre travail qui n'est pas encore terminé.
- `TerminateTask` : permet de terminer le travail en cours de la tâche appelant le service. Ainsi, son état passe de `RUNNING` à `SUSPENDED` si elle n'a pas un autre travail qui n'est pas encore terminé. Notons qu'une tâche ne peut pas en terminer une autre.
- `ChainTask` : permet de terminer le travail de la tâche appelant le service et d'activer un nouveau travail de la tâche dont l'identifiant est passé en paramètre.
- `Schedule` : permet de mettre à l'état `RUNNING` la tâche prête la plus prioritaire. C'est un service qui entraîne un ré-ordonnancement.
- `GetTaskState` : renvoie l'état actuel de la tâche dont l'identifiant est passé en paramètre.
- `GetTaskID` : renvoie l'identifiant de la tâche en cours d'exécution.

Classes de conformité : le système d'exploitation OSEK définit des classes de conformité qui permettent de couvrir une large gamme d'applications avec différents niveaux de complexité. Ainsi quatre classes de conformité sont distinguées selon le type des tâches, la possibilité d'activations multiples et la possibilité d'attribution d'une même priorité à plusieurs tâches. Notons que l'activation multiple correspond à la possibilité d'activer

un ou plusieurs nouveaux travaux de tâche alors qu'elle possède déjà au moins un autre travail qui n'est pas encore terminé. Ces quatre classes sont :

- **BCC1 :** correspond aux applications qui ne contiennent que des tâches basiques sans activations multiples et une seule tâche par niveau de priorité.
- **BCC2 :** c'est une extension de la classe *BCC1* qui autorise des activations multiples d'une tâche et la possibilité d'avoir plusieurs tâches par niveau de priorité.
- **ECC1 :** correspond aux applications supportant des tâches basiques et étendues avec une seule demande d'activation par tâche et une seule tâche par niveau de priorité.
- **ECC2 :** c'est une extension de la classe *ECC1* qui autorise des activations multiples d'une tâche basique et la possibilité d'avoir plusieurs tâches par niveau de priorité.

5.1.4 Ordonnancement des tâches

L'ordonnanceur est le composant du noyau de l'OS permettant de gérer l'allocation des processeurs aux tâches pour leur exécution. Le standard OSEK définit une politique d'ordonnancement à priorité fixe pour les tâches. La priorité d'une tâche est ainsi assignée statiquement. La tâche la moins prioritaire se voit attribuer la valeur de priorité la plus basse. Ainsi, l'ordonnanceur se base sur ces priorités pour faire passer les états des tâches de `READY` à `RUNNING` et vice versa. Il existe trois types d'ordonnancement possibles en OSEK :

- *Ordonnancement préemptif :* si une tâche prête a une priorité plus élevée que celle qui est en cours d'exécution, l'ordonnanceur préempte cette dernière pour attribuer le processeur à la tâche la plus prioritaire.
- *Ordonnancement non-préemptif :* les tâches ne peuvent pas être préemptées. Ainsi, une tâche prête plus prioritaire peut s'exécuter seulement quand celle qui est en cours d'exécution a terminé ou est mise en attente d'un événement.
- *Ordonnancement mixte :* dans ce type d'ordonnancement, la règle suivie dépend de la tâche qui s'exécute. Si elle est définie comme ne pouvant pas être préemptée, alors elle ne le sera pas. Sinon, elle pourra être préemptée par une tâche plus prioritaire.

5.1.5 Gestion des événements récurrents

Le standard OSEK propose un processus en deux étapes pour pouvoir gérer les événements récurrents dans le temps (tels que les activations périodiques des tâches) :

- au niveau de l'OS : des compteurs sont implémentés par des objets permettant d'enregistrer des "*ticks*" provenant d'une horloge.
- au niveau de l'application : des alarmes associées aux compteurs sont disponibles. Chaque alarme expire à une valeur prédéfinie du compteur qui lui est associé et déclenche l'action qui lui est assignée (e.g. activation d'une tâche).

Notons qu'une action d'alarme peut être une activation d'une tâche ou une signalisation d'un événement. En OSEK, les alarmes et les compteurs sont déclarés et définis

statiquement. Ainsi, l'association alarme/compteur et l'action à exécuter sont également prédéfinies avant le démarrage du système.

5.2 AUTOSAR

AUTOSAR (*AUtomotive Open System ARchitecture*) [AUT03] a été fondé en 2003 en tant que partenariat de développement des logiciels embarqués pour l'automobile. Son objectif est de fournir une norme industrielle commune pour l'architecture logicielle automobile entre fournisseurs et constructeurs. cette norme permet de maîtriser la complexité croissante des architectures électroniques dans le domaine de l'automobile en séparant la gestion des logiciels et du matériel. Cette approche accroît l'indépendance, la maintenabilité et la réutilisation des logiciels tout en réduisant leurs coûts de développement. Il se base sur le standard OSEK/VDX pour la spécification des systèmes d'exploitation pour l'embarqué. La norme AUTOSAR offre également un support pour les systèmes d'exploitation multicœurs depuis sa version 4.0 [C$^+$09].

5.2.1 Architecture d'AUTOSAR

L'architecture proposée par AUTOSAR est fondée sur une approche modulaire permettant de faciliter la construction du logiciel à partir de briques élémentaires tout en abstrayant la couche matérielle. Ceci permet de pouvoir exécuter la même application de manière transparente sur différentes plateformes matérielles. Ainsi, cette architecture est constituée de trois couches logicielles qui s'exécutent sur un microcontrôleur constituant la couche matérielle. Ces couches sont présentées sur la figure 5.2 et détaillées ci-dessous :

- **la couche applicative (*Application Layer*) :** c'est la couche la plus haute qui contient les différents composants logiciels applicatifs appelés SWCs (*pour Software Components*) à exécuter. Ces composants sont divisés en fonctions écrites en langage C appelées des *Runnables*. Ces derniers mettent en œuvre les traitements applicatifs permettant la gestion des capteurs/actionneurs, des services de proxy, etc. Ils sont intégrés par le RTE (la couche ci-après) sous forme de tâche à exécuter au sein du système d'exploitation.
- **le RTE (*Runtime Environment*) :** c'est une interface qui constitue un lien de communication entre les différents SWCs. Elle permet également aux services de la couche applicative de communiquer avec ceux de la couche basse et contrôle leurs interactions. Plus précisément, le RTE est une implémentation du VFB (*pour Virtual Function Bus*) qui est un environnement de communication abstrait permettant de connecter les SWCs et d'être indépendant de la plate-forme matérielle sous-jacente. Il organise également l'exécution des *Runnables* en les passant sous forme de tâches au système d'exploitation.
- **la couche basse (*Basic Software Layer*) :** elle fournit les services de base permettant l'exécution des SWCs. Elle est constituée des sous-couches suivantes :

- *le système d'exploitation :* les caractéristiques d'un exécutif conforme à AUTOSAR sont largement héritées du standard OSEK/VDX. Le concept d'un *runnable* étant masqué, puisque le RTE l'intègre en tant que tâche du système, le système d'exploitation maintient ainsi un ensemble de tâches géré par un ordonnanceur pour leur exécution.
- *la couche des services (Services Layer)* : elle regroupe des services de haut niveau utilisés par les SWCs tels que les services de gestion de mémoire, services de communication réseau, etc.
- *la couche d'abstraction d'ECU* [2] *(ECU Abstraction Layer)* : cette couche fournit l'accès à toutes les fonctionnalités d'un ECU telles que la gestion de la communication, de mémoire et des entrées/sorties. Que ces fonctionnalités fassent partie du microcontrôleur ou qu'elles soient implémentées par des composants périphériques, la couche permet de les abstraire vis-à-vis des services applicatifs.
- *la couche des pilotes de périphérique (Complex Device Drivers)* : cette couche regroupe les pilotes des périphériques qui ne sont pas standardisés en AUTOSAR.
- *la couche d'abstraction du microcontrôleur (Microcontroller Abstraction Layer)* : cette couche contient des pilotes spécifiques au matériel pour l'accès à la mémoire, aux entrées/sorties et au bus de communication du microcontrôleur.

5.2.2 Développement d'une application en AUTOSAR

Le développement d'une application logicielle pour un ECU en AUTOSAR est réalisée en suivant une configuration statique en quatre étapes :

- configuration du système : durant cette étape l'ensemble des composants de l'application est spécifié (SWCs, ports, interfaces, connecteurs, etc). L'interconnexion entre ces composants ainsi que les protocoles de communication doivent également être définis. Cette spécification est générée dans un fichier XML (`System Description`) qui est utilisé dans l'étape qui suit.
- extraction des informations spécifiques à l'ECU : cette étape consiste à définir l'implémentation des SWCs en déterminant le comportement des *Runnables* qui les composent. Cette implémentation est générée dans un fichier XML (`ECU Exctract of System Configuration`) qui est utilisé dans l'étape suivante.
- configuration de l'ECU : dans cette étape le *mapping* entre les *Runnables* et tâches de l'OS est défini dans un fichier XML (`ECU Configuration Description`).
- génération de l'exécutable : l'exécutable de l'application est généré à ce stade en compilant et liant les fichiers de configuration générés dans les étapes précédentes.

[2] Unité de Contrôle Électronique : c'est une unité d'un système électronique dans le domaine d'automobile qui se compose d'un calculateur électronique et d'un logiciel embarqué permettant la gestion des dispositifs matériels du système.

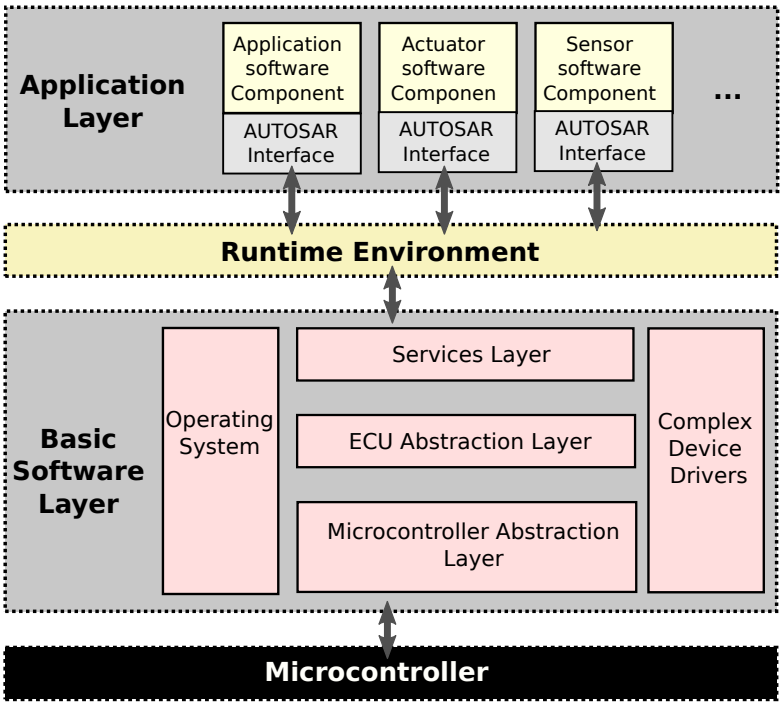


FIGURE 5.2: L'architecture d'AUTOSAR

5.3 L'exécutif temps réel Trampoline

5.3.1 Présentation

Trampoline [BBFT06] est un système d'exploitation temps réel statique qui se base sur le standard OSEK/VDX [G+05] et son successeur AUTOSAR [FMB+09] pour le support du multicœur. Trampoline est développé depuis 2005 au sein de l'équipe STR (*Systèmes Temps Réel*) au LS2N [3]. Il a été conçu dans le but d'un usage interne pour la recherche scientifique, l'enseignement et pour l'expérimentation de nouveaux algorithmes d'ordonnancement, mais a été également utilisé dans un cadre industriel. L'exécutif est disponible en deux versions : une version monocœur et une autre en multicœur. Dans ce qui suit, nous nous focalisons sur l'aspect multicœur de l'OS.

5.3.2 Architecture de Trampoline

L'architecture de Trampoline en multicœur comprend trois parties majeures qui sont également présentées dans la figure 5.3 :

L'API (*Application Programming Interface*) : c'est une interface qui regroupe tous les services OSEK et AUTOSAR qui sont offerts à l'application. Ces services peuvent

[3] Laboratoire des Sciences du Numérique de Nantes (`www.ls2n.fr`)

être activés ou désactivés selon les besoins de manière à n'inclure que les services nécessaires à l'application. Les services de gestion des tâches OSEK présentés dans la section 5.1.3 sont inclus dans l'API. D'autres services existent pour la gestion des alarmes, des compteurs, des ressources, etc. AUTOSAR ajoute d'autres services permettant la gestion des ISR, des compteurs logiciels, de la communication inter-OS Applications, du temps global, etc.

Le Noyau (*Kernel*) : il regroupe toutes les fonctions de bas niveau sur lesquelles s'appuient les services de l'API. Ces fonctions sont implémentées en langage C. Elles permettent de gérer le démarrage et l'arrêt du système d'exploitation, le démarrage, l'arrêt et la synchronisation des tâches, l'ordonnancement, etc. Il est constitué de quatre composants majeurs :

- *le « dispatcheur » d'interruptions :* suite à la réception d'une interruption, ce composant permet d'effectuer certaines des opérations suivantes : (i) l'incrémentation d'un ou plusieurs compteurs ; (ii) l'appel d'une ISR1 ; (iii) l'activation d'une ISR2 (cf. Sec. 5.1.2).
- *le gestionnaire des compteurs :* il est sollicité par le *dispatcheur* des interruptions quand la source d'interruption exige l'incrémentation d'un ou plusieurs compteurs. Il permet de gérer également les alarmes liées aux compteurs. Ainsi, lorsque la valeur incrémentée par un compteur correspond à la valeur d'expiration d'une alarme, l'action associée à cette alarme est effectuée (activation de tâche ou signalisation d'un événement).
- *le gestionnaire de tâches* : ce composant contient toutes les fonctions du noyau qui permettent la gestion de l'activation, la terminaison, la mise en attente d'un événement ou la libération d'une tâche. Il est appelé par le gestionnaire de compteurs ou le dispatcheurs d'interruption.
- *l'ordonnanceur :* il s'agit du composant principal du noyau qui permet de gérer l'exécution des tâches. Il est appelé par le gestionnaire des tâches et le dispatcheur des interruptions. Le principe de fonctionnement de ce composant au sein de Trampoline est présenté de manière plus détaillée dans la section 5.4.

Le BSP *(Board Support Package)* : cette partie regroupe les fonctions bas niveau qui sont dépendantes de la cible d'exécution de l'OS. Le langage d'implémentation dépend également de la cible. Le BSP comprend cinq modules :

- *le gestionnaire des interruptions externes :* il permet de traiter les interruptions provenant des sources externes telles qu'un *timer*. Si l'interruption reçue provoque un ré-ordonnancement, il doit alors en informer l'ordonnanceur.
- *le gestionnaire des interruptions inter-cœurs :* ce composant permet de gérer les interruptions qui assurent la communication entre les cœurs dans une architecture multicœur.
- *le gestionnaire des appels système* : ce composant permet de gérer les appels des services de l'API et aussi les passages du mode utilisateur vers le mode noyau.

- *Le gestionnaire de changement de contexte :* ce composant assure la sauvegarde du contexte de la tâche qui perd le CPU et la restauration de celui de la tâche qui va y être exécutée.
- *Le gestionnaire de la protection mémoire :* ce composant s'occupe de la programmation de l'Unité de la Protection Mémoire qui gère les droits d'accès à la mémoire de l'OS.

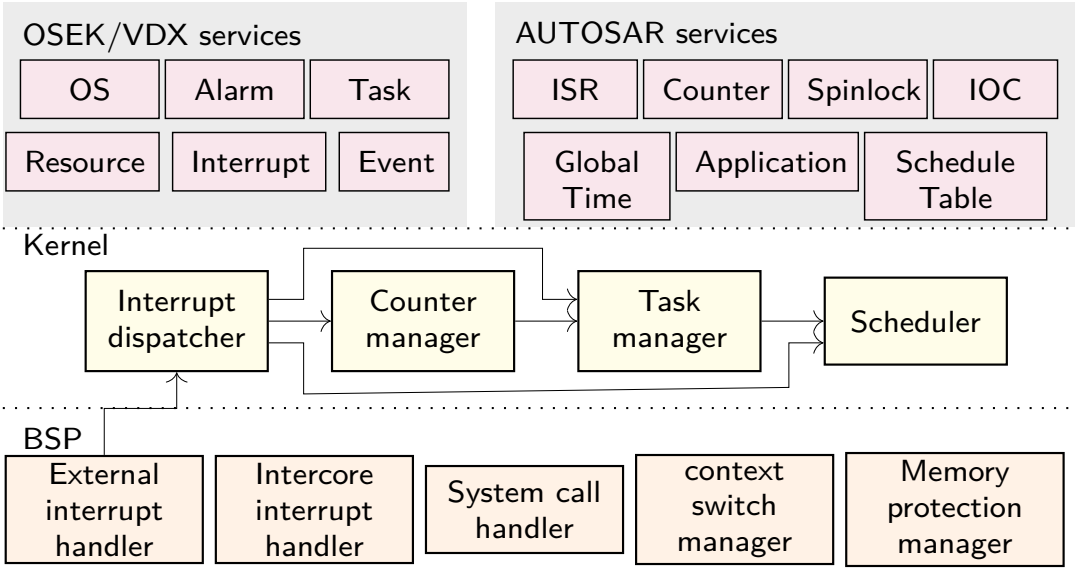


FIGURE 5.3: L'architecture de Trampoline

5.3.3 Exécution d'une application sous Trampoline

Trampoline étant un système d'exploitation temps réel statique, il n'offre pas la possibilité de créer des objets en ligne et doit être configuré hors ligne. Ceci revient à déclarer et initialiser toutes les structures de données et les objets gérés par le système avant la phase de la compilation. Les objets OSEK (tâches, ISRs, alarmes, compteurs, événements, etc.) sont définis par le biais du langage OIL.

Une description OIL du système peut être réalisée manuellement comme elle peut être générée automatiquement par un outil de configuration système. Elle est composée essentiellement de deux parties : (i) une définition d'implémentation dans laquelle sont déclarés tous les objets avec leurs propriétés ; (ii) une définition d'application qui contient les valeurs attribuées (initiales ou par défaut) aux objets déclarés dans la définition d'implémentation. Cette description est vérifiée par le compilateur GOIL qui fait appel à un interpréteur de Templates pour la génération des fichiers (.c/.h) contenant des structures de données qui sont inclues dans le code source de Trampoline.

En multicœur, les composants d'une application AUTOSAR (SWCs, ports, connecteurs, etc.) sont également définis dans des fichiers XML en plus de leur implémentation

en *runnables* et le *mapping* avec des tâches de l'OS (cf. § 5.2.2). Les structures de données correspondantes sont générées avec un outil de génération conforme à AUTOSAR.

Le corps de l'application est ensuite élaboré en y incluant les fichiers (.h) du code source de Trampoline et est compilé avec ce dernier et les fichiers des structures de données. Les fichiers objets générés sont liés avec un éditeur de liens pour obtenir l'exécutable de l'application. Ce processus est illustré dans la figure 5.4.

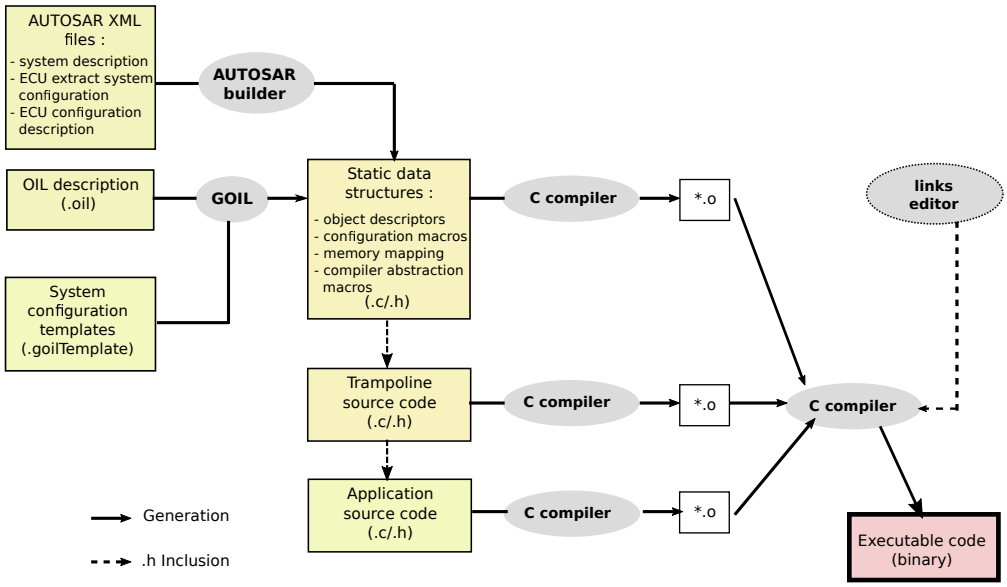


FIGURE 5.4: Construction d'une application en Trampoline

5.3.4 Généralités sur l'implémentation de l'OS

Les tâches en Trampoline : en Trampoline, les tâches sont des objets représentés par des descripteurs. Un descripteur sert à regrouper en mémoire l'ensemble des informations relatives à l'objet qu'il décrit. Le descripteur de tâche est divisé en deux parties : (i) un descripteur statique contenant les données relatives à la tâche qui ne sont pas susceptibles de varier et peuvent être stockées dans la mémoire morte (*ROM : Read-Only Memory*) ; (ii) un descripteur dynamique qui réunit l'ensemble des données de la tâche qui sont mises à jour pendant le fonctionnement du système et qui doivent être stockées dans la mémoire vive (*RAM : Random Access Memory*). Ces deux descripteurs sont implémentés sous forme de structures en langage C (`struct`). Dans ce qui suit, nous présentons les principaux attributs d'une tâche Trampoline stockés dans ses descripteurs.

a **Les attributs statiques d'une tâche :** ils sont présentés dans l'ordre de leur apparition dans le *listing* 5.1 :

- ***l'identifiant*** (`id`) de la tâche.
- ***la priorité*** (`base_priority`) de la tâche fournie dans la description OIL par l'utilisateur.

- ***le nombre maximum des activations*** (`max_activate_count`) fourni dans la description OIL. En effet, Trampoline permet d'activer successivement une tâche même si elle possède un travail qui n'est pas encore terminé. Ces activations peuvent se faire jusqu'à une certaine limite fixée statiquement qui correspond au nombre maximum des activations.
- ***le type*** (`type`) qui indique s'il s'agit d'une tâche basique ou étendue.
- ***le point d'entrée*** (`entry_point`) qui désigne un pointeur vers la première instruction du code de la tâche. Avec cette entrée, le système d'exploitation peut exécuter la tâche en question.
- ***le contexte (`context`) :*** lorsqu'une tâche est en train de s'exécuter, elle utilise les registres internes du cœur sur lequel elle s'exécute et accède à la mémoire en y stockant des valeurs propres à elle. Son contexte est défini par l'état courant de ces registres. Le contexte doit ainsi être sauvegardé quand la tâche est préemptée au profit d'une autre ou mise en attente d'un événement, il doit également être restauré quand elle reprend son exécution. Ce processus est présenté dans ce qui suit.
- ***l'identifiant de l'application (`app_id`)*** : identifiant de l'application OSEK ou AUTOSAR à laquelle la tâche en question appartient.

```
struct TPL_PROC_STATIC {
   tpl_task_id            id;
   tpl_priority           base_priority;
   tpl_activate_counter   max_activate_count;
   tpl_proc_type          type;
   tpl_proc_function      entry;
   tpl_context            context;
#if WITH_OSAPPLICATION == YES
   tpl_app_id             app_id;
#endif
   ...
};
```

Listing 5.1: Descripteur statique

b **Les attributs dynamiques d'une tâche :** ils sont présentés dans l'ordre de leur apparition dans le *listing* 5.2 :

- ***le compteur d'activations*** (`activate_count`) : il correspond au nombre de travaux non terminés que la tâche possède à l'instant courant. Quand un nouveau travail de la tâche est activé, la valeur enregistrée dans cet attribut est comparée au nombre maximum des activations. Si elle est supérieure ou égale à ce nombre, l'activation échoue. Sinon, l'activation est enregistrée et le compteur d'activations est incrémenté.
- ***l'identifiant du cœur*** (`core_id`) : il enregistre l'identifiant du cœur sur lequel la tâche s'exécute. Cet attribut est utilisé seulement dans la version multicœur de Trampoline.
- ***l'état de la tâche*** (`state`) : c'est une variable de type énuméré qui enregistre l'état courant de la tâche tel que défini par le standard OSEK : **`SUSPENDED`**, **`READY`**,

`RUNNING` ou `WAITING` dans le cas d'une tâche étendue (cf. 5.1.3). En plus de ces états, Trampoline définit deux états supplémentaires pour la gestion interne des tâches (cf. Fig. 5.5) :

- `AUTOSTART` : cet état concerne seulement les tâches qui doivent s'activer automatiquement au démarrage de l'OS. C'est un état initial dans lequel une telle tâche est placée avant que l'OS ne l'active.
- `READY_AND_NEW` : c'est un état utilisé pour désigner une tâche ayant un travail prêt nouvellement activé mais pas encore exécuté pour la première fois. Ainsi, si ce travail est mis en exécution et préempté par la suite, la tâche en question ne revient pas à l'état `READY_AND_NEW`, elle se met plutôt dans l'état `READY`.

```
struct TPL_PROC {
   tpl_activate_counter   activate_count;
#if NUMBER_OF_CORES > 1
   tpl_core_id            core_id;
#endif
   tpl_proc_state         state;
   ...
};
```

Listing 5.2: Descripteur dynamique

Il faut bien noter que lorsqu'une tâche fait état de plusieurs travaux non terminés à un instant donné (en raison d'activations multiples successives), c'est toujours son travail le plus ancien qui doit être géré par l'ordonnanceur. Par conséquent, les données enregistrées dans son descripteur dynamique (tel que la date d'échéance, l'état, etc.) concernent son travail le plus ancien.

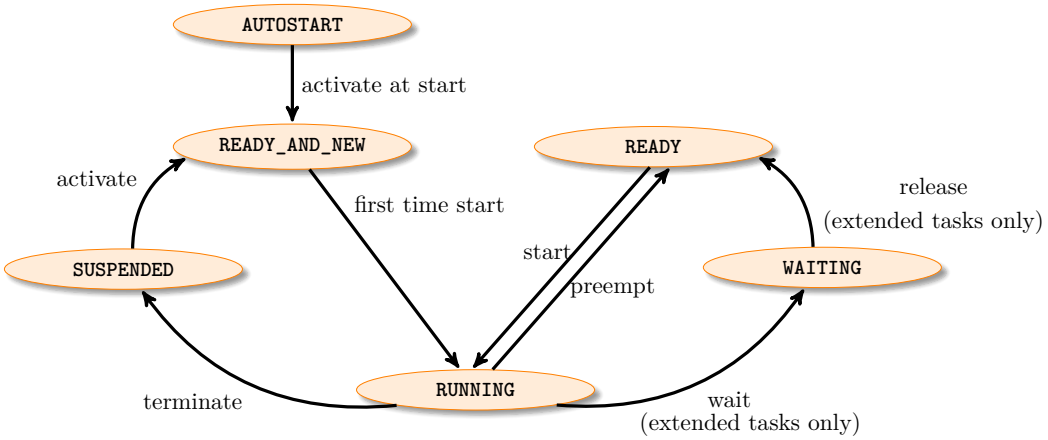


FIGURE 5.5: Les états d'une tâche en Trampoline : `AUTOSTART` correspond à l'état initial seulement pour les tâches devant démarrer avec le système d'exploitation. Les autres tâches ont `SUSPENDED` pour état initial.

Le code d'une tâche : certains attributs d'une tâche doivent être définis par l'utilisateur. Il s'agit d'une description OIL à travers un objet OSEK appelé `TASK` permettant

ensuite de générer les structures de données de la tâches tel que c'est indiqué par le processus de compilation d'une application de la figure 5.4. Un exemple de description OIL d'une tâche OIL est fourni dans le *listing* 5.3.

```
TASK TaskName
{
    PRIORITY   = 1;
    AUTOSTART  = TRUE;
    ACTIVATION = 2;
    ...
};
```

Listing 5.3: Exemple de description d'une tâche AUTOSTART dont la priorité est fixée à 1 et qui peut avoir au maximum deux travaux non terminés au même instant

Trampoline fournit également une macro en langage C appelée également `TASK` permettant de définir le code exécuté par la tâche dans une application. Cette macro prend en paramètre le nom de la tâche tel que c'est indiqué dans la description OIL. Elle est exécutée quand la tâche est élue par l'ordonnanceur. À la fin de son exécution, elle finit par appeler le service OSEK `TerminateTask()` permettant de terminer la tâche (cf. *listing* 5.4).

```
TASK (TaskName)
{
   /* code of the Task  */
   ...
   TerminateTask();
}
```

Listing 5.4: Le code d'une tâche

La notion de la tâche *Idle* : c'est une tâche basique qui est activée au démarrage de l'OS. Sa priorité est égale à 0 qui correspond à la priorité la plus basse dans le système (les autres tâches ont impérativement une priorité supérieure ou égale à 1). La tâche *idle* s'exécute sur un cœur lorsqu'aucune autre tâche ne doit s'y exécuter, c'est à dire au démarrage initial du système ou lorsque le cœur est oisif. Comme toutes les autres tâches, elle est gérée par l'ordonnanceur et elle possède également deux descripteurs, statique et dynamique, dans lesquels ses attributs sont stockés.

Le changement de contexte d'une tâche : une tâche est associée à une portion de code séquentiel susceptible (hors contexte non préemptif) d'être interrompue en tout point de son avancement. C'est pourquoi il est indispensable que le contexte d'exécution associé à ce point (c'est à dire les valeurs courantes que contiennent les registres internes du cœur), puisse alors être mémorisé et ensuite restauré pour que la suspension d'exécution soit transparente. Trampoline, comme tout autre système d'exploitation, met en

place un mécanisme de changement de contexte qui est géré par le gestionnaire de changement de contexte de la couche BSP (cf. l'architecture de Trampoline fig. 5.3). Ainsi, quand une tâche τ_i est préemptée ou mise en attente, les valeurs des registres du cœur C_k qu'elle occupe sont copiées dans l'attribut `context` de son descripteur ce qui correspond à une opération de sauvegarde de contexte. Si une autre tâche τ_j est sélectionnée pour s'exécuter sur C_k, le contexte enregistré dans son descripteur est chargé dans les registres du cœur, ceci correspond à une opération de chargement ou restauration de contexte. Pareillement, quand τ_i reprend son exécution, son contexte préalablement sauvegardé est restauré sur le cœur de son exécution.

L'exécution d'un service de l'OS : les services en Trampoline constituent la partie principale de l'API offerte aux applications (cf. l'architecture de Trampoline fig. 5.3). Du point de vue applicatif, un appel de service désigne tout simplement un appel d'une fonction de cette couche. Pour son exécution, un basculement vers le **mode superviseur** ou **mode noyau** est nécessaire. Ce mode correspond à un niveau de privilège permettant un accès total aux ressources de la machine et toutes les instructions peuvent être exécutées. Contrairement au **mode utilisateur**, qui est utilisé au niveau de l'application, et permet d'empêcher l'accès aux données du noyau et ainsi de prévenir la corruption des structures de données de l'OS. Le passage en mode noyau se fait via un *appel système* qui est une interruption logicielle dont la routine conduit à l'appel d'une des fonctions du noyau au bénéfice d'un programme de l'application. Il est géré par le gestionnaire des appels système. Ce dernier effectue l'appel du service en se basant sur son identifiant qui est renseigné par la fonction de l'API correspondante. Le service appelé peut conduire à un ré-ordonnancement avec pour conséquence un changement de contexte (eg. service d'activation d'une tâche). Le cas échéant, le gestionnaire de changement de contexte est appelé pour l'effectuer. Une fois que cela est fait, l'exécution est reprise par la tâche qui vient d'être ordonnancée en mode utilisateur. Ce processus est illustré dans la figure 5.6.

Le verrou global : dans la version multicœur de Trampoline, il n'y a qu'une seule instance du noyau de l'OS qui peut s'exécuter à un instant donné et ce, sur un seul cœur. Toutefois, de manière séquentielle, ce noyau peut s'exécuter sur deux cœurs différents. En effet, l'accès au noyau est séquencé grâce à un verrou global. Le noyau est verrouillé lors d'un appel de service du système d'exploitation (qui se traduit par un appel système) ce qui interdit à plus d'un cœur d'exécuter son code ou de traiter les interruptions simultanément. À titre d'exemple, lorsque l'ordonnanceur est appelé sur un cœur, le service conduisant à un ré-ordonnancement acquiert le verrou du noyau et ne le retourne qu'à la fin du ré-ordonnancement avant de quitter le mode noyau. Ainsi, si un autre ré-ordonnancement est nécessaire sur un autre cœur, celui-ci se met en attente de la libération du verrou.

Afin de mieux illustrer ce processus, nous considérons l'exemple d'un ensemble de quatre tâches $\tau = \{\tau_0, \tau_1, \tau_2, \tau_3\}$ devant s'exécuter sur deux cœurs (C_0 et C_1) en Trampoline telles que $\tau_0 << \tau_1 << \tau_2 << \tau_3$ ($<<$ signifie « moins prioritaire que ») (cf. Fig. 5.7). Nous supposons également que les tâches sont partitionnées de telle manière à ce

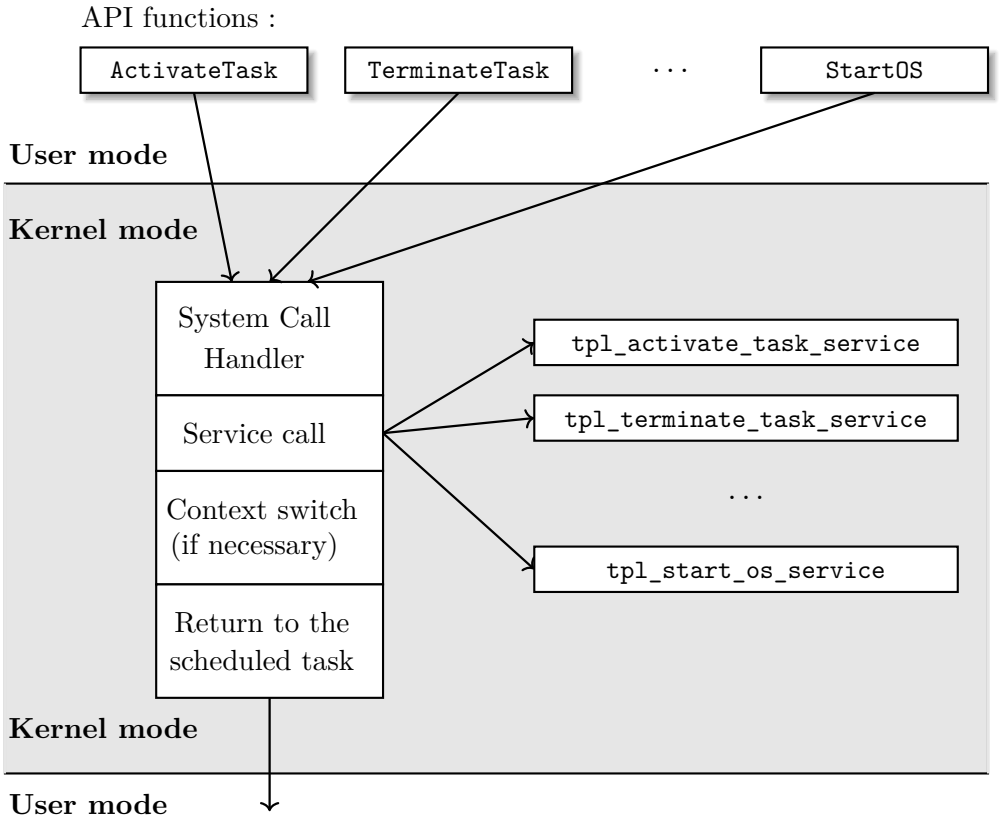


FIGURE 5.6: Processus d'un appel de service.

que τ_3 s'exécute sur C_0, tandis que τ_0 , τ_1 et τ_2 s'exécutent sur C_1. Initialement, τ_3 et τ_0 s'exécutent respectivement sur C_0 et C_1. Si la tâche τ_3 fait appel au service de l'API `ActivateTask` pour activer la tâche τ_1 (qui doit s'exécuter sur C_1), C_0 doit acquérir le verrou noyau puisque le service d'activation l'exige. Ainsi, le ré-ordonnancement pour C_1 est effectué et en résulte une notification de changement de contexte qui lui est adressée avec une interruption inter-cœur. Celle-ci est différée étant donné que le verrou est acquis par C_0. Si pendant ce temps la tâche τ_2 est activée sur C_1, son activation n'est traitée qu'à la fin de la libération du verrou par C_0. Une fois que cela est fait, C_1 récupère le verrou et un ré-ordonnancement local y est effectué conduisant à un changement de contexte de τ_0 vers τ_2 en fonction des résultat fournis par l'ordonnanceur [4]. L'interruption envoyée par C_0 est ensuite prise en compte, le changement de contexte, ayant déjà lieu, aucun autre n'est effectué. Ainsi, la tâche τ_1 se met en exécution après la libération du verrou.

[4]La structure `tpl_kern`, illustrant dans cet exemple les résultat d'ordonnancement, est présentée en § 5.4.3

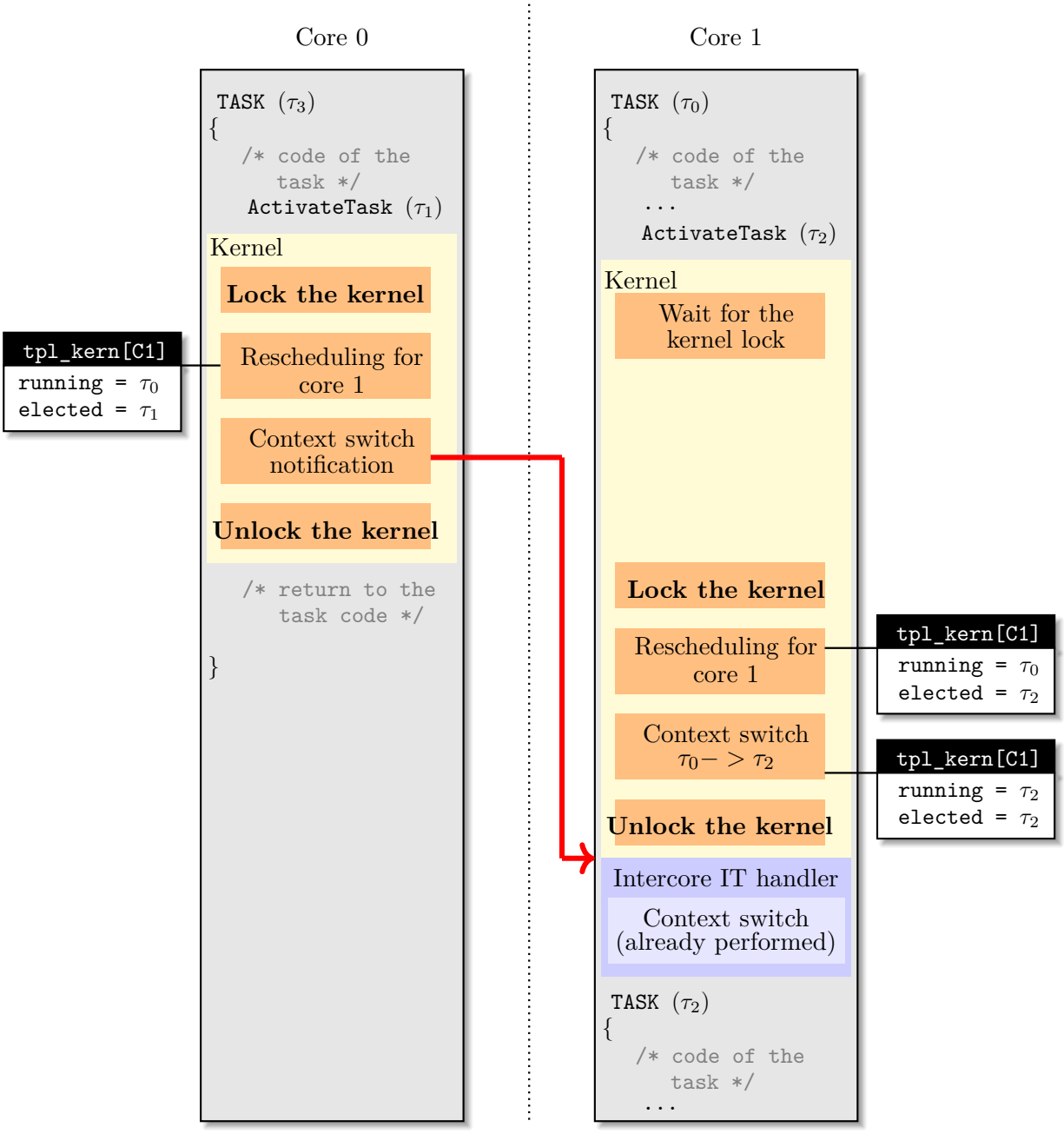


FIGURE 5.7: Processus d'acquisition du verrou lors d'un appel de service.

5.4 L'ordonnancement au sein de Trampoline

Les objets ordonnançables dans Trampoline sont les tâches et les ISRs de catégorie 2. Quand une ISR2 est détectée (activée), tout comme une tâche, elle produit un travail prêt et son état devient `READY` pour être prise en charge par l'ordonnanceur. En revanche,

une ISR2 ne peut pas avoir plus d'un travail non terminé à un instant donné. Son cycle de vie est similaire à celui d'une tâche basique. Dans la suite de ce manuscrit, nous traitons uniquement l'ordonnancement des tâches en se focalisant seulement sur le cas des tâches basiques.

L'ordonnanceur de Trampoline s'occupe de l'attribution des travaux prêts aux cœurs disponibles pour leur exécution. Étant conforme au standard OSEK/VDX, une gestion dynamique des priorités n'est pas prise en charge et la décision d'ordonnancement en Trampoline est calculée selon une politique à priorité fixe pour les tâches. Ainsi, la priorité de chacune des tâches est définie statiquement dans la description OIL et enregistrée dans son descripteur statique. La valeur 0 correspond à la priorité la plus basse, elle est attribuée à la tâche *idle* (cf. § 5.3.4). Sur la base de ces priorités, l'ordonnanceur décide lesquelles des tâches disposant d'un travail prêt doivent passer de l'état `READY` à l'état `RUNNING` et vice-versa.

5.4.1 Ordonnancement monocœur

En monocœur, l'ordonnanceur de Trampoline gère une unique liste de travaux prêts rangés et classés par ordre de priorité. Cette liste ne contient que les travaux des tâches dont l'état est `READY`. Les travaux ayant le même niveau de priorité sont classés selon leur ordre d'activation. Cette liste est implémentée sous forme d'un tableau de files d'attente FIFO. Chaque élément du tableau correspond à un niveau de priorité et donne accès aux travaux prêts de ce niveau dans l'ordre de leur activation tel que présenté dans la figure 5.8. Il est à noter qu'un travail préempté, étant forcément le plus ancien parmi les autres travaux, est toujours remis en tête de la file correspondant à sa priorité. De la même manière, un travail nouvellement activé, étant forcément le plus récent, est toujours mis en queue de la file.

Le tableau implémentant cette liste indexée par la priorité peut conduire à une occupation importante de mémoire dans le cas où certaines priorités ne sont pas attribuées par l'utilisateur. Pour remédier à ce problème, les priorités données par l'utilisateur sont recalculées et compactées de manière à ce qu'elles soient contiguës dans le tableau.

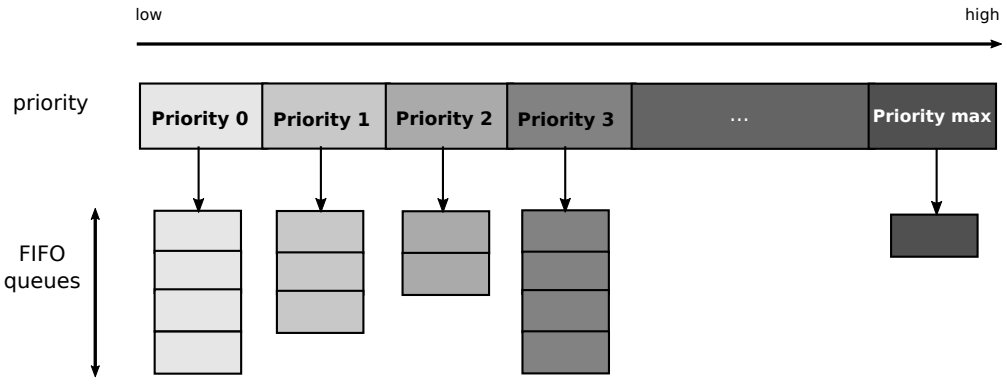


FIGURE 5.8: Structure de la liste des travaux prêts

5.4.2 Ordonnancement multicœur

La version multicœur de Trampoline implémente une politique d'ordonnancement de type partitionné selon des priorités fixes pour les tâches. Chacune des tâches est allouée statiquement à un cœur pour son exécution et chacun des cœurs dispose de sa propre liste de travaux prêts. L'implémentation de cette liste dans la version multicœur de Trampoline est réalisée avec un autre type de structure de données qui ne nécessite pas d'avoir des priorités dans un ordre contigu. Il s'agit du tas binaire (*heap*) dont la clé correspond à la priorité de la tâche. Plus de détail sur la mise en oeuvre d'une telle structure est fourni en § 6.4.

