Analyses pour l'ordonnançabilité et la flexibilité de systèmes temps-réel

Christophe Prêvot

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTE

UNIVERSITE GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Christophe Prévot

Thèse dirigée par **Alain Girault, Directeur de Recherche Première Classe, INRIA Grenoble Rhône-Alpes**,
et codirigée par **Sophie Quinton, Chargée de Recherche, INRIA Grenoble Rhône-Alpes**

préparée au sein du **Laboratoire INRIA Grenoble Rhône-Alpes**
et du **Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Analyses pour l'ordonnançabilité et la flexibilité de systèmes temps-réel

Analyses for schedulability and flexibility of real-time systems

Thèse soutenue publiquement le **15 novembre 2019**,
devant le jury composé de :

Madame, Claire, Pagetti
Ingénieure de recherche, ONERA Toulouse, Rapporteure
Monsieur, Emmanuel, Grolleau
Professeur des universités, ISAE-ENSMA, Rapporteur
Monsieur, Nicolas, Navet
Professeur des universités, Université du Luxembourg, Examinateur
Monsieur, Giuseppe, Lipari
Professeur des universités, Université de Lille, Examinateur
Monsieur, Alain, Girault
Directeur de recherche, INRIA Grenoble Rhône-Alpes, Directeur de thèse
Madame, Sophie, Quinton
Chargée de recherche, INRIA Grenoble Rhône-Alpes, Co-directrice de thèse
Monsieur, Laurent, Rioux
Ingénieur/chercheur, Thales R&T, Co-encadrant industriel
Monsieur, Rafik, Henia
Ingénieur/chercheur, Thales R&T, Co-encadrant industriel

Résumé en français Les systèmes temps réel sont développés pour de nombreuses applications dans les domaines de l'avionique ou l'automobile. Pour ces systèmes les contraintes temporelles sont aussi importantes que les contraintes fonctionnelles. Ainsi pour chaque fonctionnalité il faut calculer le temps maximum, appelé latence au pire cas, entre l'acquisition d'une donnée et la restitution du résultat correspondant. Compte tenu des besoins rencontrés au sein de l'entreprise Thales, nous considérons des systèmes mono-processeurs ordonnancés sous ordonnanceur préemptif à priorités fixes sur lesquels sont alloués des chaînes de tâches. Chaque chaîne implémente une fonctionnalité distincte et chaque tâche possède une priorité fixe et unique. La terminaison d'une tâche au sein d'une chaîne entraîne le démarrage de la tâche suivante. Quant au processeur, il exécute toujours la tâche de priorité maximale. Nous utilisons ensuite l'analyse d'ordonnançabilité pour caractériser le comportement temporel des systèmes et calculer la latence au pire cas des chaînes. Si la latence au pire cas de chaque chaîne du système est inférieure à son échéance temporelle alors ce système est ordonnançable. Pour garantir l'ordonnançabilité d'un système, on calcule des majorants, qui sont supérieurs ou égaux à la latence au pire cas, et qui peuvent être sur-approximés en fonction de la précision des analyses. Pour un système donné, si les sur-aproximations sont trop importantes alors il faut le sur-dimensionner afin de garantir son ordonnançabilité, ce qui n'est pas souhaitable dans un contexte industriel. Pour résoudre ce problème de sur-dimensionnement, nous calculons un majorant qui est plus précis et/ou plus général que ceux présentés dans l'état de l'art. Une fois le majorant calculé, il est utile de mesurer l'écart maximum entre la latence au pire cas d'une chaîne et son majorant. Dans cet objectif, nous calculons un minorant sur la latence au pire cas pour évaluer la précision du majorant. Dans l'état de l'art, les minorants sont calculés par simulation. Nous proposons quant à nous d'utiliser l'analyse d'ordonnançabilité pour calculer des minorants que nous définissons comme des scénarios réalisable compte tenu du modèle du système. Les calculs des minorants sont développés en utilisant les analyses développées pour le calcul des majorants. Pour finir, compte tenu des long cycles de vies ou des technologies qui évoluent rapidement, de nombreux systèmes sont amenés à évoluer. Une évolution pertinente dans un contexte industriel consiste au rajout d'une chaîne de tâches à un système existant. Pour garantir l'ordonnançabilité du système dans ce contexte, nous présentons une méthode pour calculer le plus grand temps d'exécution qui peut être alloué à la nouvelle chaîne tout en garantissant l'ordonnançabilité du nouveau système. Cette analyse peut être utilisée pour analyser la sensibilité et/ou la robustesse d'un système face à des changements de paramètres comme par exemple les temps d'exécution des tâches. Cela consiste à trouver les valeurs extrêmes d'un paramètre temporel de telle sorte que le système reste ordonnançable. Dans cette thèse, nous avons présenté des analyses pour majorer et minorer la latence au pire cas d'une chaîne et permettre à un système d'ordonnancer une nouvelle chaîne tout en étant garanti ordonnançable. Ces travaux pourront être étendus en analysant des systèmes plus complexes et en adaptant le calcul des minorants à d'autres types d'analyses. Pour finir, les analyses développées sont complexes, il serait intéressant de les formaliser via un assistant informatique de manière à garantir leur exactitude.

Résumé en anglais Real time system are often used for applications in avionic or automotive domains. For those systems timing constraints are as important as functional constraints. Thus for each functionality it is necessary to compute the maximum duration, denoted worst case latency, between data acquisition and the corresponding restitution of results. Given the needs that we identify at Thales firm, we consider uni-processor systems scheduled under fixed priority preemptive scheduling with task chains mapped on it. Each chain implement a functionality and each task has a fix and unique priority. The finishing time of a task correspond to the activation of the following one in the chain. The processor always execute the task with the highest priority. Then we use scheduling analysis to characterize the timing behavior of systems and compute the worst case latency of chains. If the worst case latency of each chain is lower than or equal to its timing constraint then the system is schedulable. To guarantee the schedulability of a system, we compute upper bounds that are higher than or equal to the worst case latency, and these bounds may be over-approximated depending on the precision of the analysis. For a given system, if over-approximations are too important then it is necessary to over-dimension the system to guarantee its schedulability, that is not desirable in a industrial context. To solve this over-dimension problem, we compute a more precise and/or general upper bound than in the state of the art. When the upper is computed, it is useful to measure the gap between the worst case latency and its upper bound. To achieve this, we compute a lower bound on the worst case latency to evaluate the precision of the upper bound. In the state of the art, lower bounds are computed using simulation. We propose to use schedulability analysis to compute lower bounds that we define as execution scenarios that are realizable considering the system model. Lower bounds are developed using equation establish to compute upper bounds. Finally, considering long lifetime of systems and the quick evolution of technologies, many systems have to evolve. A relevant evolution in an industrial context consist in adding a new chain in an existing system. To guarantee the schedulability of a system in this context we present a methodology to compute the largest execution time of a new chain while giving guarantee on the schedulability of the new system. This analysis may be use to analyze the sensitivity and/or the robustness of a system with respect to parameter changes as execution times of tasks. This consists in finding extreme values of a timing parameters such that the system remains schedulable. In this thesis, we present analysis to compute upper and lower bounds on the worst case latency and to schedule a new chain while giving guarantee on the schedulability. These works may be extended to handle more complex systems and computation of lower bound could be adapted to other analysis. Finally, developed analysis are complex, it would be interesting to formalize them using a proof assistant to guarantee that they are exact.

Table des matières

Chapitre 1

Introduction

Sommaire

Cette thèse est motivée par des problèmes rencontrés, au sein de l'entreprise Thales, dans le cadre de ses activités de développement logiciel pour des systèmes embarqués temps-réels. Dans ce chapitre, nous décrivons le contexte précis dans lequel s'inscrit cette thèse, les problèmes que nous avons identifiés, et nos contributions en vue de les résoudre.

1.1 Contexte industriel

Les **systèmes embarqués** font partie de notre quotidien, que ce soit dans l'automobile et l'avionique ou des secteurs nouveaux tels que les objects connectés, la domotique, etc. Dans les nouveaux secteurs d'activité, l'utilisation des sytèmes embarqués permet de fournir de nouveaux services comme la vidéo-surveillance ou le contrôle à distance du système de chauffage d'une maison. Dans des secteurs plus anciens (comme l'avionique et l'automobile), l'apparition et le développement des systèmes embarqués a permis de remplacer des systèmes mécaniques, de renforcer la sécurité des utilisateurs ou encore d'augmenter le confort. Un système embarqué est un système électronique et informatique dédié à l'exécution de tâches précises (pour lesquelles il a été développé) et qui est soumis à de nombreuses contraintes non fonctionnelles. Ces contraintes peuvent être, entre autres, le temps de calcul, la taille, le poids ou encore la consommation d'énergie. Les appareils électro-ménagers, comme le lave-linge, contiennent un ou plusieurs systèmes embarqués

qui permettent de contrôler les différentes options. Ce type de système est soumis des contraintes comme la taille ou le temps de réponse. Nous nous focalisons dans cette thèse sur les systèmes **temps-réel** et/ou **critiques**.

1.1.1 Systèmes temps-réel critiques

Un système est dit temps-réel si chacune de ses fonctionnalités doit finir de s'exécuter dans un temps borné. Chaque fonctionnalité du système est donc contrainte par une **échéance temporelle**. Les systèmes temps-réel sont utilisés, entre autres, dans les secteurs de l'automobile, l'avionique, l'aérospatiale, etc. Il est possible de classer les différents systèmes temps-réel en fonction de leurs besoins de performance et de la criticité des fonctionnalités exécutées. Dans certains cas, il importe de traiter/transmettre, dans un intervalle de temps, le plus de données possibles. Dans ce cas, le concepteur a pour objectif de connaître et maximiser la *performance moyenne* du système, c'est à dire de maximiser la quantité de données traitées par unité de temps. Dans cette thèse, nous nous intéressons aux systèmes pour lesquels il est nécessaire de donner des *garanties concernant la performance au pire cas*.

Prenons pour exemple le système de contrôle d'un avion : le pilote doit, à tout moment, pouvoir contrôler la direction et la vitesse de l'avion. Ainsi, lorsqu'il actionne la manette des gaz ou les commandes de vol, la réaction (le comportement) de l'avion doit correspondre à l'action effectuée par le pilote. Chaque fonctionnalité a des contraintes, non seulement fonctionnelles, mais également temporelles. De toute évidence, les conséquences seraient catastrophiques si exécuter la commande de vol prenait quelques minutes. Dans cette situation, le temps mis par le système pour exécuter la commande, appelé **latence**, est aussi important que la fonction réalisée. Pour chaque fonctionnalité, il est donc nécessaire de calculer la **latence au pire cas**, c'est à dire le temps maximum nécessaire pour exécuter l'ensemble de la fonctionnalité. Autrement dit, il faut calculer la durée maximum entre l'activation externe (exemple : action du pilote) et la réponse produite par le système (exemple : modification de la vitesse, la direction). Dans tous les cas, le temps mis pour exécuter la fonction doit être inférieur à son échéance temporelle. Les systèmes, pour lesquels il faut évaluer la performance au pire cas, peuvent être classifiés en plusieurs catégories, dont les deux principales sont :

— les systèmes *temps-réel mous*, pour lesquels il n'est pas problématique de dépasser les échéance temporelles. C'est le cas du système de divertissement des passagers (en anglais In Flight Intertainement).

— les systèmes *temps-réel durs*, pour lesquels manquer une échéance temporelle peut avoir des conséquences catastrophiques. C'est le cas du système de commandes de vol de l'avion.

Par la suite, nous nous intéressons aux systèmes **temps-réel durs**. De plus, nous considérons des systèmes dits *critiques*, c'est à dire pour lesquels un dysfonctionnement (fonctionnel ou temporel) peut avoir des conséquences catastrophiques. Dans l'avionique, par exemple, le dysfonctionnement de certains systèmes peut provoquer des pertes humaines et la destruction de l'avion. Les systèmes critiques peuvent être *certifiés conforme à une norme*, cela permet de prouver que ces systèmes ont été développés selon un processus strict. Dans l'avionique, pour chaque partie du système, il existe une norme qui fixe les contraintes de développement, comme la DO-254 pour le développement du support

matériel et la DO-178C pour le développement du logiciel. Ces normes développées par l'association RTCA [1] permettent à l'avioneur (qui développe et produit l'avion) de garantir le bon fonctionnement de l'avion à la compagnie aérienne (qui exploite l'avion). Un développement logiciel conforme à la DO-178C nécessite l'utilisation des *Development Assurance Levels* (DAL) qui sont des méthodes de développement plus ou moins strictes en fonction de la criticité de la fonction à réaliser. Les DALs sont répartis de A (le plus critique) à D (le moins critique). Pour les systèmes les plus critiques, il faut donner des *garanties fortes* concernant le temps de latence au pire cas des différentes fonctionnalités du système. Dans cet objectif, il faut identifier, pour chaque fonctionnalité, le comportement qui maximise le temps de latence pire cas et vérifier que ce temps de latence est inférieur à l'échéance temporelle correspondante.

Pour une fonctionnalité donnée, plus le système est complexe plus il est difficile de trouver la latence au pire cas. En effet, une unique fonctionnalité qui s'exécute sur un système composé d'un unique processeur sera plus facile à comprendre qu'une fonctionnalité qui s'exécute parmi plusieurs dizaines d'autres sur un système multiprocesseur car (i) les différentes fonctionnalités vont interférer entre elles et (ii) certaines interférences vont provenir des ressources matérielles.

1.1.2 Complexité des systèmes

Dans tous les domaines (automobile, aéronautique, etc), le besoin en terme de fonctionnalités augmente. Cette augmentation implique celle de la complexité des systèmes et, pour l'illustrer, nous utilisons le domaine de l'avionique. Intuitivement, la **complexité** d'un système est une mesure permettant d'évaluer si le fonctionnement et l'architecture interne d'un système sont faciles à comprendre. Par exemple, un lave-linge est simple car il embarque peu de fonctionnalités par rapport à un avion. L'augmentation de la complexité est présentée via les figures 1.1 et 1.2 sur lesquelles le nombre d'équipements éléctroniques, de fonctionnalités [2] et la taille du code [3] augmentent.

Le graphique figure 1.1 présente l'évolution du nombre de fonctionnalités et d'équipements électroniques embarqués dans un avion depuis 1970 jusqu'à 2005. Alors qu'en 1970 les systèmes embarqués sont inexistants dans les avions, ceux produits en 2005 embarquent une centaine d'équipements électroniques et plus de 100000 fonctionnalités.

Compte tenu de l'augmentation du nombre de fonctionnalités embarquées, la taille du code, comme présenté sur le graphique figure 1.2, a également augmenté en passant de quelques milliers à plus de 20 millions.

Augmenter le nombre de fonctionnalités nécessite d'augmenter (i) la quantité de fonctions codées et donc le nombre de lignes de code, et (ii) la quantité de ressources matérielles pour exécuter les différentes fonctionnalités. Cette augmentation du nombre de fonctionnalités et donc de la complexité des systèmes avioniques provient, entre autres, du remplacement des systèmes mécaniques par des systèmes électroniques, ou encore du développement de nouvelles fonctionnalités permettant, par exemple, l'amélioration des

1. pour "Radio Technical Commission for Aeronautics", association à but non lucratif créée en 1935

2. Jean-Bernard Itier, Artist2 - A380 Integrated Modular Avionics : The history, objectives and challenges of the deployment of IMA on A380

3. William R. Nichols, FAA Research Project on System Complexity Effects on Aircraft Safety : Candidate Complexity Metrics, May 2015

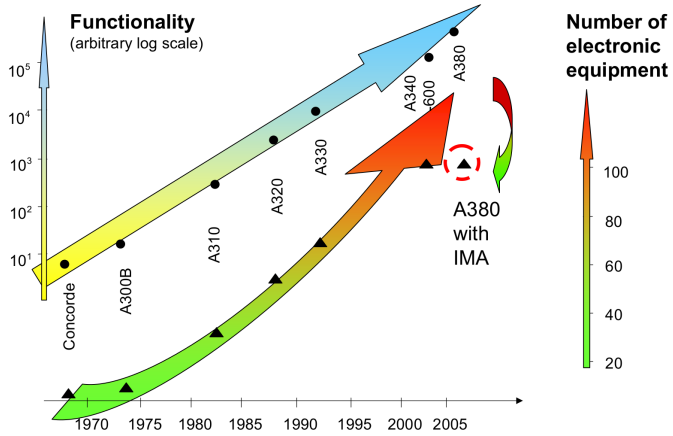

FIGURE 1.1 – Complexité des systèmes avioniques : les fonctionnalités et les ressources de calcul

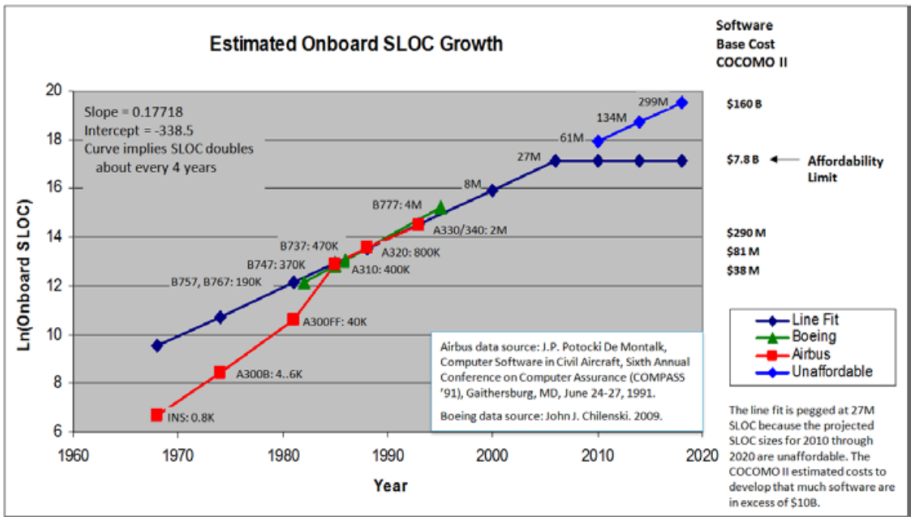

FIGURE 1.2 – Complexité des systèmes avioniques : le nombre de lignes de code

performances, de la sécurité ou encore du confort des passagers. Il est possible, via les chapitres ATA [4], d'avoir un aperçu des fonctionnalités existantes dans l'avionique. En voici quelques exemples :

— Commandes de vol électroniques
— Optimisation des plans de vol
— Diminution de la consommation de carburant
— Management du trafic aérien
— Système anti-collision
— Divertissement en vol pour les passagers

4. Pour "Air Transport Association of America", renommé ensuite "Airlines of America" (A4A), est une association américaine

Pour illustrer les problématiques liées au développement des systèmes avioniques, les objectifs et le fonctionnement du système de management du trafic aérien sont détaillés dans la section 1.2.

Voyons maintenant de manière plus précise, en utilisant l'exemple de système figure 1.3, comment les différentes parties d'un système contribuent à cette complexité. La représentation utilisée dans le cadre de cet exemple est présentée plus en détails dans la section 2.1. Globalement, le système figure 1.3 est composé de cinq ressources matérielles (en jaune) : *Utilisateur*, *Capteurs*, *Switch*, *Affichage* et *Architecture*. Cette dernière est elle-même composée de quatre ressources : *Processeur1*, *Processeur2*, *Co-processeur* et *Bus*. L'exemple est également composé de plusieurs fonctionnalités logicielles (en vert), chacune est une suite de plusieurs fonctions et chaque fonction est allouée sur les différentes ressources matérielles qui l'exécuteront.

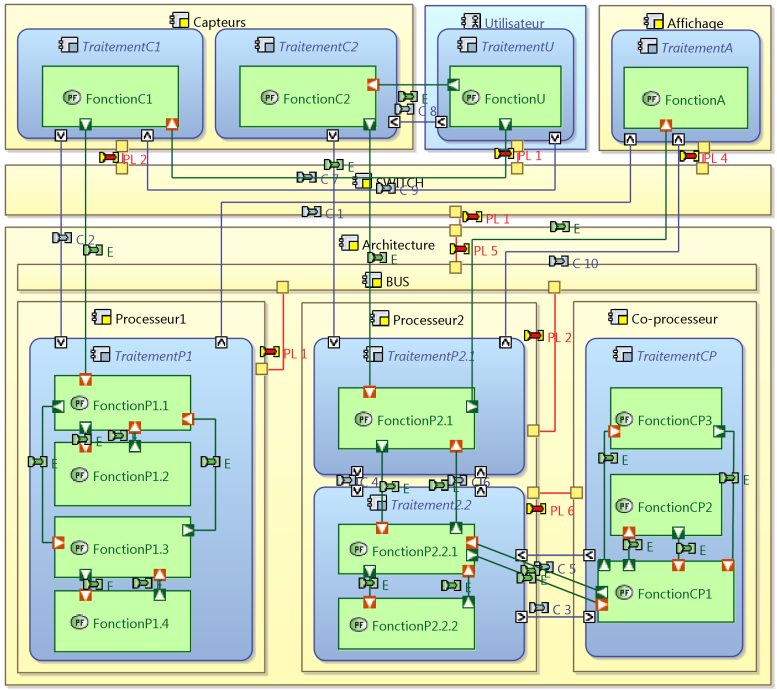

FIGURE 1.3 – Exemple de l'architecture d'un système

Chaque système est ainsi composé de deux parties principales qui sont :

1. Une partie **logicielle** : Chaque fonctionnalité logicielle et l'ensemble des fonctions qui la composent sont spécifiées et modélisées selon leur finalité. Par exemple, une fonctionnalité permettant de calculer des données de position et de les distribuer aux pilotes et aux aéroports est ainsi décomposée en plusieurs fonctions d'acquisition de données, de traitements, ou encore de cryptage. Sur l'exemple figure 1.3, l'exécution d'une fonctionnalité nécessite une action externe de l'*Utilisateur*, la récupération de données des *Capteurs*, plusieurs fonctions de traitements de données qui sont exécutées par l'*Architecture*, pour finir par la restitution des résultats via l'*Affichage*. Chaque fonction du modèle est mise en oeuvre par du code informatique. Un système est donc un ensemble complexe de différentes fonctions codées qui s'appellent entre

elles de manière à réaliser la fonctionnalité voulue. Plusieurs éléments contribuent à compléxifier le système : (i) les fonctionnalités cohabitent et interfèrent les unes avec les autres, (ii) certaines fonctions servent à réaliser des fonctionnalités différentes et (iii) plusieurs technologies (langages de programmation) peuvent être utilisées dans un même système.

2. Une partie **matérielle** : Comme le nombre de fonctionnalités augmente, il est nécessaire d'utiliser des architectures matérielles plus performantes ou d'en avoir un plus grand nombre. Dupliquer les ressources de calcul n'est économiquement pas une bonne solution (un module *Integrated Modular Avionic* pèse environ 10kg) car cela augmente le poids de l'avion, il transportera moins de passagers à chaque vol et sera donc moins rentable. Pour permettre l'augmentation du nombre de fonctionnalités embarquées dans un avion, les concepteurs utilisent des architectures plus performantes telles que des architectures multi-coeurs composées de plusieurs processeurs qui peuvent être hétérogènes. Les différentes fonctions peuvent être réparties sur les ressources de calcul, cependant, la complexité de ces architectures pose de nombreux problèmes. Dans le cas des architectures multi-coeurs hétérogènes il est difficile de prédire le comportement d'un système car (i) les ressources partagées (ex : mémoires) génèrent de nombreuses interférences et (ii) le temps nécessaire pour terminer une fonction change selon la ressource de calcul qui exécute cette fonction. Répartir les différentes fonctions contribuant à une même fonctionnalité sur des ressources différentes implique l'utilisation d'un moyen de communication (comme un *bus*) liant toutes les ressources matérielles, ce qui contribue à augmenter la complexité des systèmes. Pour finir, dans la majorité des cas, les entreprises ne développent pas le support matériel qu'elles utilisent et ne connaissent donc pas précisément son fonctionnement interne. En effet, développer l'ensemble du matériel, comme l'architecture interne du processeur, nécessite d'investir du temps et de l'argent. Pour éviter ces investissements, de nombreuses entreprises utilisent des composants dits *sur étagères* (en anglais **Component Off The Shell**), c'est à dire développés par d'autres entreprises. Elles intègrent ensuite ces composants à leurs propres architectures. Cela permet de limiter l'investissement mais cela pose un certain nombre de problèmes. Les COTS regroupent un grand nombre de composants (multi-coeurs hétérogènes) formant un système complet appelé **System on Chip**, et, comme mentionné auparavant, l'entreprise qui utilise le *COTS* ne connaît pas son architecture et son fonctionnement interne. Le comportement du *COTS* est donc difficilement prédictible.

L'aspect technique n'est pas la seule source de complexité dans un système. Compte tenu de la taille de l'avion et de l'ensemble des systèmes qui le composent, plusieurs sous-traitants et équipes contribuent à son développement. Pour un système donné, le nombre de sous-traitants augmente la difficulté des différents acteurs à comprendre le système. Si ces sous-traitants collaborent (c'est à dire partagent leurs données techniques) et contribuent aux mêmes fonctionnalités alors, il leur est possible de modéliser et comprendre le fonctionnement du système. Cependant, les sous-traitants développent souvent leurs fonctionnalités sous forme de boites noires [5] ce qui leur permet de protéger leurs compétences et leurs savoir-faire. Ce dernier point est possible car l'architecture globale du

5. c'est à dire sans divulguer le fonctionnement interne du système

système permet de garantir l'isolation temporelle des fonctionnalités les unes par rapport aux autres dans un avion. Dans le cas particulier du développement de systèmes redondants, plusieurs sous-traitants développent les mêmes fonctionnalités mais ne doivent pas collaborer. Ainsi, si en vol, un système (développé par un premier sous-traitant) ne fonctionne pas, le système redondant (développé par un second sous-traitant) permet de détecter la défaillance et de continuer à fournir des données valides. Dans ce cas, le développement du même système sans collaboration diminue les chances d'avoir simultanément la même panne sur les deux systèmes mais augmente la complexité des systèmes embarqués dans l'avion.

Le développement d'un système suit plusieurs étapes dont le but est de définir, développer et enfin tester le système. C'est dans ce contexte de développement d'un système qu'il faut pouvoir vérifier son comportement temporel. Différentes méthodes de vérification temporelle existent et elles sont plus ou moins utiles lors des différentes phases de développement d'un système.

1.1.3 Contexte de développement

Il existe de nombreuses méthodologies de développement de systèmes telles que le *cycle en V* ou les *méthodes agiles*. Ces méthodes permettent de gérer le développement des systèmes complexes soit en les découpant en plusieurs étapes (dans le cas du cycle en V) soit en permettant une gestion dynamique des fonctions à réaliser (dans le cas des méthodes agiles). Ces deux approches sont indépendantes et toutes les deux régulièrement utilisées dans l'industrie. Il est cependant plus facile de décrire les différentes étapes de développement d'un système via le cycle en V. Le cycle en V, présenté sur la figure 1.4, est composé de trois phases principales :

— La *phase descendante* durant laquelle le système est spécifié et conçu de plus en plus précisément. Les différentes propriétés (temporelles et autres) à respecter sont également décrites à ce moment.

— La *phase de développement* durant laquelle l'ensemble des composants sont développés de manière unitaire (software et hardware)

— La *phase montante* qui est une phase d'intégration durant laquelle les différents composants sont mis en place dans le système final. Les différentes propriétés requises en phase descendante sont également testées et vérifiées au cours de cette phase.

Après avoir conçu et validé le système, ce dernier peut être produit et mis en service. Les méthodes agiles diffèrent du cycle en V car il n'y a pas trois phases distinctes mais un ensemble de fonctionnalités qui sont choisies et développées de manière dynamique par les développeurs. Même si la manière de gérer le projet change, les différentes tâches de spécification, conception, développement, intégration et validation sont toujours nécessaires lors de l'utilisation des méthodes agiles.