Considération de l'ordre d'activation : une opération de calcul d'une priorité dynamique pour les tâches est utilisée en multicœur pour la prise en charge de l'ordre des activations. Elle consiste concaténer la priorité de la tâche donnée par l'utilisateur, avec son ordre d'activation. Ainsi, chaque tâche a une priorité dynamique contenant deux parties : (i) une partie de base pour la priorité statique, elle est codée sur un nombre de bits dépendant du nombre de niveaux de priorité dans le système ; (ii) un rang d'activation qui est codé sur un nombre de bits dépendant de la valeur maximale des nombres maximum d'activations des tâches.

5.4.3 Fonctionnement de l'ordonnanceur

La structure `tpl_kern` : les opérations d'ordonnancement en Trampoline ont toujours lieu sur le cœur où se produit l'événement déclencheur du ré-ordonnancement. Toutefois, un ré-ordonnancement peut conduire à une préemption et un changement de contexte concernant un autre cœur. Afin de mieux séparer les cœurs gérés par l'ordonnanceur de celui sur lequel il s'exécute, une structure appelée `tpl_kern` est implémentée. Il s'agit d'une structure de données qui contient toutes les informations relatives à un cœur pendant l'exécution des tâches. Ainsi, elle est dupliquée en multicœur selon le nombre des cœurs de la plateforme. Les informations stockées pour chaque cœur sont détaillées ci-dessous :

- `running_id` : l'identifiant de la tâche en cours d'exécution ;
- `s_running` : un pointeur vers le descripteur statique de la tâche en cours d'exécution
- `running` : un pointeur vers le descripteur dynamique de la tâche en cours d'exécution ;
- `elected_id` : l'identifiant de la tâche élue pour le cœur ;
- `s_elected` : un pointeur vers le descripteur statique de la tâche élue pour le cœur ;
- `elected` : un pointeur vers le descripteur dynamique de la tâche élue pour le cœur ;
- `need_schedule` : un booléen indiquant si un ordonnancement doit avoir lieu ;
- `need_switch` : un booléen indiquant si un changement de contexte entre la tâche élue et la tâche en cours d'exécution doit avoir lieu.

Lorsqu'une tâche est en cours d'exécution sur un cœur, les attributs `running_id` et `elected_id` dans la structure `tpl_kern` de ce cœur contiennent la même valeur qui correspond à l'identifiant de la tâche exécutée. Lorsqu'un ré-ordonnancement a lieu, les

opérations d'ordonnancement ne considèrent et éventuellement ne modifient que l'attribut `elected_id` en cas de préemption. À l'issue d'un ré-ordonnancement, si les attributs `elected_id` et `running_id` sont différents, un changement de contexte se fait entre la tâche qui est en cours d'exécution et celle qui est élue. Une copie des attributs `elected` de la structure `tpl_kern` vers les attributs `running` doit également être réalisée pour effectuer la même procédure pour les prochains ré-ordonnancements.

Notion de tâche élue : quand l'ordonnanceur décide de préempter une tâche sur un cœur pour y affecter une autre, un changement de contexte doit être effectué. Cette opération consiste à sauvegarder le contexte de la tâche qui perd le CPU afin de restaurer celui de la tâche désignée par l'ordonnanceur (cf. § 5.3.4). Ainsi, l'opération de changement de contexte peut s'avérer coûteuse en temps processeur, plus particulièrement quand il y plusieurs ré-ordonnancements à effectuer consécutivement « au même instant » et qui peuvent engendrer plusieurs changements de contexte successifs concernant le même cœur. Si nous reprenons l'exemple de la figure 5.7, deux ré-ordonnancements qui concernent le cœur 1 sont effectués l'un après l'autre. Le premier conduit à préempter τ_0 et élire τ_1. Le deuxième conduit à élire τ_2 à la place de τ_1 (qui est moins prioritaire). De sorte à éviter ce genre de changements de contexte inutiles (de τ_0 vers τ_1 ensuite de τ_1 vers τ_2), Trampoline, dans sa version multicœur, remédie à ce problème par l'introduction d'un pseudo-état de tâche appelé élue (*elected*). Ce pseudo-état caractérise une tâche qui est élue par l'ordonnanceur pour être exécutée sur l'un des cœurs mais qui n'y est pas encore en cours d'exécution étant donné que le système est en mode noyau, pendant lequel toute exécution de tâche est bloquée.

Ainsi, une distinction est faite entre une tâche élue et une tâche en cours d'exécution. Si plusieurs ré-ordonnancements sont à faire pendant que le mode noyau n'est pas encore quitté, les tests de priorités sont effectués sur la tâche élue. Ceci garantit que la tâche en cours d'exécution ne soit pas affectée par les ré-ordonnancements multiples. À l'issue de ces ré-ordonnancements, un seul changement de contexte est effectué de la tâche qui était en cours d'exécution avant l'appel de l'ordonnanceur vers la dernière tâche élue, et les attributs `elected` de `tpl_kern` sont copiés dans les attributs `running` par la fonction `tpl_run_elected()` qui sera présentée dans le paragraphe suivant (cf. Fig.5.9).

Les fonctions d'ordonnancement : l'ordonnanceur peut consulter et/ou modifier les listes des travaux prêts au moyen de trois fonctions principales :

- `tpl_put_new_proc()` : permet d'ajouter à la liste des travaux prêts un travail d'une tâche qui vient de quitter l'état `SUSPENDED` (ie. nouvellement activé). En monocœur, le travail est placé à la fin de la FIFO correspondant à sa priorité. En multicœur, le travail est inséré dans le tas en fonction de sa priorité calculée (ie. priorité statique et rang d'activation).
- `tpl_put_preempted_proc()` : permet d'ajouter à la liste un travail qui vient d'être préempté. En monocœur, s'agissant forcément du travail le plus ancien parmi les autres travaux prêts, un travail préempté est placé en tête de la FIFO correspondant

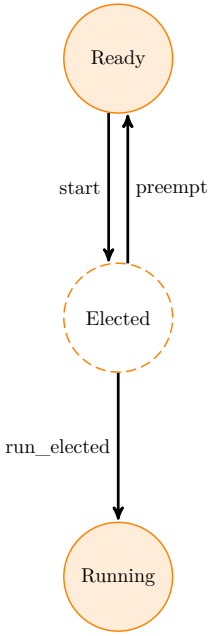


FIGURE 5.9: Le pseudo-état `elected` d'une tâche pendant l'ordonnancement.

à sa priorité. En multicœur, le travail est inséré dans le tas en fonction de sa priorité calculée.

- `tpl_remove_front_proc()` : retire le travail le plus prioritaire de la liste des travaux prêts et le retourne à la fonction appelante.

L'ordonnanceur permet également de manipuler les tâches devant être exécutées et/ou préemptées au moyen des fonctions exposées ci-dessous. Il s'agit des fonctions correspondant aux transitions d'états dans la figure 5.9 :

- `tpl_start()` : correspond à la transition `start` dans les graphes d'états. Dans cette fonction, le travail en tête de la liste des travaux prêts est récupéré et élu par l'ordonnanceur. Pour cela, ses informations sont copiées dans les attributs `elected` de la structure `tpl_kern` du cœur concerné.
- `tpl_preempt()` : correspond à la transition `preempt` dans les graphes d'états. Elle permet de faire passer une tâche préemptée par l'ordonnanceur de l'état `RUNNING` à l'état `READY`.
- `tpl_run_elected()` : cette fonction est appelée une fois que tous les réordonnancements consécutifs sont effectués. Elle permet de faire passer la tâche élue par l'ordonnanceur de l'état `READY` vers l'état `RUNNING` juste avant son exécution, et de copier tous les attributs `elected` en `running` du cœur concerné.

Les fonctions présentées ci-dessus sont appelées par la fonction principale de l'ordonnanceur `tpl_schedule_from_running()` qui s'occupe de choisir les tâches à exécuter ou à préempter selon leurs priorités.

5.4.4 Les interactions de l'ordonnanceur au sein de Trampoline

Pour les tâches basiques, deux types d'événement peuvent conduire à un ré-ordonnancement en Trampoline et donc à l'appel de l'ordonnanceur :

- l'activation d'un nouveau travail d'une tâche ;
- la terminaison du travail en cours d'exécution d'une tâche.

Dans ce qui suit, nous présentons un synoptique exposant la logique et la chronologie des actions qui mènent vers un appel de l'ordonnanceur si l'un des événements susmentionnés survient. Cette analyse s'appuie principalement sur l'architecture de Trampoline présentée dans la figure 5.3 et précise les interactions de l'ordonnanceur avec les autres composants du système d'exploitation. L'objectif principal de cette analyse est de bien comprendre les interactions menant vers l'ordonnanceur afin de pouvoir l'adapter pour supporter les politiques d'ordonnancement global. Les fonctions de gestion des travaux prêts étant déjà présentées, elles ne sont pas discutées à ce niveau. Nous nous concentrons sur la démarche d'appel de l'ordonnanceur au sein de Trampoline. Les noms des fonctions et macros permettant ces interactions sont conservés scrupuleusement tels qu'ils apparaissent dans le code source de Trampoline.

5.4.5 Appel de l'ordonnanceur dans le cas d'une activation de tâche

L'activation d'un nouveau travail de tâche en Trampoline peut être effectuée selon deux mécanismes : soit par une expiration d'alarme, ou bien par un appel système.

Activation par alarme : une alarme permet de traiter les événements récurrents tels qu'une activation périodique d'une tâche à l'aide d'un compteur (cf. § 5.1.5). Quand la valeur du compteur de *tick* correspond à la date d'expiration de l'alarme qui lui est associée, la source liée au compteur envoie une interruption. Cette interruption est interceptée par le gestionnaire des interruptions externes qui en informe le dispatcheur d'interruptions. Celui-ci invoque le gestionnaire du compteur afin de mettre à jour ses attributs et déclencher l'action associée à l'alarme. Dans ce cas, l'action de l'alarme correspond à l'appel du service OSEK `ActivateTask`. Une fois l'exécution du service terminée, le dispatcheur des interruptions invoque l'ordonnanceur avec la fonction `tpl_schedule_from_running()` en monocœur ou la fonction `tpl_multi_schedule()` en multi-cœur (cf. Fig. 5.10).

Activation par un appel système : une tâche en Trampoline peut également être activée par une autre tâche. Ceci correspond à un appel direct du service OSEK `ActivateTask` au sein du code de la tâche appelante en passant l'identifiant de la tâche à activer en paramètre. Ceci correspond à l'appel d'un service de l'API qui génère une interruption logicielle. Le dispatcheur des interruptions l'intercepte et en informe le gestionnaire des appels système. Celui-ci effectue les opérations nécessaires à l'activation d'une tâche et invoque l'ordonnanceur en utilisant la fonction `tpl_schedule_from_running()` que ce soit en monocœur ou en multicœur (cf. Fig. 5.11).

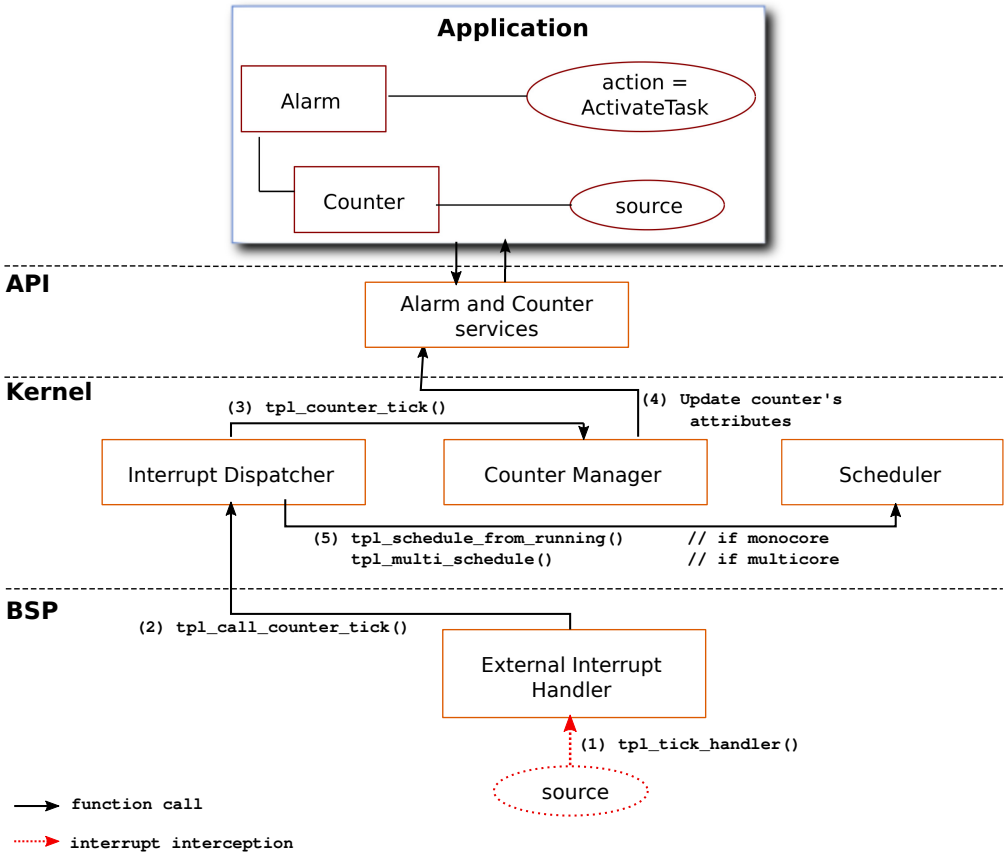


FIGURE 5.10: Procédure d'appel de l'ordonnanceur en cas d'activation par alarme

5.4.6 Appel de l'ordonnanceur dans le cas d'une terminaison de tâche

Le processus de terminaison d'une tâche suit la même logique que celle de l'activation d'une tâche par un appel système. Ceci découle du fait qu'une tâche ne peut se terminer que par elle-même en appelant le service OSEK `TerminateTask` dans son programme. Cet appel se traduit par un appel système qui passe à travers le dispatcheur des interruptions et le gestionnaire des appels système avant d'appeler l'ordonnanceur avec la fonction `tpl_terminate()`. La figure 5.11 illustre le processus de l'appel de l'ordonnanceur suite à un appel de service OSEK (`ActivateTask` ou `TerminateTask`).

5.5 Conclusion

Ce chapitre a été consacré au système d'exploitation temps réel Trampoline. Nous avons d'abord présenté les deux standards sur lesquels il se base. Ensuite son architecture et le fonctionnement de son noyau ont été exposés avec un focus sur l'ordonnanceur. Cette étude de l'OS nous permet de bien identifier les fonctions, les structures de données et

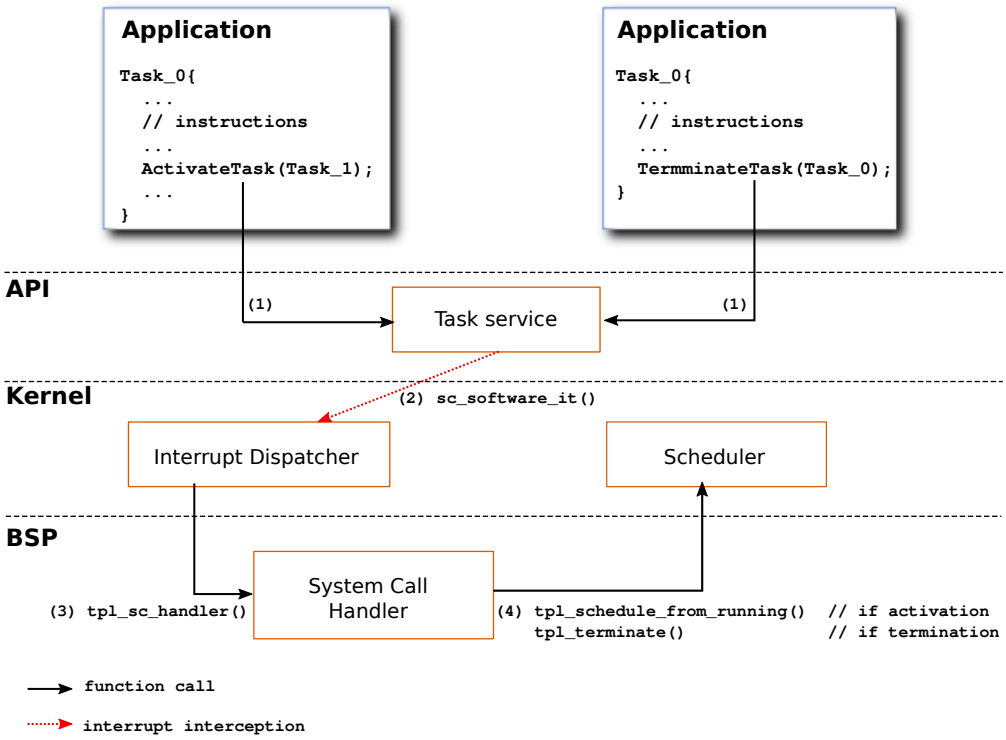


FIGURE 5.11: Procédure d'appel de l'ordonnanceur en cas d'une activation/terminaison par un appel système

les variables de l'OS liées à l'ordonnancement ainsi le fonctionnement de l'ordonnanceur au sein de l'OS. Ceci indispensable afin de pouvoir intégrer une nouvelle politique d'ordonnancement au sein de Trampoline, ce qui fait l'objet du prochain chapitre.

CHAPITRE 6

Implémentation de Global EDF

6.1 Présentation des besoins

La politique G-EDF fait apparaître lors de son implémentation au sein de Trampoline quelques exigences. Pour ce qui relève de sa règle d'ordonnancement basée sur les dates d'échéance des travaux, deux points majeurs sont identifiés :

- le calcul, la représentation et la comparaison des dates d'échéance ;
- la gestion de la liste des travaux prêts tout en prenant en considération les activations multiples des travaux d'une même tâche.

Pour ce qui concerne la mise en œuvre d'un ordonnanceur global, d'autres contraintes doivent être prises en considération :

- la gestion de l'affectation dynamique des travaux et de leur migration entre les cœurs ;
- la gestion du changement de contexte.

Dans le reste de ce manuscrit, seulement les tâches basiques et indépendantes sont manipulées. Ainsi, les exigences de l'ordonnancement en matière de protocoles de gestion des ressources et des événements ne sont pas considérées dans cette implémentation. De ce fait, deux événements donnent lieu à un ré-ordonnancement : (i) activation d'un nouveau travail d'une tâche ; (ii) terminaison d'un travail en cours d'exécution.

6.2 L'adaptation de Trampoline pour l'ordonnancement global

La mise en œuvre d'un ordonnanceur global à priorité fixe pour les travaux au sein d'un système d'exploitation statique et qui supporte l'ordonnancement partitionné doit

passer par une étape d'adaptation du noyau. En outre, l'implémentation initiale de l'ordonnanceur en Trampoline est de type « *kernel-based* ». Autrement dit, il n'y a pas de séparation entre le code du noyau de l'OS et le code de l'ordonnanceur. Pour cela, des modifications majeures ont été mises en place au sein du code de l'OS. Dans la présente section sont exposés les changements initiaux qui concernent les descripteurs des tâches et les structures de données. La mise en œuvre des fonctions relevant de l'ordonnanceur global est discutée dans les sections qui suivent.

6.2.1 Modification des descripteurs des tâches

Le descripteur de tâche est scindé en deux parties : statique et dynamique. Le descripteur statique regroupe les attributs de la tâche qui ne sont pas susceptibles d'évoluer pendant son exécution. En revanche, le descripteur dynamique stocke les attributs variables de la tâche. Ces descripteurs sont déclarés sous forme de structures dans le code de Trampoline (cf. § 5.3.4). Ils sont générés par le compilateur GOIL en se basant sur leur définition fournie dans un template `task_descriptor.goilTemplate`. Ainsi, les modifications présentées ci-dessous sont effectuées dans ce fichier :

- déplacement de l'identifiant du cœur d'exécution (`core_id`) : il s'agit d'un attribut qui indique le cœur sur lequel la tâche doit s'exécuter. Dans l'implémentation initiale de Trampoline, cet attribut est stocké dans le descripteur statique. En passant vers l'ordonnancement global, une tâche est susceptible de migrer entre les cœurs de la plateforme et donc l'attribut `core_id` doit pouvoir varier. Ainsi, il est retiré du descripteur statique et déplacé vers le descripteur dynamique dans notre implémentation.
- suppression des attributs de la priorité : rappelons qu'une tâche en Trampoline possède deux types de priorité (cf. § 5.4.2) : (i) une priorité statique de base fournie par l'utilisateur qui est stockée dans le descripteur statique ; (ii) une priorité dynamique dépendant du rang d'activation du travail de la tâche en cours qui est stockée dans le descripteur dynamique. Pour la politique EDF, ces deux attributs ne sont plus nécessaires. Ainsi, ils sont enlevés des descripteurs.
- ajout de l'échéance relative et l'échéance absolue : un ordonnanceur se basant sur EDF nécessite d'avoir accès aux dates d'échéance des tâches. Pour cela, un attribut renseignant l'échéance relative est ajouté au descripteur statique, et un autre représentant l'échéance absolue est ajouté au descripteur dynamique (cf. *listing* 6.1). Nous rappelons qu'une seule instance des descripteurs existe. Ainsi, l'échéance absolue stockée dans le descripteur dynamique d'une tâche caractérise son plus ancien travail qui n'est pas encore terminé.

6.2.2 Modification du booléen `need_schedule`

En partitionné, quand l'ordonnanceur est appelé, le ré-ordonnancement ne concerne qu'un seul cœur à la fois. C'est pour cela que chaque cœur dispose de l'attribut `need_schedule` dans sa structure `tpl_kern`, qui permet de notifier la nécessité de

```
// Absolute deadline generation in the dynamic descriptor:

%if exists task::ABSOLUTE_DEADLINE then%
  /* absolute deadline        */  %!task::ABSOLUTE_DEADLINE %,
%else%
  /* idle absolute deadline   */  0,
%end if%

// Relative deadline generation in the static descriptor:

%if exists task::DEADLINE then%
  /* relative deadline            */  %!task::DEADLINE %,
%else %
  /* idle deadline is 0       */  0,
%end if%
```

Listing 6.1: Exemple de génération des descripteurs d'une tâche

ré-ordonnancement lui concernant. En revanche, en ordonnancement global, un ré-ordonnancement concerne tous les cœurs de la plateforme, ce qui ne nécessite qu'un seul attribut l'indiquant globalement. Pour cela, un unique booléen `tpl_need_schedule` est commun à chacun des cœurs et se substitue à ceux des structures `tpl_kern`.

6.3 Mécanisme de la gestion de temps

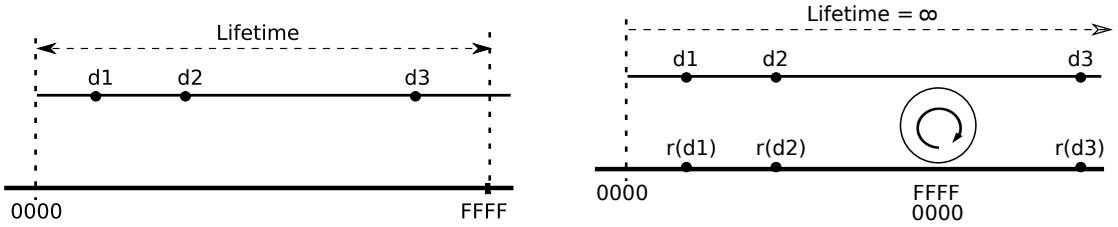
Avec G-EDF, il est nécessaire de disposer d'un mécanisme temporel permettant le calcul et la comparaison des dates d'échéance. Un tel mécanisme doit être capable de : (i) représenter et comparer les dates ; (ii) gérer les éventuels débordements dans la représentation des dates. Ce mécanisme doit gérer les dates en utilisant le minimum possible de mémoire et sans générer des « *overheads* » d'exécution importants.

6.3.1 Types de représentation du temps

De manière schématique, dans les systèmes embarqués, l'écoulement du temps est mesuré à l'aide d'un compteur matériel (*timer*) qui, associé à une horloge de référence, est configuré de manière à générer périodiquement des *ticks*. L'intervalle de temps entre deux *ticks* successifs définit la « *résolution* » temporelle du système. A chaque occurrence d'un *tick*, une variable logicielle de comptage du temps est incrémentée. Si cette variable est codée sur n bits, elle peut compter de 0 à $2^n - 1$. Ainsi, si la résolution du système est de r ms et que la variable de temps indique une valeur de k, ceci correspond à une date $r \times k$ ms. Un tel mécanisme peut être exploité selon 2 modes pour disposer d'une représentation du temps : linéaire ou circulaire. La figure 6.1 illustre un exemple des deux modèles pour un système utilisant une variable temporelle codée sur 16 bits.

La représentation linéaire : dans ce modèle, la variable de temps évolue de manière linéaire de 0 jusqu'à $2^n - 1$ (cf. Fig. 6.1(a)). La durée de vie des systèmes adoptant cette représentation est limitée et égale à son horizon ($horizon = resolution \times (2^n - 1)$). Par

exemple, pour les systèmes d'exploitation Unix, le temps est représenté en utilisant un entier signé de 32 bits et une résolution d'une microseconde. La mesure de son écoulement a commencé le 1er janvier 1970 et finira le 19 janvier 2038. Lorsque cette date sera franchie, cette représentation débordera et il ne sera plus possible de représenter l'heure Unix correctement avec le modèle linéaire. Ce phénomène s'appelle le « bug de l'an 2038 » [Tim08] et touche principalement les machines utilisant le noyau Linux en 32 bits. Ainsi, une solution doit être prévue pour gérer ce débordement d'horloge.



(a) Modèle de temps linéaire

(b) Modèle de temps circulaire

FIGURE 6.1: Types de représentations du temps.
d_1, d_2 et d_3 sont trois dates réelles successives

La représentation circulaire : contrairement à la représentation linéaire du temps, le modèle circulaire permet d'éviter « *l'apocalypse* » des systèmes d'exploitation en offrant une durée de représentation infinie. En effet, la variable de temps évolue en cycles de 0 jusqu'à $2^n - 1$ puis repasse à 0 afin d'éviter le débordement, ce qui fait que la durée de vie du système n'est plus limitée (cf. Fig. 6.1(b)). Dans une telle représentation, chaque date possède une référence temporelle qui correspond à sa position dans le cycle de temps et pas à sa valeur réelle. La seule contrainte que représente ce modèle, c'est que deux dates différentes peuvent partager la même référence temporelle si elles sont espacées d'une durée qui est égale à exactement un cycle. Dans ce qui suit, la référence sur le cycle d'une date d_i est notée $r(d_i)$.

6.3.2 Algorithme ICTOH pour la comparaison des dates d'échéance

Bien que la représentation circulaire des dates remédie au problème de la « fin des temps », elle présente des limites quand il s'agit de comparer deux dates survenues sur deux cycles différents. Afin de mieux illustrer ce problème, considérons l'exemple de la figure 6.2 représentant trois travaux de tâches τ_1, τ_2 et τ_3 ayant respectivement d_1, d_2 et d_3 pour date d'échéance. Les travaux sont ordonnancés selon EDF avec $d_1 < d_2 < d_3$ ce qui signifie que $\tau_1 >> \tau_2 >> \tau_3$. Cependant, dès que le travail de τ_3 se réveille, il s'exécute prioritairement en préemptant les deux autres travaux. Les échéances d_i des trois travaux sont représentées par les références $r(d_i)$. Leur représentation conduit à un ordre de priorité qui n'est pas celui des valeurs réelles ($r(d_3) < r(d_1) < r(d_2)$). En effet, l'échéance du travail de τ_3 est représentée sur un cycle différent de celui des deux autres

travaux avec une référence dont la valeur est petite que celles des deux autres travaux. Il en résulte ainsi que la comparaison des dates ne peut être correcte ce qui peut conduire au non-respect des échéances réelles.

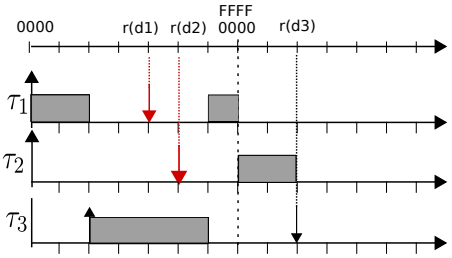


FIGURE 6.2: Exemple de comparaison erronée des dates d'échéances

Afin de prendre en compte cette situation, Carlini et Buttazzo ont proposé un algorithme permettant d'effectuer la comparaison des dates représentées suivant le modèle circulaire, tout en gérant le cas des dates appartenant à des cycles différents. Il s'agit de l'algorithme ICTOH (*Circular Timer's Overflow Handler*) [CB03]. Il a l'avantage d'introduire peu d'*overhead* par rapport à d'autres techniques de comparaison [Fon01].

Le principe consiste à représenter les dates sur un cycle temporel en adoptant la représentation circulaire. Ainsi, chaque cycle a une longueur de $P = 2^n$ pour des dates codées sur n bits. Cette longueur désigne la période du cycle temporel qui représente l'intervalle de temps minimum entre deux dates non simultanées caractérisées par la même représentation temporelle.

La comparaison de deux dates d_i et d_j est effectuée selon le principe suivant :

1. représentation de d_i et d_j respectivement par les références $r(d_i)$ et $r(d_j)$. Ces références sont codées en entiers non signés sur n bits (*unsigned integer*).
2. évaluation de la différence entre $r(d_i)$ et $r(d_j)$ en entiers non signés.
3. comparaison du résultat avec la moitié de la période du cycle ($\frac{P}{2}$) :
   (a) si $unsigned(r(d_i) - r(d_j)) > \frac{P}{2}$, alors d_i précède d_j.
   (b) si $unsigned(r(d_i) - r(d_j)) < \frac{P}{2}$, alors d_j précède d_i.
   (c) si $unsigned(r(d_i) - r(d_j)) = 0$, alors d_i et d_j sont simultanées.

La différence entre $r(d_i)$ et $r(d_j)$, calculée en entiers non signés, représente également la distance évaluée entre les deux références en direction des valeurs croissantes. Il est à noter que le résultat de cette comparaison n'est valide que sous la condition suivante :

$$|d_i - d_j| < \frac{P}{2} \tag{6.1}$$

Autrement, le résultat de la comparaison n'est pas correct. En d'autres termes, la distance en valeur absolue entre les références des deux dates ne doit pas dépasser la moitié de la période du cycle (cf. Fig. 6.3).

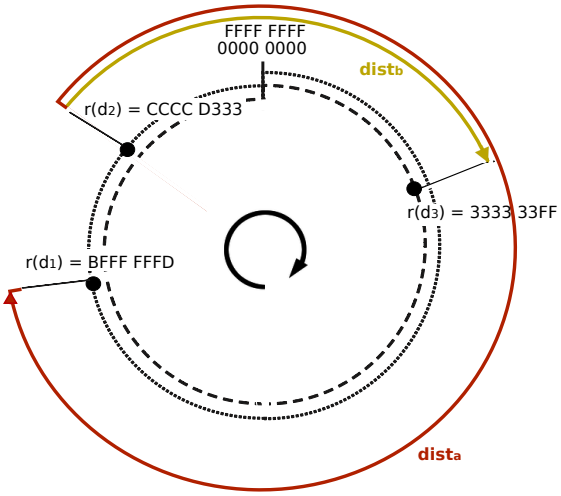


FIGURE 6.3: Évaluation et comparaison des dates selon l'algorithme ICTOH.
d_1, d_2 et d_3 sont représentées circulairement en utilisant une variable de 32 bits. Chaque cycle a une période $P = 2^{32}$. La comparaison de d_1, d_2 et d_3 donne : $dist_a = unsigned(r(d_1) - r(d_2)) = F3332CCC > 80000000 = P/2$ et $dist_b = unsigned(r(d_3) - r(d_2)) = 666660CC < P/2$. Ainsi, d_1 est avant d_2 et celle-ci est avant d_3.

6.3.3 Gestion des dates en Trampoline

En Trampoline, l'horloge système est implémentée grâce à un compteur programmable appelé PIT (*Programmable Interval Timer*) qui envoie périodiquement une interruption au CPU pour générer un tick d'horloge. La période d'envoi de cette interruption est d'une microseconde, ce qui correspond à la résolution temporelle. Pour l'implémentation de G-EDF, nous manipulons une variable de comptage de temps de 32 bits, qui enregistre la date courante selon la représentation circulaire. Ce qui donne un horizon d'une heure environ. Elle est incrémentée à chaque tick d'horloge. La date d'échéance d_i d'un travail est calculée au moment de son activation en additionnant son échéance relative D_i à la date courante t : $d_i = D_i + t$. Elle est ensuite enregistrée en utilisant la représentation circulaire avec une variable temporelle non signée codée sur 32 bits. Ainsi, chaque cycle a une période de 1 heure et 11 minutes.

La condition 6.1 du fonctionnement correct de l'algorithme ICTOH relative à notre implémentation de G-EDF, impose donc que les dates d'échéance comparables ne soient pas éloignées de plus de de $\frac{P}{2}$ (35 minutes dans notre implémentation). Nous ne jugeons pas cette contrainte limitative au regard des échelles temporelles des applications temps réel visées.

6.4 Gestion des tâches prêtes

6.4.1 Besoins

La structure de données requise dans notre implémentation de G-EDF est destinée principalement au stockage des travaux prêts selon leur priorité. Cette priorité dépend de leurs dates d'échéance. D'un autre côté, Trampoline autorise les activations successives des tâches jusqu'à une certaine limite définie statiquement (cf. § 5.3.4). Cela signifie que l'activation d'un nouveau travail de tâche est possible même si celle-ci en possède un autre qui n'est pas encore terminé. Toutefois, deux travaux de la même tâche ne doivent pas être exécutés en même temps sur deux cœurs. Aussi, la priorité d'exécution doit être attribuée au travail le plus ancien d'une tâche. La liste des travaux prêts doit également prendre en considération les contraintes suivantes :

- la liste doit être triée dans un ordre croissant des échéances absolues. Ainsi, quand un travail est activé ou préempté, il doit être inséré au bon endroit ;
- si une tâche a plusieurs travaux prêts en même temps, la liste doit les organiser de manière à respecter leur ancienneté.
- si plusieurs travaux de différentes tâches partagent la même date d'échéance, ils doivent être triés selon leur ancienneté ;
- la liste doit être partagée entre tous les cœurs tel que cela est requis pour l'ordonnancement global.

6.4.2 Choix des structures de données

Parmi les différents travaux prêts dont une tâche peut faire état, nous faisons distinctions entre deux types de travaux prêts que nous qualifions de :

- *un travail actif* : celui qui désigne le travail le plus ancien de la tâche qui n'est pas encore terminé. Il n'y en a au plus qu'un par tâche.
- *un travail en attente* : correspond à un travail qui n'est pas actif. Il peut y en avoir plusieurs par tâche.

Ainsi, afin d'éviter l'exécution de deux travaux d'une tâche en même temps, nous choisissons de séparer le stockage des travaux actifs et celui des travaux en attente. Pour ce faire, différentes listes de travaux prêts sont considérées dans notre implémentation :

- une liste servant à stocker les travaux actifs, que nous appelons `ReadyList`. Cette liste doit être triée dans un ordre croissant des dates d'échéance des travaux qu'elle contient. L'ordonnanceur ne doit sélectionner que des travaux actifs pour les exécuter. Pour ce faire, il ne doit consulter que cette liste. Concernant les travaux ayant la même date d'échéance, ils doivent être gérés en fonction de leur ancienneté.
- des listes dans lesquelles sont stockés les travaux en attente de chaque tâche, que nous appelons `PendingJobList`. Ainsi, une `PendingJobList` par tâche est considérée. Ces listes doivent être triées en fonction de l'ancienneté des travaux qu'elles contiennent.

Le choix des structures de données pour l'implémentation de ces deux types de liste dépend essentiellement de la complexité des opérations. De manière schématique, trois opérations sont fréquemment effectuées en manipulant une liste de travaux prêts : (i) **l'insertion (*enqueue*) :** d'un travail nouvellement activé ou un travail qui vient d'être préempté ; (ii) **la recherche (*research*) :** d'un travail en utilisant sa position dans la liste ; (iii) **l'extraction (*dequeue*) :** d'un travail selon sa position dans la liste afin de l'exécuter.

Une étude empirique a été conduite par Douglas W. Jones [Jon86] dans le but de comparer la complexité des opérations pour plusieurs structures de données, notamment pour les opérations d'insertion et d'extraction. Les expérimentations ont montré que le tas binaire *binary heap* est l'une des structures de données les plus simples et efficaces pour des ordonnanceurs se basant sur une priorité fixe ou dynamique pour les travaux. Les propriétés de cette structure de données en terme de complexité calculatoire sont : $O(1)$ pour rechercher la tâche la plus prioritaire et $O(\log N)$ pour les opérations d'insertion et d'extraction, avec N le nombre de nœuds dans le tas. Concernant une liste qui se base sur une priorité fixe pour le stockage des travaux, une FIFO demeure un choix idéal étant donné que toutes les opérations peuvent se faire en temps constant $O(1)$.

Par conséquent, nous choisissons le tas binaire (*min-heap*) pour l'implémentation de la `ReadyList` et un tableau de FIFO pour l'implémentation des `PendingJobLists`.

6.4.3 Le tas binaire

> **Définition 6.1.** *Une « min heap » est un arbre binaire dont chaque nœud enregistre une* clé. *Chaque nœud possède au plus deux fils au niveau inférieur, communément appelés gauche et droit. Le nœud dont ils sont issus au niveau supérieur est appelé parent.*
>
> *Pour chaque nœud A sauf la racine, si p est le parent de A alors :*
>
> $$cle(p) \leq cle(A)$$

Cette définition détermine la propriété d'ordre du min-tas binaire affirmant que la clé d'un nœud est toujours plus petite que celle de ses fils. Dans notre implémentation, les clés de l'arbre sont les dates d'échéance des travaux. Ainsi, elle renseigne la priorité de ces travaux. Afin de respecter la propriété de l'ordre, les opérations sur un tas se font comme suit :

- **insertion d'un nouveau nœud :** se fait en ajoutant ce nœud à la position située la plus à gauche et à la plus grande profondeur du tas. Ensuite, tant que sa clé est plus petite que celle de son père, il échange de place avec son père. Cette opération est répétée à chaque niveau jusqu'à ce que la racine soit atteinte ou que le père soit d'une échéance plus petite. En Trampoline, ce processus de ré-organisation du tas est assuré par la fonction de `tpl_bubble_up()`.
- **extraction du nœud racine :** se fait en enlevant le nœud de la racine et en positionnant à sa place le nœud situé le plus à droite et à la plus grande profondeur du

tas. Ensuite, tant que sa clé est plus grande que celle de son fils le plus prioritaire, ils échangent de place. Ce processus est répété à chaque niveau jusqu'à ce que le père soit plus prioritaire que ses fils ou qu'il n'ait plus de fils. En Trampoline, ce processus est assuré par la fonction de `tpl_bubble_down()`.

- **extraction d'un nœud arbitraire :** se fait en positionnant à sa place le nœud situé le plus à droite à la plus grande profondeur. Ensuite, le processus décrit ci-dessus est effectué pour la ré-organisation de l'arbre. En Trampoline, la fonction `tpl_get_proc()` assure ce traitement.
- **recherche du nœud le plus prioritaire :** cette opération se fait en temps constant car elle renvoie simplement le nœud contenu dans la racine du tas. En Trampoline, la fonction `tpl_get_front_proc()` assure ce traitement.

6.4.4 Implémentation de la liste des tâches prêtes

L'implémentation de la liste des travaux actifs : pour l'implémentation de la `ReadyList`, nous utilisons un tableau de telle sorte que le premier élément contienne la racine du tas, les deux éléments qui le suivent contiennent ses deux fils, les quatre suivants contiennent leurs fils, ainsi de suite. En d'autres termes, pour chaque élément à la position i du tableau, ses fils gauche et droit sont situés respectivement aux positions $2i$ et $2i + 1$. Sauf pour l'élément 0 du tableau, nous l'utilisons pour stocker la taille du tas qui correspond au nombre d'éléments stockés dans le tableau.

La clé dans le tas correspond à une date d'échéance, cette structure impose que deux nœuds aient des clés différentes. Ainsi, deux travaux actifs ayant la même date d'échéance doivent être associés au même nœud. Afin d'implémenter cela dans le tableau, une liste chaînée est utilisée dans chacun des éléments du tableau pour lier les travaux ayant la même date d'échéance. Les opérations d'insertion et d'extraction dans cette liste se font dans l'ordre d'activation des travaux. Pour cela, chaque élément du tableau est une structure de données contenant trois éléments (cf. *listing* 6.2) : (i) `key` : correspond à l'échéance absolue des travaux stockés dans le nœud ; (ii) `ptr_front` : correspond à l'identifiant du travail se trouvant en tête de la liste chaînée ; (iii) `ptr_end` : correspond à l'identifiant du travail se trouvant à la fin de la liste chaînée. Les opérations d'insertion / extraction dans la `ReadyList` sont discutées en § 6.5.4.

```
typedef struct{
        VAR(uint32, TYPEDEF)                key;
        VAR(tpl_proc, TYPEDEF)              ptr_front;
        VAR(tpl_proc, TYPEDEF)              ptr_end;
}tpl_heap_entry;
```

Listing 6.2: La structure de données du tas binaire
`tpl_heap_entry` correspond à la déclaration du type de la structure. `tpl_proc` est un type défini en Trampoline pour désigner l'identifiant d'un processus.

Les listes chaînées des travaux ayant les mêmes dates d'échéance sont toutes implémentées sous forme d'un unique tableau `tpl_all_nodes` ayant le nombre de tâches comme taille maximale et indexé par leurs identifiants (cf. *listing* 6.3). Chaque élément de ce tableau contient soit l'identifiant du travail suivant qui lui est lié ; soit -1 s'il n'y a aucun travail suivant. Étant donné que chaque nœud du tas contient l'identifiant du premier et dernier élément de la liste qui lui est liée, l'insertion et l'extraction dans ce tableau se fait toujours en temps constant O(1). Un exemple de cette implémentation est présenté dans la figure 6.4.

```
VAR(tpl_proc_id, OS_VAR) tpl_all_nodes[TASK_COUNT] = {[0 ... TASK_COUNT - 1]
    = -1};
```

Listing 6.3: Déclaration et initialisation en Trampoline du tableau regroupant les listes chaînées des travaux ayant la même date d'échéance.

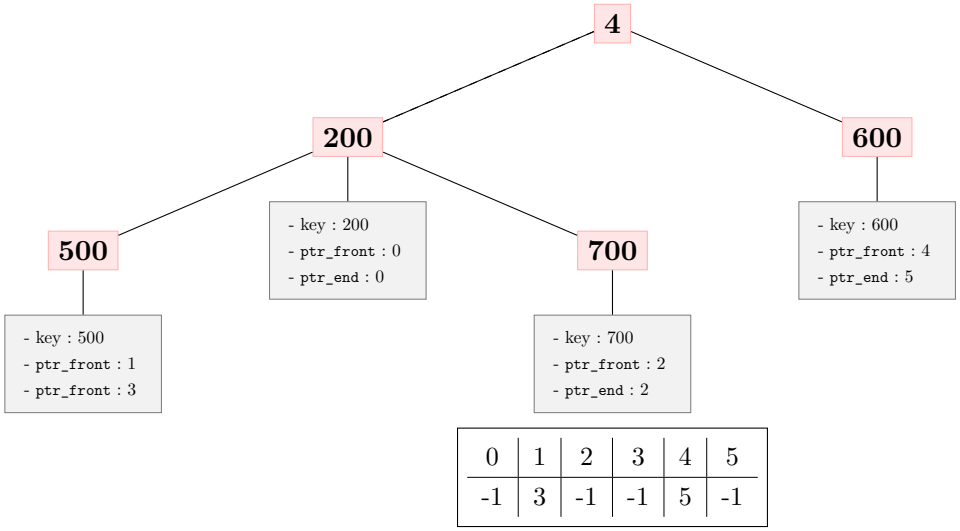


FIGURE 6.4: Implémentation de la `ReadyList`
Le tas stocke 6 travaux τ_i / $i \in \{0, 1, .., 5\}$ et a une taille de 4 qui correspond au nombre de dates d'échéance présentes dans l'arbre. Une seule tâche (τ_0) est liée à l'échéance 200. τ_1 et τ_3 sont liées l'une après l'autre à l'échéance 500. τ_2 a une échéance de 700. τ_4 et τ_5 sont liées l'une après l'autre à l'échéance 600. Le tableau regroupe les chaînes listes chaînées réunissant les travaux ayant la même date d'échéance. Un indice du tableau i correspond à l'identifiant d'une tâche τ_i, l'élément associé à cet indice correspond soit à l'identifiant du travail suivant dans la liste chaînée, soit à -1 s'il n'y pas de travail suivant.

L'implémentation de la liste des travaux en attente : concernant les `PendingJobLists`, chaque tâche en possède une qui regroupe ses travaux en attente. Elle est implémentée sous forme d'un tableau et gérée en FIFO en fonction de l'ancienneté

des travaux. Ce tableau a une taille maximale égale au nombre maximum d'activations de la tâche - 1 (cf. *listing* 6.4). Nous rappelons qu'il existe une seule instance des descripteurs des tâches, une tâche ayant plusieurs travaux activés au même instant ne peut stocker qu'une seule date d'échéance dans son descripteur qui correspond à celle de son travail actif. Les dates d'échéances des travaux en attente sont enregistrées dans les `PendingJobLists`.

```
VAR(uint32, OS_VAR) tpl_pending_job_list_task0[task0.max_activate_count - 1]=
    {[0 ... task0.max_activate_count - 2] = -1};
```

Listing 6.4: Déclaration et initialisation en Trampoline de la `PendingJobList` d'une tâche `task0`.

6.5 Ordonnancement

6.5.1 Architecture générale de l'implémentation de G-EDF

En Trampoline, plusieurs composants et fonctions de bas niveau du noyau sont impliqués dans le calcul de la décision d'ordonnancement. Dans notre implémentation, nous avons délimité ce que nous appelons « le périmètre de l'ordonnancement » c'est à dire les composants pré-existants de Trampoline et d'autres qui sont nouvellement intégrés devant coopérer afin de calculer la décision d'ordonnancement selon G-EDF. La figure 6.5 illustre l'architecture de l'implémentation de G-EDF. Dans ce qui suit, nous fournissons une description de la contribution de chacun de ces composants dans la décision d'ordonnancement. Nous rappelons que notre implémentation ne traite que des tâches basiques et indépendantes.

6.5.2 Le gestionnaire des tâches (*Task Manager*)

Dans notre implémentation, ce composant regroupe les fonctions bas niveau nécessaires à l'activation et la terminaison d'un nouveau travail de tâche. Il se charge également de positionner à vrai le booléen `tpl_need_schedule` (cf. §6.2.2) afin d'appeler l'ordonnanceur. C'est un composant pré-existant de l'OS. Ainsi, son fonctionnement a été adapté à la politique G-EDF au moyen de deux fonctions principales :

- `tpl_edf_activate_task()` : cette fonction est appelée soit à la suite d'un appel système issu du service OSEK `ActivateTask`, soit après une expiration d'alarme conduisant à l'activation d'un nouveau travail de tâche. Au début, la date d'échéance du travail nouvellement activé est calculée en se servant des fonctions du composant « gestionnaire de temps » (cf. § 6.5.3). Si le compteur d'activation de la tâche est supérieur à zéro, ceci signifie qu'elle possède déjà un travail actif, le nouveau travail est enregistré avec son échéance dans la `PendingJobList`. Sinon, il est mis dans la `ReadylList` et l'état de la tâche devient `READY_AND_NEW`. Ce processus est résumé dans le pseudo

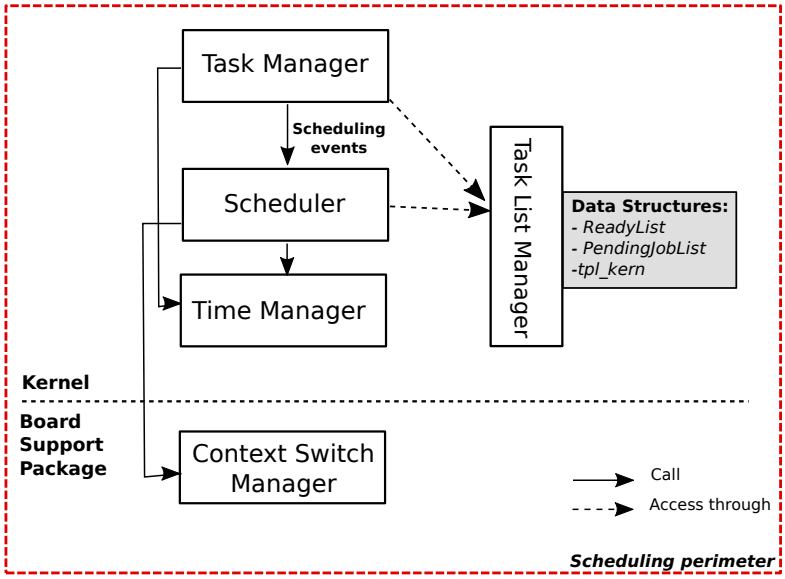


FIGURE 6.5: Les composants impliqués dans la mise en œuvre de G-EDF en Trampoline

algorithme fourni dans la figure 6.6. Les opérations d'insertion du travail dans une liste se font en passant par le composant « gestionnaire des listes de tâches » (cf. § 6.5.4).

- `tpl_edf_terminate_task()` : cette fonction est appelée à la suite d'un appel système issu du service OSEK `TerminateTask`. Elle commence par vérifier la `PendingJobList` de la tâche dont le travail vient de se terminer. Si un travail en attente y existe, il est retiré et inséré dans la `ReadyList` en fonction de sa date d'échéance et l'état de la tâche devient `READY`. Sinon, son état devient `SUSPENDED`. Ce processus est résumé dans le pseudo algorithme fourni dans la figure 6.7. Les opérations d'extraction et d'insertion du travail se font en passant par le gestionnaire des listes de tâches.

L'activation et la terminaison sont des services qui requièrent un ré-ordonnancement. Ainsi, le booléen `tpl_need_schedule` est mis à vrai dans les deux fonctions présentées ci-dessus pour appeler le composant « ordonnanceur » (cf. § 6.5.5).

6.5.3 Le gestionnaire de temps (*Time Manager*)

Ce composant est ajouté à l'OS afin de pouvoir gérer le calcul de la date courante suite à la demande du gestionnaire des tâches avec la fonction `tpl_get_current_date()`. Cette fonction récupère la valeur du compteur de ticks de l'horloge système et retourne la date courante en microsecondes selon une gestion circulaire (cf. § 6.3.3). Le gestionnaire de temps est également appelé par l'ordonnanceur afin d'effectuer la comparaison des dates d'échéance selon l'algorithme ICTOH (cf. § 6.3.2) en utilisant la fonction `tpl_compare_deadline()`.

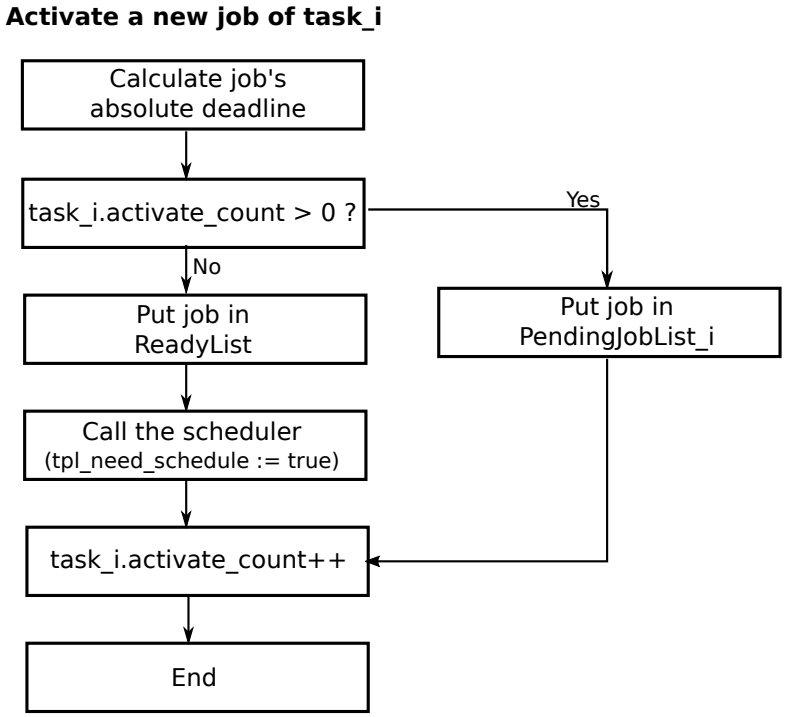


FIGURE 6.6: Pseudo algorithme d'activation d'un travail d'une tâche τ_i dans l'implémentation de G-EDF.

6.5.4 Le gestionnaire des listes de tâches (*Task List Manager*)

Gestion des opérations dans les listes de tâches : ce composant s'occupe de l'insertion/extraction des travaux de la `ReadyList` et des `PendingJobLists` sur la demande du gestionnaire des tâches et de l'ordonnanceur. Pour ce faire, il met en œuvre diverses fonctions dont les principales sont :

- `tpl_put_new_proc()` : permet d'insérer un travail nouvellement activé dans la `ReadyList` en fonction de sa date d'échéance. Si elle contient déjà une clé correspondant à la date d'échéance de ce travail, celui-ci est lié à la fin de la liste chaînée associée à la clé en question.
- `tpl_put_preempted_proc()` : permet d'insérer un travail qui vient d'être préempté dans la `ReadyList`. Si elle contient déjà une clé correspondant à la date d'échéance de ce travail, celui-ci, étant le plus ancien, est inséré en tête de la liste chaînée associée à la clé en question.
- `tpl_remove_front_proc()` : permet de retirer le travail qui se trouve en tête de la `ReadyList`.
- `tpl_put_pending_job()` : permet d'insérer un travail nouvellement activé dans la `PendingJobList` de sa tâche en enregistrant sa date d'échéance.
- `tpl_remove_pending_job()` : permet de retirer un travail en tête de la `PendingJobList` d'une tâche et d'en récupérer la date d'échéance.

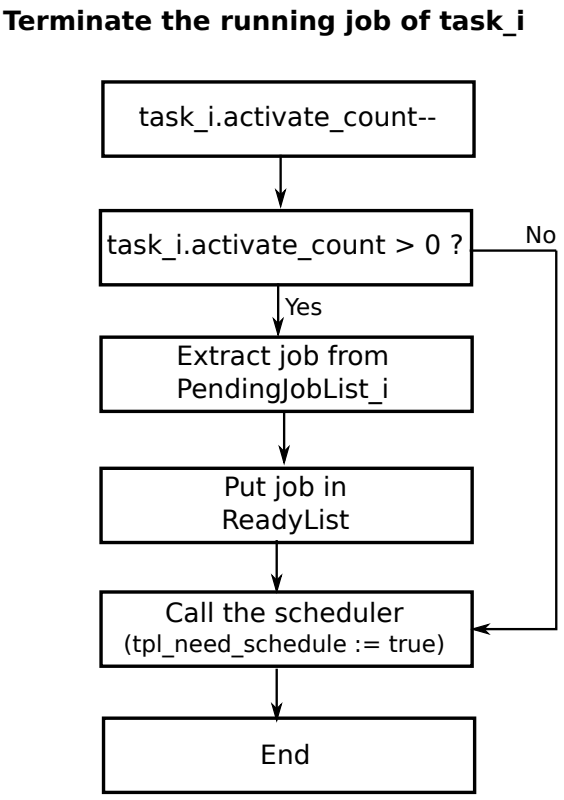


FIGURE 6.7: Pseudo algorithme de terminaison du travail en cours d'exécution d'une tâche τ_i dans l'implémentation de G-EDF.

Le gestionnaire des listes de tâches assure le tri de `ReadyList` selon un ordre croissant des échéances absolues au moyen de deux fonctions qui sont appelées à chaque opération d'insertion et/ou d'extraction : `tpl_bubble_up()` et `tpl_bubble_down()` (cf. § 6.4.3).

Gestion du conflit des dates d'échéance : dans le théorie de l'ordonnancement temps réel, il a été noté [GFB02] que la méthode utilisée pour résoudre les conflits entre les travaux ayant la même date d'échéance peut influer sur l'ordonnançabilité du système de tâches considéré. Dans ce cadre, des études permettant de résoudre ce problème ont été proposées, notamment en analysant l'impact de la vitesse des processeurs de la plateforme d'exécution des travaux [FG02].

Dans notre travail, notre objectif n'est pas d'étudier l'ordonnançabilité des systèmes de tâches considérés. Ainsi, lorsque deux travaux ont la même date d'échéance, nous faisons le choix de donner la priorité au travail le plus ancien. Ainsi, lorsqu'un travail est préempté, il est considéré prioritaire par rapport aux autres travaux prêts ayant la même date d'échéance que lui, étant nécessairement le plus ancien. Lorsque deux travaux sont activés au même instant et ont la même date d'échéance, le travail inséré en premier dans la `ReadyList`, en raison de l'exécution séquentielle des instructions dans un système

d'exploitation, est considéré le plus ancien et donc le plus prioritaire. C'est pour cette raison que chaque liste chaînée associée à un nœud de la `ReadyList` est triée dans l'ordre d'activation des travaux qu'elle contient. Ceci est géré dans la `ReadyList` au moyen des fonctions `tpl_put_new_proc()`, `tpl_put_preempted_proc()` et `tpl_remove_front_proc()` présentées ci-dessus. Ainsi, un travail nouvellement activé est inséré à la fin de la liste chaînée associée à sa date d'échéance tandis qu'un travail préempté est inséré en tête de cette liste. Quant à l'extraction pour l'exécution, elle se fait toujours de la tête de la liste chaînée associée à un nœud, ce qui correspond à sélectionner le travail le plus ancien ayant la date d'échéance associé à ce nœud. Ceci permet ainsi de favoriser l'exécution des travaux déjà préemptés lorsqu'ils ont la même date d'échéance que d'autres travaux nouvellement activés.

6.5.5 L'ordonnanceur (*Scheduler*)

Dans notre implémentation, ce composant de Trampoline a été adapté afin d'effectuer l'ordonnancement selon G-EDF au moyen de trois fonctions `tpl_schedule()`, `tpl_start()` et `tpl_preempt()`. `tpl_schedule()` est la fonction d'appel de l'ordonnanceur, elle est invoquée lorsque le booléen `tpl_need_schedule` est mis à vrai par le gestionnaire des tâches. Le rôle de l'ordonnanceur consiste à sélectionner les travaux les plus prioritaires en consultant la `ReadyList` et en accédant aux résultats de comparaison des échéances fournies par le gestionnaire du temps. Étant donné que la `ReadyList` est triée par le gestionnaire des listes, l'ordonnanceur ne doit sélectionner que les m travaux, au plus, en tête de la liste pour les exécuter sur m cœurs libres au plus. Pour ce faire, deux tests sont exécutés de manière itérative (jusqu'à épuisement des processeurs libres ou des travaux de la `ReadyList`) dans la fonction d'ordonnancement (cf. Fig. 6.8) :

1. *Est-ce qu'il y a un cœur libre alors que la* `ReadyList` *n'est pas vide ?*
   Dans ce cas, le travail en tête de la `ReadyList` est retiré et est élu pour s'exécuter sur le cœur. Ainsi, l'attribut `elected_id` de la structure `tpl_kern` du cœur en question prend pour valeur l'identifiant du travail élu. Cette opération est assurée par la fonction `tpl_start()` interne à l'ordonnanceur. L'attribut `need_switch` de la structure est également mis à jour pour indiquer la nécessité d'un changement de contexte. Ce processus est répété tant qu'il y a des cœurs libres et des travaux prêts dans la liste.
2. *Est-ce que le travail en tête de la* `ReadyList` *a une priorité plus élevée qu'un travail en cours d'exécution ?*
   Dans ce cas, le travail en cours d'exécution est préempté, remis dans la `ReadyList` et son état devient `READY`, ceci est réalisé par la fonction interne à l'ordonnanceur `tpl_preempt()`. Ensuite, la fonction `tpl_start()` est appelée : le travail en tête de la `ReadyList` est retiré pour l'exécution et son identifiant est enregistré dans l'attribut `elected_id` de la structure `tpl_kern` du cœur sur lequel il est élu. L'attribut `need_switch` de la structure est également mis à jour pour indiquer la nécessité d'un changement de contexte.

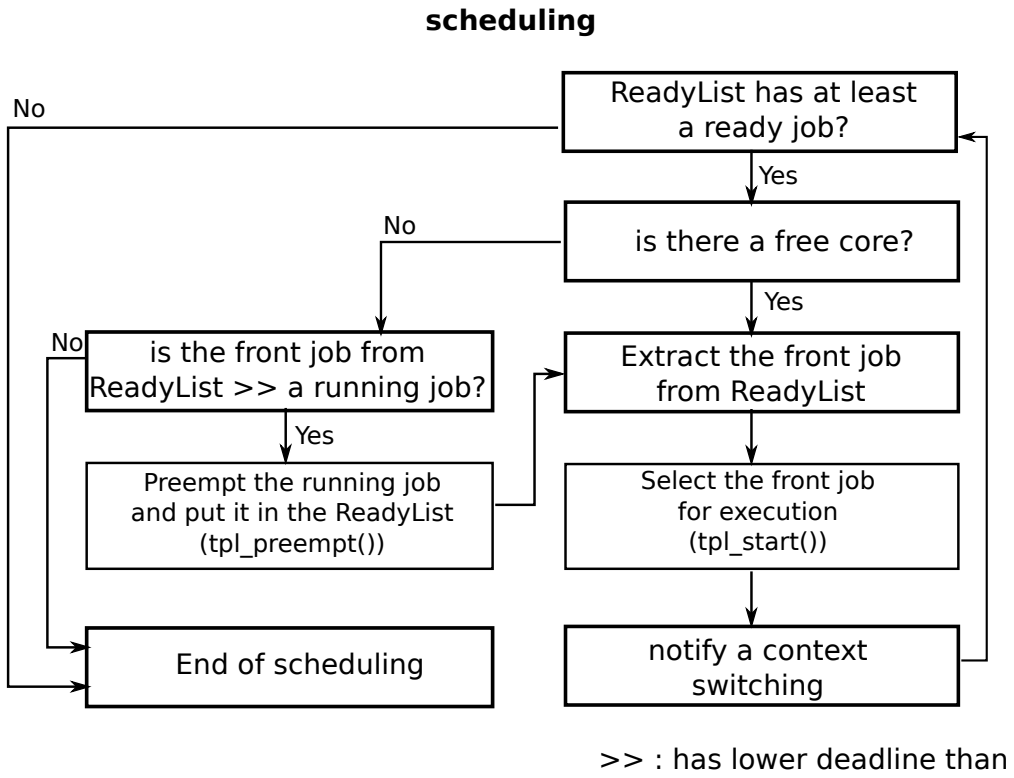


FIGURE 6.8: Pseudo algorithme de l'ordonnancement dans l'implémentation de G-EDF.

6.5.6 Le gestionnaire de changement de contexte (*Context Switch Manager*)

Ce composant est appelé quand l'ordonnanceur indique, par le biais du `need_switch`, qu'il est nécessaire d'effectuer un changement de contexte. Ce champ peut prendre trois valeurs possibles :

- `NO_NEED_SWITCH` : dans le cas où il n'y a aucun changement de contexte à opérer.
- `NEED_SWITCH` : ce qui signifie qu'il est nécessaire d'effectuer un changement de contexte sans avoir à sauvegarder celui du travail qui perd le CPU. Cela peut se produire lorsqu'une tâche oisive [1] était en cours d'exécution avant le ré-ordonnancement, ou dans le cas d'une terminaison où il n'est pas nécessaire de sauvegarder le contexte du travail qui se termine.
- `NEED_SWITCH | NEED_SAVE` : ce qui indique la nécessité de sauvegarder le contexte du travail qui perd le CPU et de restaurer celui du travail élu.

A l'issue du changement de contexte, les champs `elected_id` doivent être copiés dans `running_id` de la structure `tpl_kern` doivent ainsi avoir la même valeur. Ainsi, à

[1] La tâche oisive (*Idle task*) est une tâche sans action autre que d'occuper le cœur pendant le temps creux laissé par le système quand il n'y a aucune autre tâche qui s'exécute sur le CPU.

partir de ce point, les travaux élus commencent effectivement à s'exécuter sur les cœurs d'exécution.

6.6 Conclusion

Ce chapitre avait pour objectif la discussion de notre travail de mise en œuvre de la politique G-EDF au sein de Trampoline. Ainsi, nous avons présenté les étapes d'adaptation de l'OS afin qu'il puisse supporter l'ordonnancement global. Ensuite nous avons détaillé les choix faits dans le cadre de l'implémentation : la gestion des dates d'échéance, des travaux prêts et des fonctions d'ordonnancement, ainsi que la définition du périmètre d'ordonnancement qui rassemble les composants de l'OS contribuant à la décision d'ordonnancement. La mise en œuvre de G-EDF réalisée est intégrée au code source du noyau de Trampoline. Actuellement, elle n'est portée sur aucune des cibles de l'OS. Une éventuelle continuité de ce travail de mise en œuvre serait de mener cette dernière sur l'une des cibles embarquées sur laquelle Trampoline peut être déployé : ARM7, AVR (Atmel), ARM Cortex-M, ARM Cortex-A, PowerPC, POSIX, etc.

Le travail d'implémentation présenté dans ce chapitre n'est pas une tâche facile du fait que le code des composants contribuant à la décision d'ordonnancement est réparti sur différentes parties du noyau. Ainsi, un travail futur d'implémentation d'ordonnanceur au sein de Trampoline nécessite une étude exhaustive du fonctionnement de l'ordonnancement au sein de l'OS et du code de celui-ci. Ainsi, une seconde perspective serait d'étudier la possibilité de séparer le code du noyau et celui de l'ordonnanceur à travers une couche intermédiaire permettant de gérer les échanges entre eux (notification d'événements d'ordonnancement, décision d'ordonnancement, etc.). Ceci peut faciliter l'intégration de nouvelles politiques d'ordonnancement en se limitant à l'étude du composant responsable de l'ordonnancement l'OS sans avoir besoin d'une compréhension complète du fonctionnement de l'OS.

Dans le présent travail, l'objectif de l'implémentation de G-EDF est de l'utiliser dans un premier temps pour l'instanciation de notre démarche de vérification par model-checking. Ainsi, la prochaine partie portera sur la construction du modèle qui spécifie l'implémentation de la politique G-EDF.

Troisième partie

Modélisation d'une implémentation d'ordonnanceur

Notre démarche de vérification des implémentations d'ordonnanceur est basée sur le model-checking. Telle que nous l'avons présenté dans l'introduction, une telle démarche consiste à mener la vérification sur un modèle décrivant le fonctionnement de l'implémentation à vérifier. Une première phase de cette démarche est d'élaborer les modèles des implémentations d'ordonnanceur que nous souhaitons vérifier. Ainsi, un modèle de l'implémentation de G-EDF est construit à partir de son code source. Sur la base de celui-ci, un deuxième modèle modélisant une implémentation de la politique EDF-US[ξ] au sein de Trampoline est élaboré sans passer par un code source développé. Trampoline a déjà été modélisé et vérifié dans sa version d'ordonnancement partitionné [BRT18] avec des automates finis étendus et temporisés en utilisant UPPAAL. Ainsi, en partant de ce modèle, nous l'adaptons pour l'ordonnancement global et nous y intégrons les deux modèles d'implémentation en utilisant le même outil et le même formalisme.

À travers cette partie, nous discutons le formalisme et les règles utilisés pour une modélisation fidèle d'un système d'exploitation. Nous présentons ensuite le modèle préexistant de Trampoline et ses propriétés. Enfin nous exposons l'architecture du modèle de l'implémentation de G-EDF réalisée et celui de l'implémentation d'EDF-US[ξ] qui est menée directement sur UPPAAL.

CHAPITRE 7

Modélisation de Trampoline par automates finis étendus

7.1 Introduction aux automates finis étendus

7.1.1 Automates finis

Définition 7.1. *(Automate fini)*
Un automate fini A est défini par un quintuplet $A = (S, s_0, F, \Sigma, \rightarrow)$ tel que :

- *S est un ensemble fini d'états.*
- *$s_0 \in S$ est l'état initial (unique).*
- *$F \subseteq S$ est l'ensemble des états finaux (acceptants).*
- *Σ est un ensemble fini de symboles ou caractères (l'alphabet) correspondant à des actions réalisables.*
- *$\rightarrow \subseteq S \times \Sigma \times S$ est la relation de transition.*

De manière informelle, un automate est une machine à états pouvant prendre un nombre fini d'états. Il s'agit d'une machine abstraite qui reçoit en entrée un mot écrit sur un alphabet fini Σ et qui évolue d'état en état en fonction de la valeur des caractères composant ce mot. L'alphabet étant l'ensemble de caractères utilisés dans l'automate, et le mot désigne une suite finie de caractères de cet alphabet.

Ainsi, chaque automate a un langage qui est constitué de l'ensemble des mots qu'il reconnaît (ou qu'il accepte). La reconnaissance d'un mot est faite selon le processus suivant :

- initialisation de l'automate : son état initial est fixé à s_0. Un automate ne peut avoir qu'un seul état initial ;

- lecture des caractères du mot l'un après l'autre et évolution de l'automate : pour cela, la relation de transition $\rightarrow$ est employée pour le passage d'un état s à un autre état s'.
  Ainsi, si l'état courant de l'automate est s, que le caractère courant est σ et qu'il existe une transition $\rightarrow$ associée à σ, on parle alors du franchissement d'une transition de l'état s vers l'état s' : $s \xrightarrow{\sigma} s'$.
- acceptation du mot : le mot est reconnu par l'automate si et seulement si le dernier état atteint après la lecture du dernier caractère du mot est un état *acceptant* (i.e, l'état appartient à l'ensemble F). Notons qu'un automate peut avoir plusieurs états acceptants.

Exemple 7.1. La figure 7.1 illustre un exemple d'automate dont l'alphabet est $\Sigma = \{a, b, c\}$. Chaque état de cet automate est représenté par un cercle (un nœud). Dans cet exemple, l'état initial s_0 est repéré par une flèche entrante. Il présente un unique état final s_2 repéré par un double cercle dans la figure. Une transition d'un état à un autre est représentée par un arc étiqueté par un caractère de l'alphabet Σ.

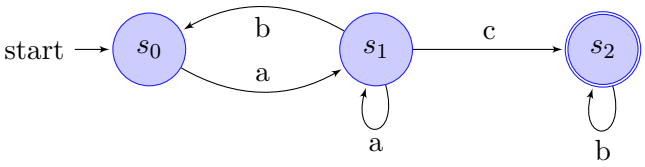


FIGURE 7.1: Exemple d'un automate : *aac* est un mot reconnu par cet automate. En revanche, *aba* est un mot qui n'est pas reconnu étant donné que la lecture de son dernier caractère *a* s'arrête à l'état s_1 qui ne correspond pas un état final.

Les actions associées aux transitions dans un automate fini ne peuvent pas être des variables. Ainsi, la modélisation d'un système manipulant des variables ne peut pas être réalisée avec de tels automates. Une extension avec des variables existe. Il s'agit des *automates finis étendus* (cf. § 7.1.2).

7.1.2 Automates finis étendus

Définition 7.2. *(Automate fini étendu)*
Un automate fini étendu A est un tuple $A = (S, s_0, F, X, \mathcal{U}(X), \mathcal{G}(X), \pi_0, \Sigma, \rightarrow)$, tel que :

- *S est un ensemble fini d'états.*
- *$s_0 \in S$ est l'état initial.*
- *$F \subseteq S$ est l'ensemble des états finaux.*
- *X est un ensemble fini de variables prenant un ensemble fini de valeurs entières bornées.*

- $\mathcal{U}(X)$ *est une liste de mises à jour sur* X *tel que :* $\pi(x) \in \mathcal{U}(X)$ *est une mise à jour de* $x \in X$*.*
- $\mathcal{G}(X)$ *est un ensemble de contraintes linéaires sur* X *tel que :* $g(x) \in \mathcal{G}(X)$ *note une contrainte linéaire sur* $x \in X$*.*
- $\pi_0 \subseteq \mathcal{U}(X)$ *est une liste des affectations initiales sur* X*.*
- Σ *est un ensemble fini d'actions réalisables.*
- $\rightarrow \subseteq S \times \Sigma \times \mathcal{G}(X) \times \mathcal{U}(X) \times S$ *est un ensemble fini de transitions.*

Dans un modèle à automates finis étendus, chaque transition peut être complétée par :

- des conditions sur les variables (ie. les contraintes linéaires $g(x)$). Dans le reste de ce manuscrit, ces conditions sont appelées des *gardes*. Ainsi, une transition qui contient une garde n'est franchissable que si sa garde est satisfaite.
- des mises à jour d'une ou plusieurs variables qui sont effectuées dès le franchissement de la transition à laquelle elles sont associées.

Soit $t = \langle s, \sigma, g, \pi, s' \rangle \in \rightarrow$, une transition notée $s \xrightarrow{\sigma,g,\pi} s'$. Cette notation représente une transition de l'état s à l'état s' du système contenant l'action σ, contrainte par la garde g et permettant d'effectuer la mise à jour π. Notons que si la garde est absente, la relation de transition s'écrit $s \xrightarrow{\sigma,\pi} s'$ et la transition t est supposée toujours franchissable. Si la mise à jour π est absente la relation de transition s'écrit $s \xrightarrow{\sigma,g} s'$ et aucune variable n'est mise à jour quand la transition t est franchie.

Exemple 7.2. La figure 7.2 illustre un exemple d'un automate fini étendu, contenant deux états s et s' et manipulant une variable x. La transition a_0 entre les deux états est contrainte par la garde $(x \leq 1)$. Son franchissement est accompagné de la mise à jour de la valeur de x avec $x := 3$.

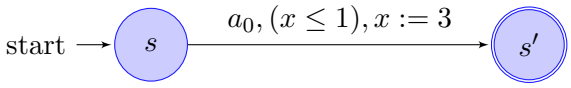


FIGURE 7.2: Exemple d'un automate fini étendu

7.1.3 Automates temporisés

Un automate fini étendu peut être étendu avec des variables d'horloge et inclure ainsi l'écoulement de temps dans ses états. Dans ce cas, il s'agit des automates temporisés.

Définition 7.3. *(Automate temporisé)*
Un temporisé A *est un tuple* $A = (S, s_0, F, X, \mathcal{U}(X), \mathcal{G}(X), \mathcal{C}(X), I, \Sigma, \rightarrow)$*, tel que :*

- S *est un ensemble fini d'états.*
- $s_0 \in S$ *est l'état initial.*

- $F \subseteq S$ *est l'ensemble des états finaux.*
- X *est un ensemble fini de variables à valeurs réelles positives ou nulles (horloges).*
- $\mathcal{U}(X)$ *est l'ensemble des mises à jour sur* X*.*
- $\mathcal{G}(X)$ *est l'ensemble des contraintes linéaires sur* X *associées aux transitions (gardes).*
- $\mathcal{C}(X)$ *est l'ensemble des contraintes linéaires sur* X *associées aux états (invariants).*
- $I : S \to \mathcal{C}(X)$ *est une fonction associant un invariant à un état.*
- Σ *est un ensemble fini d'actions réalisables.*
- $\to \subseteq S \times \Sigma \times \mathcal{G}(X) \times \mathcal{U}(X) \times S$ *est un ensemble fini de transitions.*

La théorie des automates temporisés a été développée par Alur et Dill [AD90]. Il s'agit d'automates étendus avec des variables d'horloge continues prenant des valeurs réelles supérieures ou égales à zéro. Ces variables permettent de modéliser l'écoulement de temps dans un état et de contrôler les instants où les transitions de l'automate peuvent être franchies. Ainsi, des contraintes sur les valeurs des variables d'horloge peuvent être exprimées dans les transitions sous forme de garde. Lorsqu'une transition est franchie, les horloges peuvent être remises à zéro afin de mesurer le temps écoulé à partir de l'instant de son franchissement. Ceci permet de mesurer, ainsi que de contrôler, le temps écoulé entre des actions associées à plusieurs transitions. Le franchissement d'une transition est instantané, tandis que dans les états, le temps peut s'écouler, c'est à dire que les valeurs des variables d'horloge peuvent évoluer. Un état de l'automate peut également aussi se voir associer des contraintes sur les variables d'horloge. Ces contraintes sont appelées des *invariants* et elles permettent de contrôler la durée pendant laquelle le système peut rester dans cet état selon deux règles : (i) le système **peut rester** dans un état si l'invariant qui lui est associé est satisfait ; (ii) le système **doit quitter** l'état dès que l'invariant qui lui est associé n'est plus satisfait

Exemple 7.3. À titre illustratif, considérons l'exemple de la figure 7.3 d'un automate temporisé manipulant une variable d'horloge notée x qui est initialisée à zéro au démarrage. Le fonctionnement de cet automate débute à son état initial s_0, un invariant sur la variable d'horloge lui est associé. Cet invariant indique que le système peut rester dans cet état tant que x est inférieur ou égal à 2 unités de temps. Il peut également le quitter avant l'écoulement de cette durée pour franchir la transition a_0 et passer à l'état s_1. Si les deux unités de temps se sont écoulées alors que le système est toujours dans s_0, il doit immédiatement le quitter. À s_1, un invariant est associé indiquant que le système ne peut être dans cet état que si la valeur de l'horloge est inférieure ou égale à 3. Il peut également le quitter en empruntant la transition a_1 et remettant x à zéro dès que cet invariant n'est plus satisfait.

Dans la suite de ce document, nous nous focalisons sur les automates finis étendus avec et/ou sans variable d'horloge. Ce sont ces automates que nous utilisons dans notre travail de modélisation.

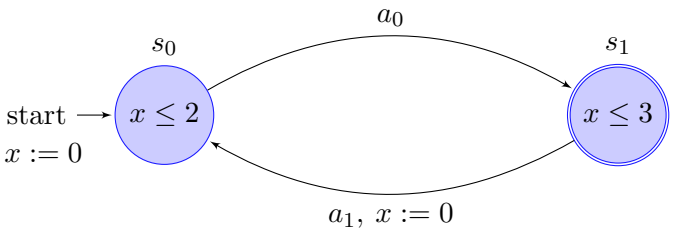


FIGURE 7.3: Exemple d'un automate temporisé.

7.1.4 Réseaux d'automates finis étendus

Un système complexe ne peut pas être simplement abstrait par un seul automate. Schématiquement, la modélisation d'un tel système se fait en modélisant chacun de ses composants ou fonctions par un automate A_i. L'ensemble des automates est mis en parallèle et synchronisé entre eux avec une fonction de synchronisation (cf. définition 7.4). Ainsi, l'ensemble synchronisé correspond à un automate global du système $\mathcal{A}$ appelé *réseau d'automates finis étendus.*

Définition 7.4. *(Fonction de synchronisation d'automates)*
Soit $A_1, ..., A_n$, n automates finis étendus tel que chaque automate A_i est défini par le tuple : $A_i = (S_i, s_{0i}, F_i, X, \mathcal{U}_i(X), \mathcal{G}_i(X), \pi_{0i}, \Sigma, \rightarrow_i)$, tel que tous les A_i partagent le même ensemble des variables X.

*Une fonction de synchronisation f est une fonction partielle définie par $f : (\Sigma \cup \{\bullet\})^n \rightarrow \Sigma$ impliquant un ou plusieurs automates. Lorsque plusieurs automates A_i sont impliqués dans une synchronisation, ils franchissent chacun une transition parallèlement et effectue ainsi une action $\sigma \in \Sigma$ de manière synchrone dans un **rendez-vous**. On note $f(\sigma_1, \sigma_2, ...) = \sigma$ tel que :*

- *Σ désigne l'ensemble des actions σ_i réalisables par les automates.*
- *« $\bullet$ » est un symbole qui désigne l'action d'un automate qui n'est pas impliqué dans la synchronisation.*

Informellement la synchronisation permet à plusieurs transitions d'être franchies simultanément et de contribuer toutes à l'évolution globale du système modélisé (ie. le passage d'un état à un autre). Ainsi, leurs actions sont effectuées de manière parallèle. Du point de vue du système, ceci est similaire au franchissement d'une seule transition ayant σ comme action. Dans le cas où une transition n'est pas impliquée dans l'évolution du système dans son état courant, son action est remplacée par $\bullet$ dans f car elle n'est pas concernée par la synchronisation. Notons que $f(\bullet, \bullet, ..., \bullet)$ n'est pas défini, c'est pour cela que la fonction f est dite partielle. Cette fonction de synchronisation permet ainsi de former un réseau d'automates.

Définition 7.5. *Réseau d'automates finis étendus*

Soit $A_1, ..., A_n$, n automates finis étendus tels que chaque automate A_i est défini par le tuple : $A_i = (S_i, s_{0i}, F_i, X, \mathcal{U}_i(X), \mathcal{G}_i(X), \pi_{0i}, \Sigma, \rightarrow_i)$, où les A_i partagent le même ensemble des variables X et le même ensemble des actions réalisables Σ, et f une fonction de synchronisation partielle définie sur Σ.

Un réseau d'automates finis étendus est un automate global $\mathcal{A} = (S, s_0, F, X, \mathcal{U}(X), \mathcal{G}(X), \pi_0, \Sigma, \rightarrow)$ constitué des automates A_i et de la fonction de synchronisation f.

Ainsi, nous avons :

- *$S = S_1 \times ... \times S_n$: tout état de $\mathcal{A}$ est un ensemble des états courants des A_i.*
- *$s_0 = s_{01} \times ... \times s_{0n}$: l'état initial de $\mathcal{A}$ est l'ensemble des états initiaux des A_i.*
- *$F = \bigcup_{i=1 \rightarrow n} F_i$: tout état final d'un automate A_i est un état final de $\mathcal{A}$.*
- *$\mathcal{U} = \{\mathcal{U}_1, ..., \mathcal{U}_n\}$: la liste des mises à jour de $\mathcal{A}$ est une liste ordonnée formée de la concaténation des listes des mises à jour des A_i.*
- *$\mathcal{G}$: l'ensemble des gardes de $\mathcal{A}$ contient la conjonction des ensembles des gardes des A_i et l'union des ensembles des gardes des A_i.*
- *$\pi_0 = \{\pi_{01}, ..., \pi_{0n}\}$: la liste des affectations initiales de $\mathcal{A}$ est la concaténation des listes des affectations initiales des A_i.*

Évolution d'état au sein d'un réseau d'automates finis étendus : soit $A_1, ..., A_n$, n automates finis étendus tels que chaque automate A_i est défini par le tuple : $A_i = (S_i, s_{0i}, F_i, X, \mathcal{U}_i(X), \mathcal{G}_i(X), \pi_{0i}, \Sigma, \rightarrow_i)$, où les A_i partagent le même ensemble des variables X et le même ensemble des actions réalisables Σ, et f une fonction de synchronisation partielle définie sur Σ. Soit $\mathcal{A}$ le réseau d'automates constitué des A_i et de f.

L'évolution d'état au sein du réseau $\mathcal{A}$ peut résulter de deux possibilités :

- franchissement d'une transition d'un automate A_i : $s_i \xrightarrow{g_i, \sigma_i, \pi_i} s'_i$. Ainsi, aucune synchronisation n'est effectuée.
- franchissement de plusieurs transitions des automates participant à une synchronisation avec la fonction f. Dans ce cas nous avons :
  $s = (s_1, ..., s_n) \xrightarrow{g, \sigma, \pi} s' = (s'_1, ..., s'_n)$ est une transition de $\mathcal{A}$ synchronisant plusieurs automates avec la fonction f si et seulement si :
  - $\exists (\sigma_1, ..., \sigma_n) \in (\Sigma \cup \{\bullet\})^n$ tel que $f(\sigma_1, ..., \sigma_n) = \sigma$.
  - $g = g_1 \wedge g_2 ... \wedge g_n$: la conjonction entre les gardes de toutes les transitions participant à la synchronisation.
  - $\pi = \{\pi_1, ..., \pi_n\}$: la concaténation des mises à jour de toutes les transitions participant à la synchronisation.
  - pour tout i :
    - si $\sigma_i = \bullet$ alors $s_i = s'_i$: l'automate A_i n'est pas impliqué dans la synchronisation. Ainsi, son état n'évolue pas.

- si $\sigma_i \in \Sigma$ alors $s_i \xrightarrow{g_i,\sigma_i,\pi_i} s_i'$ alors l'automate A_i est impliqué dans la synchronisation et son état évolue avec le système.

Concurrence de lecture/écriture des variables dans le réseau d'automates : rappelons que les automates A_i constituants le réseau $\mathcal{A}$ partagent le même ensemble des variables X. Cependant, en passant d'un état à un autre dans le système, tous les automates peuvent lire, dans une garde, une variable $x \in X$ donnée. Toutefois, l'écriture dans une telle variable par une mise à jour ne peut être effectuée que par un seul automate.

7.2 Mise en œuvre en UPPAAL

7.2.1 UPPAAL : outil pour la vérification formelle

UPPAAL est un outil logiciel, fruit d'une collaboration au sein de l'université d'Upsaala en Suède et l'université d'Aalborg en Danemark [BLL+96]. Il est conçu pour la modélisation et la vérification formelle des systèmes temps réel en utilisant des réseaux d'automates finis étendus avec des variables d'horloge et de données, ainsi que des fonctions écrites dans le langage UPPAAL. La syntaxe du langage UPPAAL est très proche de celle du langage C. L'outil est composé de deux parties principales : une interface graphique (GUI) qui s'exécute sur la machine de l'utilisateur et lui permet de créer des modèles, et un moteur de vérification (*VerifyTA*) permettant de vérifier des propriétés sur le modèle élaboré.

7.2.2 La sémantique en UPPAAL

UPPAAL manipule des automates finis étendus et/ou temporisés. Les actions et conditions liées aux transitions dans ces automates peuvent abstraire des instructions présentes dans le code du système à modéliser. UPPAAL offre également la possibilité de déclarer des variables globales et de définir des fonctions (appelées fonctions UPPAAL) en dehors des automates qui peuvent être accessibles par l'ensemble du réseau d'automates. Les actions, les déclarations et les fonctions sont écrites dans une syntaxe similaire à celle du langage de programmation C. Les automates et les fonctions utilisent un ensemble de variables et constantes présentées comme suit :

- variables de temps qu'on appelle des horloges. Ce sont des variables réelles positives. Toutes les variables d'horloge progressent de manière synchrone en UPPAAL.
- des variables de données (entières ou booléennes) qui peuvent être déclarées avec des domaines par défaut ou avec des intervalles déterminés ;
- des constantes dont les valeurs sont initialisées à la déclaration et ne peuvent pas changer durant l'évolution du système modélisé.

```
clock t1, t2, ..., tn;  //declaration of a clock variable

int x1, x2, ..., xn;     //integer with default domain
int[a,b] x1, x2, ..., xn;        //integer with domain from a to b
int x[m];                        //integer array with m elements
bool a;                  //declaration of a boolean
const int x1 = 2;                //integer constant
```

Listing 7.1: Syntaxe de déclaration des variables et constantes

Plusieurs fonctionnalités d'UPPAAL peuvent être employées dans la modélisation d'un système temps réel. Dans ce qui suit, sont citées celles qui sont utilisées principalement dans le présent travail.

Les *templates* : en UPPAAL, chaque automate est élaboré dans un *template*. Celui-ci offre la possibilité de passer à l'automate qu'il contient des paramètres de n'importe quel type supporté par UPPAAL (e.g., int, bool, etc.). Ces paramètres pouvant ainsi être manipulés dans l'automate en question.

Les transitions : une transition en UPPAAL peut contenir une garde qui modélise la condition de son franchissement. Quand elle est franchie, elle peut effectuer une mise à jour de la valeur d'une variable du système modélisé et/ou une synchronisation avec une ou plusieurs transitions des autres automates du système. Cette synchronisation est assurée au moyen d'un canal. Nous présentons dans ce qui suit les types de canaux de synchronisation offerts en UPPAAL.

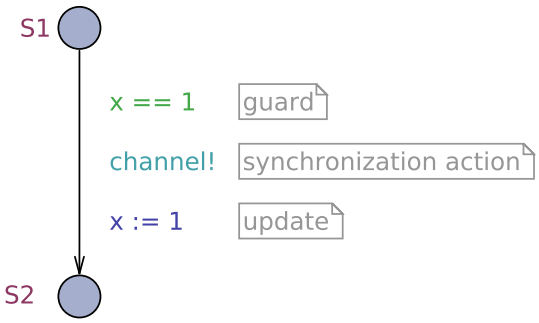


FIGURE 7.4: Exemple d'une transition en UPPAAL.

Les canaux réguliers *(*regular channels*)* : c'est un canal qui permet d'effectuer une synchronisation binaire entre deux transitions réalisant une action `channel`. Classiquement, nous définissons une transition *émettrice* étiquetée `channel?` et une transition *réceptrice* étiquetée `channel!` (cf. Fig. 7.5).

Les canaux de diffusion *(*broadcast channels*)* : dans ce type de canal, une transition *émettrice* étiquetée `channel?` peut se synchroniser avec 0 ou plusieurs transitions

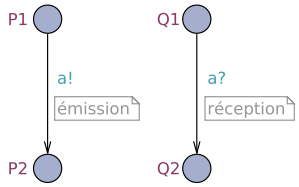


FIGURE 7.5: Transitions en synchronisation binaire via un canal régulier.

réceptrices étiquetées `channel!`. Ainsi, il s'agit d'un canal non bloquant pour l'émission. C'est à dire que la transition émettrice peut être franchie même quand il n'y a aucune transition en attente de la réception.

Les canaux urgents *(*urgent channels*)* : c'est un canal qui réalise une synchronisation dite urgente, c'est à dire qui ne doit pas être retardée. Ainsi, une transition contenant une action de synchronisation urgente ne doit contenir aucune contrainte sur une variable d'horloge.

```
chan a;                    //regular channel
broadcast chan b;                //broadcast channel
urgent chan hurry;               //urgent channel
```

Listing 7.2: Syntaxe de déclaration des canaux de synchronisation

Les états urgents *(*urgent states*)* : ce sont des états dans lesquels l'écoulement de temps n'est pas autorisé. Ainsi, les variables d'horloge sont figées dans ces états. Un état urgent est équivalent à un état normal contenant un invariant $x \leq 0$ sur une variable d'horloge x qui est initialisée à zéro à son entrée. Cet état est toujours étiqueté par la lettre `U`.