Dans ce contexte de développement, les concepteurs peutvent utiliser les **approches basées sur les modèles** (en anglais *Model Driven Engineering*) et cette thèse s'inscrit dans le cadre de ces approches. Un modèle, comme présenté sur la figure 1.3, est une représentation simplifiée d'un système réel sur laquelle apparaissent les spécifications du système. L'utilisation de modèles permet de prendre en compte les différentes contraintes

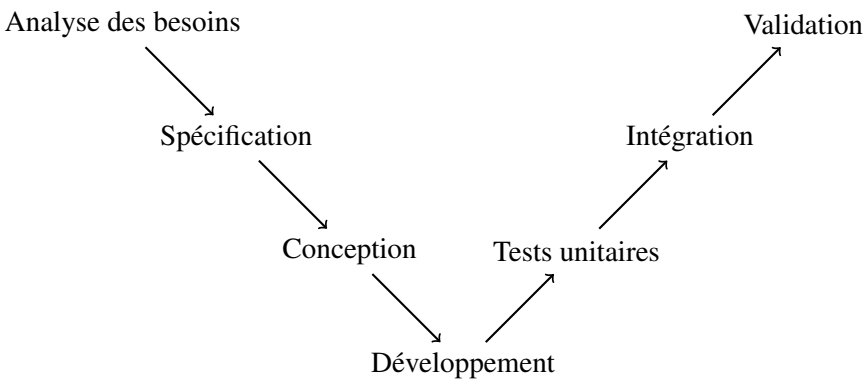

FIGURE 1.4 – Conception d'un système en utilisant le cycle en V

fonctionnelles ou non fonctionnelles d'un système. L'utilisation des modèles permet également de séparer les différentes étapes du développement d'un système, chaque étape a un ou plusieurs modèles dédiés. La phase de spécification est ainsi séparée des phases de conception qui elles même sont divisées en plusieurs étapes qui peuvent être : la conception de l'architecture matérielle, la conception de l'architecture logicielle ou encore l'allocation des composants logiciels sur le matériel. Il est à noter que, plus le modèle est proche de la réalité, plus l'utilisation du modèle et les résultats des analyses de ce modèle seront pertinents. Différents outils permettent de modéliser des systèmes comme l'outil open-source **Capella** [6] développé par Thales et utilisé dans le cadre de plusieurs projets. La modélisation d'un système sous Capella se fait lors de la phase descendante du cycle cependant les analyses peuvent être utiles durant toutes les étapes du cycle de vie d'un système. Modéliser un système sous Capella se déroule en 5 étapes :

1. L'analyse opérationnelle : Il faut commencer par définir le cadre dans lequel le système évoluera et à quels besoins le système doit répondre. Qui et/ou quoi interagira avec le système ? Quels sont les rôles et les objectifs du système ?
2. L'analyse du système : Il faut raffiner l'étape précédente en décrivant précisément les besoins du système et donc en décrivant les différentes fonctionnalités.
3. L'architecture logique : Il faut décrire l'ensemble des fonctionnalités avec la granularité la plus fine possible pour avoir les dépendances. À la fin de cette étape, l'architecture logicielle est entièrement définie.
4. L'architecture physique : Il faut spécifier l'ensemble de l'architecture matérielle, définir les couches logicielles qui vont permettre de synchroniser le logiciel avec le matériel (Operating System, middleware) et allouer les fonctionnalités logicielles sur les composants matériels. À la fin de cette étape, l'ensemble du système, matériel et logiciel, est spécifié.
5. L'intégration : Pour finir, il faut définir une stratégie d'intégration qui permettra de réunir tous les éléments au sein du même système.

Quelle que soit la compléxité du système, s'il est temps-réel et critique, il est nécessaire de borner le temps mis par une fonctionnalité pour s'exécuter. Les différentes méthodologies utilisées pour valider le comportement temporel d'un système sont plus ou moins

6. https ://www.polarsys.org/capella/

utiles en fonction de l'état de développement du système. Dans la section suivante, nous présentons le cadre général de la vérification temporelle et le cas particulier de l'analyse d'ordonnançabilité.

1.1.4 Validation temporelle

Il existe de nombreuses méthodes pour mesurer ou calculer les latences des différentes fonctionnalités d'un système comme : le test, le *model checking*, la simulation ou encore l'analyse d'ordonnançabilité. Ces méthodes peuvent être différenciées selon deux types d'approches, les approches basées sur les modèles et les autres qui utilisent l'architecture réelle du système. Chaque approche a ses avantages lors du cycle de vie d'un système.

L'utilisation de l'architecture réelle permet de limiter les approximations dues à l'utilisation de modèles qui sont, pour rappel, une abstraction de la réalité, et donc d'avoir des résultats qui correspondent exactement au comportement du système déployé. Pour cette raison, le test sur architectures réelles est une étape nécessaire afin de vérifier avant la phase de production/mise en service que le système fonctionne. C'est d'ailleurs la méthode la plus utilisée pour valider le comportement temporel d'un système. Cependant, en utilisant le test, toutes les exécutions possibles ne se réalisent pas nécessairement et le pire cas peut ne pas être identifié. De plus il est nécessaire d'avoir le système à tester, c'est-à-dire le matériel et/ou les logiciels (Operating System et applications). Il est possible de valider uniquement le matériel (resp. logiciel) en simulant la partie logicielle (resp. matérielle), cette méthode est appelé *Software In the Loop* (resp. *Hardware In the Loop*). Dans ce cas le concepteur doit avoir accès à au moins une partie du système, matérielle ou logicielle. Après le développement du système, dans la phase montante, le test permet d'exhiber des exécutions réelles du système.

Au début de la conception, le matériel tout comme les applications n'existent pas encore mais il est possible d'anticiper le comportement du système en utilisant des modèles. L'utilisation des modèles dans la phase descendante du cycle en V permet donc de caractériser le comportement temporel d'un système, d'anticiper son comportement et les pertes de performance qui pourront apparaître par la suite. Ces approches peuvent également être utilisées lors de la phase montante du cycle en V pour faciliter l'intégration des fonctionnalités au système final. Si le système est déployé avant l'intégration de certaines fonctionnalités, l'utilisation des modèles permet également d'anticiper les pertes de performances même si elle ne preut pas remplacer le test.

Pour résumer :

- Approches basées sur les modèles :
  - Avantages : (i) peut être utilisée à tous les instants du cycle de vie du système, (ii) permet d'anticiper le comportement et/ou les éventuelles pertes de performance du futur système.
  - Inconvénients : (i) le modèle est une abstraction de la réalité (approximation du comportement réel), (ii) l'abstraction augmente avec la complexité du système.
- Approches basées sur l'architecture réelle :
  - Avantages : (i) donne des résultats réalistes du comportement du système.
  - Inconvénients : (i) nécessite d'avoir le système à tester, (ii) avoir des tests représentatifs prend beaucoup de temps.

Compte tenu des besoins industriels, cette thèse traite uniquement le cas de l'**analyse d'ordonnançabilité** qui est une approche basée sur les modèles. L'objectif de cette analyse est de définir le comportement des fonctionnalités les unes par rapport aux autres de manière à calculer la latence au pire cas d'une fonctionnalité. Le *model checking* consiste à explorer l'ensemble des états possibles d'un système pour en déduire la latence au pire cas de chaque fonctionnalité. Cette méthode est très précise mais son utilisation n'est plus possible si le système est trop complexe ; ainsi l'utilisation de l'analyse d'ordonnançabilité est plus pertinente si les systèmes sont complexes. La simulation réalise successivement plusieurs exécutions du même système avec des paramètres différents (ex : temps d'exécution des fonctions, scénarios d'activations) et permet d'obtenir, pour chaque fonctionnalité, la plus grande valeur de latence parmi tous les résultats de ces simulations. La simulation est rapide à mettre en oeuvre même pour des systèmes très complexes mais ne fournit aucune garantie au pire cas. Basé sur le modèle, la simulation permet de fournir une valeur de latence réalisable, c'est-à-dire sans approximations cependant l'analyse d'ordonnançabilité est plus pertinente s'il faut calculer des latences au pire cas.

1.2 Exemple : le système de gestion du trafic aérien

Pour illustrer nos problématiques, nous introduisons un cas d'étude industriel : un système de gestion du trafic aérien (appelé en anglais Air Traffic Management). Un système ATM est un système temps-réel qui assiste les contrôleurs aériens et chaque pilote durant l'ensemble du vol, depuis l'aéroport de départ jusqu'à celui d'arrivée. Leur rôle principal est de maintenir des distances de sécurité entre les différents avions durant le vol ainsi que les phases de décollage et d'atterrissage. Le système ATM est un système temps-réel dur dont les contraintes temporelles sont critiques car recevoir une donnée trop tard peut mener à une collision. Il est important de pouvoir calculer le plus précisément possible les latences des différentes fonctions réalisées par le système pour garantir le respect de ces contraintes temporelles.

Commençons par préciser les différents rôles remplis par le système ATM. La position de chaque avion est calculée précisément à chaque instant durant le vol. Différents types de radars au sol permettent de calculer la position des avions et ces derniers calculent eux-même leur position via les satellites. Les différentes méthodes de calcul sont plus ou moins précises et sont utilisées dans différents contextes. Lors des phases de décollage et d'atterrissage (c'est-à-dire à l'approche des aéroports), le trafic est plus dense donc le risque de collision est plus important, et par conséquent les données doivent être plus précises et calculées plus souvent. Des données météorologiques sont également intégrées au système ATM, comme le gel, la neige ou encore le vent. En effet, les conditions difficiles peuvent imposer aux pilotes de modifier leurs plans de vol (ex : changer la destination), ce qui modifie le trafic aérien. Finalement, le système ATM assiste le pilote si la visibilité est mauvaise ou pour prévenir les vols d'oiseaux qui peuvent causer des défaillances des moteurs. Les données de position ainsi que celles en provenance des différents capteurs sont compilées et distribuées, par le système, aux pilotes ainsi qu'aux contrôleurs aériens afin qu'ils puissent gérer l'ensemble du trafic. Ces mécanismes permettent de gérer un trafic aérien important, d'éviter les collisions entre avions, de sécuriser les phases de décollage et d'atterrissage et de diminuer le temps de vol en diminuant le temps d'attente d'un avion avant le décollage et l'atterrissage. De plus, prendre en compte les aléas de la

météo permet de réduire le retard des différents vols.

Avec l'augmentation du trafic aérien, le système ATM doit régulièrement évoluer pour permettre de gérer un nombre toujours plus grands d'avion, exploiter les nouvelles technologies, améliorer les services existants ou encore automatiser certains mécanismes pour faciliter la gestion du trafic par des contrôleurs aériens [7].

Dans ce contexte, il est nécessaire d'analyser :

1. la latence au pire cas des fonctionnalités de l'ATM pour garantir le respect de ces échéances temporelles,
2. la capacité de l'ATM à évoluer pour savoir dans quelle mesure les futures évolutions pourront être intégrées.

1.3 Définition du problème

Cette thèse concerne l'analyse d'ordonnançabilité qui est une analyse basée sur les modèles. Commençons donc par présenter le lien entre l'exemple de système figure 1.3 modélisé avec l'outils Capella et l'analyse d'ordonnançabilité. Les concepteurs doivent définir les architectures matérielle et logicielle. Concernant l'architecture logicielle, ils doivent spécifier des fonctions, les allouer sur des ressources matérielles et les relier entre elles afin de réaliser les fonctionnalités spécifiées. Dans un système, chaque fonctionnalité qui s'exécute a une latence. Pour rappel, ce système est composé de plusieurs ressources matérielles (en jaune) et la ressource nommé *Utilisateur* (en bleu) est également matérielle mais l'utilisateur interagit directement avec elle. Les liens matériels sont modélisés par des traits rouges avec des carrés jaunes aux extrémités. Chaque fonctionnalité logicielle (en vert) correspond à une séquence de fonctions élémentaires (carrés verts). Le lien entre deux fonctions logicielles consécutives est modélisé par un trait vert avec, au niveau de la fonction source un carré vert et une flêche blanche et, au niveau de la fonction destinataire un carré orange avec une flêche blanche. Sur l'exemple figure 1.3, il y a deux fonctionnalités :

1. La première fonctionnalité est activée par l'*Utilisateur* (FonctionU, en haut au milieu), vient ensuite l'acquisition de données des *Capteurs* (FonctionC1, en haut à gauche), différents traitements (FonctionP1.x,) puis l'*Affichage* des données (FonctionA).
2. La seconde fonctionnalité est activée par l'*Utilisateur*, vient ensuite l'acquisition de données des *Capteurs* (FonctionC2), différents traitements (FonctionP2.x et FonctionCPx) puis l'*Affichage* des données (FonctionA).

Chacune des fonctions est alouée sur un processeur. Il est a noter que l'outil Capella impose aux concepteurs de regrouper, pour chaque processeur, les fonctions selon leur objectif (carré bleu). Nous ignorons ces regroupements qui ne sont pas pertinents pour l'analyse d'ordonnançabilité.

Pour rappel, la latence d'une fonctionnalité est la durée entre l'activation de la première fonction de la fonctionnalité (ex : action externe, arrivée d'une donnée en provenance d'un capteur) et la fin de la dernière fonction (ex : modification du système, envoie d'une donnée).

7. Ces évolutions font parties des objectifs cités dans les projets américain NextGen [3] et européen SESAR [4].

Dans notre exemple figure 1.3, la latence d'une fonctionnalité est la durée entre l'activation par l'*Utilisateur* (FonctionU) et la fin de la fonction d'*Affichage* (FonctionA). La latence peut changer d'une exécution à une autre en fonction des différentes interférences et perturbations qui interviennent durant son exécution.

La figure 1.5 illustre les variations possibles de la latence d'une fonctionnalité. Parmi toutes les exécutions possibles, la latence est comprise entre deux bornes, la plus petite valeur de latence est dite au *"meilleur cas"* tandis que la plus grande valeur est dite au *"pire cas"*. Dans cette thèse, nous nous interéssons à la latence au pire cas qui sert à garantir que chaque fonctionnalité respecte son échéance temporelle. Par définition, si chaque fonctionnalité respecte son échéance alors le système est ordonnançable.