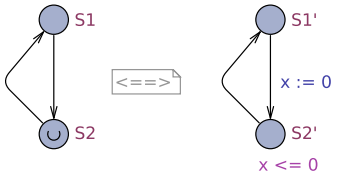


FIGURE 7.6: État urgent : l'urgence de `S2` est spécifiée avec le symbole `U`. En `S2'`, l'urgence est assurée par l'invariant $x \leq 0$ associé à cet état (x est une variable d'horloge). Cet invariant indique que l'on peut rester en `S2'` au maximum jusqu'à ce que x soit inférieur ou égal 0. Étant donné que x est initialisé à zéro, ceci revient à forcer que l'état soit quitté immédiatement.

Les états engagés *(*committed states*)* : il s'agit des états dont les transitions sortantes sont considérées prioritaires. Ainsi, à un état donné de l'exécution, s'il y a plusieurs transitions franchissables dans d'autres automates du réseau, la transition sortant

d'un état engagé est franchie avant toutes les autres. Les états engagés sont étiquetés par la lettre `C`.

7.3 Les techniques de modélisation d'un programme C en UPPAAL

7.3.1 Modélisation d'un système avec des automates finis étendus

Un programme séquentiel peut être abstrait avec un réseau d'automates finis étendus et/ou des automates temporisés en utilisant les actions et conditions associées aux transitions de ces automates. Ainsi, les gardes, les mises à jour et les synchronisations peuvent être utilisées afin de décrire le flot d'instructions du programme. Ces mises à jour et conditions sont réalisées sur un ensemble X des variables qui correspondent aux mêmes variables que celles du système à modéliser. Il s'agit également des mêmes mises à jour et conditions retrouvées dans le programme. Il est à noter qu'une séquence d'instructions peut, dans le cas où l'on ne souhaite pas une modélisation fine du système, être abstraite par une seule transition de l'automate contenant des mises à jour et/ou des gardes. Ainsi, leur exécution est considérée *atomique*. Prenons l'exemple d'un automate à deux états `S1` et `S2` et deux variables x et y ayant pour valeurs initiales 0 (cf. Fig. 7.7). On suppose que la transition de `S1` à `S2` doit mettre à jour les valeurs des deux variables à 2. Ces deux mises à jour se font de manière atomique puisqu'elle sont liées à la même transition, ce qui veut dire que les états pour lesquels $x = 2$ et $y = 0$ ou $x = 0$ et $y = 2$ ne font pas partie de l'espace d'états du modèle.

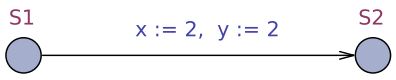


FIGURE 7.7: Exemple d'un automate fini étendu avec des instructions exécutées de manière atomique.

Le système que nous souhaitons modéliser en UPPAAL, dans le cadre de notre travail, est une implémentation d'ordonnanceur au sein de Trampoline. Le code de cette implémentation, tout comme celui de Trampoline, est écrit en langage C. Ainsi, la modélisation consiste à traduire les instructions de ce code sous forme d'automates finis étendus, d'automates temporisés et/ou des fonctions écrits en langage UPPAAL. La syntaxe des deux langages est similaire. Toutefois, certains types de variables ou traitements en C peuvent ne pas exister en UPPAAL. Pour cela, nous présentons dans la suite de cette section, un ensemble de règles utilisées dans notre modélisation, afin d'avoir une « équivalence » entre le code de l'implémentation, élaboré en C, et son modèle élaboré en UPPAAL. Ces règles permettent de modéliser ainsi, les variables, les structures de données, les instructions et/ ou fonctions manipulées en Trampoline de manière à ce qu'elle conservent le même comportement en passant en UPPAAL.

7.3.2 Modélisation des variables

UPPAAL offre la possibilité de manipuler des variables entières bornées ou booléennes (cf. § 7.2.2). Ces variables peuvent être lues, mises à jour et soumises aux opérations arithmétiques communes. En Trampoline, les variables qui sont utilisées sont des variables entières non signées. Afin de pouvoir garder cette spécificité dans un modèle en UPPAAL, il est possible de déclarer une variable appartenant à un intervalle défini. La définition d'un nouveau type dans UPPAAL avec un intervalle défini est également possible (cf. *listings* 7.3 et 7.4).

```
uint8 task_id;
```

Listing 7.3: Déclaration d'une variable entière codée sur 8 bits en Trampoline

```
typedef int[0, 255] uint8;
uint8 task_id;
```

Listing 7.4: Déclaration d'une variable entière codée sur 8 bits en UPPAAL

7.3.3 Opérations arithmétiques

L'environnement d'UPPAAL offre la possibilité d'effectuer des opérations arithmétiques sur l'ensemble des variables qu'il manipule. Toutefois, pour certains types de variables que nous avons ajoutées dans notre modèle, tels que les entiers non signés, le débordement n'est pas géré. Pour cela, nous élaborons pour chaque type de données ajouté, les opérations arithmétiques correspondantes sous forme de fonctions, de manière à pouvoir gérer le débordement correctement. *Listing* 7.5 illustre un exemple de fonction UPPAAL permettant d'effectuer l'opération de l'addition pour des entiers non signés codés sur 8 bits tout en gérant le débordement.

```
uint8 addition8(uint8 first integer, uint8 second integer)
{
   if(first_integer + second_integer < 256)
      return first_integer + second_integer;
   else
      return first_integer + second_integer - 256;
}
```

Listing 7.5: Modélisation d'une addition en 8 bits.

7.3.4 Modélisation des objets

Les objets que nous souhaitons modéliser sont les objets traités par Trampoline. L'objet principal est le descripteur qui stocke en mémoire les informations relatives à une entité du système à un instant t. Dans le code source de Trampoline, un descripteur est défini par une structure en langage C (`struct`). Étant donné que la syntaxe UPPAAL est similaire à celle du C, la définition d'un objet en UPPAAL se fait également grâce à des structures. Prenons l'exemple d'une tâche de Trampoline. C'est une entité décrite par deux descripteurs (cf. § 5.3.4) : un descripteur statique et un descripteur dynamique. La

modélisation de ces deux descripteurs sous UPPAAL est illustrée dans les *listings* 7.6 et 7.7.

```
typedef struct {
    tpl_task_id id;
    int max_activate_count;
    int type;
    uint32 deadline;
    ...
} tpl_proc_static;
```

Listing 7.6: Modèle UPPAAL du descripteur statique d'une tâche en Trampoline

```
typedef struct {
    int activate_count;
    uint32 absolute_deadline;
    int state_d;
    tpl_core_id core_id;
    ...
} tpl_proc;
```

Listing 7.7: Modèle UPPAAL du descripteur dynamique d'une tâche en Trampoline

7.3.5 Modélisation des pointeurs

UPPAAL ne permet pas de déclarer des pointeurs. Pour cela, ces derniers sont modélisés par des tableaux du même type que les éléments vers lesquels ils pointent. L'accès aux éléments pointés est ensuite fait en utilisant les indices du tableau. Par exemple, en Trampoline le pointeur vers l'identifiant d'une tâche est déclaré comme : `CONSTP2VAR(tpl_proc_id, AUTOMATIC, OS_APPL_DATA) proc_id`. Cette déclaration est traduite dans le modèle par le tableau `tpl_proc_id proc_id[ ]`.

7.3.6 Modélisation des structures alternatives

Les structures alternatives `if` et `if ... else` orientent l'exécution d'un programme vers un ensemble d'instructions ou vers un autre. Ce principe peut être modélisé en UPPAAL en utilisant deux transitions sortant du même état. L'une avec une garde `condition` traduisant la condition de `if` et l'autre avec une garde logiquement complémentaire pour traduire la condition du `else` (cf. Fig 7.8).

```
if(condition)
{
    // instruction 1
    }
else // if(!condition)
{
    // instruction 2
}
```

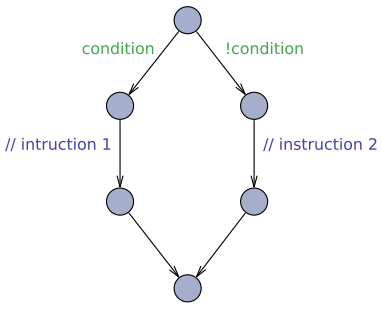


FIGURE 7.8: Modélisation d'une structure alternative en UPPAAL

7.3.7 Modélisation des structures répétitives

Les structures répétitives `for`, `while` et `do ... while` permettent de répéter l'exécution d'une séquence d'instructions un certain nombre de fois ou tant qu'une condition est satisfaite. Une telle structure de contrôle peut être modélisée par un retour à un état précédent tant que la condition d'arrêt est fausse (pour les boucles `while` et `do ... while`) ou le nombre d'itérations (pour la boucle `for`) n'est pas atteint. La figure 7.9 illustre la modélisation de la boucle `while` par un automate fini étendu.

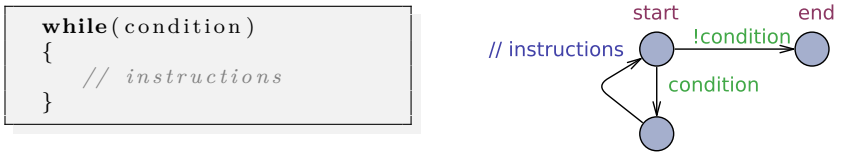


FIGURE 7.9: Modélisation de la boucle `while` en UPPAAL

7.3.8 Modélisation des appels de fonction

Un appel de fonction peut être modélisé par le mécanisme de synchronisation binaire et en utilisant des variables partagées entre deux automates modélisant les fonctions impliquées dans l'appel. Ces variables permettent de gérer le bon entrelacement des exécution d'une part, et de passer les paramètres des fonctions entre les automates d'autre part. Ainsi, prenons l'exemple de la figure 7.10 de deux fonctions `caller_func` et `callee_func` telles que la première appelle la deuxième pendant son exécution. Pour modéliser cet appel, un canal régulier `sync` est utilisé avec une variable partagée `ack` pour indiquer la fin de l'exécution de la fonction appelée. Ceci permet d'assurer une exécution ordonnée conforme à l'exécution séquentielle réelle. Le passage des paramètres est assuré par les variables partagées `param_1, param_2` qui sont initialisées par la fonction appelante au début de la synchronisation. Leurs valeurs peuvent ainsi être lues et/ou modifiées par la fonction appelée. La technique de modélisation d'un appel de fonction présentée dans cette partie constitue un choix de modélisation adopté dans notre travail. Il existe toutefois d'autres mécanismes pouvant modéliser un appel de fonction.

7.4 Modèle pré-existant de Trampoline

7.4.1 Introduction

Un modèle de Trampoline en version monocœur et multicœur a été proposé dans le cadre d'une étude préalable visant à modéliser l'OS sous forme d'un réseau d'automates finis étendus dans le but d'y enlever le code mort [TBFR17]. Nous présentons brièvement dans cette section la version multicœur de ce modèle qui est une extension de sa version monocœur supportant l'ordonnancement partitionné. Notons que ce qui est présenté dans la suite correspond à l'architecture initiale de Trampoline avant l'intégration de l'implémentation de G-EDF (cf. § 5.3). Ainsi, certains composants du périmètre

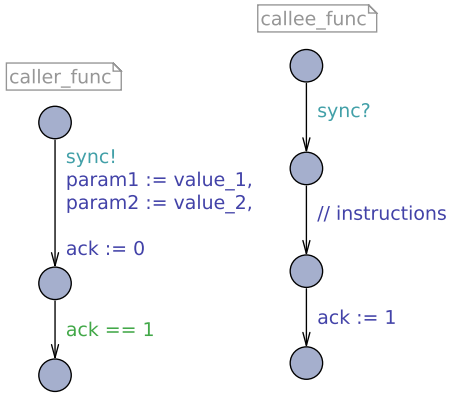


FIGURE 7.10: Exemple d'un appel de fonction : l'automate appelant libère l'exécution de l'automate appelé avec une synchronisation via le canal `sync` ("!" pour émettre et "?" pour recevoir) et met la variable `ack` à 0. Une garde sur cette variable bloque l'exécution de l'automate appelant. Il est libéré dès que l'automate appelé termine l'exécution de ses instructions et remet cette variable à 1.

d'ordonnancement présentés dans la section 6.5 peuvent ne pas figurer dans les modèles décrits dans cette section.

7.4.2 Modélisation du noyau

Le modèle du noyau regroupe les modèles de quatre composants :

- le *dispatcheur* d'interruptions : le modèle de ce composant est construit avec des fonctions UPPAAL et des automates finis étendus permettant de gérer l'appel des ISR1s ou l'activation des ISR2s.
- le gestionnaire de tâches : le modèle de ce composant est constitué de deux automates finis étendus modélisant deux fonctions : (i) `tpl_activate_task()` responsable de l'activation d'une tâche ; (ii) `tpl_terminate()` responsable de la terminaison d'une tâche.
- le gestionnaire des compteurs : ce composant assure la gestion de toute interruption matérielle provenant d'un *timer*. Il est modélisé en utilisant trois automates : (i) `tpl_call_counter_tick` qui permet d'appeler la fonction gérant l'interruption reçue et d'appeler ensuite l'ordonnanceur ; (ii) `tpl_counter_tick` qui permet d'incrémenter les valeurs des compteurs système à partir d'une fonction `CounterTick()` modélisée par une fonction UPPAAL ; (iii) `tpl_raise_alarm` si une alarme expire suite à la réception de l'interruption, cet automate effectue l'action liée à l'alarme.
- l'ordonnanceur : le modèle de ce composant est constitué des automates finis étendus et des fonctions UPPAAL abstrayant ses fonctions présentées dans la section 5.4.

7.4.3 Modélisation des services de l'API

L'API contient 26 fonctions permettant à une tâche d'accéder à Trampoline. L'ensemble de ces fonctions sont abstraites par des automates finis étendus. Un exemple de modélisation d'une fonction de l'API `TerminateTask` est présenté dans la figure 7.11

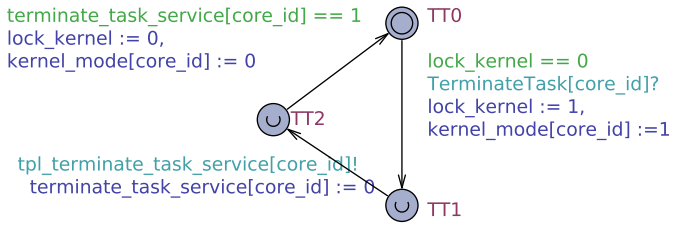


FIGURE 7.11: Modèle de la fonction d'API `TerminateTask` : les opérations de terminaison sont assurées par la fonction `tpl_terminate_task_service` avec qui la fonction de l'API se synchronise.

7.4.4 Gestion du temps

Modélisation d'une source d'interruption : certains événements dans l'OS sont reçus suite au déclenchement d'une interruption. Dans le modèle, les interruptions liées au temps sont générées par un automate temporisé appelé *Timer*. Il contient un seul état et manipule une variable d'horloge appelée `timer` permettant de contrôler l'intervalle entre le déclenchement de deux interruptions (cf. Fig 7.12). À chaque unité de temps écoulée, une transition rebouclée sur l'état de l'automate est franchie. Ceci correspond au déclenchement d'une interruption matérielle. Ainsi, le *Timer* se synchronise avec un autre automate `tpl_call_counter_tick` permettant d'incrémenter la date courante et de libérer tous les événements qui doivent se produire à cet instant.

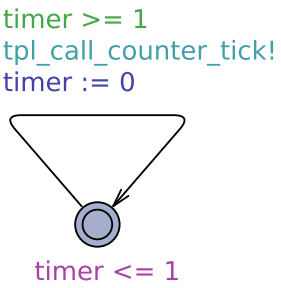


FIGURE 7.12: Modèle du `Timer`.

Le temps au niveau du noyau : dans les automates qui modélisent les fonctions du noyau de Trampoline, aucune variable d'horloge n'est utilisée. En effet, tous les états de ces automates, hormis les états initiaux, sont urgents (cf. Sec. 7.2.2). Ce traitement permet d'interdire l'écoulement à l'intérieur du modèle du noyau. Ainsi, du point de

vue des autres composants du système, l'exécution du modèle du noyau est effectuée en temps nul. Ce choix de modélisation était motivé par le fait que le temps d'exécution du noyau et les *overheads* qu'il engendre ont été considérés négligeables vis-à-vis de l'échelle temporelle des tâches.

Le temps au niveau de l'application : le modèle d'une application Trampoline regroupe des automates modélisant les tâches qui la constituent (cf. § 7.4.5). Dans ces automates, des variables d'horloge peuvent être utilisées afin de reproduire un comportement d'exécution similaire à une exécution réelle (cf. § 9.4).

7.4.5 Modèle d'application

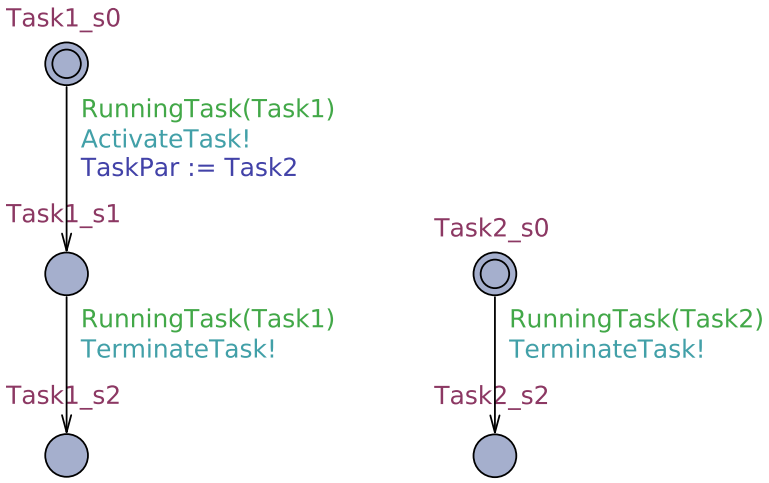
Rappelons qu'une application en Trampoline contient un ensemble de tâches qui interagissent avec le système d'exploitation par le biais des appels système. Elle possède une description OIL contenant une déclaration des objets qu'elle manipule, et une définition de son corps contenant le code C de toutes ses tâches (cf. § 5.3.3). Afin de modéliser une telle application, les structures de données générées à partir de sa description OIL sont récupérées et sont traduites en structures de données en langage UPPAAL. Quant au code source de l'application, chaque tâche est modélisée par un automate temporisé. Cet automate décrit l'ensemble des appels système effectués par la tâche. Chaque automate contient une variable d'horloge x_i qui est mise à 0 quand la tâche correspondante est activée. Les états contiennent des invariants de la forme $x_i \leq WCRT_i$, où $WCRT_i$ correspond au pire temps de réponse de la tâche [1] τ_i. La figure 7.13 présente un exemple de modélisation d'une application se composant de deux tâches `Task1` et `Task2` (cf. *listing* 7.8). Ainsi, deux automates sont utilisés. Dans cet exemple `Task1` permet d'activer `Task2` en faisant un appel système `ActivateTask` prenant comme paramètre la tâche à activer à travers la variable `TaskPar`. Une fonction `RunningTask()` est utilisée comme garde dans les transitions des deux automates. Elle permet de vérifier, en retournant un booléen, si la tâche passée en paramètre est en cours d'exécution ou non.

```
TASK(Task1){
    ActivateTask(Task2);
    // instructions
    TerminateTask();
};

TASK(Task2){
    // instructions
    TerminateTask();
};
```

Listing 7.8: Code source d'une application

[1] Le temps de réponse d'un travail correspond à la durée entre son activation et la fin de son execution. Le pire temps de réponse d'une tâche correspond au maximum des temps de réponse de ses travaux.



(a) Modèle de la tâche 1 (b) Modèle de la tâche 2

FIGURE 7.13: Modélisation de l'application fournie dans le *listing* 7.8.

7.4.6 Propriétés du modèle de l'OS

La bornitude : le modèle de l'OS composé de tous les automates modélisant les fonctions du noyau est un modèle borné du fait qu'il utilise les mêmes variables retrouvées dans l'OS. En effet, les variables utilisées dans Trampoline, étant un système d'exploitation statique, sont soit bornées, telles que les valeurs de priorité des tâches, soit de type énuméré, telles que l'état d'une tâche. Ces variables sont équivalentes à des entiers dans un domaine fini étant donné qu'elles peuvent prendre un ensemble fini de valeurs. Par exemple, certaines de ces variables sont utilisées pour enregistrer une liste d'objets de l'OS (eg. tâches, alarmes, etc.). De telles variables sont ainsi bornées par le nombre d'objets qu'elles référencent.

L'accessibilité : cette propriété peut être décidable pour un automate fini étendu et pour un automate temporisé ayant des variables bornées. Ceci est valide pour le modèle de l'OS et de l'application. Ces modèles sont construits avec des automates finis étendus ayant des variables bornées, et des automates temporisés dont les états possèdent des invariants limitant la progression des variables d'horloge.

Les chemins d'exécution : le modèle complet, correspondant à la combinaison du modèle de l'OS, du Timer et d'une application donnée, contient tous les chemins pouvant être traversés durant l'exécution de l'OS pour cette application. D'une part, les variables, les structures de données et les instructions de l'OS sont inclus dans le modèle complet. D'un autre côté, l'espace d'états du modèle complet contient tous les états dans lesquels le système (OS + application) peut être. Ceci conduit donc à ce que tous les chemins potentiellement traversés durant l'exécution d'une application en Trampoline soient couverts par le modèle construit.

7.5 Conclusion

Dans ce chapitre, nous avons présenté les règles de modélisation d'un système d'exploitation temps réel avec des automates finis étendus et/ou temporisés en utilisant UPPAAL. Sur la base de ces règles, le modèle de Trampoline a été construit dans sa version monocœur et multicœur avec un ordonnancement partitionné à priorité fixe pour les tâches. Ainsi, le flot de contrôle de l'OS a été modélisé par une combinaison d'automates et fonctions UPPAAL, dans lesquels on retrouve les mêmes variables, structures de données et instructions de Trampoline. Les opérations effectuées sur ces variables et structures au niveau des transitions des automates ou des instructions des fonctions UPPAAL sont les mêmes opérations retrouvées dans le code source de l'OS. L'ensemble de ces règles sera employé dans le cadre de notre démarche de vérification des implémentations d'ordonnanceur. Nous les utilisons pour l'élaboration du modèle de l'implémentation de G-EDF dans un premier temps puis celle d'un modèle modélisant une implémentation d'EDF-US[ξ] au sein de Trampoline. Ceci fera ainsi l'objet du prochain chapitre.

CHAPITRE 8

Modélisation d'une implémentation d'ordonnanceur

8.1 Introduction

Dans ce chapitre, nous abordons la première phase de notre démarche de vérification des implémentations d'ordonnanceur, qui consiste à les modéliser (cf. Fig. 1.2). Nous présentons d'abord le modèle élaboré pour l'implémentation de G-EDF au sein de Trampoline à partir de son code source. Sur la base de ce modèle, nous construisons un deuxième modèle décrivant le fonctionnement d'une implémentation d'EDF-US[ξ] au sein de Trampoline. Par souci de simplicité, nous nommons ce deuxième modèle, dans la suite de ce manuscrit, « modèle d'implémentation d'EDF-US[ξ] », bien qu'il ne s'agisse pas d'une implémentation proprement dite puisqu'elle n'a pas fait l'objet de l'écriture du code source associé.

L'implémentation d'un ordonnanceur au sein de Trampoline ne concerne que les composants impliqués dans la décision d'ordonnancement (cf. § 6.5.1). Ainsi, notre travail de modélisation se focalise uniquement sur ces derniers. Toutefois, ces composants étant partie prenante de l'OS, leurs modèles sont destinés à être intégrés dans le modèle préexistant de l'OS comme nous le verrons dans ce chapitre. Étant donné que le modèle de l'OS est conçu en UPPAAL, notre travail de modélisation pour les deux politiques G-EDF et EDF-US[ξ] est effectué en utilisant le même outil et en suivant les mêmes règles présentées dans la section 7.3.

8.2 Modélisation d'une implémentation de G-EDF

8.2.1 Architecture du modèle de l'implémentation de G-EDF

La modélisation de l'implémentation de G-EDF nécessite de s'intéresser à tous les composants contribuant de manière directe ou indirecte à la prise de décisions de l'ordon-

nancement (cf. Fig. 6.5). Une description détaillée de la structure du modèle élaboré et les notations qui y sont employées est fournie dans ce qui suit.

La structure du modèle : dans Trampoline, chaque composant du périmètre d'ordonnancement est implémenté sous forme d'une ou plusieurs fonctions écrites en langage C. Dans notre modélisation, chaque fonction d'un composant est modélisée par un automate fini étendu ou une fonction UPPAAL. L'ensemble des automates et/ou fonctions UPPAAL correspondant à un composant constitue ce que nous qualifions de *bloc*. Ainsi, le modèle de l'implémentation est constitué de plusieurs blocs dont chacun modélise un composant du périmètre de l'ordonnancement. Ces blocs interagissent entre eux par le biais de synchronisations. La figure 8.1 illustre l'architecture du modèle proposé pour l'implémentation de G-EDF.

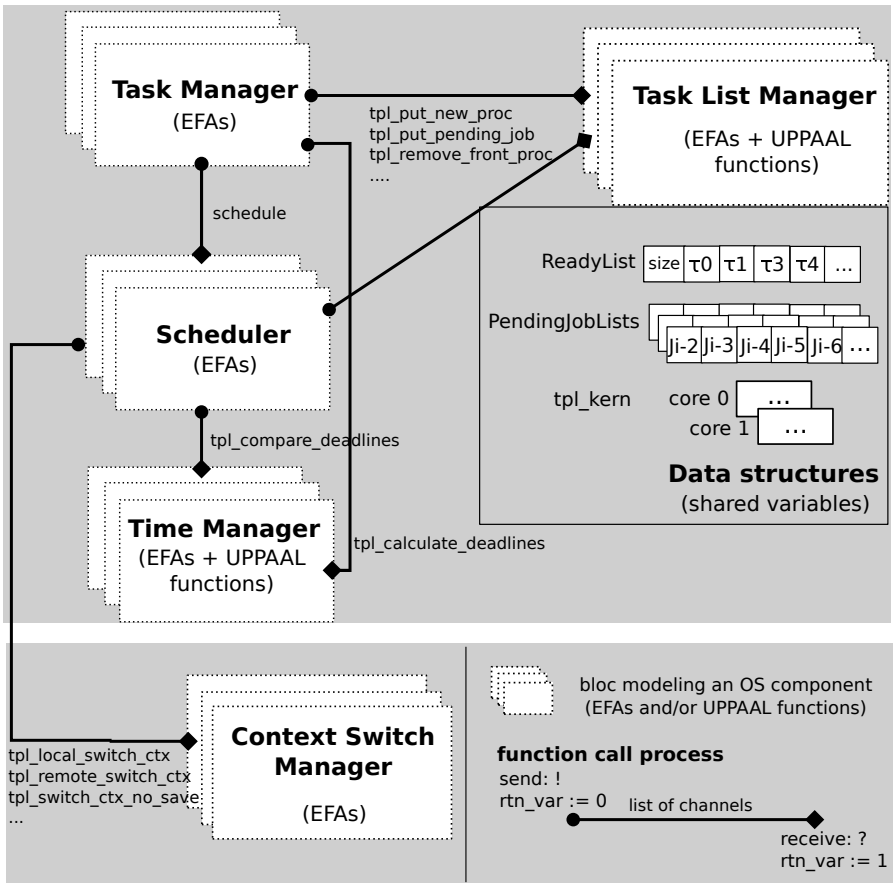


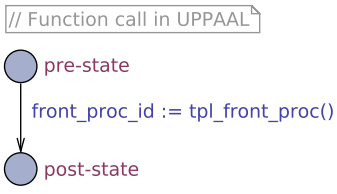
FIGURE 8.1: Architecture du modèle du périmètre d'ordonnancement dans l'implémentation de G-EDF.

Dans cette architecture, deux niveaux de finesse sont adoptés pour la modélisation des fonctions des composants de l'implémentation :

- *modélisation fine :* une fonction de l'OS est modélisée finement si chaque instruction de son code source est traduite par une transition d'un état à un autre dans l'automate qui la modélise. Ainsi, l'état du système peut être observé, en cas de besoin, avant et après l'exécution de chaque instruction de la dite fonction. Ce traitement est destiné aux fonctions nécessitant une vérification détaillée de ses instructions telles que la fonction de l'ordonnanceur.
- *modélisation atomique :* nous utilisons cette modélisation pour des fonctions dont le code source ne nécessite pas une vérification fine instruction par instruction (eg. fonctions d'incrémentation ou de comparaison de variables). Une telle fonction est modélisée par une fonction écrite en langage UPPAAL et peut être appelée par un automate. Ainsi, son exécution dans le modèle est atomique et l'état du système ne peut être observé qu'avant et après son appel. Un exemple de traduction d'une fonction écrite en C vers le langage UPPAAL est présenté dans le *listing* 8.1 et la figure 8.2.

```
// Definition in C language
FUNC(tpl_proc_id, OS_CODE) tpl_front_proc(void){
    GET_READY_LIST(ready_list)
    VAR(tpl_heap_entry, AUTOMATIC) front_proc = READY_LIST(ready_list)
        [1];
    VAR(tpl_proc_id, AUTOMATIC) front_proc_id = front_proc.ptr_front;
    return front_proc_id;
}
```

Listing 8.1: Définition de la fonction `tpl_front_proc` en langage C. Cette fonction permet de récupérer l'identifiant de la tâche en tête de la liste des tâches prêtes.




```
// Definition in UPPAAL language
tpl_proc_id tpl_front_proc(){
 ready_list = GET_READY_LIST()
 tpl_heap_entry front_proc = ready_list
     [1];
 tpl_proc_id front_proc_id = front_proc.
     ptr_front
 return front_proc_id;
}
```

FIGURE 8.2: Modélisation de la fonction `tpl_front_proc` en langage UPPAAL.

Les notations dans le modèle : il est à noter que les noms de fonction en Trampoline sont préfixés par la mention `tpl` (alias de Trampoline). Dans le modèle, cette notation est conservée pour désigner les canaux de synchronisation utilisés pour les appels de fonction. En outre, chaque automate est paramétré par l'identifiant du cœur sur lequel la fonction correspondante doit s'exécuter. En l'occurrence, une fonction de Trampoline

appelée `tpl_function()` sera appelée dans le modèle avec le processus de synchronisation binaire via le canal `tpl_function[core_id]?`.

Le respect de la séquentialité entre fonction et son appelant est assurée à l'aide de variables partagées qui permettent d'acquitter la fin d'exécution de l'automate d'une fonction appelée avant de retourner à l'automate de la fonction appelante (cf. § 7.3.8). Chaque variable utilisée à cette fin est notée par le nom de la fonction qu'elle doit acquitter en y enlevant l'alias `tpl`. À titre d'exemple, la fin d'exécution d'une fonction appelée `tpl_function()` est signalée par la variable `function`.

Le passage des paramètres et les retours de fonction dans le modèle sont également assurés par le biais de variables partagées. Les noms de ces variables sont construits à partir de leurs noms originaux suffixés par `_var`. Ainsi, un paramètre noté `param` dans le code de Trampoline devient `param_var` dans le modèle et est partagé entre la fonction appelante et la fonction appelée. Similairement, un résultat en retour de fonction noté `result` dans le code, est désigné par `result_var` dans le modèle.

8.2.2 Modèles des composants d'ordonnancement

Les composants qui interviennent dans le périmètre d'ordonnancement sont modélisés à partir de leurs codes sources en suivant les règles présentées dans la section 7.3 et en utilisant une combinaison des automates finis étendus et des fonctions UPPAAL. Les modèles produits doivent également respecter les consignes présentées dans la section 8.2.1. Dans ce qui suit sont présentées les principales fonctions de chacun de ces composants.

Modélisation du gestionnaire des tâches (*Task Manager*) : le modèle du gestionnaire des tâches est constitué de deux automates finis étendus qui correspondent aux fonctions d'activation et de terminaison de tâches. Ces fonctions sont rappelées ci-dessous :

- `tpl_edf_activate_task()` : appelée lorsqu'un un travail d'une tâche est activé. Elle se charge de l'incrémentation du compteur d'activation de la tâche, de la mise à jour de son descripteur et de la demande de la mise du travail nouvellement activé dans la `ReadyList` ou dans la `PendingJobList` en question.
- `tpl_edf_terminate_task()` : appelée lorsqu'un travail d'une tâche est terminé. Elle se charge de la décrémentation du compteur d'activation de la tâche et de la mise à jour de son descripteur. Elle demande également le retrait du travail en attente le plus ancien de la `PendingJobList` et sa mise dans la `ReadyList`.

Dans cette section, nous présentons uniquement le fonctionnement et le modèle de la fonction `tpl_edf_activate_task()`. La figure 8.3 illustre le passage du code source vers le modèle de cette fonction. Le rôle de cette fonction est de calculer la date d'échéance du travail qui vient d'être activé et de demander auprès du gestionnaire des listes de tâches de le mettre dans la `ReadyList` (*travail actif*) ou dans la `PendingJobList` (*travail en attente*) en fonction du compteur d'activation de la tâche.

Dans le modèle, l'automate est initialement en attente d'être appelé via la synchronisation `tpl_edf_activate_task[core_id]?` (`core_id` étant le cœur sur lequel la fonction est censée être exécutée). Une fois que cette synchronisation est franchie, les descripteurs statique et dynamique de la tâche ainsi que la date courante sont récupérés. Un premier branchement au niveau de l'état `S_1` modélise un test sur la limite du compteur d'activation de la tâche. Si cette limite n'est pas atteinte, un deuxième test est effectué pour vérifier si le travail nouvellement activé doit être mis dans la `ReadyList` via la synchronisation `tpl_put_new_proc[core_id]!`, ou dans la `PendingJobList` via `tpl_put_pending_job[core_id]!`. Rappelons que si le travail est actif alors l'ordonnanceur doit être appelé. Ceci se fait dans l'automate en positionnant à vrai le booléen `tpl_need_schedule` au niveau de la transition sortant de l'état `S_4`.

Modélisation du gestionnaire des listes de tâches (*Task List Manager*) : ce composant s'occupe de la gestion des structures de données servant à stocker les travaux prêts : la `ReadyList` pour les travaux actifs et les `PendingjobLists` pour les travaux en attente. Cette gestion est assurée au moyen de sept fonctions (cf. § 6.5.4). Les fonctions de la gestion de la `ReadyList` sont modélisées par des automates finis étendus. En ce qui concerne les `PendingJobLists`, ce sont de simples FIFOs avec des d'insertion et d'extraction ne nécessitant pas le ré-ordonnancement de la liste (opérations effectuées en temps constant). Ainsi, les fonctions de gestion de ces listes sont modélisées avec des fonctions UPPAAL.

Modélisation du gestionnaire du temps (*Time Manager*) : le modèle de ce composant est constitué d'automates finis étendus modélisant les fonctions responsables du calcul de la date courante selon une gestion circulaire et les fonctions responsables de la comparaison des dates d'échéance selon l'algorithme ICTOH (cf. § 6.3.2). D'autres fonctions mineures de ce composant permettant de réaliser des opérations d'addition et de soustraction d'entiers non signés sont modélisées par des fonctions UPPAAL (cf. § 7.3.3).

Modélisation du gestionnaire de changement de contexte (*Context Switch Manager*) : ce composant permet de gérer les opérations de sauvegarde et chargement de contexte selon les décisions de l'ordonnancement. Toutes les fonctions de gestion de ces opérations sont modélisées par des automates finis étendus. La figure 8.4 illustre le modèle d'une de ces fonctions qui permet d'envoyer une interruption inter-cœur afin d'informer un autre cœur d'un changement de contexte devant s'y exécuter.

Modèle de l'ordonnanceur (*Scheduler*) : le modèle de l'ordonnanceur est formé de trois automates finis étendus pour abstraire le comportement des trois fonctions décrites ci-dessous.

- `tpl_schedule()` : c'est la fonction principale de l'ordonnanceur qui permet de décider parmi les travaux actifs ceux qui doivent s'exécuter ou être préemptés selon la politique G-EDF.

```
FUNC(tpl_status, OS_CODE) tpl_edf_activate_task(CONST(tpl_task_id,
AUTOMATIC) task_id){

  VAR(uint32, AUTOMATIC) pending_job_deadline;
  VAR(tpl_status, AUTOMATIC) result = E_OS_LIMIT;

  /** Get the task's descriptor */
  CONSTP2VAR(tpl_proc, AUTOMATIC, OS_APPL_DATA) task = tpl_dyn_proc_table
  [task_id];
  CONSTP2VAR(tpl_proc_static, AUTOMATIC, OS_APPL_DATA)  s_task =
  tpl_stat_proc_table[task_id];

  /** Get the current date for deadline calculation     */
  VAR(uint32, OS_VAR)    current_date = tpl_get_crrent_date();

  if (task->activate_count < s_task->max_activate_count)
  {
      if(task->activate_count == 0){
          task->absolute_deadline = current_date + s_task->deadline;
          task->state = (tpl_proc_state)READY_AND_NEW;

          /** Put the job in the ready list      */
          tpl_put_new_proc(task_id);
          tpl_need_schedule = TRUE;
      }
      else{
           pending_job_deadline = current_date + s_task->deadline;
           tpl_put_pending_job(task_id, task->activate_count,
           pending_job_deadline);
      }
      task->activate_count++;
      result = E_OK;
  }
  return result;
}
```

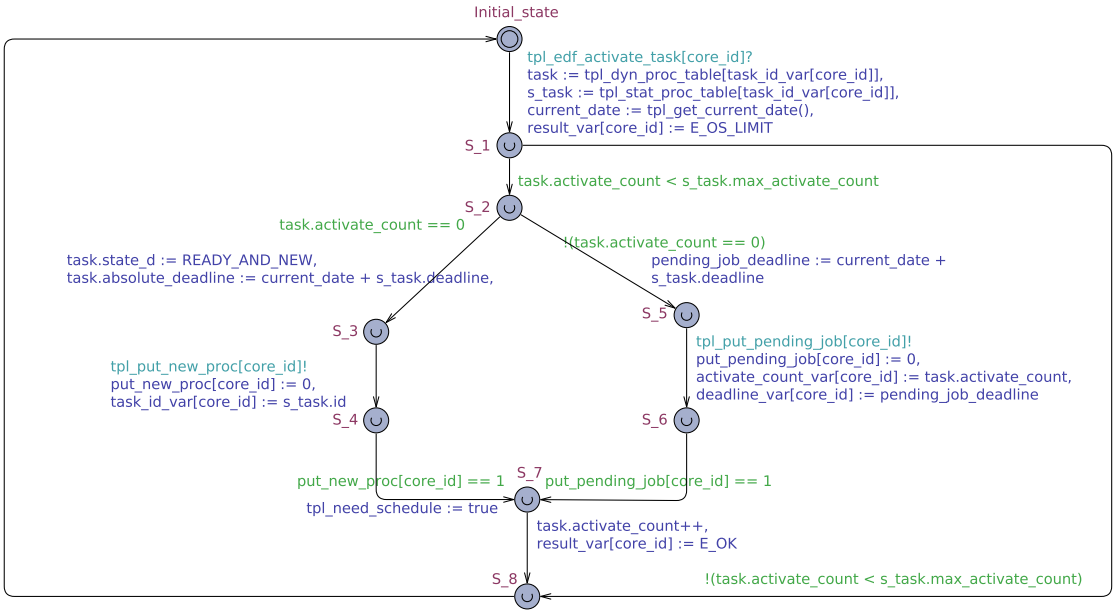


FIGURE 8.3: Modélisation de la fonction du gestionnaire de tâches `tpl_edf_activate_task()`. Nous rappelons que : l'état initial est sous forme de double cercle. Les états urgents sont marqués par le symbole `U`. Les actions de synchronisation se terminent par `?` ou `!` et représentent un appel de fonction. Les variables dont le nom finit par la mention `_var` sont des paramètres passés en arguments ou des résultats de fonction.

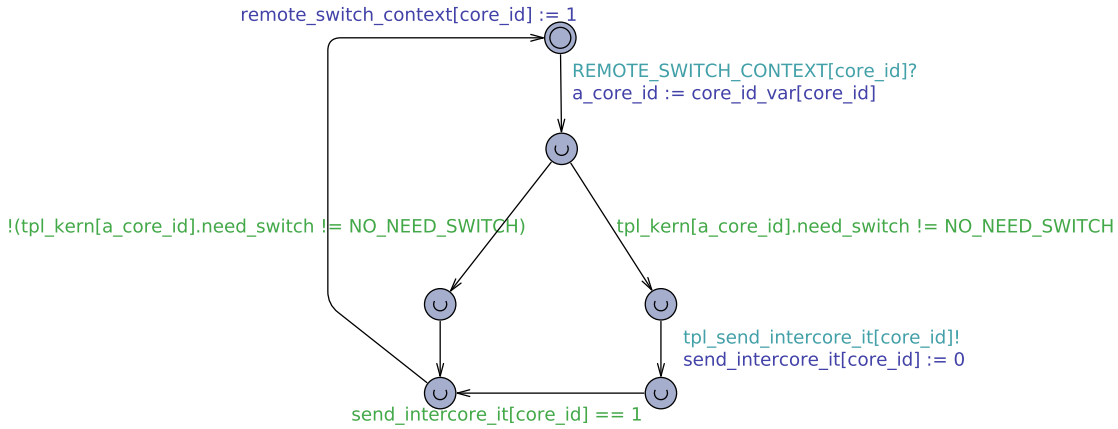


FIGURE 8.4: Modélisation d'une fonction de signalisation d'un changement de contexte distant. Cette signalisation est effectuée, lorsque c'est nécessaire, en envoyant une interruption au cœur concerné via la synchronisation `tpl_send_intercore_it`.

- `tpl_start()` : c'est une fonction appelée par la fonction `tpl_schedule()` pour faire passer un travail de l'état `READY` à l'état `RUNNING`.
- `tpl_preempt()` : c'est une fonction appelée par la fonction `tpl_schedule()` pour faire passer un travail de l'état `RUNNING` à l'état `READY`.

La figure 8.5 illustre le code source de la fonction `tpl_schedule()` et sa traduction sous forme d'automate qui suit le même algorithme présenté dans la figure 6.8. Dans le modèle élaboré, la fonction est appelée via l'action de synchronisation `tpl_schedule[core_id]?`. Ensuite, l'ordonnancement se fait suivant deux traitements assurés par deux boucles consécutives contraintes par la garde `(ready_list_var[0].key > 0)` qui correspond au nombre de nœuds contenus dans la `ReadyList`. Cette condition permet de garantir que l'ordonnanceur est exécuté uniquement quand la `ReadyList` n'est pas vide. Le premier traitement consiste à élire les travaux les plus prioritaires pour exécution sur les cœurs libres (`S_2` à `S_5`). Tandis que le deuxième sert à vérifier s'il existe un travail dans la `ReadyList` qui a une priorité supérieure à celle d'un travail élu (ie. une date d'échéance plus petite) (`S_6` à `S_10`).

8.3 Élaboration d'un modèle d'implémentation d'EDF-US[ξ]

8.3.1 Introduction

Dans cette section, nous présentons un modèle élaboré qui décrit le fonctionnement d'une implémentation de la politique EDF-US[ξ] au sein de Trampoline. À l'inverse du modèle de l'implémentation de G-EDF présenté au début de ce chapitre, celui-ci n'est pas élaboré à partir d'un code source préalablement développé. Ainsi, pour l'implémentation d'EDF-US[ξ], nous proposons de passer plutôt par l'élaboration d'un modèle modélisant le fonctionnement de la politique, conduire la vérification sur ce modèle et en dériver

```
FUNC(void, OS_CODE) tpl_schedule (void){

  VAR(uint32, AUTOMATIC) core_id;
  VAR(tpl_proc_id, AUTOMATIC) front_proc;

  GET_READY_LIST(ready_list)
  VAR(uint8, AUTOMATIC) need_switch = NO_NEED_SWITCH

  if(ready_list[0].key){      /* ready_list[0].key stores the size of the heap */
    for(core_id = 0; core_id < NUMBER_OF_CORES; core_id++){

        GET_TPL_KERN_FOR_CORE_ID(core_id, kern)
        if(TPL_KERN_REF(kern).elected_id == TASK_COUNT + core_id &&
        size > 0){
           tpl_start(core_id);
           TPL_KERN_REF(kern).need_switch = NEED_SWITCH;
        }
    }
  }
  if(ready_list[0].key){
    for(core_id = 0; core_id < NUMBER_OF_CORES; core_id++){
         front_proc = tpl_front_proc();
         GET_TPL_KERN_FOR_CORE_ID(core_id, kern)
         if((front_proc.key) < tpl_dyn_proc_table[TPL_KERN_REF(kern).
         elected_id]->absolute_deadline){
             tpl_preempt(core_id);
             TPL_KERN_REF(kern).need_switch = NEED_SWITCH | NEED_SAVE;
             tpl_start(core_id);
          }
    }
  }
}
```

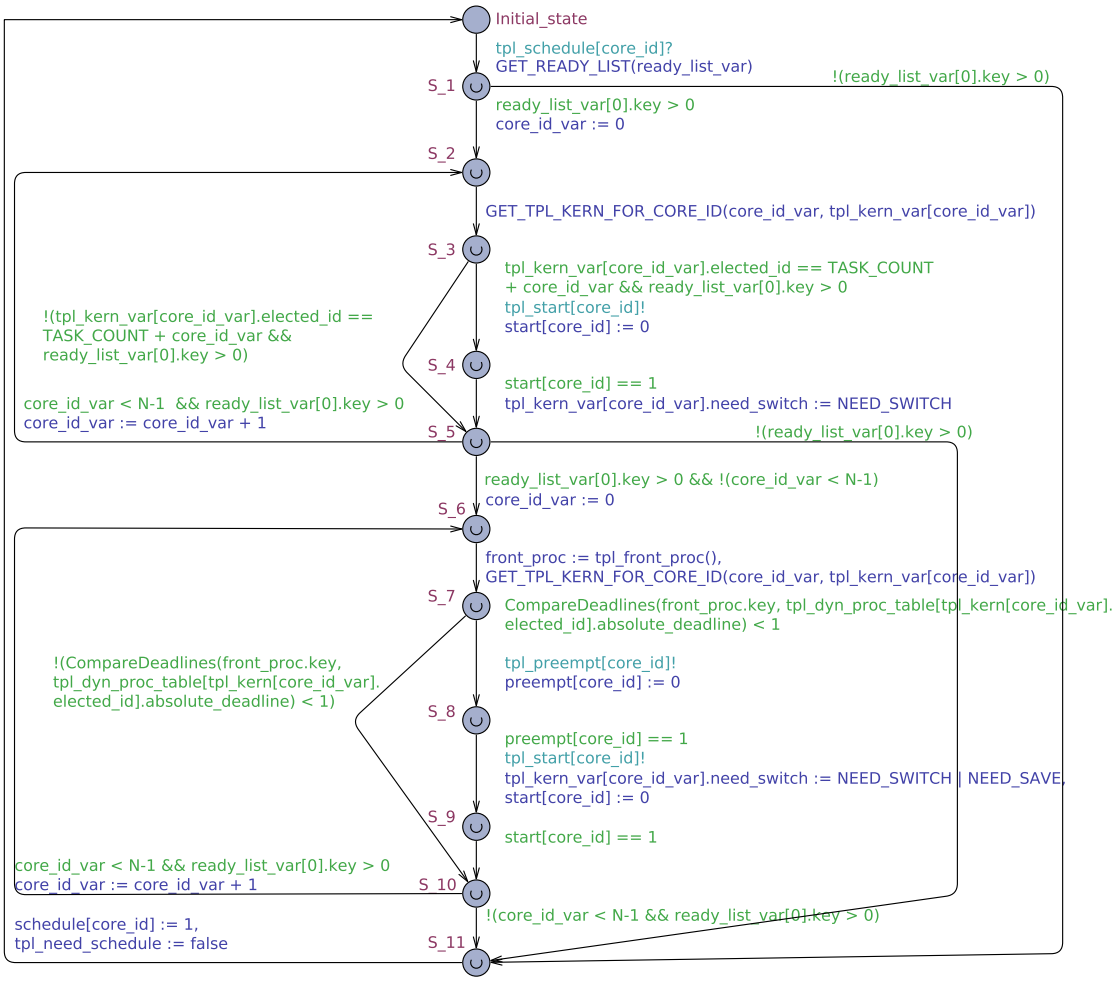


FIGURE 8.5: Modélisation de la fonction `tpl_schedule()`.

le code source de l'implémentation une fois le processus de vérification accompli. Ce choix de modélisation est motivé par le fait que nous disposons préalablement d'un modèle de l'OS vérifié et qui a été adapté pour pouvoir supporter l'ordonnancement global en y intégrant le modèle de l'implémentation de G-EDF. D'un autre côté, en raison de la similitude entre la politique G-EDF et EDF-US[ξ], le modèle à développer pour l'implémentation de cette dernière peut facilement être construit sur la base de celui développé pour l'implémentation de G-EDF. En outre, nous estimons qu'il serait intéressant de tester la démarche de vérification dans le sens où l'on élabore un modèle sur la base d'une spécification formelle du fonctionnement de la politique au sein de Trampoline et d'en déduire le code source à l'issue de la démarche.

8.3.2 Architecture du modèle d'implémentation d'EDF-US[ξ]

Rappelons que la politique EDF-US[ξ] est inspirée de G-EDF (cf. § 2.4.1). Elle définit deux classes de tâches : (i) des tâches lourdes dont le taux d'utilisation dépasse un seuil prédéfini ξ ; (ii) des tâches légères dont le taux d'utilisation est inférieur à ξ. La règle de priorité selon EDF-US[ξ] fait que les tâches lourdes sont plus prioritaires que les tâches légères. De plus, la résolution des conflits entre ces tâches est arbitraire. En revanche, la priorité entre les travaux des tâches légères est déterminée selon la politique EDF. C'est pour cette raison que le modèle à construire pour l'implémentation de cette politique se base sur le modèle préalablement construit pour l'implémentation de G-EDF en intégrant la notion du poids des tâches : lourde ou légère. L'adaptation du modèle initial pour pouvoir supporter la politique EDF-US[ξ] est discutée ainsi dans la suite de cette section.

L'adaptation du modèle de G-EDF pour l'ordonnancement selon EDF-US[ξ] : le fonctionnement d'une implémentation d'EDF-US[ξ] au sein de Trampoline fait intervenir les composants du périmètre d'ordonnancement. Nous proposons un pseudo algorithme permettant de représenter ce fonctionnement (cf. § 8.3.3). La modélisation de l'implémentation d'EDF-US[ξ] est faite sur la base de cet algorithme. Pour cela, plusieurs modifications ont été apportées sur le modèle de G-EDF afin de l'adapter pour une implémentation d'EDF-US[ξ], nous les discutons dans ce qui suit :

- *classification des tâches :* le fonctionnement d'un ordonnanceur de type EDF-US[ξ] se base sur une connaissance préalable de la durée d'exécution de la tâche pour pouvoir calculer son taux d'utilisation et la qualifier de lourde ou légère. Ceci doit se faire avant le déroulement de l'ordonnancement de manière à ce que l'on connaisse le poids des tâches avant le démarrage du système. Pour ce faire, nous introduisons au sein du descripteur statique de la tâche un champ qui renseigne cette information. À cette fin, une variable booléenne est mise en place avec la convention de valeur : `vrai` dans le cas d'une tâche lourde et `faux` dans le cas d'une tâche légère (cf. *listings* 8.2 et 8.3).
- *listes des travaux prêts :* en se basant sur la variable renseignant le poids d'une tâche, le stockage des travaux prêts peut être fait selon que la tâche est lourde ou légère. Nous conservons la notion de travail prêt actif et travail prêt en attente introduite

dans la section 6.4. Ainsi, trois listes de tâches sont utilisées pour le stockage des travaux prêts et qui sont gérées par le gestionnaire des listes de tâches :

- `HeavyList` : cette liste stocke les travaux prêts actifs des tâches lourdes. Étant donné que les conflits de priorité entre tâches lourdes sont gérés de manière arbitraire, cette liste est modélisée par une FIFO. Les travaux y sont insérés dans l'ordre de leur activation et retirés selon leur ancienneté.
- `ReadyList` : cette liste sert à stocker les travaux prêts actifs des tâches légères. La priorité entre ces tâches est gérée selon l'algorithme EDF. Ainsi, nous conservons l'implémentation sous forme de tas de cette liste avec tri selon les dates d'échéance des travaux.
- `PendingJobList` : cette liste permet de stocker les travaux en attente d'une tâche donnée, qu'elle soit lourde ou légère. Ainsi, à l'image de ce qui a été fait pour G-EDF, chaque tâche possède sa propre `PendingJobList`. L'insertion dans cette liste est faite en fonction de l'ancienneté des travaux de la tâche en question. Ainsi, elle est implémentée sous forme d'une FIFO.

```
typedef struct {
    tpl_internal_resource
        internal_resource;
    tpl_task_id id;
    int max_activate_count;
    int type;
    uint32 deadline;
    bool weight;     //weight of
        tasks
} tpl_proc_static;
```

Listing 8.2: Descripteur statique de la tâche

```
//weight of tasks
const bool HEAVY = true;
const bool LIGHT = false;
```

Listing 8.3: Définition des poids de la tâche

La structure du modèle : le modèle de l'implémentation de la politique EDF-US[ξ] regroupe les modèles élaborés pour chaque composant du périmètre d'ordonnancement. Sa structure suit la même logique de celle du modèle construit pour l'implémentation de G-EDF présentée dans la figure 8.1. Ainsi, chaque composant est modélisée par des automates finis étendus et/ou des fonctions UPPAAL abstrayant les fonctions qui le composent. Les interactions entre les automates sont assurées par des canaux de synchronisation. En récupérant le modèle de l'implémentation de G-EDF, les modèle de certains de ses composants y ont été modifiés en adéquation avec le fonctionnement de la politique EDF-US[ξ]. Tandis que d'autres ont été conservés du fait que leur fonctionnement demeure le même pour les deux politiques. La figure 8.6 illustre la structure du nouveau modèle élaboré avec les composants dont le fonctionnement a été modifié ou conservé, et les nouvelles structures de données ajoutées. Notons que le fonctionnement du gestionnaire de temps (*Time Manager*) et du gestionnaire de changement de contexte (*Context Switch Manager*) ne change pas pour la politique EDF-US[ξ]. De ce fait, leurs modèles sont récupérés sans modification. Ainsi, les composants dont les modèles ont été modifiés sont :

- le gestionnaire des tâches (*Task Manager*) : les fonctions de ce composant ont été modifiées afin de pouvoir intégrer la gestion des tâches lourdes à l'activation ou la terminaison d'un nouveau travail. Le fonctionnement de ces fonctions est détaillé dans le paragraphe suivant ;
- le gestionnaire des listes de tâches (*Task List Manager*) : avec la considération d'une nouvelle structure de données stockant les travaux de tâches lourdes, des fonctions ont été ajoutées à ce composant. Elles gèrent les différentes opérations d'insertion/extraction dans la `HeavyList` ;
- l'ordonnanceur (*Scheduler*) : les fonctions de l'ordonnanceur sont toutes modifiées afin de pouvoir intégrer le traitement des tâches lourdes dans le processus d'ordonnancement. Celui ci est détaillé dans le paragraphe suivant.

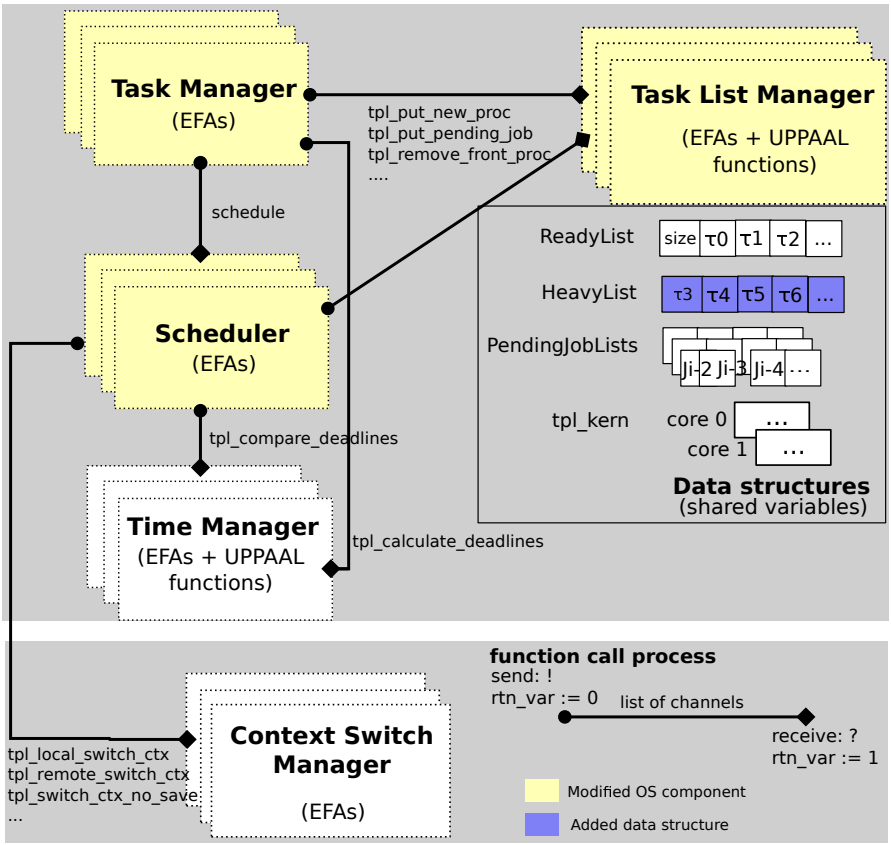


FIGURE 8.6: Structure du modèle de l'implémentation d'EDF-US[ξ].

8.3.3 Processus d'ordonnancement dans le modèle d'implémentation d'EDF-US[ξ]

Dans cette section, nous détaillons le déroulement de l'ordonnancement selon la politique EDF-US[ξ] au sein du modèle élaboré pour son implémentation dans Trampoline.

Par souci de simplicité, nous choisissons d'illustrer ce processus à travers des pseudo-algorithmes au lieu de fournir l'ensemble des automates modélisant les composants de cette implémentation.

Pour la politique EDF-US[ξ], deux événements peuvent conduire à un ré-ordonnancement dans le cas d'une application composée uniquement de tâches basiques : (i) activation d'un nouveau travail ; (ii) terminaison d'un travail en cours d'exécution. Rappelons que dans Trampoline, les opérations liées à ces deux événements sont assurées par le gestionnaire des tâches qui s'occupe ensuite d'appeler l'ordonnanceur à l'issue de son exécution. Ce dernier calcule la décision d'ordonnancement en fonction de ses règles et de l'état des différentes listes de tâches. Si la décision d'ordonnancement conduit à un changement de contexte, le gestionnaire de changement de contexte en est tenu informé par l'ordonnanceur. Ce processus est conservé et adapté pour la politique EDF-US[ξ] dont les étapes sont présentées dans ce qui suit :

- **activation d'un nouveau travail d'une tâche τ_i :** suite à cet événement, le gestionnaire des tâches débute par le test du compteur d'activation de la tâche τ_i : (i) si le compteur d'activation est supérieur à zéro, le travail nouvellement activé doit être mis en attente dans la `PendingJobList` de la tâche correspondante. Si cette tâche est légère, la date d'échéance du travail activé est calculée par le gestionnaire de temps et enregistrée dans la `PendingJobList` ; (ii) sinon, le travail doit être mis soit dans la `HeavyList` dans le cas d'une tâche lourde, soit dans la `ReadyList`, après le calcul de sa date d'échéance dans le cas d'une tâche légère. Nous rappelons que l'ordonnancement des tâches lourdes ne se base pas sur leurs dates d'échéance. Ainsi, il n'est pas nécessaire de les calculer ou les enregistrer pour ces tâches. L'insertion dans les listes des tâches est gérée par le gestionnaire des listes de tâches qui assure également leur tri. Le gestionnaire de tâche finit par informer l'ordonnanceur de la nécessité d'un ré-ordonnancement. Ces étapes sont illustrées dans la figure 8.7.
- **terminaison du travail en cours d'exécution d'une tâche τ_i :** dans ce cas, le gestionnaire des tâches commence par décrémenter le compteur d'activation de la tâche τ_i et effectue un test : si le compteur d'activation est supérieur à zéro, ceci veut dire que la tâche dont le travail vient de se terminer possède au moins un travail en attente. Dans ce cas, ce dernier est extrait de la `PendingJobList` de la tâche et mis soit dans la `HeavyList` dans le cas d'une tâche lourde, soit dans la `ReadyList` dans le cas d'une tâche légère. L'ordonnanceur est appelé à la fin de ce test (cf. Fig. 8.8).
  La figure 8.10 présente l'automate modélisant la fonction du gestionnaire des tâches appelée suite à la terminaison d'un travail en cours d'exécution. Cet automate se base le pseudo algorithme de la figure 8.8.
- **déroulement de l'ordonnancement :** la décision d'ordonnancement est calculée par l'ordonnanceur en consultant et manipulant les deux listes de travaux actifs prêts à travers le gestionnaire des listes de tâches. Il se base également sur le résultat des comparaisons des dates d'échéance des travaux en concurrence fourni par le gestionnaire du temps. Une fois que sa décision est prise, celle-ci est appliquée par le gestionnaire de changement de contexte. Ainsi, l'ordonnanceur calcule l'ordonnancement en deux étapes effectuées successivement :

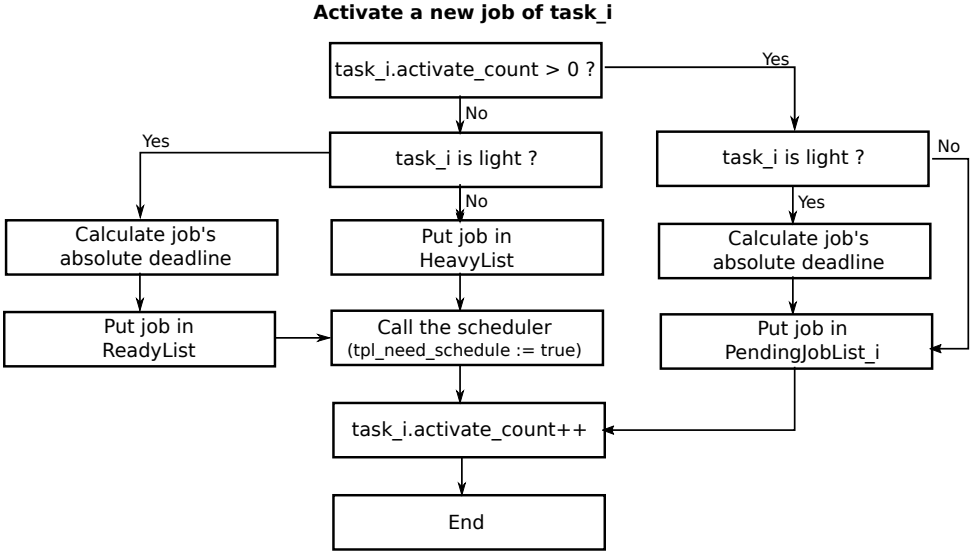


FIGURE 8.7: Pseudo algorithme d'activation d'un travail d'une tâche τ_i.

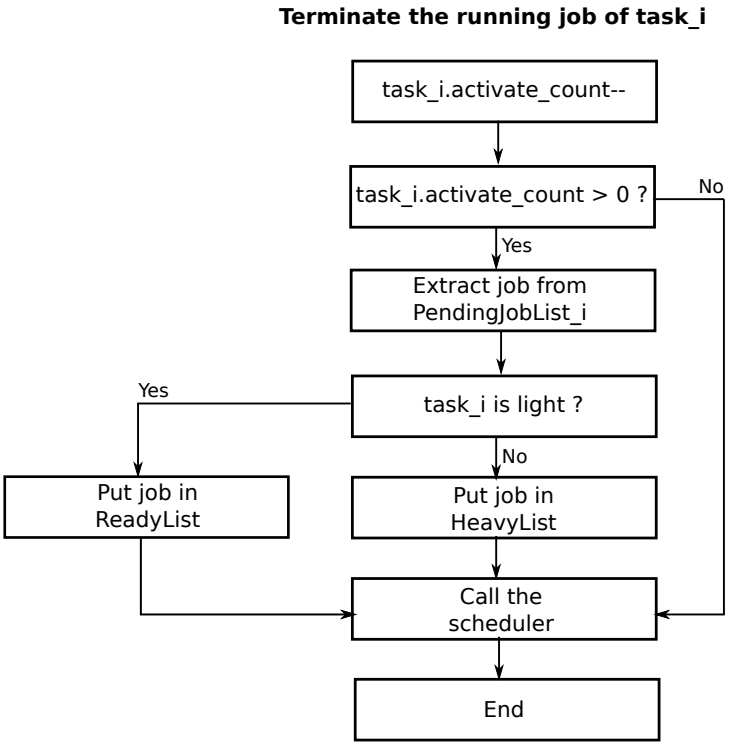


FIGURE 8.8: Pseudo algorithme de terminaison du travail en cours d'exécution d'une tâche τ_i.

1. *ordonnancement des tâches lourdes :* elles sont ordonnancées en premier étant donné qu'elles sont plus prioritaires que les tâches légères. Ainsi, si un travail

d'une tâche lourde est prêt (ie. la `HeavyList` n'est pas vide) et qu'il existe un cœur libre, ce dernier se voit assigné le travail pour son exécution et l'ordonnanceur informe le gestionnaire de contexte de la nécessité d'effectuer un changement de contexte. De la même manière, si un travail d'une tâche lourde est prêt alors qu'un autre d'une tâche légère est en cours d'exécution, celui-ci est préempté et le travail lourd est sélectionné pour s'exécuter. Ces deux opérations sont répétées tant que la `HeavyList` n'est pas vide et qu'il y a des cœurs libres ou des travaux de tâches légères en cours d'exécution. La priorité entre les tâches lourdes étant gérée arbitrairement, une fois qu'un travail d'une tâche lourde est sélectionné pour être exécuté, le choix a été fait de ne pas le préempter avant la fin de son exécution.

2. *ordonnancement des tâche légères :* dans cette étape, la `ReadyList` est examinée pour voir si elle contient au moins un travail prêt et qu'il y a au moins un cœur libre, . Dans ce cas, le travail est extrait de la liste et sélectionné pour exécution et un changement de contexte est notifié. Dans le cas contraire, les travaux de tâches légères en cours d'exécution sont examinés pour voir si l'un d'eux possède une date d'échéance plus grande que celle du travail en tête de la `ReadyList`. Si tel est le cas, le travail en cours d'exécution est préempté en faveur de celui qui est à la tête de la liste. Ce dernier est sélectionné pour être exécuté et un changement de contexte est notifié. Ce traitement est répété tant qu'il y a des travaux prêts plus urgents dans la `ReadyList` et des cœurs libres.

- **processus de sauvegarde et de changement de contexte :** cette opération est gérée par le gestionnaire de changement de contexte. Elle reste inchangée par rapport à l'implémentation de G-EDF (cf. § 6.5.6) : quand un changement de contexte est nécessaire, l'ordonnanceur met à jour le booléen `need_switch` du cœur sur lequel le changement doit être effectué. Ainsi, le gestionnaire de changement de contexte s'occupe de sauvegarder le contexte du travail qui perd le CPU et charger celui du travail élu par l'ordonnanceur.

8.4 Modèle du Timer

Rappelons que le modèle du noyau de Trampoline est non temporisé et qu'il s'exécute en temps nul (cf. § 7.4.4). Néanmoins, un ordonnanceur de type G-EDF ou EDF-US[ξ] se base sur les dates d'échéance des travaux (de tâches légères pour EDF-US[ξ]) afin de calculer la décision d'ordonnancement. Ainsi, il doit disposer de ces dates. À cette effet, il est nécessaire de disposer d'un mécanisme qui permet de récupérer des dates courantes d'exécution afin de calculer des dates d'échéance. Nous proposons pour cela un modèle permettant décrire le fonctionnement d'un `Timer` et qui est utilisé dans les deux modèles d'implémentation de G-EDF et d'EDF-US[ξ]. Il s'agit d'un automate temporisé qui gère une variable continue d'horloge représentant la progression du temps lors de l'exécution de tâches. L'automate gère également une variable partagée discrète dans laquelle est enregistrée la date courante appelée `micro_sec_date`. Elle modélise ainsi le temps enregistré en Trampoline en microseconde. La microseconde correspond à la granularité

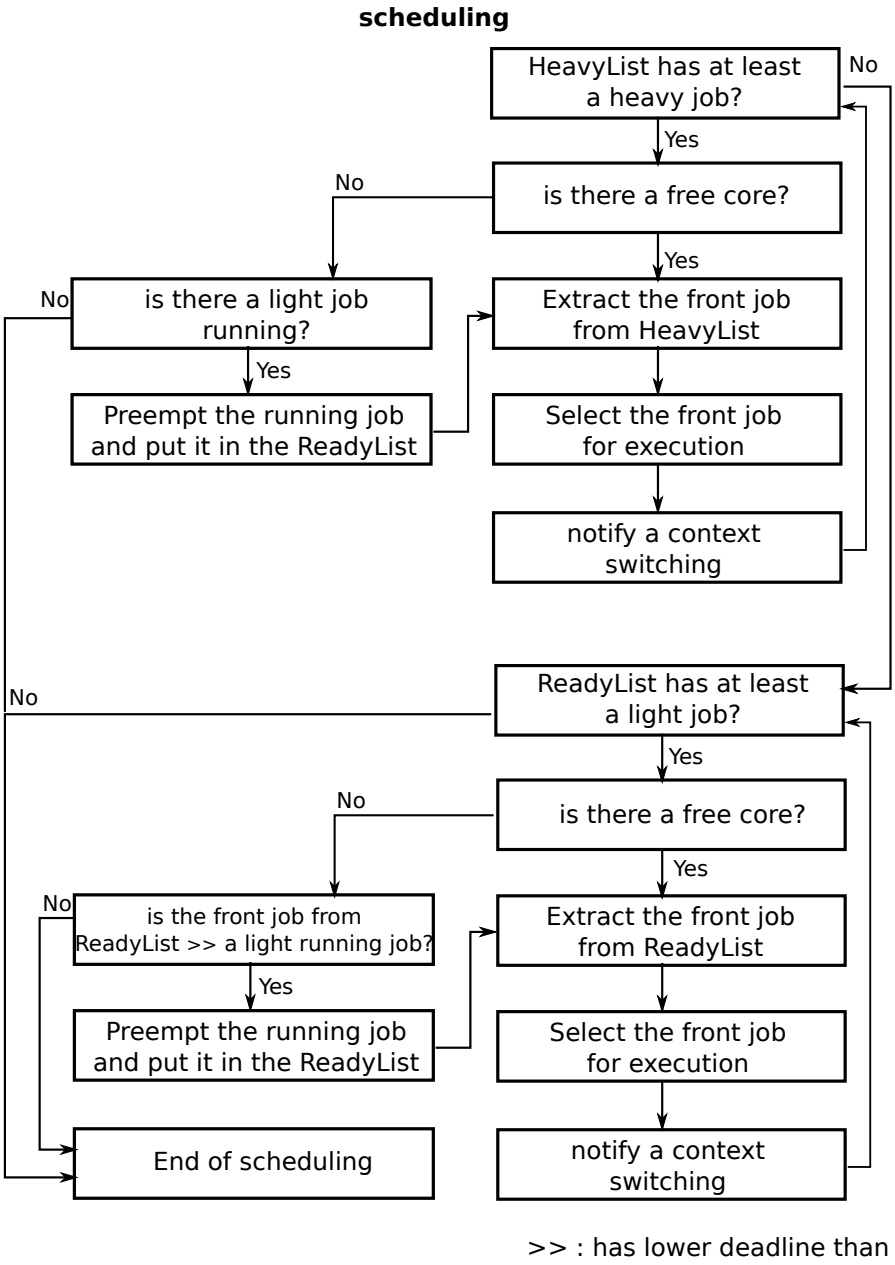


FIGURE 8.9: Pseudo algorithme de l'ordonnanceur EDF-US.

choisie dans l'implémentation en Trampoline. Tel que présenté sur la figure 8.11, cette variable est incrémentée à chaque fois que l'automate du `Timer` est synchronisé, via le canal `MicroSecondsInc`, avec d'autres automates dans lesquels le passage du temps est nécessaire (eg. un automate modélisant l'exécution d'une tâche). L'incrémentation de la variable `micro_sec_date` est gérée par la fonction UPPAAL `micro_seconds_inc_func()`

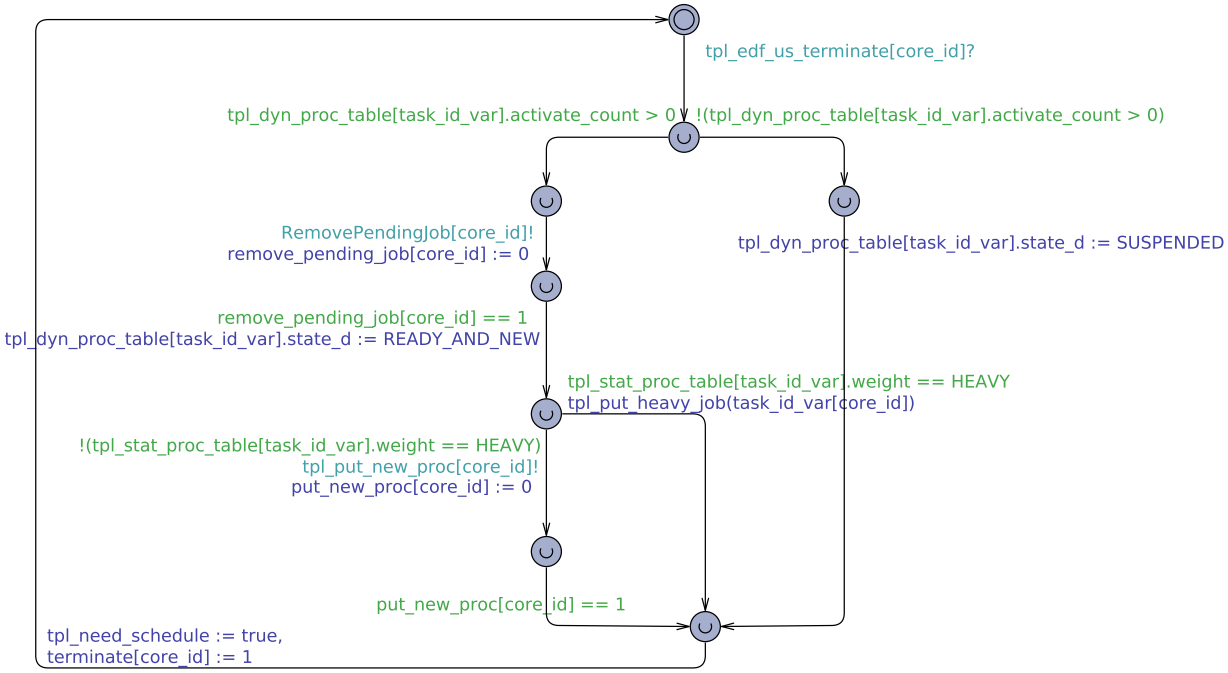
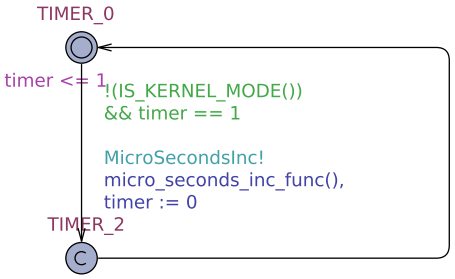


FIGURE 8.10: Automate modélisant la fonction `tpl_edf_us_terminate`.

qui doit également gérer le débordement. Notons que l'évolution de l'horloge, et notamment l'incrémentation de la variable discrète du temps, n'a pas lieu en mode noyau puisque tous les états dans ce mode sont urgents. Notre modèle du `Timer` est différent de celui présenté dans la figure 7.12 puisqu'il permet de modéliser l'enregistrement des ticks d'horloge et ainsi les dates courantes en Trampoline.

Il est indispensable de bien faire la différence entre ce modèle de `Timer` et celui du gestionnaire de temps (*Time Manager*). Le premier sert à modéliser une progression de temps pendant l'exécution des tâches, tandis que le deuxième représente une abstraction d'un composant de l'OS qui est responsable du calcul et de la comparaison des dates d'échéance.




```
void micro_secondes_inc_func()
{
  if(micro_sec_date == (1 << 31) -1)
    micro_sec_date = 0;
  else
    micro_sec_date = micro_sec_date +
        1;
}
```

FIGURE 8.11: Modèle du `Timer`.

8.5 Conclusion

Dans ce chapitre, nous avons présenté deux modèles qui décrivent de manière fidèle le comportement de deux implémentations d'ordonnanceur au sein de Trampoline : G-EDF et EDF-US[ξ]. L'architecture des deux modèles d'implémentation comme les modèles des composants du périmètre d'ordonnancement ont été abordés. Dans ce travail de modélisation d'implémentation, nous avons examiné deux voies possibles : (i) la première consiste à élaborer le modèle de l'implémentation à partir de son code source. Ce qui était le cas pour le modèle de l'implémentation de G-EDF ; (ii) la deuxième a pour objectif de construire directement le modèle de l'implémentation sans passage préalable par un code source élaboré, tel que nous l'avons proposé pour la modélisation de l'implémentation d'EDF-US[ξ]. Notons que dans notre démarche de vérification, le modèle d'implémentation doit décrire fidèlement le fonctionnement de celle-ci. Ceci ne serait possible sans connaissance exhaustive des composants, variables ou fonctions impliqués dans l'implémentation à vérifier. Ainsi, un passage par l'élaboration du code source de l'une des implémentations avant sa modélisation était indispensable dans la mesure où cela a permis d'identifier le périmètre de l'ordonnancement et de maîtriser le processus d'ordonnancement au sein de Trampoline. Ainsi, une fois que le modèle de cette implémentation est construit et intégré à celui de l'OS, il devient plus aisé d'élaborer directement d'autres modèles pour d'autres implémentations et de les intégrer au modèle de l'OS sans passer par leur code source, ce qui était le cas pour le modèle d'implémentation d'EDF-US[ξ].

Dans ce chapitre, nous avons également proposé un modèle de `Timer` qui a été intégré aux modèles des deux implémentations susmentionnées. Il permet de faire évoluer le temps et enregistrer les dates courantes dans le but de récupérer des dates d'échéance pour les tâches. Ce `Timer` est également important dans le cas où l'on souhaite stimuler le modèle du système d'exploitation par une application dans laquelle il est nécessaire de faire passer le temps.

À l'issue de ce travail de modélisation, nous disposons de deux modèles de Trampoline avec deux versions d'ordonnanceurs : le premier avec une implémentation de G-EDF et le deuxième implémente la politique EDF-US[ξ]. Ces deux modèles seront utilisés dans la deuxième phase de notre démarche de vérification présentée dans la partie suivante.

Quatrième partie

Vérification d'implémentation par model-checking

Cette partie est consacrée à la deuxième phase de notre approche de vérification et ses résultats. Ainsi, nous identifions quatre étapes que nous suivons dans cette deuxième phase pour mener la vérification. Ce processus est instancié dans un premier temps sur le modèle de l'implémentation de G-EDF. Ensuite, nous proposons un mécanisme permettant de générer de manière non déterministes des événements d'ordonnancement pouvant stimuler le modèle de l'OS dans le processus de vérification. Ce mécanisme sera utilisé dans le cadre de la vérification d'une implémentation d'EDF-US au sein de Trampoline. Notre démarche de vérification, étant basée sur le model-checking, conduit à une explosion de l'espace d'états exploré. Ainsi, nous proposons également dans cette partie des techniques de réduction de cet espace.

CHAPITRE 9

Approche de vérification

9.1 Introduction

Dans ce chapitre, nous détaillons la deuxième phase de la démarche que nous proposons pour la vérification des implémentations d'ordonnanceur par model-checking. Cette phase consiste à vérifier la correction fonctionnelle de l'implémentation en vérifiant le respect des exigences de sa spécification sur son modèle préalablement élaboré. Nous entendons par correction fonctionnelle la cohérence du comportement effectif produit par l'implémentation par rapport au comportement attendu. Autrement dit, la justesse des opérations et résultats produits par l'ordonnanceur par rapport à la politique d'ordonnancement considérée. Ainsi, les exigences de la spécification à vérifier expriment ce comportement attendu. Nous souhaitons proposer une démarche la plus générique possible, la plus indépendante possible de la politique d'ordonnancement à vérifier, et pouvant être adaptée selon le système d'exploitation cible, le nombre et la nature des composants liés à l'ordonnanceur. Dans un premier temps, nous instancions notre démarche sur l'implémentation de G-EDF au sein de Trampoline présentée dans le chapitre 6.

La deuxième phase de notre démarche de vérification consiste en quatre étapes principales que nous développons dans la suite de ce chapitre (cf. Fig. 9.1) :

1. traduire les propriétés liées à la politique implémentée en exigences décrivant le comportement attendu de l'implémentation (cf. § 9.2) ;
2. formaliser les exigences ainsi établies afin de pouvoir les vérifier formellement (cf. § 9.3) ;
3. générer des événements d'ordonnancement permettant de stimuler le modèle de l'implémentation et le soumettre à des scénarios d'exécution permettant de vérifier les exigences établies (cf. § 9.4) ;

4. conduire une vérification modulaire des exigences sur le modèle de l'implémentation en fonction des scénarios d'événements d'ordonnancement (cf. § 9.5).

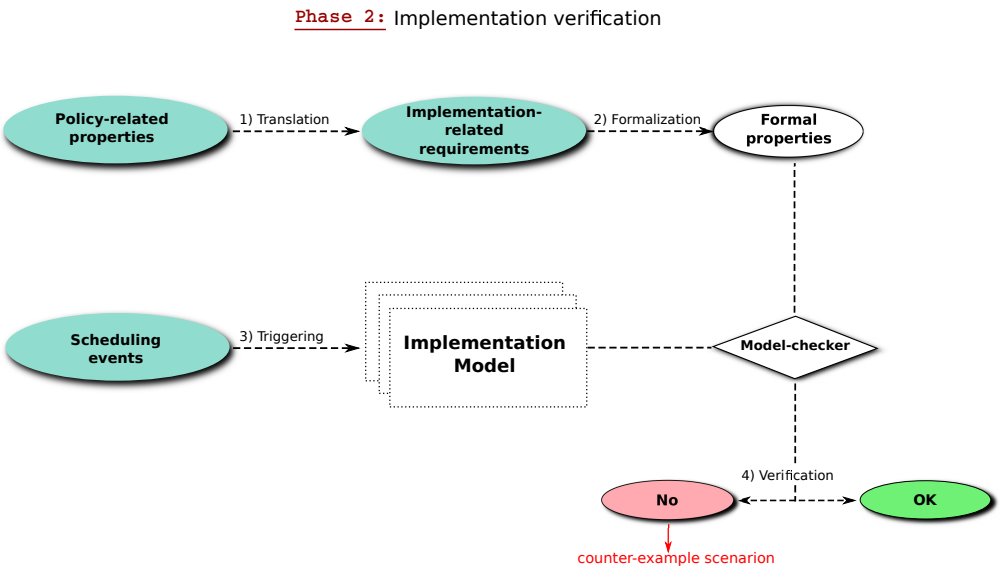


FIGURE 9.1: Les étapes de la deuxième phase de la démarche de vérification des implémentations.

9.2 Spécification des exigences

9.2.1 Prise en compte des contraintes d'implémentation

La vérification de la correction d'un ordonnanceur implémenté correspond à vérifier qu'il produit des décisions d'ordonnancement en accord avec sa spécification telle qu'énoncée dans la littérature. Pour les politiques d'ordonnancement basées sur la priorité, ceci revient à vérifier qu'à chaque instant t, les tâches en cours d'exécution sont les tâches les plus prioritaires selon la définition de la priorité de la politique mise en œuvre. En l'occurrence, pour la politique G-EDF, deux exigences intrinsèques peuvent être énoncées d'après la définition littéraire de l'algorithme EDF établie à l'origine par Liu et Layland [LL73] :

- **exigence de priorité :** à tout instant t, les travaux en cours d'exécution sont ceux qui ont les dates d'échéance les plus proches parmi les travaux prêts.
- **exigence de non oisiveté :** un processeur ne doit jamais être libre tant qu'il y a des travaux prêts.

Les exigences ainsi exprimées concernent spécifiquement ce qui relève du "calcul" de la décision d'ordonnancement et donc du seul module d'ordonnancement. Elles supposent de manière implicite que les données sur lesquelles l'ordonnanceur repose (*e.g.*

états des tâches, dates d'échéance, périodicité des activations, état des processeurs, etc.) sont préalablement fournies. Or, comme la présentation de l'implémentation de G-EDF l'a montré (cf. chapitre 6), la gestion de ces données dans une réelle plateforme n'est pas une hypothèse systématique, elle est plutôt assurée par divers composants de l'OS. En effet, même si les deux exigences intrinsèques de la politique G-EDF sont correctement vérifiées, le résultat de l'ordonnancement produit par l'implémentation peut être faux si le comportement de ces composants est incorrect. Ainsi, certaines situations, pour lesquelles les exigences intrinsèques ne sont pas violées, peuvent démontrer que l'implémentation n'est pas correcte dans le cas où les composants interagissant avec l'ordonnanceur ne fonctionnent pas correctement :

situation 1 : le gestionnaire de tâches ne prévient pas l'ordonnanceur quand un travail vient d'être activé ou de se terminer. Dans une telle situation, aucun ordonnancement n'a lieu étant donné que l'ordonnanceur n'est pas appelé. Le résultat de vérification des deux exigences reste toutefois vrai.

situation 2 : le gestionnaire des listes de tâches ne met pas à jour la liste des travaux prêts lorsque nécessaire conduisant à ce que cette liste ne reflète pas la situation réelle des travaux susceptibles d'être exécutés. Dans une telle situation, même si l'ordonnanceur sélectionne, comme il se doit selon les exigences intrinsèques, les m travaux en tête de liste pour s'exécuter sur les m cœurs libres, des erreurs en terme de séquence d'exécution en résulteront.

situation 3 : le gestionnaire de tâches ne met pas à jour le compteur d'activations d'une tâche après une activation ou une terminaison. Ceci peut conduire soit à une perte d'un travail nouvellement activé, soit à la prise en considération d'un travail qui est supposé être terminé. Dans le premier cas, si le travail est ignoré, il ne sera pas considéré dans le calcul de la décision d'ordonnancement par l'ordonnanceur. Dans le second cas, le travail devant être terminé, peut être sélectionné pour l'exécution par l'ordonnanceur au détriment des autres travaux éligibles.

situation 4 : le gestionnaire de changement de contexte n'applique pas la décision d'ordonnancement calculée par l'ordonnanceur. Ainsi, la séquence d'exécution peut ne pas être conforme à ce que l'ordonnanceur a décidé. En outre, les ré-ordonnancements ultérieurs seront basés sur un état erroné en comparant les échéances absolues des travaux prêts avec celles des travaux qui ne devaient pas être en exécution.

Devant de telles situations, la vérification des exigences intrinsèques de la politique G-EDF ne peut pas être suffisante du moment où l'on ne vérifie pas que l'ordonnanceur dispose des informations nécessaires lui permettant de calculer sa décision (eg. les dates d'échéance des tâches prêtes) et sans tenir compte de la correction du comportement des autres composants. Les exigences à spécifier doivent être ainsi étendues pour englober l'ensemble des composants du périmètre d'ordonnancement tel que nous l'avons défini en § 6.5.1. Nous précisons que nous n'étendons pas la vérification au-delà de ce périmètre car nous partons de l'hypothèse que la correction des autres composants de l'OS, non

liés à l'ordonnanceur, a déjà été établie dans le cadre de la thèse de Toussaint Tigori [BRT18]. Les exigences intrinsèques de la politique G-EDF susmentionnées doivent donc être rapportées au niveau du périmètre d'ordonnancement et tenir compte des spécificités de l'implémentation. Il faut ainsi identifier, pour chacun des composants de ce périmètre, quelles sont ses propres exigences comportementales à vérifier. Pour ce faire, nous analysons dans quelle mesure il doit contribuer à la satisfaction des exigences de G-EDF et identifions ainsi son comportement attendu. Cette analyse est basée également sur une étude rigoureuse du code source de Trampoline et des interactions de ses composants en se focalisant sur les composants du périmètre d'ordonnancement. Ceci nous permet de dresser une liste des exigences étendant celles de la politique G-EDF qui est discutée dans la section qui suit. La vérification de ces exigences est effectuée aux instants d'appel de l'ordonnanceur qui correspondent aux événements d'ordonnancement. En effet, il n'est pas nécessaire de vérifier l'implémentation « à tout instant », où les instants désignent dans notre implémentation les ticks d'horloge, étant donné que les tâches en cours d'exécution ne peuvent changer que lorsqu'un ré-ordonnancement est effectué. En outre, les composants du périmètre d'ordonnancement ne sont exécutés qu'aux instants d'appel de l'ordonnanceur. Ainsi, si leur comportement est vérifié à ces instants, il n'est plus utile de le vérifier ailleurs où ils ne s'exécutent pas.

9.2.2 Les exigences des composants de l'implémentation de G-EDF

Dans notre étude, nous optons pour une démarche modulaire de spécification et de vérification des exigences de l'implémentation. Ceci signifie que pour chaque composant du périmètre d'ordonnancement, nous spécifions les exigences décrivant son comportement et nous les vérifions de manière individuelle. Outre la localisation plus précise des éventuelles erreurs de mise en œuvre de la politique d'ordonnancement, une telle approche favorise sa réutilisation dans le cadre d'implémentation d'autres politiques pouvant partager un même sous-ensemble de composants. Ainsi, dans cette section, les exigences établies pour chaque composant sont présentées.

Le gestionnaire des tâches (*Task Manager*) : afin de s'assurer que ce composant effectue correctement les actions liées aux activations et terminaisons des tâches (cf. § 6.5.2), nous formulons les exigences suivantes :

- quand un travail d'une tâche τ_i est activé :
  - si le compteur d'activation de la tâche τ_i est nul, le travail nouvellement activé doit être inséré dans la `ReadyList`.
  - si le compteur d'activation de la tâche τ_i est nul, l'état de la tâche doit devenir `READY_AND_NEW`.
  - si le compteur d'activation de la tâche τ_i est entre 1 et son nombre d'activations maximum, le travail nouvellement activé doit être inséré dans la `PendingJobList_i`.
  - si le compteur d'activation de la tâche τ_i est égal à son nombre d'activations maximum, le travail nouvellement activé doit être ignoré.