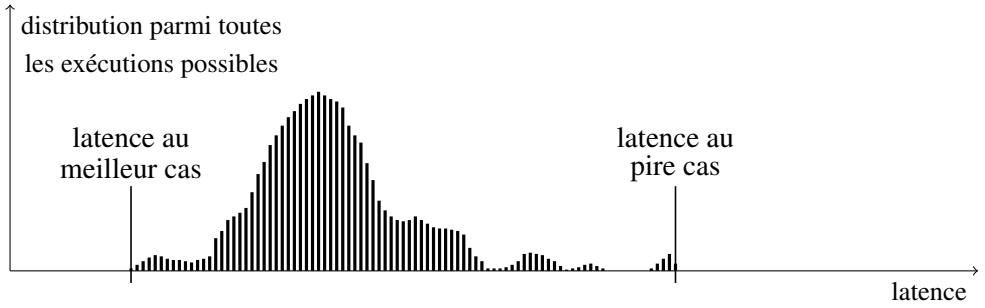

FIGURE 1.5 – Définition du problème d'incertitude

Dans le cadre de l'utilisation de l'analyse d'ordonnançabilité, plus le système est complexe plus il y a d'approximations. Tout d'abord, nous souhaitons **limiter les approximations lors du calcul d'un majorant sur la latence**, et, s'il y a des approximations, nous souhaitons **calculer un minorant pour évaluer la précision du majorant**. Les systèmes évoqués sont également soumis à de constantes évolutions compte tenu de leurs longs cycles de vie, de l'apparition continue de nouveaux besoins et de la nécessité d'adapter les systèmes aux technologies émergeantes. Dans ce contexte, nous voulons, en utilisant l'analyse d'ordonnançabilité, **garantir, avant son évolution, qu'un système sera ordonnançable après avoir évolué**.

1.3.1 Evaluation de la latence au pire cas

Étant donné que les systèmes considérés sont temps-réels et critiques, les concepteurs doivent caluler la latence au pire cas des différentes fonctionnalités pour garantir leur ordonnançabilité. Pour calculer cette latence, il faut tenir compte de la compléxité du système, et plus le système est complexe, plus il est difficile de caractériser son comportement temporel.

Comme nous l'avons vu précédemment le *model checking*, permet de calculer précisément des latences au pire cas mais ces méthodes ne conviennent pas aux systèmes complexes car plus le système est complexe plus l'analyse est longue et dans certains cas elle ne termine jamais. L'analyse doit converger vers une valeur de latence dans un temps raisonnable. Pour traiter les systèmes complexes, l'analyse d'ordonnançabilité est mieux adaptée, cependant elle ne permet pas d'obtenir la latence au pire cas mais un **majorant**, ce qui signifie que le calcul comporte des surapproximations. Garantir l'ordonnançabilité en utilisant des majorants conduit à surdimensionner le système car un majorant est supérieur

à la latence au pire cas. Ainsi pour respecter les échéances temporelles, l'architecture doit être plus performante. Dans un contexte industriel, une marge d'erreur de 20% mène à un surdimensionnement acceptable. Sur l'exemple figure 1.6, considérons que la latence pire cas est de 100 ms, alors si le majorant est inférieur à 120 ms, le surdimensionnement du système qui permet de garantir l'ordonnançabilité est acceptable. Un surdimensionnement plus grand est considéré comme trop onéreux, il est donc important de calculer des majorants le plus proche possible du pire cas mais également d'utiliser des minorants sur la latence au pire cas.

Calculer un **minorant** sur la latence au pire cas d'une fonctionnalité permet d'estimer la précision du majorant correspondant. Un minorant sur la latence au pire cas est une valeur de latence dont la réalisation est garantie possible compte tenu du modèle. Etant donné que la simulation consiste à exécuter un nombre borné de scénarios en fonction du modèle du système, il est possible d'exhiber un scénario d'exécution qui réalise la latence maximum obtenue par simulation. La latence maximum obtenue par simulation est nécessairement inférieure ou égale à la latence au pire cas et correspond ainsi à un minorant sur la latence au pire cas (cf figure 1.6). Une latence simulée permet donc d'évaluer la précision du majorant. La différence entre les deux bornes, minorant et majorant, représente la **précision** ou l'**incertitude** sur la latence au pire cas. Plus les approximations sont importantes, plus l'incertitude est grande (ou plus la précision est faible). Il est essentiel, pour réduire l'incertitude (ou augmenter la précision), d'avoir le moins d'approximations possibles entre le minorant et la latence au pire cas ainsi qu'entre le majorant et la latence au pire cas.

Il est à noter que, de la même manière que pour le calcul du majorant, le minorant peut être éloigné de la latence au pire cas ce qui mène à une mauvaise précision lors de l'évaluation de la précision du majorant. Considérons par exemple que, sur la figure 1.6, le calcul du majorant permette de trouver la latence au pire cas, si le calcul du minorant comporte d'importantes sous-approximations alors la précision sera mauvaise. Dans ce contexte, il faut trouver un minorant le plus proche possible de la latence au pire cas et nous souhaitons évaluer les avantages de l'analyse par rapport à la simulation pour calculer un minorant le plus proche possible du pire cas.

Pour rappel, le concepteur doit garantir que la latence au pire cas d'une fonctionnalité est inférieure à son échéance et donc que la fonctionnalité est ordonnançable. Si chaque fonctionnalité d'un système est ordonnaçable alors ce dernier est également déclaré ordonnançable.

Comparer l'échéance d'une fonctionnalité aux bornes (minorant et majorant) permet au concepteur de garantir ou non l'ordonnançabilité de la fonctionnalité. Les trois différents cas sont les suivants :

1. Si l'échéance est supérieure au majorant alors la fonctionnalité respecte son échéance temporelle et est garantie ordonnançable.
2. Si l'échéance est inférieure au minorant alors la fonctionnalité est garantie non ordonnançable et le système doit être modifié pour être ordonnançable.
3. Si l'échéance est comprise entre les deux bornes alors il est impossible de savoir si le système est ordonnançable et le concepteur n'a aucune garantie sur l'ordonnançabilité de la fonctionnalité.

Un majorant avec d'importantes surapproximations peut mener à déclarer un système non ordonnançable même s'il l'est et donc à surdimensionner inutilement le système. De

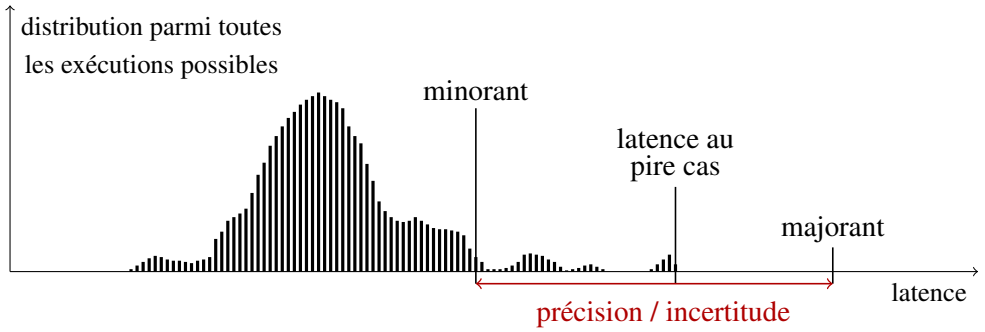

FIGURE 1.6 – Définition du problème d'incertitude

plus, l'absence de minorant ou un minorant avec d'importante sous-approximations ne permet pas au concepteur d'évaluer de manière pertinente la précision du majorant.

1.3.2 Le problème d'évolution

Les systèmes considérés dans le cadre de cette thèse embarquent des technologies qui évoluent rapidement et beaucoup de ces systèmes ont une longue durée de vie (du début de la spécification jusqu'au moment où le système est obsolète). Dans le cas de l'avionique, par exemple, la durée de vie d'un avion dépasse 30 ans. Il est donc nécessaire pour créer un système viable lors de toute sa durée de vie d'anticiper les possibles changements. Nous nous intéresserons au comportement des systèmes dans le cas où ces derniers doivent évoluer.

Les systèmes sont amenés à évoluer pendant l'ensemble de leur cycle de vie aussi bien lors du développement (spécification, conception, développement, tests et intégration voir figure 1.4) que lors des phases de production ou après déploiement (maintenance) pour différentes raisons comme la sécurité, la compétitivité industrielle, les enjeux environnementaux ou encore les nouveaux besoins utilisateurs comme le confort ou la connectivité.

Faire évoluer un système peut concerner les architectures matérielles ou logicielles. Dans le cadre de l'avionique, utiliser une architecture matérielle plus performante permet de gérer plus de fonctionnalités tout en diminuant le nombre de ressources matérielles et donc d'être plus compétitif. En effet, cela permet de diminuer le poids de l'avion et l'espace requis pour les équipements électroniques ce qui permet de transporter plus de passagers par voyage. Les évolutions matérielles ne sont pas traitées dans le cadre de cette thèse et nous nous intéressons exclusivement aux **évolutions logicielles**.

Une évolution logicielle peut apparaître pour plusieurs raisons : (i) corriger une erreur (ii) prendre en compte un nouveau standard ou encore (iii) répondre aux besoins des utilisateurs. Ces changements peuvent apparaître à différentes étapes du cycle de vie d'un système et il est nécessaire de les anticiper pour plusieurs raisons. Tout d'abord, les tests nécessaires après une évolution peuvent prendre du temps et l'analyse permet d'anticiper les pertes de performance et donc de réduire le temps d'immobilisation d'un système. De plus, faire évoluer un système et tester ensuite son ordonnançabilité peut être long et peu efficace. En effet, en fonction des évolutions possibles cela obligerait les concepteurs à tester l'ensemble des cas possibles pour définir ceux qui permettront de garantir l'ordonnançabilité du système. Il est donc important de fournir une analyse qui

permet d'identifier et de garantir les possibles évolutions pour un système sans refaire les analyses d'ordonnançabilité.

L'évolution d'un système peut également faire partie de la méthode de développement du système. C'est le cas de la méthode du Minimum Viable Product (MVP) qui consiste à développer un système par itérations successives comme présenté sur la figure 1.7.

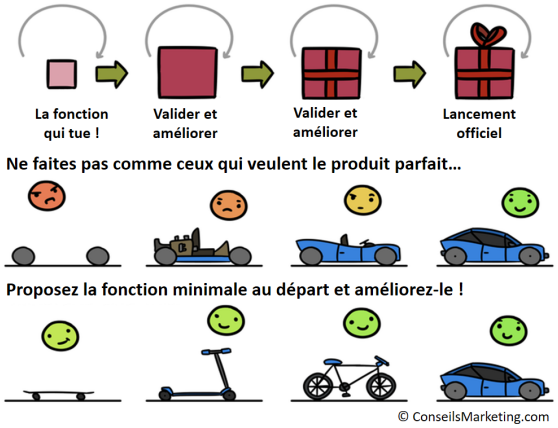

FIGURE 1.7 – Minimum Viable Product

La première version du système doit réaliser les fonctions élémentaires tandis que les versions suivantes réaliseront des fonctions de plus en plus avancées. A chaque itération, le destinataire du produit spécifie les fonctions supplémentaires à ajouter pour l'itération suivante. Afin de maîtriser chaque itération, il est nécessaire de calculer des marges afin de savoir quelles fonctionnalités pourront être ajoutées et quelles sont les contraintes sur les fonctionnalités qui permettront de garantir le respect des échéances temporelles.

Illustrons maintenant la problèmatique d'évolution en utilisant l'exemple figure 1.8 qui illustre la répartition des latences des différents jobs possibles d'une fonctionnalité. La répartition originale représentée en noir correspond à celle du graphique 1.6 tandis que la répartition des latences après avoir fait évoluer le système est représentée en vert. Faire évoluer un système correspond au rajout d'une ou plusieurs fonctionnalité(s), la suppression d'une fonctionnalité n'étant pas un scénario pertinent pour l'industrie. Rajouter une nouvelle fonctionnalité peut changer la répartition des jobs et par conséquent le majorant sur la latence au pire cas sera plus grand qu'avant l'évolution. Dans ce cas, il est important pour les concepteurs de savoir, avant d'intégrer l'évolution, si les garanties temporelles seront préservées ou non. De plus, afin d'anticiper les évolutions possible pour un système il est utile aux concepteurs d'identifier les évolutions les plus contraingnantes pour lesquelles il est possible de garantir l'ordonnançabilité. L'objectif est de maximiser l'utilisation du processeur par les différentes fonctionnalités. Ce premier problème correspond à la **flexibilité** des systèmes face aux évolutions.

Un problème lié au précédent est celui de la **sensibilité/robustesse** des systèmes concernant l'incertitude de certains paramètres temporels. Par exemple, le temps que prend une fonctionnalité ou une fonction pour s'exécuter peut être incertain. Dans ce cas, il est nécessaire de connaitre le plus long temps d'exécution que peut prendre la fonctionnalité (ou la fonction) pour s'exécuter tout en garantissant l'ordonnançabilité du système.

De manière générale, afin de garantir le comportement temporel des systèmes, les

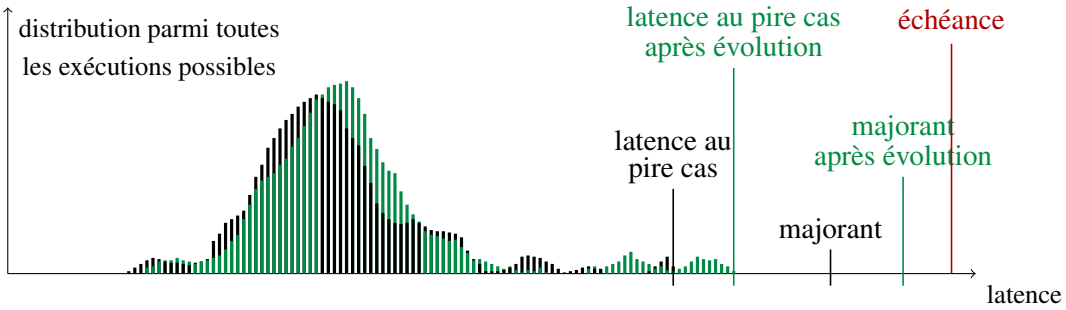

FIGURE 1.8 – Définition du problème de flexibilité

ressources sont surdimensionnées et donc sous-utilisées par les différentes fonctionnalités dans les systèmes temps-réels durs. Il est a noter que, en augmentant la précision des analyses, il devient possible de diminuer les marges qui sont gardées par le constructeur pour garantir les propriétés temporelles du système, et donc d'augmenter l'utilisation des processeurs.

1.4 Contributions

Nous avons, dans la section précédente, présenté les deux problèmes pour lesquels nous proposons une solution. Les analyses de latence de la littérature concernent des systèmes relativement simples et comportent un certain nombre d'approximations. Nous choissisons de traiter des systèmes plus complexes (avec des dépendances entre les fonctions) et nous caractérisons précisément les interférences pour ce type de systèmes, cependant nous ne traitons pas le cas des systèmes composés de plusieurs resources matérielles. Il est a noter que la performance des systèmes peut être diminuée pour faciliter le calcul des temps de latence des différentes fonctionnalités. Les architectures multi-processeurs génèrent beaucoup d'incertitudes compte tenu des ressources partagées (comme les mémoires). Dans ce cas, le concepteur peut choisir de n'utiliser qu'un seul des processeurs pour augmenter la prédictibilité. Le cas des systèmes mono-processeurs est donc pertinent dans un cadre industriel pour les systèmes temps-réels critiques.