  - si le compteur d'activation de la tâche τ_i est strictement inférieur à son nombre d'activations maximum, il doit être incrémenté.

- quand un travail d'une tâche τ_i est terminé :
  - si le compteur d'activation de la tâche τ_i est égal à 1, l'état de la tâche doit devenir `SUSPENDED`.
  - si le compteur d'activation de la tâche τ_i est strictement supérieur à 1, l'état de la tâche doit devenir `READY`.
  - si le compteur d'activation de la tâche τ_i est strictement supérieur à 1, le travail en attente le plus ancien doit être retiré de la `PendingJobList_i` et mis dans la `ReadyList`.
  - le compteur d'activation de la tâche τ_i doit être décrémenté.

- pour chaque activation ou terminaison de travail d'une tâche τ_i :
  - si le compteur d'activation de la tâche τ_i est strictement supérieur à 1, la taille de sa `PendingJobList_i` doit être égale au compteur d'activation - 1.
  - si la tâche τ_i n'a aucun travail en cours d'exécution ou dans `ReadyList`, sa `PendingJobList_i` doit être vide.
  - l'ordonnanceur doit être appelé.

Le gestionnaire des listes des tâches (*Task List Manager*) : étant donné que l'ordonnanceur est implémenté de manière à ce qu'il sélectionne les m travaux en tête de la `ReadyList`, il est nécessaire de s'assurer que ces travaux sont justement les plus prioritaires. Autrement dit, la `ReadyList` et les `PendingJobLists` doivent être gérées correctement. À cette fin, les exigences suivantes sont vérifiées avant l'appel de l'ordonnanceur :

- pour chaque nœud A (hors nœud racine) de la `ReadyList`, si P est le parent de A, alors : clé(A) $\leq$ clé(P). Rappelons que la clé désigne la date d'échéance liée au nœud en question.
- deux nœuds de la `ReadyList` doivent avoir deux clés différentes.
- la `ReadyList` ne doit pas contenir plus d'un travail de la même tâche.
- l'insertion dans la `ReadyList` d'un travail nouvellement activé doit être faite à la fin de la liste chaînée liée au nœud dont la clé correspond à l'échéance du travail.
- l'insertion dans la `ReadyList` d'un travail préempté doit être faite en tête de la liste chaînée liée au nœud dont la clé correspond à l'échéance du travail.
- l'extraction de la `ReadyList` d'un travail sélectionné par l'ordonnanceur doit être faite en tête de la liste chaînée liée au nœud dont la clé correspond à l'échéance du travail.
- les `PendingJobLists` doivent être triées dans un ordre croissant des dates d'échéance.

Le gestionnaire de temps (*Time Manager*) : ce composant s'occupe du calcul et de la comparaison des dates d'échéance selon l'algorithme ICTOH(cf. § 6.5.3). Sa correction est indispensable à la bonne gestion de la `ReadyList` et des `PendingJobLists` par le gestionnaire des listes de tâches qui se base sur les dates d'échéance des tâches pour

les trier. Ainsi, la correction du fonctionnement de ce composant s'exprime au moyen d'une seule exigence :

- le résultat de comparaison deux dates d'échéance selon l'algorithme ICTOH est conforme avec le résultat issu d'une comparaison de ces deux dates quand elles sont représentées linéairement.

L'ordonnanceur (*Scheduler*) : ses exigences reprennent les exigences intrinsèques de la politique G-EDF :

- à la fin de l'exécution de l'ordonnanceur, les m travaux élus ont toujours une date d'échéance plus petite que celle de tout autre travail dans la `ReadyList` ou dans l'une des `PendingJobLists`.
- une tâche oisive (*idle*) ne doit jamais être élue sur un cœur alors que la `ReadyList` ou une des `PendingJobLists` ne sont pas vides.

Le gestionnaire du changement de contexte (*Context Switch Manager*) : responsable de la mise en application des décisions d'ordonnancement, nous identifions les exigences suivantes pour ce composant :

- le gestionnaire de changement de contexte doit être appelé à chaque fois que l'ordonnanceur indique qu'un changement de contexte est à effectuer.
- le changement de contexte doit être effectué en accord avec les décisions de l'ordonnanceur : sauvegarde du contexte du travail préempté et/ou restauration du contexte du travail élu si nécessaire.

9.3 Formalisation des exigences

Étant donné que la vérification des exigences, présentées dans la section précédente, est envisagée par model-checking, celles-ci doivent être préalablement formalisées sous forme de propriétés afin de pouvoir être vérifiées. Le model-checking offre ainsi deux formalisations possibles :

- énoncer formellement ces propriétés en utilisant une logique temporelle et utiliser un model-checker pour vérifier si elles représentent une tautologie ou non.
- élaborer des automates additionnels ayant un rôle d'observateurs externes en mesure de statuer sur le respect des exigences.

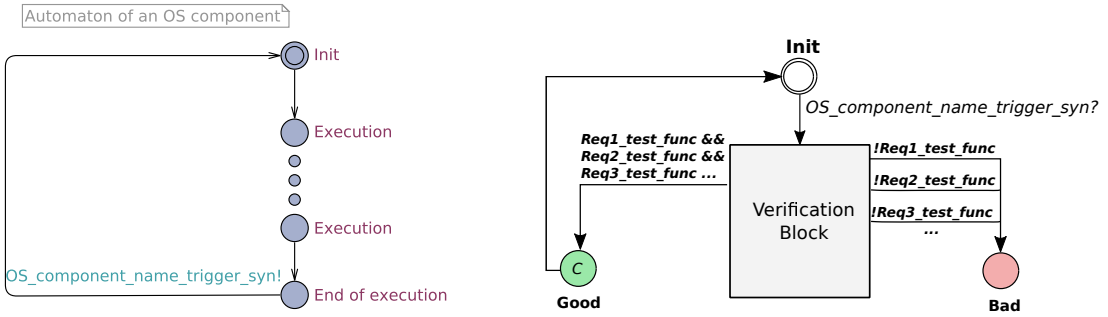
Dans le premier cas, les logiques temporelles CTL et LTL (cf. § 4.3.1) permettent d'exprimer des exigences sous forme de propriétés en UPPAAL. Néanmoins, dans le cas où les exigences à vérifier font intervenir plusieurs paramètres ou variables, elles deviennent rapidement difficiles à exprimer. Leur expression peut également aboutir à des propriétés imbriquées (ie. une propriété exprimée à l'intérieur d'une autre). Or, UPPAAL ne supporte pas ce type de formules.

De l'autre côté, l'expression d'une exigence peut être simple et systématique par le biais d'un observateur. Il peut être modélisé par un automate supplémentaire associé à la vérification d'une ou plusieurs exigences. Cet automate se synchronise de manière non intrusive avec le modèle original sans en altérer le comportement. Informé des évolutions du comportement du système qu'il observe, il évolue lui même en conséquence et l'état qu'il atteint finalement renseigne sur l'(les) exigence(s) en question. Ainsi, la vérification de la satisfaction d'une exigence se transforme-t-elle en un test d'accessibilité d'un état spécifique. Ce processus sera détaillé en section 9.5.

Dans notre travail, nous proposons une approche de vérification basée sur l'utilisation des observateurs. Conformément à notre démarche modulaire, à chaque composant est associé un observateur traduisant l'ensemble de ses exigences. Tous les observateurs utilisés suivent la même structure présentée dans la figure 9.2. Ce sont des automates qui ne contiennent que des états engagés (cf. § 7.2.2) afin de garantir leur exécution avant tout autre automate une fois déclenchés sans compromettre ainsi l'évolution du modèle du système observé.

Fonctionnement d'un observateur dans le modèle : dans un premier temps, l'observateur est en attente dans son état `Init`. Son exécution débute à la réception d'une synchronisation via un canal déclencheur (e.g. `OS_component_name_trigger_syn?`). Cette synchronisation est envoyée par le composant correspondant afin de notifier la fin de son exécution et qu'il est désormais possible de vérifier son comportement. Ceci garantit que l'observateur ne vérifie pas les exigences associées en permanence étant donné que certains états intermédiaires du système ne sont pas significatifs pour lui. En outre, certaines variables et structures de données consultées par l'observateur ne sont mises à jour qu'à la fin d'exécution des fonctions de l'OS. Le processus de vérification est assuré par un bloc de vérification qui comprend une séquence d'appels à des fonctions de test des exigences listées pour le composant en question. Ainsi, chaque exigence est traduite par une fonction de test qui renvoie vrai ou faux selon le résultat de sa satisfaction. Le bloc de vérification possède deux états de sortie : (i) un état bon (`Good`) qui est accessible seulement quand toutes les exigences sont vérifiées avec succès. L'observateur retourne ensuite à son état initial afin d'attendre le prochain déclenchement ; (ii) un état mauvais (`Bad`) qui est atteint si au moins une des exigences n'est pas satisfaite.

Modèle de l'observateur du gestionnaire des tâches : la figure 9.3 illustre le modèle d'observateur utilisé pour la vérification des exigences du composant gestionnaire des tâches (cf. § 9.2.2). L'automate de cet observateur est déclenché à l'issue de l'exécution du dit composant par le biais de deux canaux de synchronisation selon l'événement d'ordonnancement reçu : (i) `end_activate` : si l'événement reçu est une activation d'un nouveau travail ; (ii) `end_terminate` : si l'événement reçu est une terminaison d'un travail. Les exigences du gestionnaire des tâches sont vérifiées à l'aide de trois fonctions de test qui retournent un booléen selon la satisfaction de ces exigences. Ainsi, si le booléen est vrai, l'automate passe à l'état suivant vérifiant l'exigence suivante. L'état `Good` n'est atteint que si les trois exigences sont satisfaites. Sinon, l'automate passe à l'état `Bad` et



(a) Poit de rattachement d'un observateur dans un automate.

(b) Structure générale d'un observateur

FIGURE 9.2: Principe de fonctionnement d'un observateur. `OS_component_name_trigger_syn?` correspond au nom du canal de synchronisation utilisé pour déclencher l'observateur. `Req_i_test_func` correspond au résultat booléen retourné par la fonction de test vérifiant l'exigence i.

y reste bloqué. Les fonctions de test utilisées sont les suivantes :

- `check_activation_requirements` : permet de vérifier les exigences relatives à l'activation d'un nouveau travail. Ainsi, elles sont vérifiées seulement si l'événement d'ordonnancement reçu est une activation ;
- `check_termination_requirements` : permet de vérifier les exigences relatives à une terminaison de travail dans le cas où l'événement d'ordonnancement reçu est une terminaison ;
- `check_scheduling_event_requirements` : permet de vérifier des exigences devant être examinées quel que soit le type d'événement d'ordonnancement reçu.

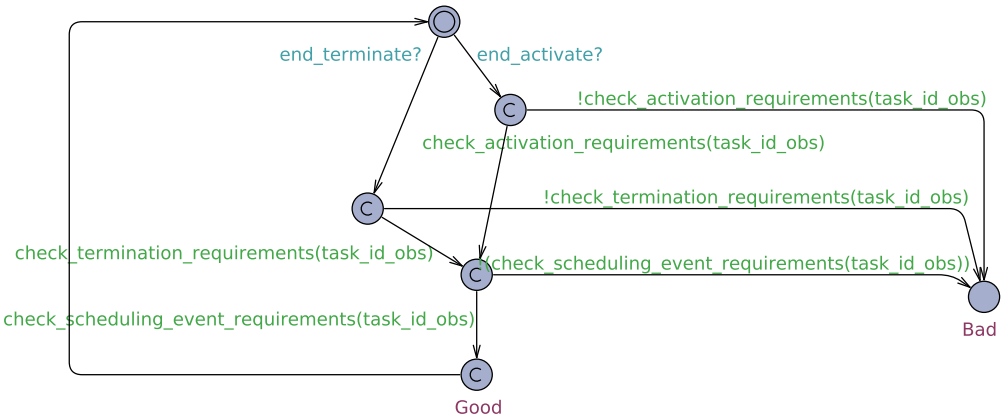


FIGURE 9.3: Le modèle d'observateur du composant gestionnaire des tâches.

9.4 Élaboration d'un modèle d'application

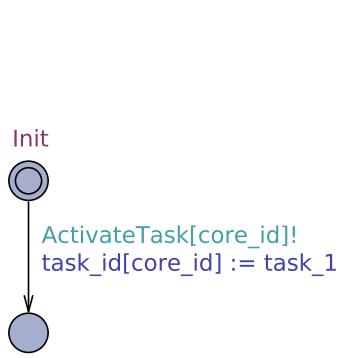
Afin de pouvoir conduire la vérification sur le modèle de l'implémentation, celui-ci doit être excité par des événements d'ordonnancement sollicitant le fonctionnement de l'ordonnanceur. Ce qui constitue la troisième étape de cette phase de vérification. Les événements d'ordonnancement ont le rôle de forcer le modèle à opérer dans certaines situations afin de pouvoir observer sa réaction et ainsi de vérifier les exigences souhaitées. À cette fin, nous fournissons dans un premier temps un modèle d'application permettant de modéliser l'exécution d'un ensemble de tâches et ainsi produire des événements pour exciter le modèle de l'implémentation. Rappelons que notre objectif est de vérifier la cohérence du comportement de l'ordonnanceur implémenté avec le comportement attendu et non pas l'ordonnançabilité d'une application. Ainsi, le modèle d'application que nous proposons dans cette section sert juste à générer les événements nécessaires pour l'excitation d'un ordonnanceur G-EDF, à savoir l'activation et la terminaison d'un ensemble de tâches d'une application.

Ainsi, nous modélisons une application en Trampoline par une partie écrite en langage UPPAAL dans une déclaration générale accessible par tous les automates du système, et des automates finis étendus. La déclaration regroupe les structures de données, les constantes et les variables manipulées par l'application et le système d'exploitation. Elle permet également d'initialiser certaines de ces variables et structures de données (cf. *listing* 9.1). Quant aux automates, il en existe un pour chaque tâche exprimant chacun le comportement de la tâche correspondante après son activation. Un automate supplémentaire est ajouté pour orchestrer les activations. Ces activations peuvent être programmées à l'aide des alarmes qui sont configurées avant le démarrage du système dans le cas de tâches périodiques (cf. Fig. 9.4(b)). D'autres tâches peuvent être activées une seule fois en utilisant un automate faisant directement appel au service `ActivateTask` de l'API (cf. Fig. 9.4(a)). Ceci permet de supporter l'activation périodique et apériodique des tâches.

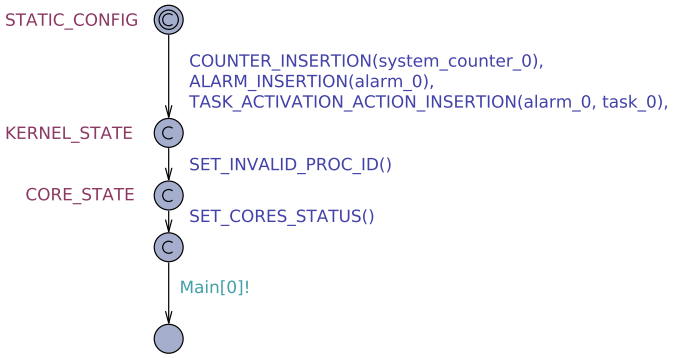
```
/** Task declaration in UPPAAL :
    Static descriptor: task_id, max_activate_count, type, relative deadline, bcet
        , wcet.
    Dynamic descriptor: activate_count, absolute_deadline, state, core_id.
*/
tpl_proc_static s_task_0 = {0, 3, BASIC_TASK, 10, 3, 5};
tpl_proc task_0 = {0, 0, SUSPENDED, -1};
```

Listing 9.1: Exemple d'initialisation des descripteurs statique et dynamique d'une tâche `task_0` dans le modèle.

Pour chacune des tâches, le modèle de son exécution est conduit par le modèle du `Timer` décrit dans la section 8.4. Nous rappelons que le `Timer` représente un automate qui incrémente une variable discrète représentant la date du système en microsecondes. L'incrémentation de cette variable est effectuée à chaque fois que le `Timer` est synchronisé avec un autre automate dans lequel il y a nécessité de faire évoluer le temps. Ceci est le cas de l'automate abstrayant l'exécution d'une tâche illustré dans la figure 9.5. Son exécution



(a) Automate d'activation de la tâche `task_1` sur le coeur `core_id` une seule fois.



(b) Automate permettant l'association de l'activation de la tâche `task_0` avec l'alarme `alarm_0` avant le démarrage de l'OS

FIGURE 9.4: Automates d'activation de tâche

est déclenchée une fois synchronisée avec le `Timer` à travers le canal `MicroSecInc`. Cette synchronisation est contrainte par la fonction `IS_RUNNING()` qui retourne *vrai* seulement quand la tâche passée en argument est élue par l'ordonnanceur pour être exécutée. Elle s'exécute en continuant à se synchroniser en boucle avec le `Timer`, ce qui représente l'écoulement de temps à travers l'incrémentation d'une variable discrète `execution` qui représente la durée d'exécution effective de la tâche. Cette durée doit être comprise entre le meilleur temps d'exécution de la tâche (`task_id_bcet`) et son pire temps d'exécution (`task_id_wcet`) qui sont renseignés dans la déclaration des variables (cf. *listing* 9.1). Quand une tâche est préemptée, l'automate passe à l'état `Preemption` étant donné que le retour de la fonction `IS_RUNNING()` devient *faux*. La terminaison est effectuée via une synchronisation avec le service de terminaison des tâches à travers le canal `TerminateTask`. Elle peut survenir de manière indéterminée entre le BCET et le WCET. Cet automate présente seulement le cycle de vie d'une tâche une fois activée. Comme présenté juste avant, son activation est gérée séparément soit par une alarme si elle est périodique, soit par un appel direct de la fonction `ActivateTask` de l'API.

Notons que le modèle d'application élaboré reste fidèle aux étapes du cycle de vie d'une tâche : activation, exécution, éventuelle préemption, terminaison. Cependant un tel modèle nous permet de vérifier nos exigences seulement par rapport à une configuration de tâches donnée. Il doit ainsi être reconstruit pour chaque nouvelle application à considérer. Comme notre objectif n'est pas d'étudier l'ordonnançabilité d'une application, mais plutôt de vérifier la réaction des composants du périmètre d'ordonnancement face à des événements d'ordonnancement, un tel modèle d'application ne peut offrir une couverture suffisante pour conclure sur la correction de l'implémentation. Ainsi, un mécanisme de génération indéterministe d'événements d'ordonnancement sera proposé (cf. chapitre 10).

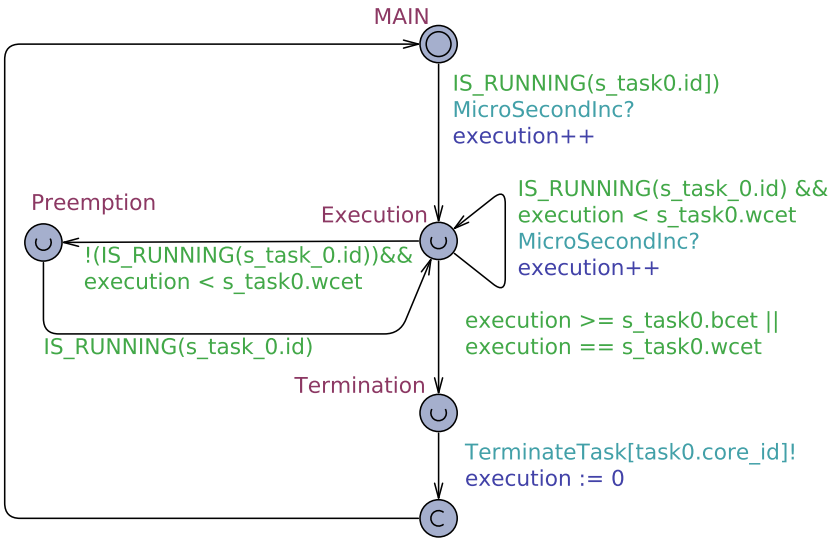


FIGURE 9.5: Modèle d'une tâche

9.5 Processus de vérification

Il s'agit de la quatrième étape de la deuxième phase de notre démarche de vérification. Elle consiste à conduire la vérification en utilisant le model-checker d'UPPAAL permettant d'explorer l'espace d'états de notre système et d'attester si les exigences énoncées sont satisfaites ou non. Pour ce faire, nous formons une chaîne de vérification regroupant les étapes présentées au début de ce chapitre (cf. Fig. 9.6). Ainsi, une combinaison du modèle de l'OS et de celui du `Timer` est formée et produit ce que nous qualifions de *modèle du système*. Ce dernier est stimulé en utilisant les événements d'activation et de terminaison de tâches produits par le modèle d'application. La vérification du modèle est effectuée en examinant la satisfaction des exigences présentées dans la section 9.3 en utilisant les observateurs. Ils peuvent s'exécuter en parallèle avec le modèle complet pour observer son comportement. De cette manière, le modèle du système stimulé par le modèle d'application est combiné avec les modèles d'observateurs et l'ensemble est injecté en entrée du model-checker `VerifyTA` d'UPPAAL. Ce dernier permet de vérifier des propriétés exprimées en logique temporelle et de générer, dans le cas de la non satisfaction d'une propriété, un contre exemple sous forme d'une trace d'exécution.

Pour cela, les exigences sont ramenées à des propriétés de sûreté exprimées sur les états `Bad` des observateurs. Rappelons qu'une propriété de sûreté affirme qu'une situation non désirée ne se produit jamais (cf. § 4.3.1). Se retrouver dans un état `Bad` d'un observateur peut être considéré comme une situation non désirée. Ainsi, les exigences peuvent être vérifiées en s'assurant que le modèle ne se retrouve jamais dans cet état, quel que soit le chemin d'exécution emprunté, ce qui revient à vérifier des propriétés de sûreté. D'un autre côté, il est également important de vérifier que les observateurs passent par leur état `Good` et donc que toutes les exigences sont satisfaites. Ceci permet de vérifier

que l'observateur a bien été appelé et s'est éventuellement exécuté afin d'exclure le cas où l'observateur n'est pas exécuté et donc ne se retrouve pas dans l'état `Bad`. En effet, si l'observateur n'est pas appelé, il reste dans son état initial et ne passe donc jamais à l'état `Bad` même si aucune des exigences n'est vérifiée.

Afin de vérifier ces propriétés de sûreté, nous utilisons des tests d'accessibilité au moyen desquels nous vérifions si les états `Good` et `Bad` des observateurs sont atteignables. Ces tests sont effectués de manière séparée pour chacun des observateurs. Ainsi, l'accessibilité est examinée en utilisant la logique CTL en exprimant deux types de formules logiques sur les états des observateurs.

- `A[] (observer.Good == true)` permet de vérifier que l'état `Good` d'un observateur est atteignable quel que soit le chemin emprunté. Si la propriété n'est pas vérifiée, une trace d'exécution d'un contre exemple la violant est générée ;
- `E<>(observer.Bad == true)` permet de vérifier s'il existe un chemin qui mène vers l'état `Bad` d'un observateur. Si la vérification de cette formule retourne vrai, ceci veut dire qu'au moins une de exigences du composant correspondant à l'observateur n'est pas vérifiée. Dans ce cas, une trace d'exécution du chemin menant vers cet état est générée par le model-checker. Ce qui permet de localiser l'erreur dans le modèle.

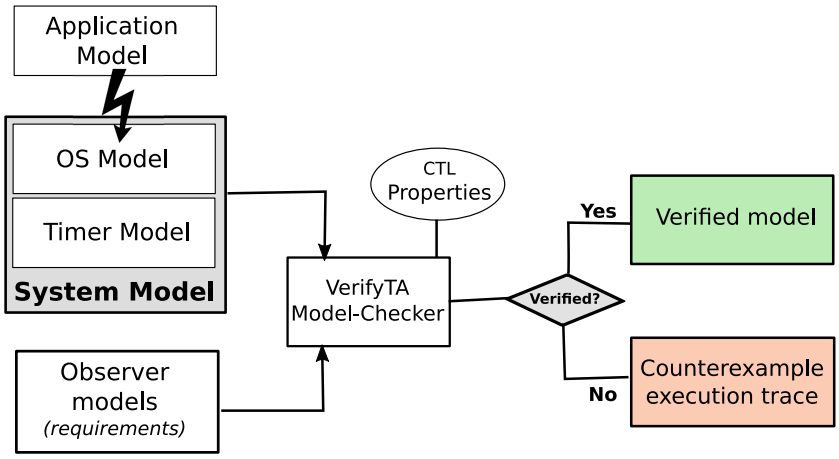


FIGURE 9.6: Le processus de vérification d'une implémentation.

9.6 Résultats de la vérification de l'implémentation de G-EDF

9.6.1 Le jeu d'applications utilisé pour la vérification

Afin de tester la faisabilité de notre approche de vérification dans un premier temps, le modèle de l'implémentation est sollicité par des configurations de tâches différentes afin de donner lieu à divers cas d'appel d'ordonnanceur et de traitement d'événements

d'ordonnancement. Notons que les tâches utilisées sont modélisées chacune par le modèle de la figure 9.5. Les caractéristiques des applications utilisées sont présentées ci-dessous :

- les tâches sont indépendantes, n'utilisent pas de ressources et ne peuvent pas se mettre en attente d'un événement.
- **mode d'activation de tâches :** synchrone et périodique.
- **le choix des périodes des tâches :** chaque tâche τ_i se voit attribuer aléatoirement une période d'activation appartenant à l'ensemble des valeurs suivantes : $T_i \in \{5, 10, 20, 25, 50\}$.
- **le choix des délais critiques des tâches :** les applications utilisées ont des tâches à échéances arbitraires. Ces échéances sont choisies de manière à satisfaire la condition d'ordonnançabilité des applications discutées dans ce qui suit.
- **l'intervalle d'étude de l'application :** la vérification est menée sur un intervalle de temps de 100 unités temporelles, ce qui correspond à l'hyper-période des configurations de tâches utilisées. Le choix de cet intervalle se base sur les résultats présents à ce jour dans la littérature concernant la périodicité de comportement d'un ordonnanceur G-EDF déterministe, sans mémoire et pour une configuration de tâches synchrones et indépendantes [CGG11, GGCG16].
- **l'ordonnançabilité des applications :** comme notre objectif est de vérifier la correction de l'implémentation, et non pas l'ordonnançabilité des applications, les configuration de tâches choisies sont ordonnançables par G-EDF. Pour cela, les durées d'exécution des tâches et les dates d'échéance sont fixées dans leurs descripteurs de manière à satisfaire la condition suffisante d'ordonnançabilité proposée par [BB09] pour un ensemble de tâches à échéances arbitraires. Il s'agit d'un test de densité basé sur le calcul des densité des tâches [1]. Ainsi, ce test indique qu'un ensemble de tâches τ peut être ordonnancé sur un ensemble de m processeurs si :

$$\sum_{i=1}^{n} \delta_i \leq m - (m-1)\delta_{max} \quad \text{où} \quad \delta_i = \frac{C_i}{min\{D_i, T_i\}} \quad \text{et} \quad \delta_{max} = \max_{i=1}^{n}\{\delta_i\} \tag{9.1}$$

Ainsi, en respectant ces caractéristiques, 31 applications ont pu être générées. Pour chaque application, un modèle équivalent a été produit pour vérifier les exigences de l'implémentation :

- 24 applications à exécuter sur deux cœurs avec un nombre de tâches allant de 1 à 24.
- 7 applications à exécuter sur trois cœurs avec un nombre de tâches allant de 1 à 7.

La vérification des exigences de l'implémentation de G-EDF pour ces applications est conduite suivant le processus discuté en § 9.5 sur une machine de 128 Mo de mémoire et 282428 MHz de fréquence de CPU. La vérification prend entre 1 seconde et 22 heures pour un nombre d'états explorés allant de 16884 jusqu'à 334280911 états. Ce nombre augmente exponentiellement avec l'augmentation du nombre de tâches et celui des cœurs

[1] La densité δ_i d'une tâche τ_i est le rapport entre sa durée d'exécution C_i et le minimum entre son échéance relative D_i et sa période T_i

jusqu'à ce que la mémoire de la machine s'épuise pour un nombre de tâches supérieur à 7 s'exécutant sur 3 cœurs (cf. Fig. 9.7). Ceci est dû au problème de l'explosion combinatoire de l'espace d'états explorés.

9.6.2 Les erreurs de l'implémentation détectées

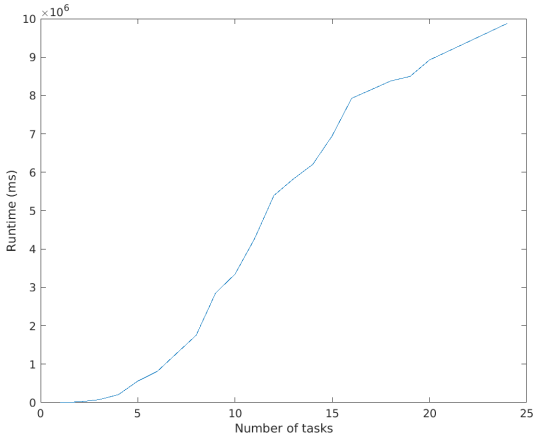
L'approche de vérification conduite sur les applications présentées en § 9.6.1 a permis de relever certaines erreurs d'implémentation dans le code source des fonctions appartenant au périmètre d'ordonnancement. Ces erreurs sont particulièrement liées aux adaptations réalisées pour le passage de l'OS d'un ordonnancement partitionné à un ordonnancement global. Une analyse minutieuse des traces d'exécution de contre-exemple, générées par le model-checker quand une exigence n'est pas satisfaite, a permis de retracer les erreurs et les corriger au niveau du code de l'implémentation. Cette analyse consiste principalement à surveiller le comportement des composants du périmètre d'ordonnancement et l'évolution des variables et structures de données de l'OS en passant d'un état du système à un autre le long la trace d'exécution générée. Les erreurs détectées sont présentées dans ce qui suit avec les scénarios d'exécution qui ont permis leur détection.

Erreur au niveau de l'ordonnanceur : dans la version initiale de Trampoline, celle qui implémente l'ordonnancement partitionné, chaque tâche est statiquement attribuée à un unique cœur pour son exécution. Ainsi, si un travail est préempté, il n'est pas possible qu'il soit sélectionné pour être exécuté sur un autre cœur. Par conséquent, quand un travail se préempte durant un ré-ordonnancement, sa remise dans la liste des travaux prêts n'a lieu qu'à l'issue du changement de contexte qui suit ce ré-ordonnancement, puisque ce travail ne peut pas prétendre à être à nouveau immédiatement élu.

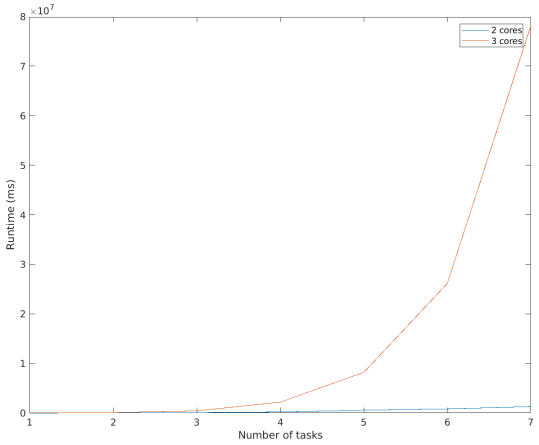
En passant à l'ordonnancement global avec G-EDF, certaines séquences d'appel des fonctions de l'OS ont été préservées. Notamment celle concernant la remise dans la `ReadyList` d'une tâche préemptée. En conséquence, ceci a conduit à l'erreur suivante : une tâche τ_i qui est préemptée sur un cœur C_i peut avoir une priorité plus élevée qu'une tâche τ_j qui s'exécute sur un cœur C_j. De la même manière, la tâche τ_i pourrait également être éligible pour exécution sur un cœur C_j sur lequel une tâche τ_j vient de se terminer. Toutefois, si la tâche τ_i n'est remise dans la `ReadyList` qu'à l'issue de tous les ré-ordonnancements prévus à l'instant de sa préemption, l'ordonnanceur ne peut pas la considérer dans son calcul d'ordonnancement. Il en résulte qu'une tâche moins prioritaire que τ_i pourrait s'exécuter à sa place.

L'erreur susmentionnée a été détectée à l'aide de l'observateur destiné à la vérification des exigences de l'ordonnanceur (cf. § 9.2.2) avec le scénario d'exécution suivant : 3 tâches notées τ_0, τ_1 et τ_2 devant s'exécuter sur deux cœurs C_0 et C_1. Chaque tâche τ_i a une échéance relative D_i. L'ordre des opérations conduisant à l'erreur est présenté comme suit :

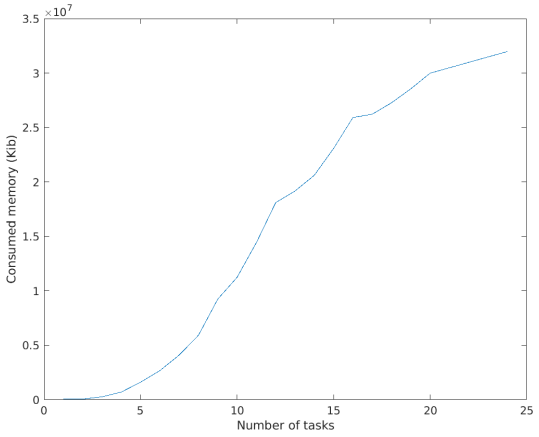
- à t_1, τ_1 et τ_2 sont activées. Leurs dates d'échéance sont respectivement $d_1 = D_1 + t_1$ et $d_2 = D_2 + t_1$ tel que $d_1 < d_2$. τ_1 est sélectionnée pour être exécutée sur le cœur C_0 et τ_2 sur C_1.



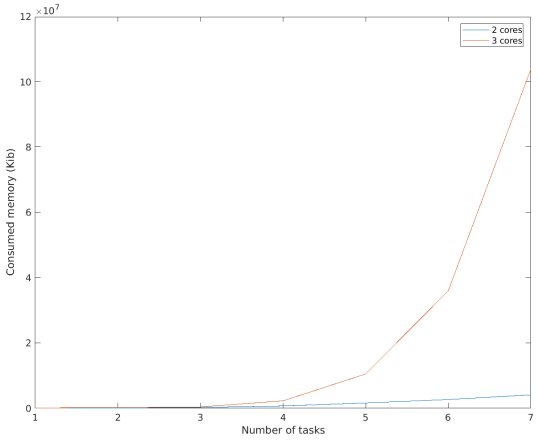
(a) Durée de vérification pour les applications sur deux cœurs.



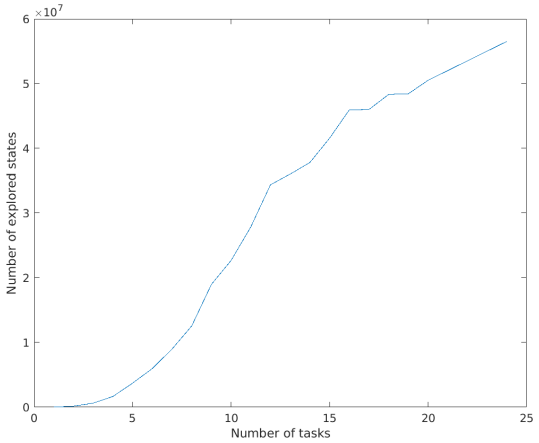
(b) Durée de vérification pour les applications sur 3 cœurs.



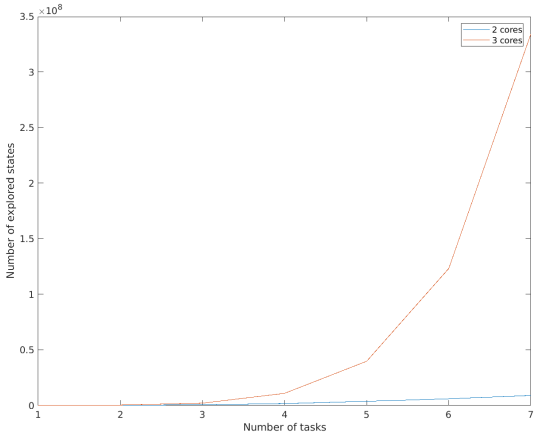
(c) Consommation en mémoire pour les applications sur deux cœurs.



(d) Consommation en mémoire pour les applications sur 3 cœurs.



(e) Nombre d'états explorés pour les applications sur deux cœurs.



(f) Nombre d'états explorés pour les applications sur 3 cœurs.

FIGURE 9.7: Les performances de la vérification

- à $t_2 > t_1$, τ_0 est activée et sa date d'échéance est calculée : $d_0 = D_0 + t_2$ tel que $d_0 < d_1 < d_2$. L'implémentation de l'ordonnanceur fait que ce dernier compare les dates d'échéance des tâches en cours d'exécution avec celles des tâches prêtes en traitant cœur par cœur. Ainsi, étant donné que $d_0 < d_1$, τ_1 est préemptée et τ_0 est sélectionnée à sa place pour s'exécuter sur le cœur C_0. La tâche τ_1 n'est pas remise dans la `ReadyList` ce qui implique que τ_2 continue son exécution sur le cœur C_1.
- à la fin du ré-ordonnancement et du changement de contexte la tâche τ_1 est remise dans la `ReadyList`.

Ainsi, le scénario présenté viole l'exigence de priorité selon la politique G-EDF puisqu'à l'issue du ré-ordonnancement, la tâche τ_2 est en cours d'exécution alors que la tâche τ_1 qui a une date d'échéance plus proche est dans la `ReadyList`.

Erreur au niveau du gestionnaire de changement de contexte : comme nous l'avons indiqué préalablement, dans la version d'ordonnancement partitionné de Trampoline, une tâche est assignée statiquement à un cœur. L'identifiant de ce cœur est initialisé au moyen d'une fonction de la couche API de Trampoline qui n'appartient pas au périmètre d'ordonnancement. Dans le cas particulier de l'activation des tâches `AUTOSTART` (des tâches activées avec le démarrage de l'OS), le gestionnaire de changement de contexte récupère l'identifiant du cœur initialisé et charge le contexte de la tâche qui y est attribuée. En passant à l'implémentation de G-EDF, pour la même raison qu'au dessus à savoir certains appels de fonctions hors du périmètre d'ordonnancement ont été conservés, il en résulte alors que le gestionnaire de changement de contexte récupère le mauvais identifiant du cœur auprès de la fonction d'initialisation, en ignorant celui indiqué par l'ordonnanceur. Ceci conduit à deux erreurs potentielles :

- exécuter une tâche sur un cœur autre que celui indiqué par l'ordonnanceur ;
- exécuter une tâche non prioritaire, et donc qui n'a pas été sélectionnée par l'ordonnanceur, sur le cœur dont l'identifiant est indiqué dans la fonction de l'API.

Ces erreurs ont été soulevées grâce à l'observateur du gestionnaire de changement de contexte et celui de l'ordonnanceur sous le scénarios décrit dans le paragraphe précédent.

D'autres erreurs mineures : certaines erreurs minimes ont également été identifiées durant le processus de vérification telles que :

1. sauvegarder le contexte d'une tâche terminée, et qui pourtant ne nécessite pas de sauvegarde, en appelant la mauvaise fonction du gestionnaire de changement de contexte.
2. demander la restauration du contexte d'une tâche nouvellement activée, ce qui est une conséquence de l'erreur 1.
3. ne pas mettre à jour la taille d'une `PendingJobList` après avoir enlevé un travail en attente.

9.7 Conclusion

La démarche de vérification conduite pour G-EDF a montré des résultats intéressants et a permis de détecter certaines erreurs dans l'implémentation de la politique. Toutefois, le nombre de situations auxquelles l'implémentation est confrontée reste limité et très dépendant du modèle d'application utilisé. Ceci peut remettre en cause la couverture du processus de vérification mené puisque la correction de l'implémentation est confirmée uniquement pour les applications considérées et face aux événements que ces dernières génèrent de manière déterministe. Une autre limite du modèle présenté dans ce chapitre est la nécessité de reconstruire un nouveau modèle d'application pour chaque configuration de tâches choisie et de le raccrocher au modèle du système d'exploitation. Pour cela, nous souhaitons proposer d'autres modèles offrant plus d'événements d'ordonnancement et pouvant mettre le modèle d'une implémentation face à diverses situations générées de manière indéterministe. Ces modèles seront utilisés pour la vérification du modèle de l'implémentation de la politique EDF-US[ξ] pour laquelle nous souhaitons également conduire notre démarche de vérification. Ceci fera ainsi l'objet du chapitre suivant.

CHAPITRE 10

Modèles d'excitation pour la vérification

10.1 Introduction

Le processus de vérification d'une implémentation présenté dans le chapitre précédent ne peut être conduit sans que le modèle à vérifier soit stimulé par des événements d'ordonnancement. Ceci permet d'observer par la suite la réaction du modèle face aux situations produites par les événements d'ordonnancement. Pour cela, nous avons élaboré, dans un premier temps, un modèle d'application décrivant le comportement d'un système de tâches et permettant de solliciter le modèle d'implémentation par des événements d'ordonnancement produits durant le cycle de vie de ces tâches.

Le modèle d'application proposé présente toutefois des inconvénients. D'une part, le comportement du modèle est très déterministe puisqu'il décrit de manière fidèle le comportement d'une tâche durant sa vie : activation périodique ou non, synchrone ou non, exécution et terminaison selon des attributs de la tâche renseignés dans son descripteur, etc. Ceci limite les scénarios d'événements d'ordonnancement produits et les situations auxquelles l'ordonnanceur peut être confronté. Ainsi, la vérification de l'implémentation est dépendante de l'application modélisée. Sa correction est attestée pour cette application et pour les événements qu'elle génère de manière déterministe et le modèle de l'application doit être reconstruit à chaque fois que le système de tâches est modifié pour avoir d'autres scénarios d'événements d'ordonnancement.

Afin de remédier à cela, nous proposons d'autres modèles permettant de générer de manière indéterministe tous les scénarios possibles d'activation et d'exécution d'un ensemble de tâches sur une durée de vérification déterminée. L'indéterminisme de cette solution vient du fait que, pour un nombre de tâches donné, les événements d'ordonnancement peuvent se produire à n'importe quel moment et dans n'importe quel ordre. Ceci est important pour vérifier la réaction de l'implémentation face à un large nombre de situations possibles et pouvoir attester de sa correction. Ces modèles permettent de sur-

monter les limites du modèle d'application proposé en § 9.4, dans lequel les événements ont lieu à des instants précis, dans un nombre de situations précis et limité, et dans un ordre déterminé en fonction de la périodicité et les durées d'exécution des tâches.

Les modèles que nous proposons dans ce chapitre sont appelés des « *moteurs de vérification* ». L'idée est de générer, pour un nombre n de tâches donné, toutes les séquences possibles d'appels d'ordonnanceur et d'exécution des tâches sur un intervalle donné. Cet intervalle est appelé « *Verification Time Interval (VTI)* ». L'évolution du temps dans ces moteurs est assurée grâce à une variable d'horloge continue pouvant prendre un nombre infini de valeurs dans l'intervalle susmentionné. À l'inverse du modèle d'application proposé en § 9.4 dans lequel l'écoulement de temps est assuré par une variable discrète ne pouvant prendre que des valeurs entières. Le choix d'une variable continue permet d'élargir l'étude et d'offrir plus d'entrelacements concernant l'arrivée des événements d'ordonnancement. En effet, au lieu d'avoir des événements arrivant à des dates précises, les moteurs de vérification génèrent des événements pouvant arriver à n'importe quelle date dans l'intervalle susmentionné.

Nous distinguons deux types de moteurs de vérification dont le fonctionnement est présenté dans la suite de ce chapitre :

- **moteurs d'activation :** permettant de générer des scénarios d'activation de tâches (cf. § 10.2) ;
- **moteurs d'exécution :** permettant de générer des scénarios d'exécution et de terminaison de tâches.

10.2 Les moteurs d'activation

Ces moteurs sont des générateurs indéterministes de scénarios d'activation d'un ensemble de tâches. Ils sont élaborés par des automates temporisés permettant de générer des événements d'activation pouvant avoir lieu à n'importe quel moment et dans n'importe quel ordre dans l'intervalle de vérification spécifié.

Ce comportement est assuré par le biais de deux automates temporisés assurant le contrôle de l'activation d'un ensemble de n tâches :

1. `activate_once` : cet automate permet d'activer au plus un travail pour chacune des tâches dans l'intervalle de vérification. L'activation de ces travaux peut se produire ou non. Les activations peuvent avoir lieu de manière indéterministe dans n'importe quel ordre. La figure 10.1 illustre un exemple de moteur d'activation utilisé pour activer au maximum deux tâches `task_0` et `task_1` ($n = 2$) dans un intervalle de temps dont la valeur est stockée dans une variable globale *VTI*. L'écoulement du temps dans l'automate est géré par une variable d'horloge `timer_c` qui évolue de manière continue. Initialement, les deux tâches se trouvent dans l'état `SUSPENDED`. Depuis l'état `Main`, l'automate peut choisir ou non de prendre l'une des transitions menant vers les états `I0` ou `I1` afin d'activer respectivement la tâche `task_0` ou `task_1`. Ces transitions peuvent être prises à tout moment tant que la variable d'horloge `timer_c` n'a pas

atteint la valeur de VTI. Une fois que le temps de vérification est expiré, l'automate passe à son état final `END`. Cet automate permet de générer plusieurs combinaisons d'activation permettant également de couvrir des applications avec des tâches ayant des dates d'échéance identiques ou différentes. Ceci mène à solliciter l'ordonnanceur pour différentes situations de comparaison entre les dates d'échéance des tâches. Afin de mieux illustrer ces combinaisons, nous considérons l'exemple de deux tâches `task_0` et `task_1` ayant respectivement des échéances relatives de D_0 et $D_1 = D_0 - \delta$ tel que $0 \leq \delta < D_0$. Plusieurs scénarios d'activations sont possibles avec le modèle de la figure 10.1, parmi lesquels nous citons :

scenario 1 : aucune des deux tâches n'est activée ;

scenario 2 : une seule des deux tâches est activée à un instant $t \in [0, VTI]$ avec une date d'échéance $d_i = D_i + t$. La valeur de t correspond à la valeur de la variable d'horloge `timer_c` lorsque la transition de l'activation est franchie ;

scenario 3 : `task_0` et `task_1` sont activées simultanément à un instant $t \in [0, VTI]$ avec des dates d'échéance respectives $d_0 = D_0 + t$ et $d_1 = D_0 - \delta + t$. Ainsi, `task_0` est moins prioritaire que `task_1` ;

scenario 4 : `task_0` et `task_1` sont activées respectivement à deux instants différents : $t0$ et $t1 = t0 + \delta$ tels que $t0, t1 \in [0, VTI]$. Ainsi, les deux tâches ont la même date d'échéance $d_0 = d_1 = D_0 + t0$.

scenario 5 : `task_0` et `task_1` sont activées respectivement à deux instants différents : $t0$ et $t1 > t0 + \delta$ tels que $t0, t1 \in [0, VTI]$. La valeur de t_i correspond à la valeur de `timer_c` lorsque la transition menant à l'activation de la tâche `task_i` est franchie. Dans ce scénario, les deux tâches ont deux dates d'échéance différentes $d_0 = D_0 + t0$ et $d_1 = D_0 + t1$ avec $d_0 < d_1$ ce qui implique que `task_0` est plus prioritaire que `task_1`.

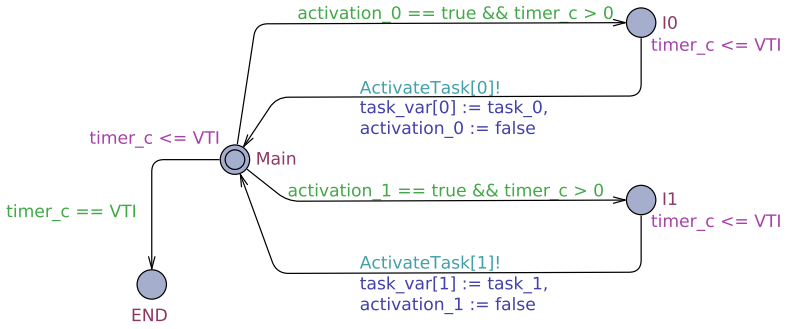


FIGURE 10.1: Modèle du moteur d'activation `activate_once`. Ce moteur permet d'activer chaque tâche une seule fois. Pour cela un booléen `activation_i` est utilisé pour contrôler l'activation de la tâche `task_i`. Sa valeur est est initialisée à vrai. Lorsque la tâche correspondante est activée, sa valeur passe à faux afin d'empêcher les activations futures.

2. `activate_several` : cet automate permet d'activer plusieurs travaux d'une même tâche tant que son compteur d'activations le permet. Le même principe de fonctionnement est conservé : les activations de tâches peuvent arriver de manière indéterministe et dans n'importe quel ordre dans un intervalle de temps donné. Cet automate couvre également certains scénarios offerts par le moteur `activate_once`. La figure 10.2 illustre un exemple d'un moteur d'activation permettant d'activer dans un ordre aléatoire plusieurs travaux de deux tâches `task_0` et `task_1` dans l'intervalle de temps borné par *VTI*. Cet automate a l'avantage de couvrir toutes les possibilités d'activations pour chacune des tâches : périodique, sporadique et apériodique. Parmi les scénarios d'activations possibles, nous citons :

   scénario 1 : seulement les travaux de `task_0` sont activés à des instants $t_1, t_2, ..., t_{max}$ tel que $0 \leq t_1 < t_2 < ... < t_max \leq VTI$. Ces instants peuvent être espacés d'une même durée, ce qui produit un comportement périodique de la tâche, ou de durées différentes, ce qui représente un comportement sporadique.

   scénario 2 : un seul travail de chaque tâche est activé avec des dates d'échéance identiques ou différentes (cf. scénario 4 et 5 du moteur `activate_once`).

   scénario 3 : plusieurs travaux de chaque tâche sont activés avec des combinaisons différentes des dates d'échéance intégrant celles détaillées dans les scénarios 4 et 5 du moteur `activate_once`.

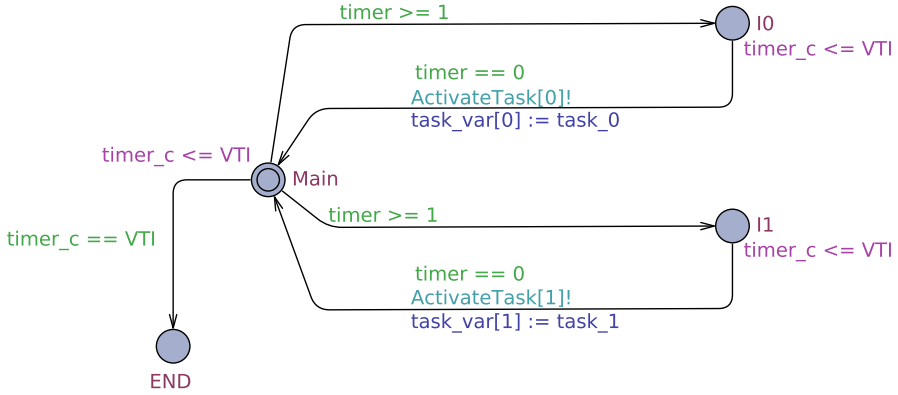


FIGURE 10.2: Modèle du moteur d'activation `activate_several`.

Notons que `activate_several` ne permet pas des activations simultanées de deux travaux. En effet, deux activations successives sont espacées au moins d'une unité de temps. Ceci est assuré par une deuxième variable d'horloge `timer` qui évolue de manière circulaire entre 0 et 1 unité de temps grâce au modèle du `Timer` présenté dans la section 8.4. L'intégration de cette contrainte était nécessaire afin d'éviter l'appel du service d'activation `ActivateTask` infiniment à un instant donné. Rappelons que les fonctions du noyau s'exécutent en temps nul du fait que tous les états des automates modélisant le noyau sont urgents (cf. § 7.4.4). Ainsi, un comportement possible de cet automate serait

d'appeler le service `ActivateTask` de manière infinie sans que le temps s'écoule. Ainsi, afin de pouvoir couvrir également le cas des activations synchrones de tâches, il est possible de combiner le moteur d'activation `activate_several` avec celui de `activate_once` étant donné que ce dernier permet les activations simultanées des tâches.

10.3 Les moteurs d'exécution

Ces moteurs correspondent à des automates permettant de décrire un comportement indéterministe d'une tâche pendant son exécution. Notre objectif n'est pas de fournir un modèle décrivant avec précision l'exécution d'une tâche, mais de développer un modèle couvrant le plus grand nombre possible de scénarios d'exécution. À cette fin, chaque tâche est abstraite par un automate temporisé qui regroupe tous les états dans lesquels une tâche basique peut se retrouver durant son cycle de vie (cf. Fig. 10.3). Le changement d'état de la tâche est enregistré dans son descripteur dynamique. Ainsi, quand une tâche n'est pas encore activée, elle est dans l'état `SUSPENDED` et l'automate d'exécution dans son état `Main`. Lorsque la tâche est activée par l'un des moteurs d'activation, son état devient `READY` et son travail est mis dans l'une des listes de tâches, mais l'automate reste toujours dans l'état `Main`. Une fois que la tâche est sélectionnée par l'ordonnanceur, son état devient `RUNNING` et la fonction `IS_RUNNING()` retourne vrai. Quand elle s'exécute, tous les scénarios dont la durée d'exécution n'excède pas le temps de vérification *VTI* sont traités par le modèle. Cette exécution est modélisée par la progression continue de la variable d'horloge `timer_c` dans l'automate jusqu'à ce que la tâche se termine ou que la variable atteigne la valeur *VTI*. Autrement dit, la tâche peut s'exécuter pour une durée indéterministe tant que *VTI* n'est pas atteint. Quand elle est préemptée, la fonction `IS_RUNNING()` retourne faux et l'état de la tâche dans son descripteur redevient `READY`. La terminaison peut se produire à n'importe quel moment avant l'expiration du *VTI*. Pour ce faire, l'automate passe à l'état `Termination` en utilisant une transition qui est gardée par la fonction `IS_RUNNING()` et appelle ensuite le service `TerminateTask`. Cette garde est utilisée pour éviter d'appeler le service lorsque la tâche n'est pas en cours d'exécution ou pendant qu'elle est préemptée. L'exécution peut être effectuée dans le meilleur des cas en temps nul, et dans le pire des cas pour une durée égale au *VTI*. Tous les scénarios intermédiaires sont également traités, ce qui permet d'élargir le champ de la vérification et de vérifier plusieurs configurations de tâches. Ainsi, en combinant ces moteurs avec ceux d'activation, les événements d'ordonnancement sont générés de manière aléatoire mettant l'implémentation de l'ordonnanceur face à différentes situations.

10.4 Utilisation des moteurs de vérification pour la vérification d'une implémentation de EDF-US

10.4.1 Introduction

Dans cette section nous proposons de conduire le processus de vérification présenté dans le chapitre précédent sur le modèle de l'implémentation de la politique EDF-US[ξ] (cf.

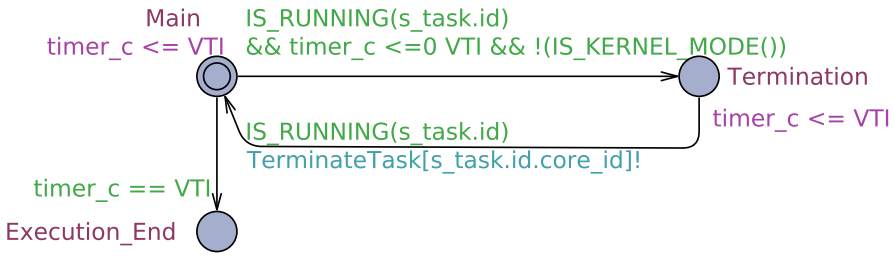


FIGURE 10.3: Modèle d'exécution d'une tâche.

§ 8.3). Ainsi, la vérification est réalisée en examinant un ensemble d'exigences élaborées en fonction du comportement que l'on attend de l'implémentation (cf. § 10.4.2). Dans ce contexte, nous souhaitons examiner l'utilisation des moteurs de vérification discutés en § 10.2 et § 10.3 dans le cadre du processus de vérification mené sur le modèle de l'implémentation d'EDF-US[ξ]

10.4.2 Les exigences des composants de l'implémentation d'EDF-US[ξ]

La vérification du modèle élaboré pour la politique EDF-US[ξ] est conduite en suivant le processus de vérification présenté dans la section 9.5. Ainsi, dans un premier temps, des exigences décrivant le comportement attendu de l'implémentation de la politique sont élaborées pour chaque composant du périmètre d'ordonnancement. Elles sont ensuite traduites sous forme d'observateurs sur lesquels l'accessibilité de leurs états est examinée pour vérifier les exigences. Dans cette section, nous présentons l'ensemble des exigences élaborées pour la vérification d'une implémentation EDF-US[ξ] au sein de Trampoline.

Rappelons que le fonctionnement du gestionnaire de temps et celui du gestionnaire de changement de contexte ne changent pas par rapport à l'implémentation de G-EDF (cf. § 8.3.2). Ainsi, leurs exigences sont conservées pour la vérification d'EDF-US[ξ].

Le gestionnaire des tâches (*Task Manager*) : pour ce composant, les exigences relatives aux opérations d'insertion/extraction dans les listes de travaux prêts pour G-EDF sont adaptées afin de prendre en considération le poids des tâches. Les autres exigences sont conservées à l'identique.

- quand un travail d'une tâche τ_i est activé :
  - si la tâche τ_i est légère et son compteur d'activation est nul, le travail nouvellement activé doit être inséré dans la `ReadyList`.
  - si la tâche τ_i est lourde et son compteur d'activation est nul, le travail nouvellement activé doit être inséré dans la `HeavyList`.
  - si le compteur d'activation de la tâche τ_i est nul, l'état de la tâche doit devenir `READY_AND_NEW`.
  - si le compteur d'activation de la tâche τ_i est entre 1 et son nombre d'activations maximum, le travail nouvellement activé doit être inséré dans la `PendingJobList_i`.

  - si le compteur d'activation de la tâche τ_i est égal à son nombre d'activations maximum, le travail nouvellement activé doit être ignoré.
  - si le compteur d'activation de la tâche τ_i est strictement inférieur à son nombre d'activations maximum, il doit être incrémenté.

- quand un travail d'une tâche τ_i est terminé :
  - si le compteur d'activation de la tâche τ_i est égal à 1, l'état de la tâche doit devenir `SUSPENDED`.
  - si le compteur d'activation de la tâche τ_i est strictement supérieur à 1, l'état de la tâche doit devenir `READY`.
  - si la tâche τ_i est légère et son compteur d'activation est strictement supérieur à 1, le travail en attente le plus ancien doit être retiré de la `PendingJobList_i` et mis dans la `ReadyList`.
  - si la tâche τ_i est lourde et son compteur d'activation est strictement supérieur à 1, le travail en attente le plus ancien doit être retiré de la `PendingJobList_i` et mis dans la `HeavyList`.
  - le compteur d'activation de la tâche τ_i doit être décrémenté.

- pour chaque activation ou terminaison de travail d'une tâche τ_i :
  - si le compteur d'activation de la tâche τ_i est strictement supérieur à 1, la taille de sa `PendingJobList_i` doit être égale au compteur d'activation - 1.
  - si la tâche τ_i n'a aucun travail en cours d'exécution ou dans `ReadyList` ou dans la `HeavyList`, sa `PendingJobList_i` doit être vide.
  - l'ordonnanceur doit être appelé.

Le gestionnaire des listes des tâches (*Task List Manager*) : pour ce composant, certaines exigences élaborées pour l'implémentation de G-EDF sont conservées à l'identique. Nous considérons d'autres exigences permettant d'assurer la bonne gestion de la `HeavyList` et en prenant en considération de nouvelles contraintes pour la `ReadyList` :

- pour chaque nœud A (hors nœud racine) de la `ReadyList`, si P est le parent de A, alors : clé(A) $\leq$ clé(P). Rappelons que la clé désigne la date d'échéance liée au nœud en question.
- deux nœuds de la `ReadyList` doivent avoir deux clés différentes.
- la `ReadyList` ne doit pas contenir plus d'un travail la même tâche.
- la `ReadyList` ne doit contenir aucun travail d'une tâche lourde.
- l'insertion dans la `ReadyList` d'un travail d'une tâche légère nouvellement activé doit être faite à la fin de la liste chaînée liée au nœud dont la clé correspond à l'échéance du travail.
- l'insertion dans la `ReadyList` d'un travail préempté d'une tâche légère doit être faite en tête de la liste chaînée liée au nœud dont la clé correspond à l'échéance du travail.
- l'extraction de la `ReadyList` d'un travail d'une tâche légère sélectionné par l'ordonnanceur doit être faite en tête de la liste chaînée liée au nœud dont la clé correspond à l'échéance du travail.

- la `HeavyList` ne doit contenir aucun travail d'une tâche légère.
- la `HeavyList` ne doit pas contenir plus d'un travail de la même tâche.
- l'insertion dans la `HeavyList` d'un travail nouvellement activé d'une tâche lourde doit être faite à la fin de la FIFO qui l'implémente.
- l'extraction de la `HeavyList` d'un travail d'une tâche lourde sélectionné par l'ordonnanceur doit être faite en tête de la FIFO qui l'implémente.
- les `PendingJobLists` doivent être triées dans un ordre croissant des dates d'échéance.

L'ordonnanceur (*Scheduler*) : pour ce composant, nous adaptons les exigences élaborées pour l'implémentation de G-EDF afin de prendre en compte le poids des tâches dans la décision d'ordonnancement :

- à la fin de l'exécution de l'ordonnanceur, si la `HeavyList` est non vide, aucun travail d'une tâche légère ne doit être en cours d'exécution.
- à la fin de l'exécution de l'ordonnanceur, les travaux de tâches légères en cours d'exécution ont toujours une date d'échéance plus petite que celle de tout autre travail léger dans la `ReadyList` ou dans l'une des `PendingJobLists`.
- une tâche oisive (*idle*) ne doit jamais être élue pour l'exécution sur un cœur alors que la `ReadyList`, la `HeavyList` ou une des `PendingJobLists` ne sont pas vides.

10.4.3 Résultats de la vérification de l'implémentation d'EDF-US[ξ]

Choix des scénarios pour la vérification de l'implémentation : tel que nous l'avons expliqué avant, notre travail se focalise sur la correction fonctionnelle de l'implémentation. Le comportement temporel étant ignoré, l'objectif est donc de vérifier la réaction de l'implémentation face à différentes situations. Ainsi, il n'est pas pertinent dans notre étude de vérifier le comportement de l'ordonnanceur vis-à-vis d'une configuration de tâche donnée, mais plutôt de diversifier les situations auxquelles il doit faire face.

Pour une implémentation d'EDF-US[ξ], le processus de stimulation d'ordonnanceur nécessite de mettre l'accent sur certaines situations significatives pour la politique : gestion de dates d'échéance identiques ou différentes, gestion de tâches lourdes ou légères, gestion de plusieurs travaux actifs d'une même tâche, etc. Les moteurs de vérification présentés dans les sections 10.2 et 10.3 s'occupent respectivement de la génération indéterministe des événements d'activation et de terminaison de N tâches au plus sur un intervalle de temps VTI [1]. Il est nécessaire d'accompagner ces moteurs par un choix de certains attributs de tâches afin de permettre la génération d'événements d'ordonnancement dans des situations différentes. Nous discutons dans la suite la manière avec laquelle nous choisissons certains attributs des tâches pour lesquels les moteurs de vérification génèrent les événements d'ordonnancement.

- **Le poids des tâches :** cet attribut est à enregistrer dans le descripteur statique des tâches. Chaque ensemble de N tâches considéré est composé de 50% de tâches légères

[1] Rappelons que le nombre de tâches et le VTI sont à déterminer par l'utilisateur dans le modèle

et 50% de tâches lourdes. Quand N est impair, nous favorisons les tâches légères étant donné que la compétition entre elles se résout selon G-EDF, tandis que la compétition entre les tâches lourdes est gérée de manière arbitraire. Ceci permet ainsi de vérifier les exigences de l'ordonnanceur relatives à la politique G-EDF.

- **Le délai critique des tâches :** pour les tâches lourdes, leurs dates d'échéance ne déterminent pas leur priorité. Par conséquent, le choix du délai critique à attribuer est uniquement utile pour les tâches légères. Nous choisissons ainsi d'attribuer la même valeur de délai critique à toutes les tâches légères. En effet, tel que nous l'avons expliqué dans les scénarios d'activation dans la section 10.2, les entrelacements des instants d'activation couverts par les deux moteurs d'activation conduisent à des situations variées en terme de positionnement des dates d'échéance des tâches lorsque les délais critiques de celles-ci sont identiques. Afin de mieux illustrer ce phénomène, nous prenons l'exemple présenté dans la figure 10.4. Nous considérons dans cet exemple deux tâches légères τ_0 et τ_1 ayant le même délai critique. Avec ces deux tâches, le moteur d'activation `activate_once` assure au moins trois scénarios mettant l'ordonnanceur face à trois situations possibles :
  - activation simultanée des deux tâches et donc leurs dates d'échéance coïncident.
  - τ_0 est activée avant τ_1. Ainsi, sa date d'échéance est plus petite que celle de τ_1.
  - τ_1 est activée avant τ_0. Ainsi, sa date d'échéance est plus petite que celle de τ_0.

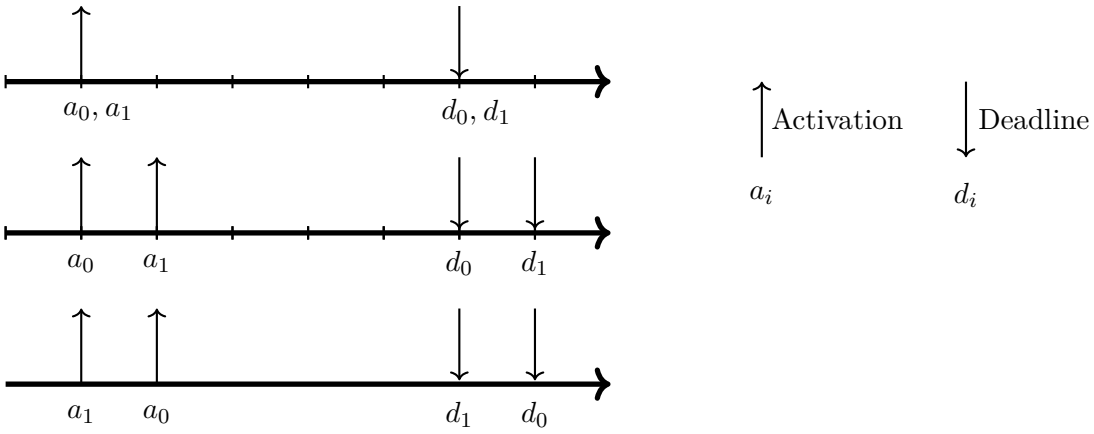


FIGURE 10.4: Exemple de scénarios d'activation générés par le moteur d'activation `activate_once`.

- **Le nombre d'activations maximum des tâches :** le choix de la valeur du nombre d'activations des tâches est essentiel pour vérifier la bonne gestion des travaux en attente et des `PendingJobLists`. Pour cela, les nombres d'activation maximaux des tâches sont initialisés à des valeurs supérieures à 1 dans leurs descripteurs statiques.

Conduite de la vérification : la vérification de l'implémentation d'EDF-US est menée selon le même processus présenté en § 9.5 et illustré dans la figure 9.6. La combi-

naison du modèle complet (OS + Timer) est stimulée par l'ensemble des trois moteurs de vérification présentés au début de chapitre. Le résultat est combiné avec des modèle d'observateurs formalisant nos exigences pour l'implémentation d'EDF-US et est injecté en entrée du model-checker d'UPPAAL afin de vérifier la correction des propriétés CTL exprimées sur les états des observateurs.

Ce processus de vérification est mené sur un ensemble de N tâches s'exécutant sur m cœurs dans un intervalle de vérification VTI renseignés dans le modèle de l'OS. L'objectif est de varier ces trois paramètres et de vérifier la correction de la réaction de l'implémentation face aux situations générées dans l'intervalle VTI. Rappelons que les événements d'activation de ces tâches sont assurés par les deux moteurs d'activation. Quant à la terminaison, pour chaque tâche, un moteur d'exécution est élaboré et intégré au modèle de l'OS afin d'assurer l'événement de terminaison correspondant. Les caractéristiques des tâches sont choisies tel que discuté au début de cette section.

Problème d'explosion combinatoire : le principe des moteurs de vérification repose sur un nombre de tâches donné et un intervalle de temps spécifié. Dans notre processus de vérification, nous nous sommes confronté au problème de *l'explosion combinatoire* de l'espace d'états [CKNZ11] en variant ces deux paramètres. Cette explosion est induite par l'exhaustivité du model checking : le nombre d'états vérifiés augmente de manière exponentielle jusqu'à ce que la taille de l'espace d'états excède la quantité de mémoire de l'ordinateur sur lequel la vérification est menée. Ceci rend la vérification des exigences impossible sur l'espace d'états. En effet, avec l'introduction des moteurs de vérification, un nombre important d'entrelacements est considéré par le model-cheker suite à l'indéterministe d'occurrence des événements d'ordonnancement. De plus, ces entrelacements se produisent dans les moteurs de vérification sur un intervalle de temps continu, manipulé par des horloges à valeurs réelles positives, ce qui génère plus d'états. Ainsi, en raison de ce problème, nous n'avons pas pu vérifier l'implémentation quand le nombre de tâches dépasse 2 pour des configurations s'exécutant sur deux cœurs. Le tableau 10.1 dresse les résultats de vérification de l'implémentation d'EDF-US[ξ] en utilisant les moteurs de vérification sur une machine de 128 Mo de mémoire. Cette mémoire s'épuise, avant que la vérification de l'ensemble des exigences n'aboutisse, dès que VTI atteint une valeur de 6 pour un nombre de tâche égal à 2.

10.5 Réduction de l'espace d'états

Le problème de l'explosion combinatoire de l'espace d'états a fait objet de certaines études proposant des techniques pour y pallier. L'objectif est de réduire la taille du modèle en créant un modèle équivalent, vis-à-vis des propriété à vérifier, mais plus petit. Nous examinons dans cette section deux techniques intégrées dans notre processus de vérification, afin de pouvoir réduire l'espace d'états exploré.

VTI	Explored states	Runtime (s)	Consumed memory (KiB)
		number of tasks = 1	
5	57621	7	63908
10	351251	41,34	119264
20	2211611	249,9	685444
50	29771991	3,3. 10^3	11117868
		number of tasks = 2	
3	322763	44,5	85236
5	17725780	2,53. 10^3	1980624
6	26592322	4,46. 10^3	3971344
7	unfinished	unfinished	unfinished
		number of tasks > 2	
		unfinished	

TABLE 10.1: Résultats de vérification du modèle de l'implémentation d'EDF-US[ξ] pour deux cœurs.

10.5.1 Réduction de l'exploration par ordre partiel

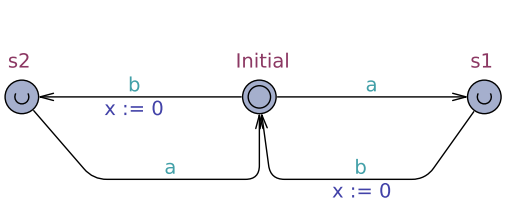
Cette technique consiste à éliminer autant que se peut les entrelacements inutiles en détectant les redondances dans les différents chemins d'exécution pour n'en parcourir qu'un seul [God90]. Compte tenu de deux ou plusieurs exécutions qui ne diffèrent que par l'ordre d'entrelacement de leurs franchissements de transitions, il suffit de vérifier la propriété sur une seule de ces exécutions à condition que l'on puisse garantir que sa valeur de vérité est la même sur toutes les autres. Afin de mieux illustrer ce principe, nous considérons l'exemple de la figure 10.5(a). Supposons que cet automate manipule une variable globale `x` pouvant être modifiée par d'autres automates du système et que l'on souhaite vérifier qu'à chaque fois que l'action `b` est réalisée, la valeur de `x` est remise à 0 avec la propriété : `AG(b ==> (x == 0))`. Depuis l'état initial, plusieurs chemins d'exécution sont possibles (*e.g.* `abba, abab, baba, baab, ababab,` etc.). Nous nous focalisons sur deux d'entre eux qui nous intéressent vis-à-vis de la propriété à vérifier :

chemin 1 (`abba`) : faire `a` et prendre la transition menant vers `s1` puis faire `b`. Ensuite, faire `b` et prendre la transition menant vers `s2` puis faire `a` et revenir à l'état initial.

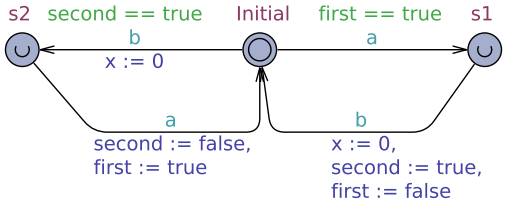
chemin 2 (`baab`) : prendre le chemin inverse de la première exécution en faisant `b` d'abord et revenir à l'état initial. Ensuite, faire `a` et revenir à l'état initial après `b`.

La propriété énoncée reste indépendante puisqu'elle dépend plutôt de la valeur de `x` après avoir fait l'action `b`. Ainsi, que l'on prenne le chemin (1) en premier ou le chemin (2), le résultat de vérification de la propriété reste inchangé. Ainsi, nous pouvons déterminer un ordre d'exécution dans cet automate afin de pouvoir réduire les entrelacements dans

cet automate. Une solution possible est d'utiliser des variables globales avec des valeurs initiales imposant l'exécution du chemin (1) avant le chemin (2) (cf. Fig. 10.5(b)).



(a) Automate avant définition d'un ordre d'exécution



(b) Automate après la définition d'un ordre d'exécution. Cet ordre est imposé grâce aux booléens `first`, initialisé à vrai, et second initialisé à faux.

FIGURE 10.5: Exemple de réduction de l'exploration par ordre partiel.

Application dans les moteurs d'activation : nous souhaitons étendre le principe expliqué dans l'exemple précédent afin de pouvoir l'appliquer dans notre processus de vérification. Notons que l'application d'une telle technique de réduction nécessite la maîtrise parfaite du modèle que l'on souhaite réduire afin de pouvoir sélectionner les chemins n'influençant pas sur la vérification des exigences à vérifier.

Dans notre modèle, ce sont les moteurs d'activation présentés dans les figures 10.1 et 10.2 qui sont la principale cause de l'explosion combinatoire. En plus du non déterminisme qu'ils offrent vis-à-vis de la date de l'occurrence d'un événement d'ordonnancement, ils contiennent plusieurs entrelacements possibles pour l'activation des tâches. En effet, chaque tâche est activée en empruntant une transition se synchronisant vers l'automate modélisant le service `ActivateTask`. Toutes les transitions sortent du même état et peuvent être prises dans n'importe quel ordre. Ainsi, pour n tâches à activer il y a $n!$ chemins d'exécution possibles pour chaque instant de 0 à VTI ce qui explique l'explosion de l'espace d'états. Par conséquent, la technique de réduction d'ordre partiel est appliquée sur ces moteurs afin de réduire les entrelacements en définissant un ordre de franchissement de leurs transitions et donc un ordre d'activation des tâches.

Nous énumérons deux critères observables permettant de vérifier, par le biais de nos exigences, la réaction de l'implémentation d'EDF-US[ξ] face aux événements d'ordonnancement et nous discutons la possibilité d'y appliquer la réduction par ordre partiel :

1. valeurs des variables et structures de données partagées (eg. compteurs d'activation, les booléens `need_schedule need_switch`, les listes des tâches, etc.) : la majorité des exigences présentées en 10.4.2 sont vérifiées en observant la variation de certaines variables ou structures de données de l'OS. Par exemple, la vérification de l'appel de l'ordonnanceur suite à l'occurrence d'un événement d'ordonnancement se traduit par la vérification de la valeur du booléen `need_schedule` et voir si sa valeur est mise à vrai. Dans ce cas, quel que soit l'ordre d'activation des tâches dans le moteur de

vérification, l'objectif est de vérifier l'appel de l'ordonnanceur après ces activations. Le même raisonnement pourra être étendu sur le reste des exigences dont la vérification est effectuée en examinant les variables de l'OS.

2. comparaison des priorités des tâches activées : le composant de l'ordonnanceur au sein de l'OS est le seul composant dont certaines exigences ne dépendent pas de la variation des valeurs des variables de l'OS mais plutôt d'un résultat de comparaison des priorité entre tâches. Ainsi, nous discutons à ce niveau l'importance de l'ordre d'exécution par rapport au résultat de cette comparaison. Rappelons que la priorité d'exécution selon EDF-US[ξ] est attribuée en fonction du poids des tâches et de leurs dates d'échéance. Dans ce cas, trois situations sont possibles suivant le type des tâches activées par les moteurs d'activation :

   (a) toutes les tâches à activer sont lourdes : dans une telle situation, l'ordre de leur activation n'est pas important étant donné que la concurrence d'exécution entre les tâches lourdes est résolue arbitrairement. Le résultat de la vérification des exigences ne dépend pas de cet ordre.

   (b) toutes les tâches à activer sont légères : la priorité d'exécution entre ces tâches est déterminée en fonction de leur date d'échéance. Tel que nous l'avons expliqué dans la section 10.4.3, toutes les tâches ont le même délai critique puisque le moteur d'activation permet de couvrir les possibilités d'avoir des tâches à échéances identiques ou non (cf. Fig 10.4). Dans ce cas, l'ordre du franchissement des transitions pour les activer n'est pas important étant donné qu'elles ont le même délai critique. Ainsi, le résultat de comparaison des dates d'échéance reste inchangé par rapport à l'ordre de franchissement des transitions.

   (c) des tâches lourdes et légères sont à activer : afin d'expliquer l'ordre que nous choisissons, nous reprenons l'exemple d'activation présenté dans la figure 10.1 permettant d'activer deux tâches `task0` et `task1`. Supposons que `task0` est lourde et `task1` est légère. Ainsi, deux cas de figure se présentent :

      - si `task0` est activée avant `task1`, l'implémentation peut être confrontée aux situations suivantes : (i) l'insertion de `task0` dans la `HeavyList` ; (ii) la sélection de `task0` pour exécution ; (iii) l'insertion de `task1` dans la `ReadyList` quand elle est activée ; (iv) la comparaison des poids de `task0` et `task1`.
      - `task1` est activée avant `task0` : l'implémentation peut être confrontée aux situations suivantes : (i) l'insertion de `task1` dans la `ReadyList` ; (ii) la sélection de `task1` pour exécution ; (iii) l'insertion de `task0` dans la `HeavyList` quand elle est activée ; (iv) la comparaison des poids de `task0` et `task1` ; (v) la préemption de `task1` en faveur de `task0`.

   En généralisant ce raisonnement au delà de deux tâches, le fait d'activer toutes les tâches légères en premier engendre une situation supplémentaire qui concerne la comparaison des dates d'échéance entre les tâches légères. Ainsi, en activant les tâches légères avant les tâches lourdes, l'implémentation pourra être confrontée à plus de situations que dans le cas inverse. En outre, l'ensemble des situations se produisant quand les tâches lourdes sont activées en premier est inclus dans l'ensemble des situations pouvant se produire quand les tâches légères sont activées

en premier tel que nous l'avons montré dans les deux cas de figure susmentionnés.

En conséquence de cette analyse, nous avons décidé d'appliquer sur les moteurs d'activation la réduction par ordre partiel des entrelacements qui favorise l'activation des tâches légères en premier. Cette méthode ne détermine pas l'ordre d'activation des tâches dans le temps, mais plutôt l'ordre séquentiel de l'appel du service `ActivateTask` pour chacune des tâches depuis l'état initial du moteur d'activation. Autrement dit, deux tâches peuvent être activées au même instant, c'est l'ordre de l'exécution du code du service d'activation pour elles qui est déterminé. Ainsi, nous déterminons un ordre de franchissement des transitions permettant d'activer les tâches tel que le montre la figure 10.6. Pour cela, nous utilisons un tableau `order` enregistrant des variables booléennes dont l'indice correspond aux identifiants des tâches. Chaque transition menant à l'activation d'une tâche `task_i` est conditionnée par la tautologie du booléen enregistré à `order[i]`. Une fois que la tâche `i` est activée, elle met à faux son booléen et met à vrai le booléen `order[i+1]` afin de permettre l'activation de la tâche suivante, ainsi de suite. Le booléen `order[0]` est initialisé à vrai afin de pouvoir lancer l'activation de la tâche 0 au début. Cette opération est répétée jusqu'à ce que le *VTI* soit atteint. Notons que l'indéterminisme concernant l'occurrence ou non des activations et la date de ces occurrences est toujours conservé. C'est seulement l'ordre des activations qui est déterminé.

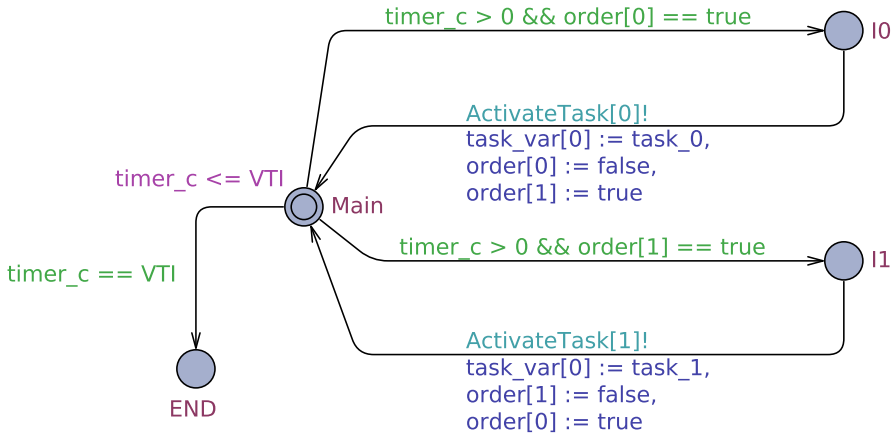


FIGURE 10.6: Moteurs d'activation avec ordre partiel.

En déterminant l'ordre d'activation des tâches dans le modèle, la vérification des exigences a pu être poussée jusqu'à couvrir l'activation de 6 tâches sur deux cœurs avec *VTI* allant jusqu'à 10. Le tableau 10.2 résume les résultats obtenus durant le processus de vérification. Pour les nombres de tâches et les *VTI* fournis dans le tableau, l'ensemble des exigences de l'implémentation EDF-US[ξ] est vérifié avec succès. Avec l'intégration de la technique d'ordre partiel, une amélioration de performances de vérification en terme de temps d'exécution et d'espace mémoire occupé est noté : un gain 63.5% en temps CPU et de 58.9% de mémoire consommée.

VTI	Explored states	Runtime (s)	Consumed memory (KiB)
		number of tasks = 1	
5	19869	1.909	24618
10	134682	15,08	47588
20	853904	118,57	294365
50	10826178	1,15. 10^3	4073821
		number of tasks = 2	
3	127072	10,29	36515
5	6596228	0,94. 10^3	732454
6	9625487	1,63. 10^3	1613289
		number of tasks = 6	
10	808168394	1,72. 10^7	
		number of tasks > 6	
		unfinished	

TABLE 10.2: Résultats de vérification du modèle de l'implémentation d'EDF-US[ξ] après réduction par ordre partiel.

10.5.2 Réduction de l'espace d'états par sélection de scénarios significatifs

Dans cette section, nous proposons une autre méthode de réduction du nombre d'états explorés durant le processus de vérification. Cette méthode consiste à cibler des scénarios d'événements d'ordonnancement significatifs au regard de nos exigences, et à mener la vérification uniquement pour ces scénarios. Le choix de ces scénarios se fait en fonction de la réaction de l'implémentation que nous souhaitons observer. Par exemple, pour vérifier que l'ordonnanceur est appelé suite à une activation, un scénario permettant de déclencher l'activation d'une tâche doit être mis en place afin d'observer la réaction de l'implémentation face à cette activation. Ce raisonnement est étendu pour toutes les exigences élaborées dans la section 10.4.2. Ainsi, une analyse est menée pour chaque composant du périmètre de l'ordonnancement afin de déterminer le scénario décrivant une situation pouvant vérifier ses exigences. Cette méthode permet de faire une réduction, dans la mesure du possible, qui ne garde que le minimum de scénarios nécessaires à la vérification de nos exigences. Nous discutons dans la suite de cette section le choix des scénarios pour chaque composant. Ce choix détermine le nombre de tâches, leurs nombres maximaux d'activations et le VTI à considérer. Le nombre de cœurs étant fixé à 2 en raison du problème d'explosion combinatoire. Mais également puisque la version actuelle multicœur de Trampoline n'a que 2 cœurs.

Scénario pour le gestionnaire de tâches : ce composant permet d'effectuer certaines opérations nécessaires à l'activation ou terminaison d'une tâche, telles que le changement d'état d'une tâche, l'incrémentation / décrémentation du compteur d'acti-

vations, l'insertion / extraction dans les listes des tâches en fonction du poids de la tâche et de son compteur d'activation, etc. Son comportement ne dépend que de l'occurrence des événements d'ordonnancement et non pas des résultats de comparaison de dates d'échéance. Ainsi, le scénario à choisir doit provoquer les opérations effectuées après une occurrence d'un événement d'ordonnancement afin de pourvoir observer la réaction du composant vis-à-vis de ces opérations. Ce scénario doit contenir au moins une tâche lourde et une tâche légère pour la vérification des exigences concernant l'insertion et/ou l'extraction dans la `ReadyList` ou la `HeavyList`. Chacune des tâches doit avoir un nombre maximum d'activations supérieur à 1 afin de pouvoir vérifier les exigences concernant l'insertion et/ou l'extraction dans la `PendingJobList`. Ceci permet également de vérifier si l'ordonnanceur est appelé quand l'une des tâches est activée ou terminée. Nous choisissons de mettre en place le scénario pour ce composant avec le moteur d'activation `activate_several` qui permet l'activation de plusieurs travaux d'une même tâches tant que son nombre maximum d'activations n'est pas atteint (cf. § 10.2). Rappelons que ce moteur impose que les activations successives soient espacées d'une unité de temps afin d'éviter l'appel du service `ActivateTask` infiniment sans que le temps évolue. Ainsi, si nous souhaitons avoir un scénario activant m travaux de n tâches, il faut choisir une valeur de VTI supérieure ou égale à $m \times n$ qui correspond au nombre maximum de travaux pouvant être activés.

Scénario pour le gestionnaire de temps : la vérification de ce composant consiste à s'assurer qu'il compare correctement deux dates d'échéance avec l'algorithme ICTOH (cf. § 6.3.2). Rappelons que cette comparaison se fait sur la base d'une représentation circulaire des dates d'échéance. Ainsi, nous vérifions que le résultat de comparaison en circulaire est conforme au résultat d'une comparaison linéaire. Pour cela, il est nécessaire de prévoir deux tâches légères dans le scénario pour provoquer une comparaison des dates d'échéance. Afin de faire la vérification d'une comparaison basée sur une représentation circulaire des dates, il est nécessaire de vérifier sur un VTI qui dépasse 2^n pour des dates représentées sur n bits. Cependant, l'introduction des moteurs de vérification a drastiquement limité le temps de vérification à cause de l'explosion combinatoire. Ainsi, pour la vérification de ce composant, nous choisissons deux tâches légères ayant des dates d'échéance représentées circulairement sur 3 bits ce qui nous permet de vérifier sur un VTI supérieur ou égal à 8. L'activation de ces tâches peut être faite avec le moteur d'activation `activate_once` (cf. Fig. 10.1), étant donné que l'objectif est de vérifier la comparaison des dates d'échéance des travaux actifs concurrents des tâches et non pas des travaux en attente.

Rappelons que les valeurs de représentation de ces dates d'échéance ne peuvent pas excéder excéder $2^n - 1$, et donc 7 pour des dates représentées sur 3 bits. Ainsi, notre implémentation de l'algorithme ICTOH doit comparer des dates représentées sur des cycles de 8. La vérification d'une telle implémentation est effectuée en comparant le résultat de la comparaison des références des dates d'échéance sur la représentation circulaire de 3 bits selon notre implémentation d'ICTOH, et de celui d'une comparaison linéaire selon la valeur réelle des dates d'échéance.

Scénario pour le gestionnaire des listes de tâches : pour ce composant, il faut vérifier les exigences de la gestion des listes de tâches. Pour cela, il est nécessaire d'avoir deux tâches légères pour la vérification des propriétés de la `ReadyList` et une tâche lourde pour la vérification de la `HeavyList` (la priorité entre les tâches lourdes n'étant pas importante). Le nombre d'activations maximal de ces tâches doit être supérieur à 1 de manière à vérifier également les `PendingJobLists`. Le moteur d'activation de plusieurs travaux peut être utilisé dans un tel scénario pour une valeur de VTI supérieure ou égale à 6.

Scénario pour l'ordonnanceur : étant donné que l'exécution est effectuée sur deux cœurs, le scénario doit assurer au moins deux tâches en cours d'exécution afin de vérifier l'exigence de la non-oisiveté de l'ordonnanceur. Nous choisissons une tâche lourde et une tâche légère afin de vérifier la correction de comparaison des poids. Pour la vérification de la correction de comparaison des dates d'échéance, une troisième tâche légère doit être considérée. Les tâches doivent avoir un nombre d'activations maximal supérieur à 1 afin de vérifier les exigences de l'ordonnanceur par rapport aux `PendingJobLists`. Pour un tel scénario, l'utilisation du moteur `activate_several` est indispensable. Ainsi, le VTI doit être d'une valeur supérieure ou égale à 6.

Scénario pour le gestionnaire de changement de contexte : pour ce composant, il est nécessaire de considérer trois tâches en compétition pour l'exécution afin de vérifier, en plus du chargement de contexte d'une tâche élue par l'ordonnanceur, la sauvegarde du contexte d'une tâche préemptée. Pour cela, nous choisissons trois tâches légères à activer par le moteur `activate_once` et donc à exécuter sur un VTI supérieur ou égal à 3 (puisque l'activation multiple n'est pas nécessaire).