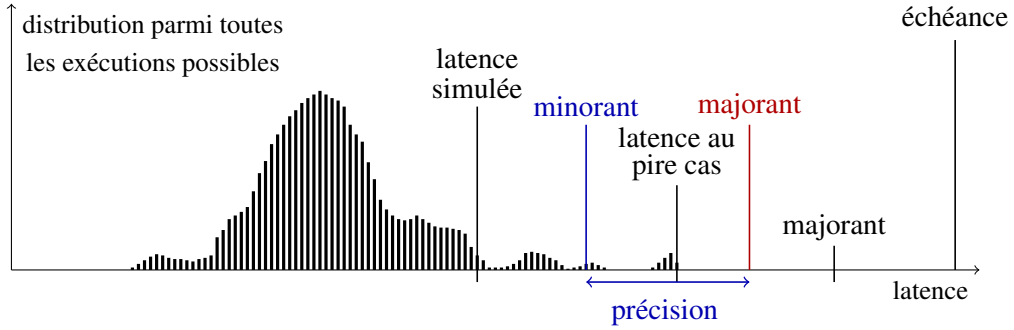

FIGURE 1.9 – Contributions

Notre contribution se décompose en trois parties :

1. Pour un système donné, nous proposons une analyse permettant de calculer un majorant sur la latence d'une fonctionnalité (en rouge sur la figure 1.9). Nous utilisons un modèle plus général que [2] et notre borne est plus précise que [1].
2. Nous adaptons l'analyse d'ordonnançabilité pour calculer un minorant sur la latence au pire d'une fonctionnalité et ainsi évaluer la précision du majorant correspondant (en bleu sur la figure 1.9). Cette borne est une alternative à la simulation qui nous permet dans certains cas d'améliorer la précision du minorant sur la latence au pire cas. L'utilisation de cette borne, plus précise, nous permet de diminuer l'incertitude.
3. Nous présentons une méthologie pour garantir l'ordonnançabilité d'un système amené à évoluer et pouvoir également calculer la robustesse/sensibilité de ce système face aux changements des paramètres temporels. Notre objectif est de maximiser l'utilisation du processeur de telle sorte que, après évolutions, le majorant se rapproche de l'échéance sans jamais la dépasser (pour garantir l'ordonnançabilité).

1.5 Plan

Le rapport est organisé ainsi : Le chapitre 2 est une description des différents modèles utilisés dont celui utilisé pour l'analyse d'ordonnançabilité. Nous utilisons ce modèle pour nos différentes analyses. Dans le chapitre 3, nous développons le calcul d'un majorant sur la latence au pire cas. Nous analysons ensuite les différentes approximations du majorant calculer un minorant dans le chapitre 4. Pour finir nous présentons une méthodologie pour faire évoluer des systèmes tout en garantissant leur ordonnançabilité dans le chapitre 5.

Chapitre 2

Modèle fonctionnel du système

Sommaire

Dans cette thèse, nous souhaitons analyser la latence au pire cas des fonctionnalités d'un système pour garantir son ordonnançabilité et, dans cet objectif, nous avons besoins d'un modèle de ce système. L'objectif de ce chapitre est de définir le **modèle temporel** qui est utilisé pour l'analyse d'ordonnançabilité.

Lors du développement d'un système, son modèle temporel n'est pas défini par les concepteurs qui utilisent des modèles dédiés principalement aux propriétés fonctionnelles. Ce modèle peut cependant être enrichi avec diverses propriétés non fonctionnelles comme la performance temporelle, la sécurité, la fiabilité, etc. Afin de définir la structure du modèle temporel d'un système, il faut commencer par comprendre ce que souhaitent modéliser les concepteurs, comment ils le modélisent et finalement comment extraire et exploiter les informations.

Dans ce chapitre, l'objectif est de formaliser la structure du modèle temporel d'un système. Dans cet objectif nous présentons les différents modèles et mécanismes utilisés dans le cadre du développement d'un système qui sont utiles à la création du modèle temporel. Ainsi, nous aborderons dans cet ordre :

1. les modèles utilisés par les concepteurs pour représenter les fonctionnalités et sur lesquels sont rajoutés des données temporelles pour spécifier le comportement du système.
2. les transformations qui permettent de résoudre le fossé sémantique entre les modèles fonctionnels et le modèle temporel.
3. le modèle temporel utilisé pour l'analyse d'ordonnançabilité.

2.1 Modèles fonctionnels du système

Les différents modèles présents dans l'outils Capella n'ont pas de sémantique formelle. Cependant, pour modéliser et analyser le comportement d'un système, les concepteurs doivent définir une sémantique formelle pour chaque modèle. Par la suite, nous présentons les modèles (et leur sémantique) utilisés par les concepteurs pour décrire le comportement fonctionnel d'un système. Afin de modéliser le fonctionnement d'un système, les concepteurs utilisent :

1. un **modèle de dépendance** pour modéliser les liens entre les différentes fonctions d'une fonctionnalité (cf l'exemple figure 2.1).
2. un **modèle d'exécution** pour modéliser les appels entre les différentes fonctions d'une fonctionnalité via un scénario d'exécution (cf l'exemple figure 2.2).

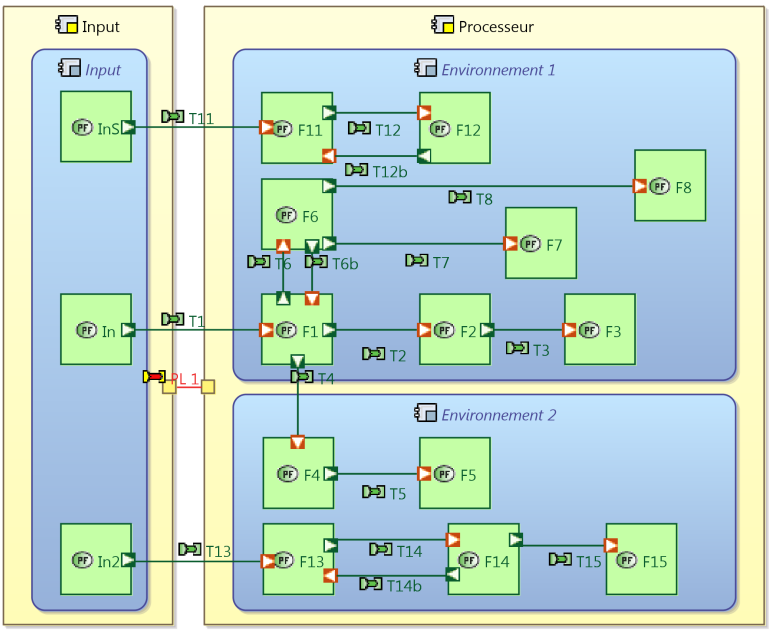

FIGURE 2.1 – Exemple de modèle de dépendance

Considérons d'abord le cas du modèle de dépendance dont un exemple est présenté sur la figure 2.1. Cet exemple est composé de deux processeurs (rectangle jaune) dont un fournissant des données d'entrées aux fonctionnalités. Dans le cadre de l'analyse d'ordonnançabilité d'un tel système, seule l'analyse du processeur qui exécute les fonctionnalités est nécessaire. La ressource fournissant les données peut être ignorée lors du calcul de latence, elle permet seulement de définir le modèle d'arrivée des données, et donc la fréquence d'activation des fonctionnalités. Pour rappel, les environnements (rectangle bleu) qui permettent de regrouper les fonctions selon leurs caractéristiques et/ou objectifs ne sont pas nécessaires à l'analyse d'ordonnançabilité et sont donc ignorés. Comme présenté auparavant, chaque fonction (rectangle vert) est ensuite allouée sur un environnement et une fonctionnalité correspond à un ensemble de fonctions reliées entre elles par des dépendances. L'exemple figure 2.1, comporte 13 fonctions et 3 fonctionnalités. Chaque

fonctionnalité a une unique activation qui correspond à l'activation de la première fonction qui permet de réaliser la fonctionnalité.

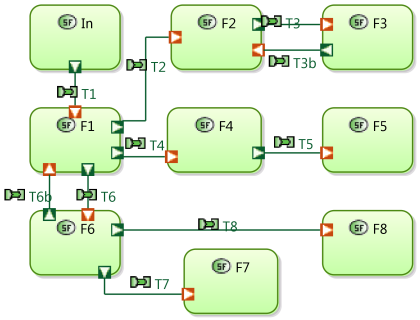

FIGURE 2.2 – Modèle d'une fonctionnalité en forme d'arbre

Considérons maintenant l'arbre fonctionnel présenté sur la figure 2.1 qui est reproduit sur la figure 2.2 et qui nécessite l'exécution des fonctions $F1$ à $F8$. Ce modèle permet de spécifier que $F1$ est la première fonction de l'arbre et elle appelle $F2$, $F4$ et $F6$. De même, $F2$ appelle $F3$, $F4$ appelle $F5$ et $F6$ appelle $F7$ et $F8$. Cependant ce modèle ne fournit aucune information sur l'ordre d'appel des fonctions par $F1$. Ainsi il est impossible de savoir l'ordre dans lequel les fonctions $F2$, $F4$ et $F6$ sont appelées par $F1$. De plus, il est impossible de savoir si les appels sont **bloquants**. L'appel de $F2$ par $F1$ est bloquant si la fonction $F1$ a besoin du résultat de $F2$ pour continuer son exécution. Pour avoir ces informations, il faut un modèle d'exécution qui permet permet d'exhiber la structure des exécutions, comme l'exemple figure 2.3 qui est un modèle d'exécution correspondant à la figure 2.2.

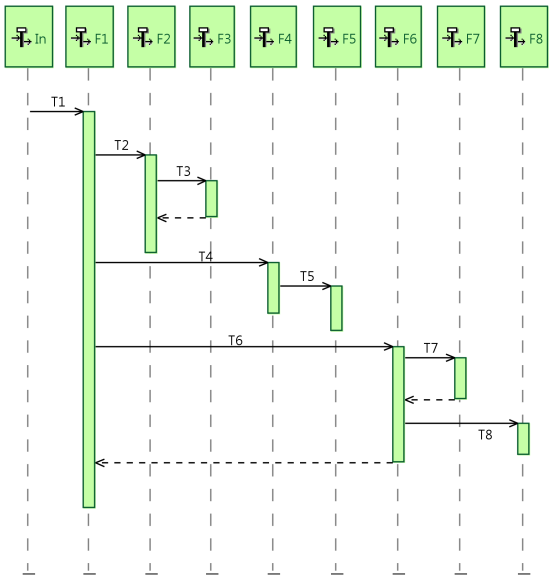

FIGURE 2.3 – Scénario d'exécution d'un arbre

Pour comprendre les deux types d'appels présents sur la figure 2.2, considérons une fonctionnalité composée de deux fonctions F_1 et F_2 telle que $F1$ appelle $F2$:

1. l'appel est **bloquant** : F_2 est activée pendant l'exécution de F_1 et F_1 a besoin du résultat de l'exécution de F_2 pour finir sa propre exécution.

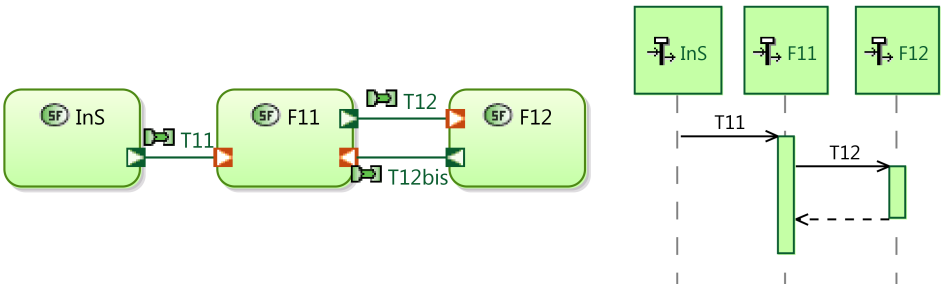

FIGURE 2.4 – Modèle fonctionnel d'exécution 3

2. l'appel est **non-bloquant** : F_2 est activée pendant l'exécution de F_1 et F_1 n'a pas besoin du résultat de l'exécution de F_2 pour finir sa propre exécution.

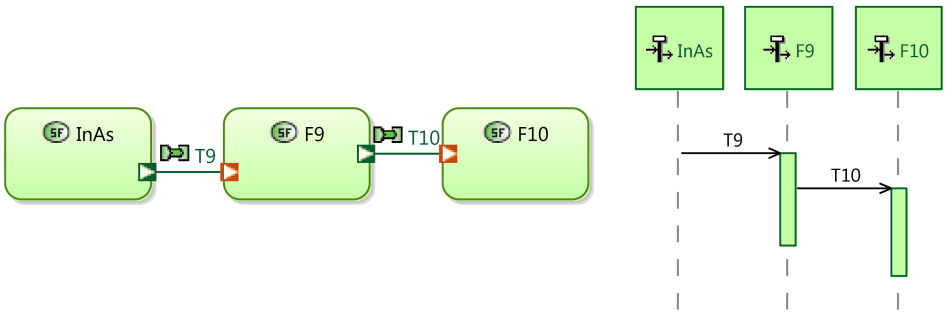

FIGURE 2.5 – Modèle fonctionnel d'exécution 4

Il est à noter que ces modifications nécessaires du modèle fonctionnel pour analyser son ordonnançabilité changent selon le type d'appel entre les fonctions de chaque fonctionnalité.

Il faut ensuite enrichir le modèle fonctionnel avec des informations temporelles en utilisant MARTE ou AADL. Cela permet, par exemple, d'annoter chaque fonctions avec un temps d'exécution et une priorité, ou encore un modèle d'activation sur les fonctionnalités. Il faut ensuite un modèle pivot afin d'extraire les données du modèle MARTE ou AADL et créer le modèle d'analyse temporelle. La description de ce modèle pivot est l'objet du projet *Waruna*.