Résultats et discussion : l'ensemble des scénarios proposés dans cette section a été utilisé pour la vérification de l'implémentation d'EDF-US[ξ] (cf. tableau 10.3). Ces scénarios utilisent les moteurs d'activation dans leur version avec ordre partiel. Les exigences de cette implémentation sont vérifiées avec succès pour les scénarios fournis. Ainsi, ceci nous permet de donner de la confiance sur la correction fonctionnelle de notre implémentation. En effet, face au problème de l'explosion combinatoire, il est nécessaire d'identifier des situations, dans lesquelles nous souhaitons un comportement spécifié par nos exigences, et voir la réaction de l'implémentation face à celles-ci. En fonction de la réponse de l'implémentation, nous pourrons ainsi conclure sur la correction de son comportement. Ceci, en tenant compte également du fait que les exigences ont été vérifiées avec succès dans les étapes précédentes (§ 10.4.3 et § 10.5.1). Certainement, nous ne pouvons pas affirmer que l'implémentation est valide pour n'importe quel scénario, étant donné que le problème de l'explosion combinatoire constitue une vraie limite pour la conduite du processus de vérification. Toutefois, l'approche que nous avons suivie permet de donner de la confiance dans la comportement de l'implémentation.

Component	Verification scenario	Explored states	Runtime (s)	Consumed memory (KiB)
Task Manager	- 1 heavy task - 1 light task - max activation count = 2 - VTI = 4	361949	48,5	88404
Time Manager	- 2 light tasks - max activation count = 1 - VTI = 9	454766	60,12	127032
Task List Manager	- 2 light tasks - 1 heavy task - max activation count = 2 - VTI = 6	67222346	10^4	6357204
Scheduler	- 2 light tasks - 1 heavy task - max activation count = 2 - VTI = 6	67222346	10^4	6357204
Context Switch Manager	- 3 light tasks - max activation count = 1 - VTI = 3	365157	53,4	121864

TABLE 10.3: Résultats de vérification du modèle de l'implémentation d'EDF-US[ξ] sur des scénarios significatifs.

10.6 Conclusion

Ce chapitre était consacré aux modèles d'excitation que nous avons proposés pour stimuler le modèle d'une implémentation lors de la conduite de notre approche de vérification. Ces modèles sont des automates temporisés permettant de générer de manière indéterministe des événements d'activation et de terminaison d'un nombre déterminé de tâches sur un intervalle de temps de vérification. L'objectif de leur introduction est de pousser la vérification des implémentations au delà d'une seule application. Nous avons appelé ces modèles des moteurs de vérification : (i) d'activation pour les générateurs d'événements d'activation ; (ii) d'exécution pour les générateur d'événements de terminaison. Ces moteurs ont été appliqués afin vérifier une implémentation d'EDF-US[ξ] au sein de Trampoline dont le modèle et les exigences ont été présentés également dans ce chapitre.

L'utilisation des moteurs de vérification permet de couvrir tous les scénarios d'activation et de terminaison de l'ensemble de tâches considéré sur un intervalle de temps. Avec cette exhaustivité, nous avons été confrontés au problème majeur d'une approche de vérification basée sur le model-checking : l'explosion combinatoire de l'espace d'états. Pour cela, nous avons proposé des techniques permettant de réduire l'espace d'états explorés. D'abord, une réduction des entrelacements dans les moteurs de vérification est proposée, en définissant un ordre d'activation des tâches. Ensuite, nous avons identifié des scénarios

de vérification en relation avec les exigences que nous souhaitons vérifier et permettant ainsi d'observer la réaction de l'implémentation face à des situations spécifiques. Nous avons pu ainsi mener la vérification de nos exigences avec succès sur l'ensemble des scénarios identifiés. Toutefois, limité par le problème de l'explosion combinatoire, nous restons conscients du fait que nous ne pouvons pas conclure sur la validation de la correction de l'implémentation.

Conclusion générale et perspectives

Bilan

Deux constatations se sont dégagées de l'analyse de l'état de l'art en matière de politiques d'ordonnancement temps réel multiprocesseur et des systèmes d'exploitation temps réel : (i) une offre riche en matière de politiques d'ordonnancement temps réel proposées par la communauté scientifique ; (ii) une offre limitée en matière de travaux d'implémentation de ces politiques au sein d'une plateforme réelle. En effet, une grande partie des systèmes d'exploitation temps réel repose toujours sur des politiques d'ordonnancement à priorité fixe pour les tâches et sur l'ordonnancement partitionné lorsque l'offre multiprocesseur/-multicœur est supportée. Une cause possible de ces constats serait la distance entre la description abstraite des politiques, qui se ramène à un ensemble de N tâches s'exécutant sur m processeurs, et le contexte réel au sein duquel il faut manipuler les variables, les structures de données, les composants et/ou les fonctions du système d'exploitation support. La mise en œuvre d'une politique d'ordonnancement dans une plateforme réelle représente ainsi une tâche qui peut s'avérer ardue et qui soulève la question suivante : pour une politique d'ordonnancement, comment s'assurer que l'implémentation réalisée produit un comportement conforme à sa description telle qu'exposée dans la littérature scientifique ?

Le travail exposé dans ce manuscrit a tenté de fournir une réponse à cette problématique. Dans ce cadre, nous avons proposé une approche basée sur les méthodes formelles visant à vérifier la correction fonctionnelle d'une implémentation d'ordonnanceur. En d'autres termes, l'objectif de l'approche est de vérifier que le comportement effectif, produit par l'ordonnanceur implémenté, correspond au comportement de l'ordonnanceur tel qu'énoncé dans la littérature. Nous nous sommes intéressés dans notre étude à l'implémentation des politiques d'ordonnancement global. Ainsi, dans ce manuscrit, nous avons présenté une instanciation de notre approche de vérification à travers une mise en œuvre des politiques G-EDF et EDF-US[ξ] au sein du système d'exploitation temps réel Trampoline. La première mise en œuvre a été réalisée en intégrant le code source de l'implémentation de G-EDF à celui de Trampoline. La deuxième a été effectuée en élaborant un modèle en UPPAAL modélisant l'implémentation d'EDF-US[ξ] et en l'intégrant au modèle de Trampoline.

Nous avons commencé ce manuscrit par une présentation du contexte général au sein duquel notre travail de recherche se situe. Ainsi, nous avons fourni dans un premier temps, un panorama sur la théorie de l'ordonnancement temps réel en nous focalisant sur l'ordonnancement multiprocesseur, les systèmes d'exploitation temps réel et les méthodes de vérification formelle. Étant donné que nos travaux d'implémentation et de vérification sont menés sur Trampoline, nous avons ainsi fourni une présentation de cet RTOS résultant d'une étude approfondie de son architecture et son fonctionnement, avec un accent sur le processus de l'ordonnancement au sein du noyau. Ceci nous a permis d'identifier l'ensemble des composants du système d'exploitation qui contribuent à la décision d'ordonnancement que nous avons qualifié de *périmètre d'ordonnancement*. Cette

analyse et la maîtrise qu'elle nous a permis d'acquérir, nous ont conduit à proposer une implémentation de G-EDF au sein de Trampoline, et de l'étendre ainsi pour supporter l'ordonnancement global.

L'approche de vérification que nous avons proposée pour la vérification des implémentations d'ordonnanceur est basée sur le *model-checking*. C'est une approche conduite en deux phases. Une première phase consiste à concevoir un modèle formel décrivant les implémentations en question et à les intégrer à un modèle pré-existant de l'OS. Le modèle de l'implémentation est élaboré en UPPAAL avec des automates finis étendus, des automates temporisés et des fonctions UPPAAL. Il décrit de manière fidèle le flot de contrôle de l'implémentation, ses variables et les séquences d'instructions telles que présentes dans le code source de Trampoline. Une deuxième phase consiste à conduire la vérification sur le modèle élaboré en suivant quatre étapes : (i) traduire les propriétés de la politique implémentée énoncées dans la littérature sous forme d'exigences décrivant le comportement attendu de l'implémentation ; (ii) formaliser ces exigences sous forme de modèles d'observateur ; (iii) générer des événements d'ordonnancement pour stimuler le modèle de l'implémentation ; (iv) vérifier les exigences sur le modèle stimulé en examinant la satisfaction de propriétés CTL exprimées sur les états des observateurs.

La vérification a été menée sur les deux modèles modélisant des implémentations de G-EDF et d'EDF-US[ξ] au sein Trampoline. Pour la première implémentation, la stimulation du modèle a été faite grâce un modèle d'application décrivant de manière fidèle le cycle de vie d'une tâche. Ceci nous a permis de confronter le modèle de l'implémentation à un jeu de 31 applications et nous a conduit à détecter et corriger des erreurs dans l'implémentation de G-EDF. Toutefois, le modèle d'application proposé reste déterministe et limité du point de vue des événements d'ordonnancement produits et des scénarios auxquels le modèle de l'implémentation doit réagir. Pour remédier à cela, nous avons élaboré des moteurs de vérification pouvant générer de manière indéterministe des événements d'ordonnancement d'un nombre de tâches pouvant survenir dans n'importe quel ordre et à n'importe quel moment sur un intervalle de temps donné. Ainsi, ces moteurs permettent une couverture la plus large possible, en terme de scénarios d'événements d'ordonnancement, et la plus indépendante possible de configurations particulières de tâches. Ils ont été appliqués pour la vérification du modèle de l'implémentation d'EDF-US[ξ]. L'exhaustivité de couverture de ces moteurs vient avec le problème d'explosion combinatoire de l'espace d'états qui représente la principale limite d'une démarche de vérification par model-checking. Ainsi, des techniques de réduction de l'espace d'états explorés ont été appliquées, notamment en identifiant des scénarios en relation avec les exigences que nous souhaitons vérifier. La vérification du modèle d'implémentation d'EDF-US[ξ] a été menée avec succès sur ces scénarios ce qui nous a permis de gagner en confiance sur sa correction.

Discussion

Ce travail de thèse nous a permis de mettre à disposition de la communauté scientifique de l'ordonnancement temps réel et des systèmes d'exploitation temps réel une démarche

de vérification des implémentations d'ordonnanceur. Une démarche qui est la plus générique possible et qui pourrait éventuellement être appliquée à d'autres implémentations d'ordonnanceur dans d'autres systèmes d'exploitation. Nous avons ainsi pu montrer la faisabilité de notre approche en présentant des résultats prometteurs en termes de détection des erreurs d'implémentation et de la couverture des situations auxquelles cette implémentation peut faire face. Toutefois, certains points restent à discuter, notamment en ce qui concerne la couverture des exigences vérifiées sur les modèles des implémentations et des scénarios d'événements d'ordonnancement les stimulant.

En ce qui concerne les exigences choisies pour chaque implémentation, elles sont élaborées à la suite d'une étude approfondie des interactions des composants liés à l'ordonnancement au sein de Trampoline et du code de celui-ci, que nous maîtrisons parfaitement. Elles expriment minutieusement, sur la base de cette analyse, ce que nous attendons comme comportement de tous ces composants afin de parvenir à un fonctionnement correct des implémentations vérifiées. Nous sommes ainsi convaincus de la pertinence du choix des exigences proposées.

En ce qui concerne la couverture de la vérification par rapport aux jeux d'application et les scénarios d'événements d'ordonnancement générés, notre objectif initial était de démontrer que le comportement des implémentations est correct face à n'importe quelle situation. Dans cette optique, la réponse des implémentations a pu être vérifiée avec succès face à une multitude d'événements d'ordonnancement produits soit par un jeu d'applications choisi ou bien par des moteurs générant de manière indéterministe ces événements. En l'occurrence, pour le modèle de l'implémentation d'EDF-US[ξ], nous avons vérifié que quel que soit l'événement d'ordonnancement produit par un ensemble de tâches sur un intervalle de temps donné, la réponse de l'implémentation est correcte. Toutefois, les valeurs du nombre de tâches et de l'intervalle considérés sont fortement limitées par le problème de l'explosion combinatoire à cause de l'indéterminisme de l'arrivée des événements d'ordonnancement. Ceci nous mène à nous poser la question suivante : « *est-ce que nous pouvons généraliser les résultats obtenus concernant la correction des implémentations par rapport aux scénarios considérés à n'importe quelle autre application avec un nombre plus élevé de cœurs ou de tâches ?* » Il convient d'affirmer qu'en l'état actuel de l'étude, il serait potentiellement possible de pousser les limites actuelles de l'intervalle de vérification en explorant d'autres pistes permettant de réduire l'espace d'états explorés, en plus de celles proposées dans ce travail. Cependant, l'exhaustivité d'une approche basée sur le model-checking et l'indéterminisme des situations auxquelles l'implémentation est confrontée pourraient éventuellement poser de nouvelles limites pour le dit intervalle.

Confrontés au problème de l'explosion combinatoire, nous arrivons à la conclusion que notre démarche est faisable pour la vérification d'une implémentation dans le cadre d'un cas d'étude, dans lequel une application avec une configuration de tâches est spécifiée et dont les objets et les paramètres sont renseignés. Elle pourrait en l'occurrence être utilisée dans le cadre d'une étude d'ordonnançabilité d'une politique d'ordonnancement par rapport à une application donnée. Nous restons toutefois conscients de l'intérêt des résultats qu'elle a présentés puisqu'elle nous a permis de détecter des erreurs dans notre

implémentation de G-EDF et gagné en confiance dans l'implémentation d'EDF-US[ξ]. Elle nous ouvre par conséquent sur des perspectives sur la poursuite du travail que nous discutons dans ce qui suit.

Perspectives

Étude d'autres techniques d'abstraction pour réduire l'exploration de l'espace d'états

Afin de tenter de pallier le problème de l'explosion combinatoire de l'espace d'états, nous avons examiné l'utilisation de deux techniques de réduction de cet espace. Il serait intéressant de parcourir d'autres techniques existantes dans le but de pousser la vérification au-delà des limites de l'intervalle de vérification auxquelles nous étions confrontés. Pour cela, plusieurs pistes possibles sont envisageables comme :

1. **la conversion des automates modélisant les fonctions de Trampoline en fonctions UPPAAL :** ceci consiste à limiter au maximum le nombre d'automates dans le modèle à vérifier afin de réduire l'espace d'états. En effet, étant donné que nous menons notre démarche de vérification sur un OS préalablement vérifié, il n'est pas nécessaire d'avoir une modélisation fine des fonctions de l'OS non liées à l'implémentation de l'ordonnanceur à vérifier. Ainsi, les automates modélisant ces fonctions peuvent être exprimés sous forme de fonctions UPPAAL qui sont exécutées dans le modèle de manière atomique et ne génèrent pas d'états intermédiaires durant le processus de vérification. Nous pouvons ne conserver alors que les automates modélisant les fonctions des composants du périmètre d'ordonnancement dont le comportement nécessite d'être vérifié. Notons que ceci ne peut s'appliquer que sur un système d'exploitation préalablement vérifié comme c'est le cas de Trampoline.
2. **l'abstraction du modèle de l'OS :** c'est une des techniques proposées dans la littérature pour remédier au problème de l'explosion combinatoire [CGL94]. Le principe consiste ainsi à construire un autre modèle qui abstrait le modèle original du système à vérifier tout en essayant de conserver au maximum le comportement effectif qu'il modélise. Concrètement dans notre étude, ceci se traduit par construire pour chaque automate A du modèle de l'OS, satisfaisant une propriété P, un automate B plus simple et avec moins d'états satisfaisant aussi P. L'automate B peut également être le résultat d'une fusion des états de l'automate A. Une telle approche peut ne pas être facilement appliquée sur un modèle décrivant de manière très fine le système d'exploitation. Ainsi, l'abstraction obtenue peut ne pas satisfaire exactement les mêmes propriétés du modèle original. Pour cela, il est nécessaire d'intégrer une étape dans le processus de vérification qui consiste à s'assurer que l'abstraction est correcte vis-à-vis des propriétés du modèle initial que nous souhaitons conserver. Dans la littérature de la vérification formelle, cette étape doit être faite à l'aide d'un assistant de preuve combinant ainsi l'utilisation du model-checking et du theorem-proving dans la même démarche de vérification [MN95].

Génération du code des implémentations vérifiées

Après la démarche de vérification des deux implémentations proposées, une perspective de ce travail est de générer le code des versions vérifiées de ces implémentations afin de l'intégrer au code source de Trampoline. Ceci correspond à un passage du modèle vérifié vers son implémentation au sein du système d'exploitation. Pour cela, un traducteur d'UPPAAL vers le langage C doit être élaboré sur la base des règles proposées dans ce manuscrit pour passer d'un code source en C vers UPPAAL, en appliquant ces règles dans le sens inverse. Il est à noter que les modèles construits en UPPAAL sont regroupés dans des fichiers XML. Ainsi, la génération du code des implémentations doit se baser sur deux éléments :

1. **un analyseur de fichiers XML :** qui doit parcourir les fichiers correspondant aux modèles des implémentations ligne par ligne. Ceci permet de déterminer si la ligne est à traiter correspond à une transition, une place ou une instruction en langage UPPAAL. Dans le cas d'une instruction, l'analyseur doit être en mesure de déterminer la nature des variables (arguments de fonctions, variables globales de l'OS, variables locales à une fonction/automate etc.) et des structures de données qui y sont traitées. Le modèle doit ainsi être annoté de manière à faire distinction entre ces données.
2. **traducteur de code :** qui contient toutes les règles permettant de passer d'un code en UPPAAL, associé à chaque transition, instruction ou déclaration de variables et/ou structures de données, à une ligne de code la traduisant en langage C.

À l'aide de ces deux éléments, il est possible de générer le code en C des implémentations en analysant chaque ligne du fichier XML et déterminant l'instruction équivalente en C sur la base des informations fournies par le traducteur de code établi.

Utilisation d'un langage dédié pour la génération du modèle

À plus long terme, une possible continuité de ce travail serait d'élaborer un langage dédié ou DSL (*Domain-Specific Language*) pour la modélisation des implémentations au sein de Trampoline. Un tel langage doit fournir des domaines de variable, des déclarations et des abstractions spécifiques au fonctionnement de l'ordonnanceur au sein de Trampoline. Ceci permet par la suite de générer automatiquement, à partir d'une spécification complète des paramètres, des objets et des fonctions de l'implémentation, le modèle de celle-ci et puis son code source en intégrant le principe de génération de code expliqué ci-dessus.

Mise en œuvre de la démarche de vérification pour des politiques d'ordonnancement plus sophistiquées

L'objectif premier de notre démarche de vérification est de s'assurer qu'une politique d'ordonnancement une fois implémentée dans un OS avec toutes les contraintes que celui-ci impose, affiche un comportement équivalent à celui exprimé par ses auteurs dans la littérature. Nous souhaitons ainsi appliquer cette démarche pour la vérification des

implémentations de certaines politiques d'ordonnancement plus sophistiquées, notamment celles dont le résultat produit est fortement basé sur le traitement des événements d'ordonnancement simultanés. En effet, pour certaines politiques (*e.g.* U-EDF [NBN+12] entre autres), lorsque plusieurs événements d'ordonnancement ont lieu au même instant, le calcul de la séquence d'ordonnancement doit tenir compte de tous ces événements et des traitements qui en résultent, même quand ces événements surviennent sur différents cœurs. Or, dans un système d'exploitation temps réel, il est difficile d'assurer la simultanéité de traitement des événements étant donné que les instructions du système sont exécutées de manière séquentielle. En outre, certains événements d'ordonnancement ne peuvent tout simplement pas survenir de manière synchrone comme certaines politiques le supposent du point de vue théorique. À titre d'exemple, dans la réalité, un événement de terminaison de tâche dépend de la durée effective de l'exécution de celle-ci et non pas des durées estimées (*e.g* WCET et BCET) sur lesquelles les descriptions littéraire des politiques d'ordonnancement se basent. Par conséquent, il est difficile de synchroniser un tel événement d'ordonnancement et son traitement avec d'autres événements. Ainsi, une continuité du présent travail serait d'étudier, en appliquant la même démarche de vérification proposée, l'implémentabilité de ces politiques, tout en tenant compte des contraintes susmentionnées. Notons qu'une telle étude pourrait être facilitée dans le cas où l'on traite en premier les trois premières perspectives discutées.

Bibliographie

[AB00] Luca Abeni and Giorgio Buttazzo. Support for dynamic qos in the hartik kernel. In *Proceedings Seventh International Conference on Real-Time Computing Systems and Applications*, pages 65–72. IEEE, 2000.

[ABD05] James H Anderson, Vasile Bud, and UmaMaheswari C Devi. An edf-based scheduling algorithm for multiprocessor soft real-time systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 199–208. IEEE, 2005.

[ABJ01] Björn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. In *Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE*, pages 193–202. IEEE, 2001.

[AD90] Rajeev Alur and David Dill. Automata for modeling real-time systems. In *International Colloquium on Automata, Languages, and Programming*, pages 322–335. Springer, 1990.

[AHL+08] Eyad Alkassar, Mark A Hillebrand, Dirk Leinenbach, Norbert W Schirmer, and Artem Starostin. The verisoft approach to systems verification. In *Working Conference on Verified Software : Theories, Tools, and Experiments*, pages 209–224. Springer, 2008.

[AJ00] Björn Andersson and Jan Jonsson. Fixed-priority preemptive multiprocessor scheduling : to partition or not to partition. In *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*, pages 337–346. IEEE, 2000.

[And08] Björn Andersson. Global static-priority preemptive multiprocessor scheduling with utilization bound 38%. In *International Conference on Principles of Distributed Systems*, pages 73–88. Springer, 2008.

[ÅNKR12] Mikael Åsberg, Thomas Nolte, Shinpei Kato, and Ragunathan Rajkumar. Exsched : An external cpu scheduler framework for real-time systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2012 IEEE 18th International Conference on*, pages 240–249. IEEE, 2012.

[Arn10] Rob Arnold. *C0, an imperative programming language for novice computer scientists*. PhD thesis, Master's thesis, Department of Computer Science, Carnegie Mellon University, 2010.

[AS99a] J Anderson and Anand Srinivasan. A new look at pfair priorities. *Submitted to Real-time Systems Journal*, 1999.

[AS99b] James H. Anderson and Anand Srinivasan. A new look at pfair priorities. Technical report, In Submission, 1999.

[AS00a] James H Anderson and Anand Srinivasan. Early-release fair scheduling. In *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*, pages 35–43. IEEE, 2000.

[AS00b] James H Anderson and Anand Srinivasan. Pfair scheduling : Beyond periodic task systems. In *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*, pages 297–306. IEEE, 2000.

[AT06] Björn Andersson and Eduardo Tovar. Multiprocessor scheduling with few preemptions. In *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)*, pages 322–334. IEEE, 2006.

[AUT03] AUTOSAR development cooperation. *site web : http ://www.autosar.org*, 2003.

[B+08] Richard Barry et al. Freertos. *Internet, Oct*, 2008.

[BA11] Konstantinos Bletsas and Björn Andersson. Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. *Real-Time Systems*, 47(4) :319–355, 2011.

[Bar97] Michael Barabanov. A linux-based real-time operating system. 1997.

[Bar04] Sanjoy K Baruah. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *IEEE Transactions on Computers*, 53(6) :781–784, 2004.

[Bar13] Sanjoy Baruah. Partitioned edf scheduling : a closer look. *Real-Time Systems*, 49(6) :715–729, 2013.

[BB09] Theodore P Baker and Sanjoy K Baruah. An analysis of global edf schedulability for arbitrary-deadline sporadic task systems. *Real-Time Systems*, 43(1) :3–24, 2009.

[BBFT06] Jean-Luc Béchennec, Mikael Briday, Sébastien Faucou, and Yvon Trinquet. Trampoline an open source implementation of the OSEK/VDX RTOS specification. In *Emerging Technologies and Factory Automation, 2006. ETFA'06. IEEE Conference on*, pages 62–69. IEEE, 2006.

[BCL05] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. New schedulability tests for real-time task sets scheduled by deadline monotonic on multiprocessors. In *International Conference On Principles Of Distributed Systems*, pages 306–321. Springer, 2005.

[BCM+92] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking : 1020 states and beyond. *Information and computation*, 98(2) :142–170, 1992.

[BCPV96] Sanjoy K Baruah, Neil K Cohen, C Greg Plaxton, and Donald A Varvel. Proportionate progress : A notion of fairness in resource allocation. *Algorithmica*, 15(6) :600–625, 1996.

[BF] JP Bodeveix and M Filali. A generic tool for expressing the development of validations. In *11th Nordic Workshop on Programming Theory NWPT*, volume 99, pages 37–37.

[BFLM07] Jean-Paul Bodeveix, Mamoun Filali, Julia L Lawall, and Gilles Muller. Automatic verification of bossa scheduler properties. *Electronic Notes in Theoretical Computer Science*, 185 :17–32, 2007.

[BG03] Sanjoy K Baruah and Joël Goossens. Rate-monotonic scheduling on uniform multiprocessors. *IEEE transactions on computers*, 52(7) :966–970, 2003.

[BGP95] Sanjoy K Baruah, Johannes E Gehrke, and C Greg Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Parallel Processing Symposium, 1995. Proceedings., 9th International*, pages 280–288. IEEE, 1995.

[Bje05] Per Bjesse. What is formal verification ? *ACM SIGDA Newsletter*, 35(24) :1–es, 2005.

[BLL+96] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL—a tool suite for automatic verification of real-time systems. *Hybrid Systems III*, pages 232–243, 1996.

[BM01] Luciano Porto Barreto and Gilles Muller. *Bossa : a DSL framework for application-specific scheduling policies.* PhD thesis, INRIA, 2001.

[BRT18] Jean-Luc Béchennec, Olivier H. Roux, and Toussaint Tigori. Formal model-based conformance verification of an OSEK/VDX compliant RTOS. In *CODIt 2018-5th International Conference on Control, Decision and Information Technologies*, 2018.

[BS93] Jonathan Bowen and Victoria Stavridou. Safety-critical systems, formal methods and standards. *Software engineering journal*, 8(4) :189–209, 1993.

[BS14] Gedare Bloom and Joel Sherrill. Scheduling and thread management with rtems. *ACM Sigbed Review*, 11(1) :20–25, 2014.

[Bur99] Alan Burns. The ravenscar profile. *ACM SIGAda Ada Letters*, 19(4) :49–52, 1999.

[But11] Giorgio C Buttazzo. *Hard real-time computing systems : predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[C+09] AUTOSAR Consortium et al. Autosar release 4.0. *Specification of multicore OS architecture v1*, 2009.

[CB03] Alessio Carlini and Giorgio C Buttazzo. An efficient time representation for real-time embedded systems. In *Proceedings of the 2003 ACM symposium on Applied computing*, pages 705–712. ACM, 2003.

[CDNQ12] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, 77(9) :1006–1036, 2012.

[CE81] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[CES83] Edmund M Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite state concurrent system using temporal logic specifications : a practical approach. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 117–126, 1983.

[CFH+04] John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James H Anderson, and Sanjoy K Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms., 2004.

[CFJ93] Edmund M Clarke, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. In *International Conference on Computer Aided Verification*, pages 450–462. Springer, 1993.

[CGG11] Liliana Cucu-Grosjean and Joël Goossens. Exact schedulability tests for real-time scheduling of periodic tasks on unrelated multiprocessor platforms. *Journal of systems architecture*, 57(5) :561–569, 2011.

[CGL94] Edmund M Clarke, Orna Grumberg, and David E Long. Model checking and abstraction. *ACM transactions on Programming Languages and Systems (TOPLAS)*, 16(5) :1512–1542, 1994.

[CHD15] Maxime Chéramy, Pierre-Emmanuel Hladik, and Anne-Marie Déplanche. Algorithmes pour l'ordonnancement temps réel multiprocesseur. *Journal Européen des Systèmes Automatisés (JESA)*, 48(7-8) :613–639, 2015.

[CKNZ11] Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*, pages 1–30. Springer, 2011.

[CLB+06] John M Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C Devi, and James H Anderson. LITMUS^{RT} : A testbed for empirically comparing real-time multiprocessor schedulers. In *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*, pages 111–126. IEEE, 2006.

[Coo71] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[CRJ06] Hyeonjoong Cho, Binoy Ravindran, and E Douglas Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 101–110. IEEE, 2006.

[CSMC18] Eugenia A Capota, Cristina S Stangaciu, Mihai V Micea, and Vladimir I Cretu. P_fenp : A multiprocessor real-time scheduling algorithm. In *2018 IEEE 12th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 000509–000514. IEEE, 2018.

[DB11] Robert I Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM computing surveys (CSUR)*, 43(4) :35, 2011.

[DDB08] Matthias Daum, Jan Dörrenbächer, and Sebastian Bogan. Model stack for the pervasive verification of a microkernel-based operating system. In *5th International Verification Workshop (VERIFY'08)*, volume 372, pages 56–70. Citeseer, 2008.

[DFH+91] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Benjamin Werner, and Christine Paulin-Mohring. *The COQ proof assistant user's guide : version 5.6*. PhD thesis, INRIA, 1991.

[DK12] Robert I Davis and Shinpei Kato. FPSL, FPCL and FPZL schedulability analysis. *Real-Time Systems*, 48(6) :750–788, 2012.

[DL78] Sudarshan K Dhall and Chung Laung Liu. On a real-time scheduling problem. *Operations research*, 26(1) :127–140, 1978.

[dOCdO18] Daniel Bristot de Oliveira, Tommaso Cucinotta, and Rômulo Silva de Oliveira. Modeling the behavior of threads in the preempt rt linux kernel using automata. In *Proceedings of the Embedded Operating System Workshop (EWiLi), Turin, Italy*, 2018.

[EAW14] Jeremy P Erickson, James H Anderson, and Bryan C Ward. Fair lateness scheduling : Reducing maximum lateness in g-edf-like scheduling. *Real-Time Systems*, 50(1) :5–47, 2014.

[EH83] E Allen Emerson and Joseph Y Halpern. " sometimes" and" not never" revisited : on branching versus linear time (preliminary report). In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 127–140, 1983.

[Ene] Enea OSE Enea. Architecture user's guide. *Rev. BL140702*.

[FG02] Shelby Funk and Joël Goossens. Modifying edf for uniform multiprocessors. In *14th IEEE EUROMICRO conference on real-time systems (Work-In-Progress session)*. Citeseer, 2002.

[FGH+14] Joao F Ferreira, Cristian Gherghina, Guanhua He, Shengchao Qin, and Wei-Ngan Chin. Automated verification of the freertos scheduler in hip/sleek. *International Journal on Software Tools for Technology Transfer*, 16(4) :381–397, 2014.

[Fit12] Melvin Fitting. *First-order logic and automated theorem proving*. Springer Science & Business Media, 2012.

[FMB+09] Simon Fürst, Jürgen Mössinger, Stefan Bunzel, Thomas Weber, Frank Kirschke-Biller, Peter Heitkämper, Gerulf Kinkelin, Kenji Nishikawa, and Klaus Lange. Autosar–a worldwide standard is on the road. In *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, volume 62, page 5, 2009.

[FN09] Shelby Funk and Vijaykant Nanadur. Lre-tl : An optimal multiprocessor scheduling algorithm for sporadic task sets. 2009.

[Fon01] Pedro Fonseca. Approximating linear time with finite count clocks. *Electrónica e Telecomunicações*, 3(4) :359–361, 2001.

[Fre72] Gottlob Frege. Conceptual notation and related articles. 1972.

[FTC+09] Dario Faggioli, Michael Trimarchi, Fabio Checconi, Marko Bertogna, and Antonio Mancina. An implementation of the earliest deadline first algorithm in Linux. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1984–1989. ACM, 2009.

[FW08] Antônio A Fröhlich and Lucas F Wanner. Operating system support for wireless sensor networks. *Journal of Computer Science*, 4(4) :272, 2008.

[G+05] OSEK Group et al. Osek/vdx operating system specification. *site web : http ://www. osek-vdx. org*, 2005.

[GAGB01] Paolo Gai, Luca Abeni, Massimiliano Giorgi, and Giorgio Buttazzo. A new kernel approach for modular real-time systems development. In *Proceedings 13th Euromicro Conference on Real-Time Systems*, pages 199–206. IEEE, 2001.

[Gal95] Bill O Gallmeister. Programming for the real world, posix. 4. *OReilly & Associates, Inc*, 1995.

[GCS11] Laurent George, Pierre Courbin, and Yves Sorel. Job vs. portioned partitioning for the earliest deadline first semi-partitioned scheduling. *Journal of systems architecture*, 57(5) :518–535, 2011.

[Ger04] Philippe Gerum. Xenomai-implementing a rtos emulation framework on gnu/linux. *White Paper, Xenomai*, pages 1–12, 2004.

[GFB02] Joël Goossens, Shelby Funk, and Sanjoy Baruah. Edf scheduling on multiprocessor platforms : some (perhaps) counterintuitive observations. In *In RealTime Computing Systems and Applications Symposium*. Citeseer, 2002.

[GFB03] Joël Goossens, Shelby Funk, and Sanjoy Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-time systems*, 25(2) :187–205, 2003.

[GGCG16] Joël Goossens, Emmanuel Grolleau, and Liliana Cucu-Grosjean. Periodicity of real-time schedules for dependent periodic tasks on identical multiprocessor platforms. *Real-time systems*, 52(6) :808–832, 2016.

[GHRF94] Dov M Gabbay, Ian Hodkinson, Mark Reynolds, and Marcelo Finger. *Temporal logic : mathematical foundations and computational aspects*, volume 1. Clarendon Press Oxford, 1994.

[God90] Patrice Godefroid. Using partial orders to improve automatic verification methods. In *International Conference on Computer Aided Verification*, pages 176–185. Springer, 1990.

[Gra09] Mentor Graphics. Nucleus rtos. *URL, Accessed October*, 2009.

[HG03] J Holzmann Gerard. Spin model checker : The primer and reference manual, 2003.

[Hoa78] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8) :666–677, 1978.

[HT05] Michael Hohmuth and Hendrik Tews. The vfiasco approach for a verified operating system. *2nd PLOS*, 2005.

[HUW14] John Harrison, Josef Urban, and Freek Wiedijk. History of interactive theorem proving. In *Computational Logic*, volume 9, pages 135–214, 2014.

[HZZ+11] Yanhong Huang, Yongxin Zhao, Longfei Zhu, Qin Li, Huibiao Zhu, and Jianqi Shi. Modeling and verifying the code-level osek/vdx operating system with csp. In *2011 Fifth International Conference on Theoretical Aspects of Software Engineering*, pages 142–149. IEEE, 2011.

[Jon86] Douglas W Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4) :300–311, 1986.

[KC11] Heeheon Kim and Yookun Cho. A new fair scheduling algorithm for periodic tasks on multiprocessors. *Information Processing Letters*, 111(7) :301–309, 2011.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4 : Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[Kow74] Robert Kowalski. Predicate logic as programming language. In *IFIP congress*, volume 74, pages 569–544, 1974.

[KS97] Ashok Khemka and RK Shyamasundar. An optimal multiprocessor real-time scheduling algorithm. *Journal of parallel and distributed computing*, 43(1) :37–45, 1997.

[KY07] Shinpei Kato and Nobuyuki Yamasaki. Real-time scheduling with task splitting on multiprocessors. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, pages 441–450. IEEE, 2007.

[KY08a] S KATO and N YAMASAKI. Semi-partitioning technique for multiprocessor real-time scheduling. in the 29th ieee real-time systems symposium. *Work-in-Progress Session (RTSS'08 WiP)*, 2008.

[KY08b] Shinpei Kato and Nobuyuki Yamasaki. Global edf-based scheduling with efficient priority promotion. In *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA'08. 14th IEEE International Conference on*, pages 197–206. IEEE, 2008.

[KY09] Shinpei Kato and Nobuyuki Yamasaki. Semi-partitioned fixed-priority scheduling on multiprocessors. In *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 23–32. IEEE, 2009.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE transactions on software engineering*, (2) :125–143, 1977.

[Lam05] Edward L Lamie. *Real-time embedded multithreading : using ThreadX and ARM*. CMP books San Francisco, 2005.

[Lam09] Edward L Lamie. *Real-time embedded multithreading using ThreadX*. Newnes, 2009.

[Lee94] Suk Kyoon Lee. On-line multiprocessor scheduling algorithms for real-time tasks. In *TENCON'94. IEEE Region 10's Ninth Annual International Conference. Theme : Frontiers of Computer Technology. Proceedings of 1994*, pages 607–611. IEEE, 1994.

[LFS+10] Greg Levin, Shelby Funk, Caitlin Sadowski, Ian Pye, and Scott Brandt. Dp-fair : A simple model for understanding optimal multiprocessor scheduling. In *2010 22nd Euromicro Conference on Real-Time Systems*, pages 3–13. IEEE, 2010.

[Lie96] Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9) :70–78, 1996.

[LL73] 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.

[LRSF04] Peng Li, Binoy Ravindran, Syed Suhaib, and Shahrooz Feizabadi. A formally verified application-level framework for real-time scheduling on posix real-time operating systems. *IEEE Transactions on Software Engineering*, 30(9) :613–629, 2004.

[M+80] Robin Milner et al. A calculus of communicating systems. 1980.

[Mat13] Aditya P Mathur. *Foundations of software testing, 2/e*. Pearson Education India, 2013.

[MBD+00] Paolo Mantegazza, E Bianchi, Lorenzo Dozio, S Papacharalambous, S Hughes, and D Beal. Rtai : Real-time application interface. 2000.

[MBTS04] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. *The art of software testing*, volume 2. Wiley Online Library, 2004.

[McK05] Paul McKenney. A realtime preemption overview. *http ://lwn. net/Articles/146861/*, 2005.

[MLD05] Gilles Muller, Julia L Lawall, and Hervé Duchesne. A framework for simplifying the development of kernel schedulers : Design and performance evaluation. In *High-Assurance Systems Engineering, 2005. HASE 2005. Ninth IEEE International Symposium on*, pages 56–65. IEEE, 2005.

[MLR+14] Ernesto Massa, George Lima, Paul Regnier, Greg Levin, and Scott Brandt. Outstanding paper : Optimal and adaptive multiprocessor real-time scheduling : The quasi-partitioning approach. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 291–300. IEEE, 2014.

[MN95] Olaf Müller and Tobias Nipkow. Combining model checking and deduction for i/o-automata. In *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–16. Springer, 1995.

[Mok83] Aloysius K Mok. Fundamental design problems of distributed systems for the hard-real-time environment. 1983.

[NBN+12] Geoffrey Nelissen, Vandy Berten, Vincent Nélis, Joël Goossens, and Dragomir Milojevic. U-edf : An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 13–23. IEEE, 2012.

[NPW02] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL : a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[NSG+14] Geoffrey Nelissen, Hang Su, Yifeng Guo, Dakai Zhu, Vincent Nélis, and Joël Goossens. An optimal boundary fair scheduling. *Real-time systems*, 50(4) :456–508, 2014.

[ORR+96] Sam Owre, Sreeranga Rajan, John M Rushby, Natarajan Shankar, and Mandayam Srivas. Pvs : Combining specification, proof checking, and model checking. In *International Conference on Computer Aided Verification*, pages 411–414. Springer, 1996.

[OY98] Sung-Heun Oh and Seung-Min Yang. A modified least-laxity-first scheduling algorithm for real-time tasks. In *Real-Time Computing Systems and Applications, 1998. Proceedings. Fifth International Conference on*, pages 31–36. IEEE, 1998.

[Pet77] James L Peterson. Petri nets. *ACM Computing Surveys (CSUR)*, 9(3) :223–252, 1977.

[PN03] QNX Real-time Plataform and QNX Neutrino. Qnx ltd software, canada, 2003.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, 1977.

[PSD11] Markus Partheymüller, Julian Stecklina, and Björn Döbel. Fiasco. oc on the scc. In *4th Many-core Applications Research Community (MARC) Symposium*, page 79, 2011.

[Riv03] Wind River. Vxworks programmer's guide, 2003.

[RLM+11] Paul Regnier, George Lima, Ernesto Massa, Greg Levin, and Scott Brandt. Run : Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 104–115. IEEE, 2011.

[Ros98] Joe Rosen. *Symmetry discovered : Concepts and applications in nature and science*. Courier Corporation, 1998.

[rtl] Rt-linux.

[SB90] Inder M Singh and Mitch Bunnell. Lynxos : Unix rewritten for real-time. *Ik_ _*, page 27, 1990.

[SB02] Anand Srinivasan and Sanjoy Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*, 84(2) :93–98, 2002.

[SGQ16] Lijun Shan, Susanne Graf, and Sophie Quinton. Rtlib : A library of timed automata for modeling real-time systems. 2016.

[SLD08] Jun Sun, Yang Liu, and Jin Song Dong. Model checking csp revisited : Introducing a process analysis toolkit. In *International symposium on leveraging applications of formal methods, verification and validation*, pages 307–322. Springer, 2008.

[SR89] John A. Stankovic and Krithi Ramamritham. The spring kernel : a new paradigm for real-time operating systems. *ACM SIGOPS Operating Systems Review*, 23(3) :54–71, 1989.

[Srl] Evidence Srl. Erika enterprise rtos. *URL : http ://www.evidence.eu.com.*

[SSTB13] Paulo Baltarejo Sousa, Pedro Souto, Eduardo Tovar, and Konstantinos Bletsas. The carousel-edf scheduling algorithm for multiprocessor systems. In *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 12–21. IEEE, 2013.

[Sta88] John A. Stankovic. Misconceptions about real-time computing : A serious problem for next-generation systems. *Computer*, 21(10) :10–19, 1988.

[TBFR17] Kabland Toussaint Gautier Tigori, Jean-Luc Béchennec, Sébastien Faucou, and Olivier Henri Roux. Formal model-based synthesis of application-specific static RTOS. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(4) :97, 2017.

[Tim08] *The Open Group Technical Standard. Base Specifications, Issue 6.* IEEE, 2008.

[WDG+97] Victor Fay Wolfe, Lisa Cingiser DiPippo, Roman Ginis, Michael Squadrito, Steven Wohlever, Igor Zykh, and Russell Johnston. Real-time corba. In *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*, pages 148–157. IEEE, 1997.

[WH08] Libor Waszniowski and Zdeněk Hanzálek. Formal verification of multitasking applications based on timed automata model. *Real-Time Systems*, 38(1) :39–65, 2008.

[WL99] Y-C Wang and K-J Lin. Implementing a general real-time scheduling framework in the red-linux real-time kernel. In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pages 246–255. IEEE, 1999.

[Zah98] Andree Zahir. Oil-osek implementation language. 1998.

[ZMM03] Dakai Zhu, Daniel Mossé, and Rami Melhem. Multiple-resource periodic scheduling problem : how much fairness is necessary ? In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 142–151. IEEE, 2003.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE NANTES

Titre : Mise en œuvre de politiques d'ordonnancement temps réel multiprocesseur prouvée

Mot clés : Politique d'ordonnancement temps réel, système d'exploitation temps réel, Implémentation d'ordonnancement, Model-checking

Résumé : L'implémentation d'une nouvelle politique d'ordonnancement au sein d'un système d'exploitation temps réel n'est pas une tâche facile. Le passage de la spécification littéraire abstraite d'une politique à une mise en œuvre sur une plateforme réelle exige que des choix de réalisation soient faits et que des contraintes de diverses natures inhérentes à cette dernière soient prises en compte. Par conséquent, l'implémentation d'un ordonnanceur doit impérativement être accompagnée d'un travail de vérification permettant d'apporter un niveau de confiance en validant la conformité du comportement de l'ordonnanceur implémenté par rapport à sa spécification d'origine.

Dans cette thèse, nous nous intéressons à l'utilisation des méthodes formelles pour la vérification des implémentations d'ordonnanceurs temps réel. Nous proposons une approche de vérification de type « model-checking », que nous conduisons sur des implémentations d'ordonnanceurs globaux menées au sein de Trampoline, un système d'exploitation temps réel conforme aux standards OSEK/VDX et AUTOSAR. Pour chaque implémentation, un modèle décrivant son fonctionnement est élaboré avec des machines à états finis et temporisées. Ce modèle est stimulé par des générateurs produisant des scénarios indéterministes d'événements d'ordonnancement afin d'observer la réaction de l'implémentation à vérifier face à diverses situations. La vérification est alors menée en examinant la satisfaction d'un ensemble d'exigences spécifiées en fonction du comportement attendu de l'implémentation tel que stipulé dans la littérature.

Cette approche a permis la vérification de la correction fonctionnelle du comportement de deux implémentations d'ordonnanceurs globaux au sein de Trampoline : G-EDF et EDF-US. Toutefois, son caractère modulaire et générique permet d'en envisager l'usage pour d'autres politiques et dans d'autres systèmes d'exploitation.

Title: Proven implementation of multiprocessor real-time scheduling policies

Keywords: Real-time scheduling policy, Real-time operating system, Scheduer implementation, Model-checking

Abstract: Implementing a new scheduling policy within a real-time operating system is not an easy rask. Moving from an abstract literary specification of a policy to its implementation within a real platform requires making choices of realization and considering various contraints inherent to the latter. Consequently, a scheduler implementation work shall imperatively be supported by a verification study allowing to bring a level of confidence by validating the conformity of the behavior of the implemented scheduler compared to its original specification.

In this thesis, we are interested in the use of formal methods for the verification of real-time scheduler implementations. We propose a "model-checking" approach, which we conduct on global scheduler implementations carried out on Trampoline, a real-time operating system compliant with OSEK/VDX and AUTOSAR standards. For each implementation, a model describing its behavior is elaborated with finite state and timed machines. This model is stimulated by generators producing indeterministic scenarios of scheduling events in order to observe the reaction of the implementation under various situations. The verification is then conducted by examining the satisfaction of a set of specified requirements according to the expected behavior of the implementation as stipulated in the literature.

This approach allowed the verification of the functional correction of the behavior of two implementations of global schedulers in Trampoline: G-EDF and EDF-US. However, its modular and generic character allows to consider its use for other policies and in other operating systems.