2.2 Résolution du fossé sémantique entre modèles

Les modèles fonctionnels sont modifiés pour deux raisons :

1. Modifier les modèles fonctionnels permet d'obtenir un modèle temporel qui lui même peut être pris en charge avec les algorithmes de l'état de l'art. Dans ce cas, la transformation permet d'obtenir un modèle commun à de nombreuses analyses.

2. Modifier les modèles fonctionnels permet d'obtenir un modèle temporel simple ainsi, même s'il n'est pas analysable avec les algorithme de l'état de l'art, l'analyse est plus simple et rapide à développer.

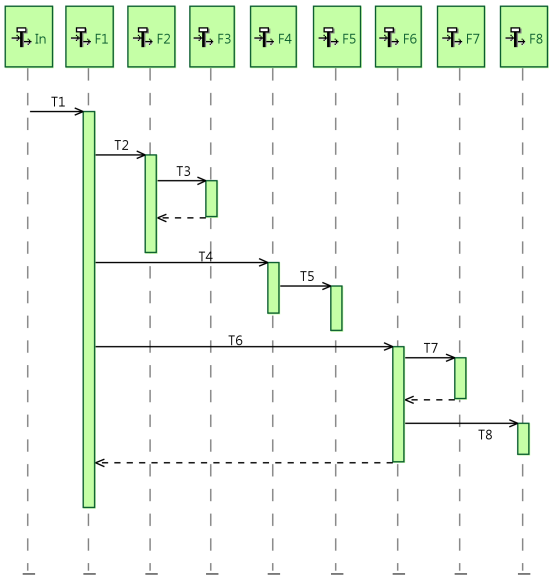

FIGURE 2.6 – Scénario d'exécution d'un arbre

Dans cette section, l'exemple figure 2.6 est utilisé pour comprendre les transformations nécessaires à l'obtention du modèle temporel. Considérons, comme dans la section précédente, une fonctionnalité composée de deux fonctions F_1 et F_2 telle que $F1$ appelle $F2$. Les transformations nécessaires sont les suivantes :

1. l'appel est **bloquant** : Soit t l'instant de F_1 auquel l'étape F_2 est activée alors l'instant t permet de séparer F_1 en F_{1A} et F_{1B}. Comme présenté sur la figure 2.7, F_2

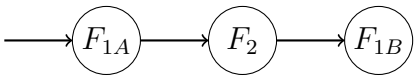

FIGURE 2.7 – Transformation résultant d'un appel non-bloquant

est activé à la fin de F_{1A} et F_{1B} est activée à la fin de F_2. Transformer l'exemple figure 2.4, revient à séparer $F11$ en deux. Ainsi, $F11$ s'exécutera jusqu'à l'activation de $F12$ qui s'exécutera à son tour et renverra des données à $F11$ afin que la fonction puisse se terminer.

2. l'appel est **non-bloquant** : Soit t l'instant de F_1 auquel l'étape F_2 est activée alors l'instant t permet de séparer F_1 en F_{1A} et F_{1B}. Comme présenté sur la figure 2.8, F_2 et F_{1B} sont activées en même temps à la fin de F_{1A}. Transformer l'exemple figure 2.5, revient à séparer $F9$ en deux. Ainsi, $F9$ s'exécutera jusqu'à l'activation de $F10$ et à ce moment s'activeront deux fonctions, $F10$ et une fonction correspondant à la fin de l'exécution de $F9$.

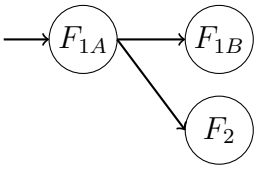

FIGURE 2.8 – Transformation résultant d'un appel non-bloquant

Compte tenu de l'exemple figure 2.6, nous savons que (i) $F1$ appelle dans cet ordre $F2$ puis $F4$ et enfin $F6$, (ii) $F1$, $F2$, $F5$ et $F8$ sont des feuille de l'arbre, (iii) $F2$ a besoin de $F3$ pour finir de même que $F6$ a besoin de $F7$ et $F1$ a besoin de $F6$. En fonction de ce modèle d'exécution 2.6, il est maintenant possible de modifier le modèle fonctionnel pour obtenir le modèle d'analyse temporelle figure 2.9. Le passage du domaine fonctionnel au domaine temporel implique le changement du vocabulaire utilisé pour décrire les modèles. Ainsi dans le domaine de l'analyse d'ordonnançabilité, les fonctions sont appelées **tâches** et les fonctionnalités sont des **arbres ou des chaînes de tâches**.

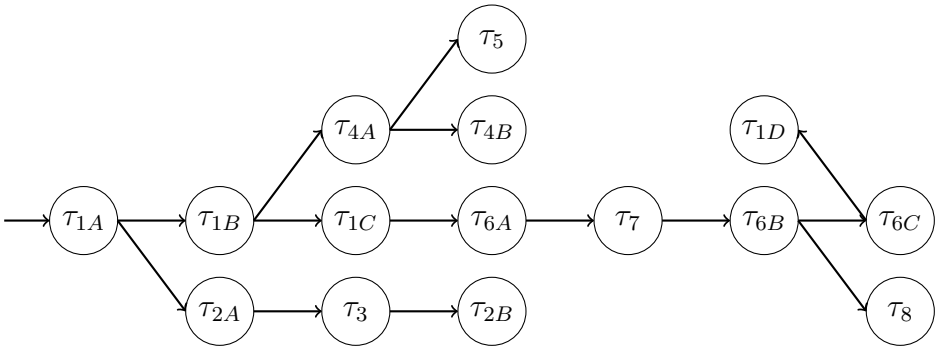

FIGURE 2.9 – Exemple d'arbre

L'arbre figure 2.9 s'active via une unique entrée, il ne dépend que d'un seul modèle d'activation et, dès qu'une tâche est activée, elle peut s'exécuter. De plus, le modèle ayant été modifié selon les appels, quand une tâche se termine, elle active immédiatement la/les tâche(s) suivante(s). En procédant ainsi pour l'ensemble des chaînes et des arbres, nous obtenons le modèle temporel figure 2.10 qui correspond au modèle temporel figure 2.1.

La différence entre le modèle temporel 2.10 et le modèle fonctionnel figure 2.1 est que chaque étape de 2.10 lorsqu'elle finit active l'étape suivante. Ce modèle temporel est toujours très complexe et l'exécution d'un arbre est prédéfinie par les priorités, autrement dit gérer cette complexité lors de l'analyse n'est pas nécessaire. Pour simplifier le modèle existant nous appliquons une nouvelle transformation qui consiste à linéariser les arbres pour ensuite analyser des chaînes. Parmi toutes les tâches qui sont allouées sur un processeur, celle qui s'exécute à un moment donné est choisie en fonction des priorités de toutes les tâches allouées sur ce processeur. La linéarisation d'un arbre dépend des priorités des différentes tâches qui composent l'arbre. Si deux tâches de priorités identiques attendent pour s'exécuter alors, la première activée est la première à s'exécuter (ordonnancement FIFO).

Prenons pour exemple la linéarisation de l'arbre 2.9 en utilisant les priorités du tableau 2.11 dans lequel la plus petite valeur correspond à la plus basse priorité. La linéarisation d'un arbre est effectuée via l'algorithme 1.

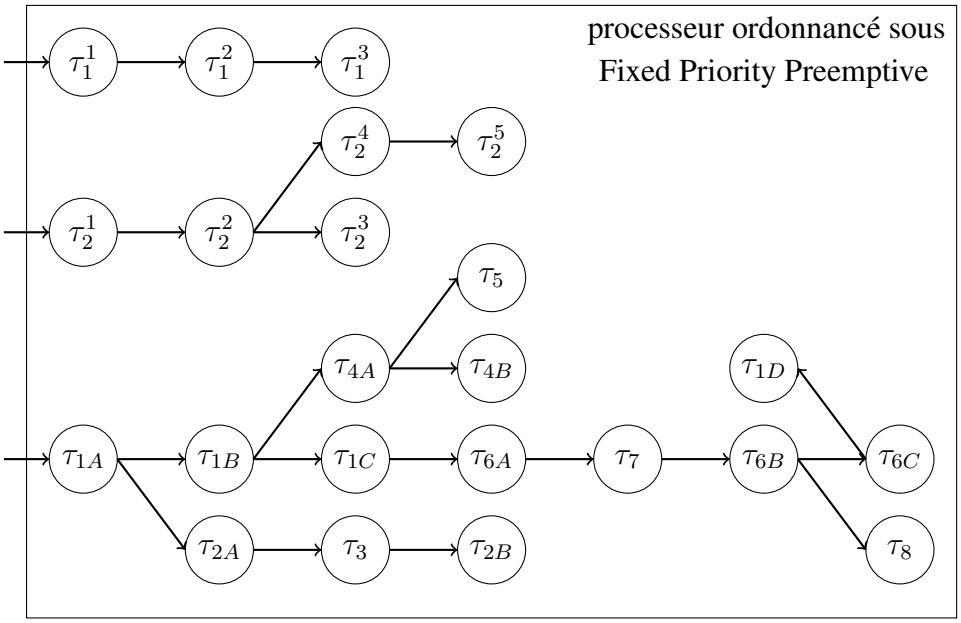

FIGURE 2.10 – Exemple de système

tâches	priorités		tâches	priorités
τ_1	5		τ_5	1
τ_2	3		τ_6	4
τ_3	8		τ_7	2
τ_4	6		τ_8	7

(2.1)

FIGURE 2.11 – Priorités des tâches dans l'arbre

La première tâche de l'arbre est la seule à être activée, elle va donc être ajoutée à la liste *chain*. Compte tenu des modifications déjà appliquées au modèle pour prendre en compte les appels, la terminaison de l'exécution de τ_{1A} active τ_{2A} et τ_{1B}. Dans ce cas, τ_{1B} est ajoutée à la liste *chain* car sa priorité est supérieure. Les tâches activées par la terminaison de τ_{1B} sont ajoutées à la liste $activated\_tasks\_list$. Ainsi, dans la liste des tâches activées τ_{1B} est alors remplacée par τ_{1C} et τ_{4A}. Parmi les tâches τ_{2A}, τ_{1C} et τ_{4A} qui sont activées, τ_{4A} s'exécute car sa priorité est supérieure. Le début de la chaîne est donc τ_{1A}, τ_{1B} puis τ_{4A}. Nous procédons ainsi jusqu'à ce que l'arbre entier soit linéarisé et on obtient la chaîne : $[\tau_{1A}, \tau_{1B}, \tau_{4A}, \tau_{4B}, \tau_{1C}, \tau_{6A}, \tau_{2A}, \tau_3, \tau_{2B}, \tau_7, \tau_{6B}, \tau_8, \tau_{6C}, \tau_{1D}, \tau_5]$.

En utilisant l'algorithme 1, il est possible de linéariser tous les arbres d'un système. On obtient ainsi un modèle entièrement composé de chaînes. Un exemple d'un tel système est présenté sur la figure 2.12.

Dans la suite de ce chapitre, nous allons formaliser ce modèle temporel pour ensuite développer nos algorithmes d'ordonnançabilité.

Algorithme 1 Linearization of a tree

```
1: activated_tasks_list=[τ_1]                ▷ On créer une liste des tâches qui sont activés
2: chain = [ ]                               ▷ On créer une liste vide qui contiendra la chaîne
3: while activated_tasks_list ≠ [ ] do
4:     task_hp = 0                           ▷ On initialise une tâche à la plus basse priorité possible
5:     for task ∈ activated_tasks_list do    ▷ On cherche la tâche de plus priorité
6:         task_p = get_priority()
7:         if task_p ≥ task_hp then
8:             task_hp = task
9:         end if
10:    end for
11:    add task_hp to chain                  ▷ Cette tâche est la suivante dans la chaîne
12:    remove task_hp from activated_tasks_list
13:    add tasks activated by task_hp to activated_tasks_list
14: end while
15: return chain
```

2.3 Modèle temporel du système

Dans cette section, nous décrivons formellement le modèle temporel sur lequel nos contributions sont basées. Un exemple est présenté de manière partielle sur la figure 2.13.

L'ensemble des paramètres utilisés pour caractériser les données temporelles d'un système sont des entiers positifs (c'est à dire dans $\mathbb{N}^+$). Ce choix permet également de traiter les systèmes avec des données temporelles dans $\mathbb{Z}^+$, pour cela il suffit de multiplier toutes les données du système par un même coefficient.

Nous considérons un système temps-réel uni-processeur (noté S) sur lequel sont allouées m chaînes de tâches qui sont ordonnancées via Fixed Priority Pré-emptive (FPP). Pour rappel, l'ordonnanceur permet de gérer les interactions entre les tâches et donc de définir, à chaque instant, celle qui s'exécute. L'ordonnanceur FPP est un ordonnanceur pour lequel, chaque tâche a une priorité fixe qui permet de définir son importance par rapport aux autre et, lorsqu'une tâche s'active, elle peut préempter et suspendre toute autre tâche de priorité plus faible. Par définition, les chaînes sont indépendantes ce qui signifie qu'elles ne peuvent pas partager une même tâche et par conséquent, il n'y a ni fork ni join. Considérer des chaînes indépendantes est possible compte tenu des différentes modifications effectuées dans la section 2.2.

Commençons par définir la notion de tâche qui est ensuite utilisée pour définir les chaînes de tâches.

Définition 2 (Tâche)**.** *Une **tâche** τ_a^i est définie par un couple (π_a^i, C_a^i) avec π_a^i la priorité et C_a^i un majorant sur le plus long temps d'exécution (appelé Worst Case Execution Time) de τ_a^i.*

La priorité de τ_a^i définit son importance, plus la priorité d'une tâche est haute plus elle est importante. À partir de son instant d'activation, chacun des jobs d'une tâche, s'ils ne sont pas interrompus, ont besoins au maximum d'une durée pour terminer leur exécution. Dans un système réel, le temps d'exécution d'une tâche varie d'un job à un autre. Dans le cas de l'analyse au pire cas, l'objectif est de maximiser la latence et dans notre cas cela

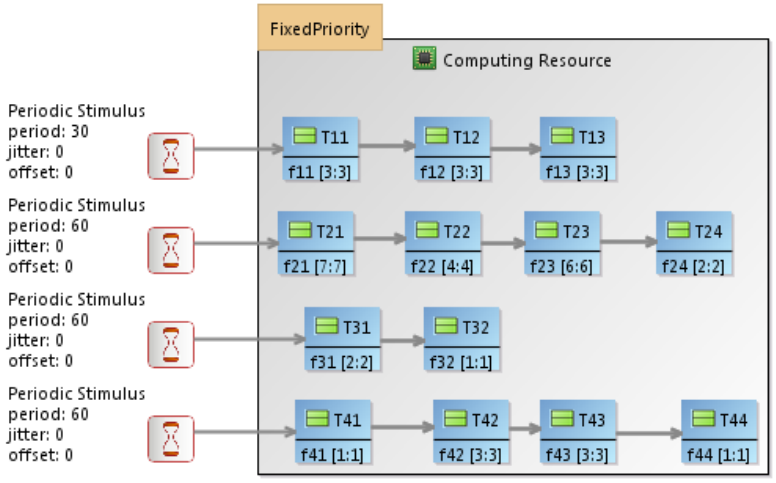

FIGURE 2.12 – Modèle d'analyse temporelle

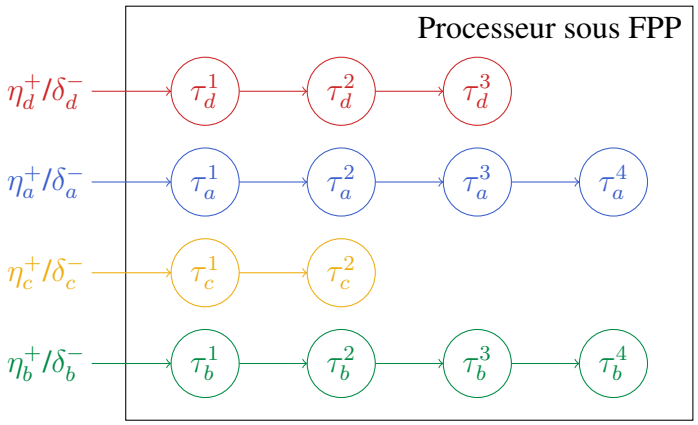

FIGURE 2.13 – Exemple de modèle d'un système

nécessite l'utilisation du plus long temps d'exécution. Ce plus long temps d'exécution étant très difficile à évaluer afin de donner des garanties au pire cas, il est courant d'utiliser un majorant sur ce plus long temps d'exécution. Dans la suite de cette thèse nous utilisons le terme WCET pour parler du majorant sur le pire temps d'exécution. Comme les analyses sont développées au pire cas, les autres notions concernant le temps d'exécution (ex : le meilleur temps d'exécution) sont inutiles.

Définition 3 (Chaîne de tâches)**.** *Une **chaîne de tâches**, notée $\sigma_a \in S$ avec $a \in \{1, \ldots, m\}$, est définie par :*

— *Une séquence finie de n_a tâches distinctes notées $(\tau_a^1, \tau_a^2, \ldots, \tau_a^{n_a})$. Il existe une relation de précédence entre les tâches de sorte que pour chaque tâche d'indice $i \in \{1, \ldots, n_a - 1\}$, τ_a^{i+1} est activée par la terminaison de la tâche τ_a^i.*

— *Un modèle d'activation (voir defs. 4 et 5) qui contraint les activations de la première tâche de la chaîne, c'est à dire τ_a^1.*

— *Une échéance relative D_a à l'instant d'activation (voir def. 13).*

— *Un modèle d'exécution, qui est soit synchrone soit asynchrone (voir def. 6).*

Par la suite, sauf indication contraire, toutes les tâches ont des priorités distinctes. Par convention, $\pi_a^i > \pi_b^j$ signifie que τ_a^i a une plus haute priorité que τ_b^j. Par conséquent, lorsque τ_a^i est activée, elle peut préempter τ_b^j. Ceci est possible grâce à la politique d'ordonnancement FPP.

Nous allons maintenant définir plus précisément les modèles d'activation des chaînes. Pour modéliser les activations d'une chaîne, nous utilisons les **courbes d'arrivées** (en anglais *arrival curves*) qui permettent de modéliser assez précisément des activations sporadiques et périodiques [5].

Définition 4 (Courbe d'arrivée)**.** *Une maximum (resp. minimum) arrival function* $\eta_a^+ : \mathbb{N} \to \mathbb{N}$ *(resp.* η_a^-*) renvoie, pour chaque intervalle* Δ*, une borne supérieure (resp. inférieure) sur le nombre d'activations successives de la chaîne* σ_a *qui peuvent arriver dans tout intervalle de temps* $[t, t + \Delta[$.

La fonction pseudo-inverse d'une fonction d'arrivée est la fonction de distance (en anglais *distance function*).

Définition 5 (Fonction de distance)**.** *Une minimum (resp. maximum) distance function* $\delta_a^- : \mathbb{N} \to \mathbb{N}$ *(resp.* δ_a^+*) retourne, pour chaque* $q \in \mathbb{N}$*, une borne inférieure (resp. supérieure) sur la longueur de chaque intervalle contenant* q *activations successives de la chaîne* σ_a*.*

Les chaînes activées sporadiquement n'ont pas de fonction de distance maximum et donc pas de fonction d'arrivée minimum. Cela signifie que deux activations d'une chaîne sporadique peuvent être arbitrairement éloignées l'une de l'autre. Par définition, l'intervalle de temps entre deux activations périodiques est fixe.

Le comportement temporel d'une chaîne de tâche σ_a est une séquence infinie de jobs, chacun est constitué d'un job de chacune des tâches de la chaîne tel que :

1. L'instant d'activation d'un job de τ_a^1 est définie par η_a^+.
2. L'instant d'activation d'un job de τ_a^{i+1} est l'instant auquel le job de τ_a^i termine.

Commençons par introduire les termes que nous utilisons dans cette thèse pour décrire les exécutions comme sur la figure 2.14. La figure 2.14 représente, entre autres, un **job** de la chaîne σ_a et par conséquent un job de chacune de ses tâches. La tâche τ_a^1 est **activée** à l'instant t_1, elle **s'exécute** de t_1 jusqu'à t_2 instant auquel elle **termine**. A l'instant t_2 la **terminaison** de τ_a^1 correspond à l'**activation** de τ_a^2 et dès son activation, τ_a^2 est **préemptée** par τ_d^1. τ_a^2 est ainsi **suspendue** de t_2 jusqu'à t_3 lorsqu'elle reprend son exécution. Peu importe son état si une tâche est activée mais pas encore terminée, elle est dite **en instance** et σ_a est en instance de t_1 à t_4.

Les **chronogrammes** sont utilisés pour représenter le comportement temporel d'un système et nous décrivons maintenant les conventions graphiques utilisés. Sur la figure 2.14, prenons pour exemple l'exécution de la chaîne σ_d. Le premier job de σ_d (ou de τ_d^1) est activé en même temps que σ_a et σ_c, et cette activation de la chaîne est représentée par une flèche pointée vers le bas. Dès qu'il est activé, il est préempté et donc suspendu, cette durée pendant laquelle le job est suspendu est représentée par un rectangle blanc. La tâche τ_d^1 commence son exécution à la terminaison de τ_a^1 et lorsqu'elle est terminée elle active la tâche suivante dans la séquence (c'est à dire τ_d^2), cette activation est représentée par la flèche en pointillés qui va de τ_d^1 vers τ_d^2. La chaîne σ_d s'exécute sans être préemptée jusqu'à la terminaison de τ_d^3 et la terminaison du job de σ_d est représentée par une flèche pointée

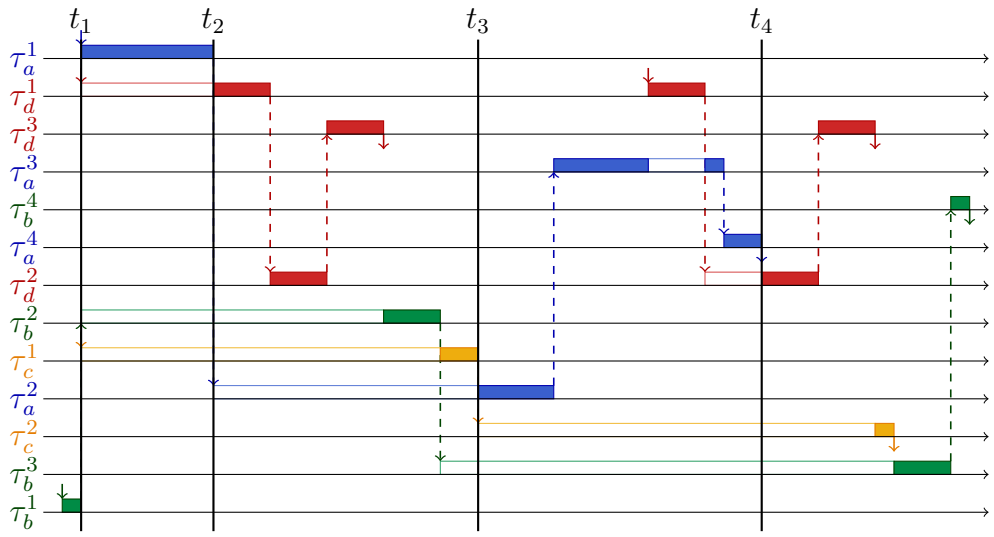

FIGURE 2.14 – Chronogramme correspondant à une exécution possible du système

vers le bas. Le second job de σ_d s'active aussi tôt que permis par la fonction d'arrivée c'est à dire $\delta_d^-(2)$ après l'activation de la première.

Les chaînes de tâches peuvent se comporter différemment si plusieurs jobs interfèrent entre eux.

Définition 6 (Chaine synchrone, asynchrone)**.** *Une chaîne peut être :*

- ***Synchrone*** *: un job de σ_a ne peut pas commencer son exécution avant que les jobs précédents de σ_a soient finis.*
- ***Asynchrone*** *: un job de σ_a peut suspendre les jobs précédents de σ_a.*

Les figures 2.15 et 2.16 représentent l'exécution d'une chaîne de trois tâches sur un processeur, seul change le modèle d'exécution de la chaîne. Sur la figure 2.15, le second job de σ_a est suspendu par l'exécution de τ_a^3 même si $\pi_a^3 > \pi_a^1$. En effet le modèle synchrone impose qu'un job soit terminé avant que le suivant puisse commencer son exécution. Ce n'est pas le cas pour le modèle asynchrone présenté sur la figure 2.16 sur lequel le deuxième job de σ_a préempte le premier job.

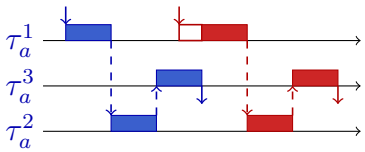

FIGURE 2.15 – Modèle d'exécution synchrone

Les jobs d'une chaîne asynchrone σ_a peuvent donc subir des interférences en provenance (i) des autres chaînes ; (ii) des jobs précédents de σ_a ; et (iii) des jobs suivants de σ_a. Une chaîne synchrone ne subira des interférences que de (i) et (ii). Il est à noter que le scénario d'exécution présenté sur la figure 2.14 ne permet pas de mettre en évidence un modèle d'exécution car aucuns jobs d'une même chaîne ne se superposent. Introduisons maintenant quelques notations.

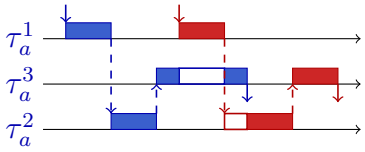

FIGURE 2.16 – Modèle d'exécution asynchrone

Notation 7. *Soit une chaîne $\sigma_a = (\tau_a^1, \ldots, \tau_a^{n_a})$, la séquence de tâches $(\tau_a^p, \ldots, \tau_a^m)$, $1 \leq p \leq m \leq n_a$, est notée $\sigma_{a[p..m]}$ et appelée une* ***sous-chaîne*** *de σ_a.*

Notation 8. *Soit une chaîne σ_a, $C_a = \sum_{i=1}^{n_a} C_a^i$ est le temps d'exécution de σ_a. Nous étendons cette notation à toutes les sous-chaînes : $C_{a[p..m]}$ est le temps d'exécution de $\sigma_{a[p..m]}$.*

Il est important de noter que nous ne faisons aucune hypothèse concernant l'ordre des priorités dans une chaîne. Les priorités ne sont ni croissantes ni décroissantes et cela augmente la complexité de l'analyse. L'exemple figure 2.14 donne un aperçu de la complexité du comportement temporel d'un système.

Exemple 9. *La figure 2.14 représente un ordonnancement possible pour un système avec quatre chaînes de tâches. On remarque que les priorités "qui se croisent" mènent à des interférences complexes. Par exemple, même si toutes les tâches de σ_a ont une priorité supérieure à τ_b^3, τ_a^2 a une plus basse priorité que τ_b^2, donc l'activation de τ_a^3 est indirectement retardée par la tâche τ_b^2 de priorité plus basse. Réussir à prendre en compte ces interférences est la principale difficulté de l'analyse de latence.*

Nous allons maintenant définir les notions nécessaires pour garantir l'ordonnançabilité du système, c'est à dire le respect des échéances temporelles.

2.4 Notions relatives à l'analyse d'ordonnançabilité

Nous commençons par définir la latence que nous souhaitons calculer puis nous définissons les propriétés que nous souhaitons vérifier.

Définition 10 (Latence)**.** *La* ***latence*** *d'un job d'une chaîne de tâches σ_a est la durée entre l'activation de τ_a^1 et la terminaison de $\tau_a^{n_a}$ du même job. La* latence au pire cas *(WCEL), notée L_a, de σ_a est la latence maximum de σ_a parmi tous les jobs possibles.*

L'objectif du chapitre 3 est de calculer le plus précisément possible la latence au pire cas de chacune des chaînes d'un système. Cette latence permet ensuite de garantir l'ordonnançabilité de chaque chaîne compte tenu de son échéance temporelle. Nous définissons maintenant la notion d'échéance temporelle. Pour rappel, l'échéance temporelle d'une chaîne est le temps maximum que peut prendre une chaîne pour s'exécuter, c'est à dire entre l'activation de la première tâche et la fin de la dernière tâche. Cette échéance est identique pour tous les jobs de la chaîne.

Définition 11 (Echéance temporelle relative)**.** *L'échéance temporelle relative d'une chaîne σ_a, notée D_a, est la durée maximale allouée pour l'exécution de tout job de σ_a.*

La notion d'échéance est inutile pour calculer la latence d'une chaîne mais permet de garantir l'ordonnançabilité.

Propriété 12 (Echéance temporelle absolue)**.** *En considérant un instant initial à $t_0 = 0$, l'échéance temporelle absolue du $q^{ème}$ job de σ_a, activé $t_a(q)$ après t_0, est :*

$$D_a(q) = t_a(q) + D_a \tag{2.2}$$

où D_a est l'échéance temporelle relative de σ_a

Nous distinguons deux types d'échéances en fonction de la valeur de l'échéance relative par rapport à la distance minimum entre deux activations consécutives.

Propriété 13 (Echéance contrainte ou arbitraire)**.** *Nous distinguons deux types d'échéances relatives pour une chaîne σ_a :* Contrainte *: $D_a \leq \delta_a^-(2)$.* Arbitraire *: $D_a > \delta_a^-(2)$.*

La distinction entre chaînes synchrones et asynchrones est inutile si toutes les chaînes ont des échéances contraintes car, dans ce cas, deux jobs ne peuvent pas se superposer (sinon les échéances temporelles ne sont pas respectées). Nous définissons maintenant la notion d'ordonnançabilité pour un système.

Définition 14 (Ordonnançable)**.** *Un système est* ordonnançable *si et seulement si toutes les jobs de toutes les chaînes du système respectent leur échéance temporelle.*

Notation 15. *Pour chaque valeur V, $\overline{V}$ et $\underline{V}$ sont respectivement un majorant et un minorant sur V.*

L'ordonnançabilité d'un système $S = \{\sigma_a\}_{a=1}^m$ est généralement établie en calculant, pour chaque chaîne σ_a, un majorant $\overline{L_a}$ sur sa latence au pire cas et en vérifiant que $\overline{L_a} \leq D_a$.

Calculer une très large sur-approximation sur L_a peut mener à déclarer des systèmes comme non ordonnançable même s'ils le sont. Le problème concernant l'analyse d'ordonnançabilité est double. Tout d'abord, il nous faut une analyse la plus précise possible. Deuxièmement, nous voulons trouver, pour chaque chaîne, un scénario *réalisable* qui est le plus proche possible du pire cas. Cette valeur sera un minorant sur le WCEL et nous permettra de mesurer le pessimisme de l'analyse du WCEL.

Chapitre 3

Majorer la latence au pire cas d'une chaîne

Sommaire

3.1 Etat de l'art et contribution

De nombreux articles concernent le calcul du temps de réponse d'une tâche, d'une chaîne, d'un arbre, etc. Nous nous intéresserons uniquement aux calculs de latence pour des tâches avec relations de précédence, c'est à dire des chaînes, comme définies dans la section 2.3.

Les deux articles qui se rapprochent le plus de nos travaux sont [2] et [1]. Dans [2] les terminologies utilisées sont différentes, en effet l'objectif est de calculer le temps de réponse de tâches mais chaque tâche est composée de une ou plusieurs sous tâches et chaque sous tâche à une priorité et un WCET. Malgré cette différence le modèle sous-jacent est le même et les deux papiers concernent le calcul de latence de chaînes sur mono-processeur ordonnancé sous ordonnanceur préemptif à priorités fixes.

Dans [2], Gonzalez Harbour et al. proposent une méthode pour analyser l'ordonnançabilité de systèmes constitués de tâches périodiques et synchrones, rappelons que les tâches du modèle de [2] correspondent à notre définition de chaîne. Nous parlerons de chaînes pour faire référence aux tâches de [2]. Tout d'abord, Gonzalez Harbour et al. utilisent une transformation pour avoir un modèle sous forme *canonique* qui permet d'avoir des tâches de priorités croissantes dans les différentes chaînes. Nous n'utilisons pas cette approche car elle n'est pas nécessaire pour calculer la latence et ne joue aucun rôle concernant la précision de l'analyse. Ils utilisent ensuite cette forme canonique pour comparer les priorités des tâches et définir des interférences et un *instant critique*. Nous améliorons cette approche selon trois directions : (1) Notre approche pour calculer un majorant sur la latence est plus formelle, elle est basée sur des preuves mathématiques et logiques ([2] propose une méthodologie sans fournir de preuves formelles). L'utilisation d'une approche formelle est nécessaire pour le calcul de minorants comme présenté au chapitre 4. (2) Notre analyse permet de prendre en compte les chaînes asynchrones avec échéances arbitraires (alors que l'hypothèse 3 de [2] exclut ce cas). (3) Et finalement notre approche peut être étendue pour calculer des *minorants* sur la latence.

Dans [1], J. Schlatow and R. Ernst étendent l'analyse compositionnelle (Compositional Performance Analysis) de [5] aux chaînes de tâches. Par rapport à l'analyse compositionnelle classique, cela réduit le pessimisme lors du calcul des majorants pour les systèmes composés de chaînes. Cependant l'analyse dans [1] comporte de nombreuses sur-approximations. Tout d'abord il n'y a qu'une seule définition pour calculer les deux concepts distincts de *busy-window* et *busy-time* sur q événements (voir sections 3.2.1 et 3.2.2), ce qui empêche de comprendre et de prendre en compte les interférences complexes tout en restant précis. Nous verrons que notre analyse permet d'améliorer de manière significative la précision des majorants. Ainsi par rapport à l'article [1], nous améliorons la précision du majorant sur la latence et nous calculons un minorant sur cette latence (ce qui permet d'évaluer la précision du majorant par rapport à la latence réelle).

Comme présenté dans l'article [6] et ensuite étendu à des systèmes plus complexes dans l'article [7], les *offsets* peuvent être utilisés pour modéliser des chaînes : les tâches avec différentes valeurs d'*offsets* sont groupées dans des *transactions* de telle sorte que les tâches d'une même *transaction* ne peuvent pas interférer entre elles. L'analyse de la latence basée sur les *offsets* permet d'améliorer l'analyse standard qui ne permet pas de prendre en compte les dépendances. Cependant J. Schlatow and R. Ernst montrent par expérimentation, que leur analyse dans l'article [1] permet de calculer des majorants plus précis que les analyses basées sur les *offsets*. Pour résumer, les articles [6] et [7] améliorent l'état concernant l'analyse de latence de chaînes mais ces analyses sont moins précises que celles développées dans l'article [1].

Une partie de l'état de l'art concerne des systèmes plus complexes, c'est le cas des articles [8] et [9]. Les systèmes considérés dans ces articles sont multiprocesseurs et les fonctionnalités sont représentées par des graphes avec *fork-join*. L'article [9] permet

également de prendre en compte le modèle de graphes orienté acyclique (en anglais *DAG*) en utilisant une transformation de modèle. Les modèles utilisés sont donc différents des nôtres car nous ne considérons pas les systèmes multiprocesseurs et nous ne considérons également pas les *fork-join*. Les politiques d'ordonnancement sont également différentes de celle que nous utilisons et dans le cas de [9] l'ordonnancement est global (une tâche est dynamiquement allouée sur le premier processeur disponible). De plus, l'objectif de ces articles est d'optimiser l'ordonnancement des graphes pour minimiser la latence et non pas de calculer la latence au pire cas. Le principal problème vient du fait que certaines tâches sont activées simultanément et donc il faut gérer ces tâches qui s'exécutent en parallèle.

Dans cette thèse, nous nous intéressons au chaînes avec des dépendances fonctionnelles dans lesquelles la terminaison d'une tâche permet d'activer la suivante ce qui n'est pas le cas des chaînes *cause-effect*. Les chaînes *cause-effect*, comme dans l'article [10] ont des dépendances de données. Les tâches sont activées indépendemment les unes des autres mais, il y a des dépendances de données entre les tâches ce qui signifie qu'une tâche lit des données produites par une autre tâche du système. L'analyse de ce type de systèmes se fait en calculant le temps maximum nécessaire pour traiter une donnée.

Comme notre analyse est proche de celles développées dans les articles [2] et [1], nous proposons une comparaison détaillée de ces deux approches avec la notre dans la section 3.6.

3.2 Principe général

Dans cette section, nous présentons l'approche globale que nous utilisons pour calculer un majorant sur la latence au pire cas d'une chaîne σ_a. Il est à noter que l'approche et les concepts nécessaires au calcul du majorant sont également utiles au calcul du minorant sur la latence au pire cas qui est développée dans le chapitre 4.

Pour calculer un majorant sur la latence au pire cas de σ_a il est nécessaire d'évaluer les **interférences** que peut subir cette chaîne. Intuitivement, l'interférence d'une chaîne σ_b sur σ_a est la durée pendant laquelle σ_b suspend σ_a dans un intervalle de temps donné. Il est important de remarquer que les interférences causées par une chaîne du système sur σ_a dépendent de la tâche de priorité minimum de chacune des deux chaînes. Nous utilisons cette observation pour définir une priorité pour chacune des chaînes. Même s'ils comparent les priorités des tâches des différentes chaînes les unes par rapport aux autres, ni [2] ni [1] n'utilisent la notion de priorité sur les chaînes ce qui rend leurs formules moins lisibles, et donc leurs analyses plus difficiles à comprendre. À l'inverse, utiliser la notion de priorité de chaîne permet une formulation proche des analyses classiques pour tâches indépendantes.

Exemple 16. *Dans l'exemple représenté figure 3.1, la tâche de σ_a qui a la priorité la plus basse est τ_a^2, de même, c'est τ_b^1 pour σ_b, τ_c^2 pour σ_c et τ_d^2 pour σ_d. Ces tâches sont ordonnées par priorité de la façon suivante : $\pi_b^1 < \pi_c^2 < \pi_a^2 < \pi_d^2$. En relation avec cet ordre de priorité, on constate que :*

1. *σ_d suspend σ_a une première fois pendant C_d (le WCET de σ_d) et une seconde fois pendant C_d^1.*
2. *σ_b (resp. σ_c) ne peut pas suspendre σ_a pendant C_b (resp. C_c) car l'exécution de σ_b (resp. σ_c) est elle-même préemptée dès l'activation de τ_b^1 ou τ_b^2 (resp. τ_c^2).*

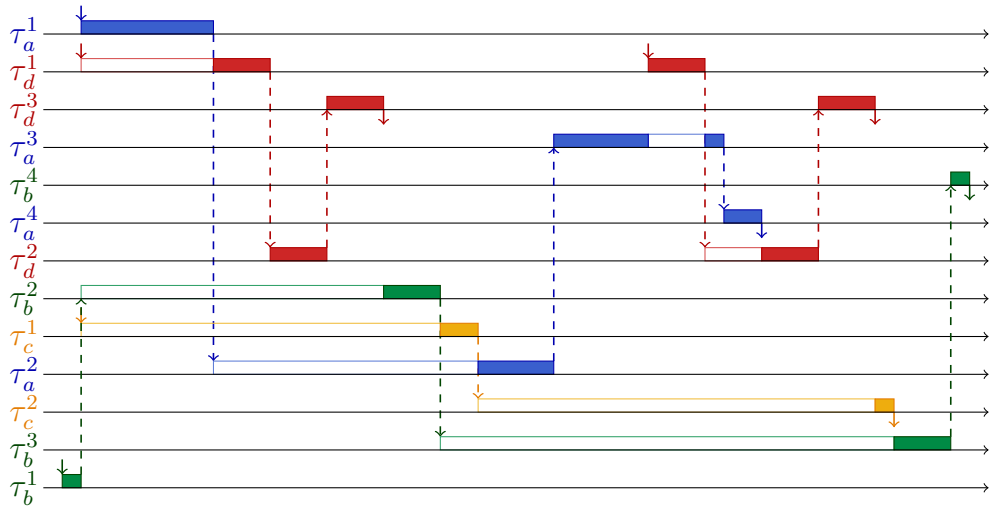

FIGURE 3.1 – Chronogramme correspondant à une exécution possible d'un système composé de 4 chaînes : σ_d (en rouge), σ_a (en bleu), σ_c (en jaune) et σ_b (en vert)

Ainsi les interférences causées par une chaîne sur σ_a dépendent de la tâche de plus basse priorité dans chacune des deux chaînes.

Définition 17 (Priorité d'une chaîne)**.** *La **priorité d'une chaîne** σ_a, notée π_a, est la priorité de sa tâche la moins prioritaire :*

$$\pi_a = \min_{i=1..n_a} \{\pi_a^i\}$$

Une tâche τ_b^j a une priorité plus basse qu'une chaîne σ_a si et seulement si $\pi_b^j < \pi_a$.

Par la suite, l'utilisation des priorités de chaînes permet de garder, pour l'analyse, une structure similaire à l'état de l'art [11]. Une chaîne σ_a a n_a différents niveaux de priorité car :

1. comme présenté dans la définition 3, n_a est le nombre de tâches dans la chaîne σ_a.
2. selon notre modèle de système défini en section 2.3, deux tâches d'un même système ne peuvent pas avoir la même priorité.

De plus, ce dernier point implique que les priorités des chaînes nous permettent de définir un *ordre total* entre les chaînes. La priorité d'une chaîne σ_a nous permet de séparer les interférences subies par σ_a dans un intervalle de temps en trois catégories : celles dues à des chaînes plus prioritaires, celles dues à des chaînes moins prioritaires et celles dues à la chaîne σ_a elle-même.

Notation 18. *$\ell p(a) = \{\sigma_b \in S | \pi_b < \pi_a\}$ (resp. $hp(a) = \{\tau_d \in S | \pi_d > \pi_a\}$) est l'ensemble des chaînes avec une priorité strictement inférieure (resp. supérieure) à σ_a. Nous utilisons également la notation $hpe(a) = hp(a) \cup \{\sigma_a\}$ et notons $\ell p_b(a)$ l'ensemble des tâches de σ_b qui ont une priorité plus basse que σ_a, c'est à dire, $\ell p_b(a) = \{\tau_b^j \in \sigma_b | \pi_b^j < \pi_a\}$.*

Cette notion de priorité peut être étendue aux sous-chaînes. La priorité d'une sous-chaîne $(\tau_a^j, \ldots, \tau_a^k)$ de σ_a est la priorité de la tâche la moins prioritaire de cette sous-chaîne

soit :

$$\pi_{a[j...k]} = \min_{i=j...k} \{\pi_a^i\}$$

L'approche proposée dans cette section pour calculer un majorant sur la latence au pire cas d'une chaîne est en certains points similaire à celle présentée dans [12] qui elle permet de calculer le temps de réponse au pire cas de tâches indépendantes allouées sur une ressource ordonnancée sous ordonnanceur non préemptif à priorités fixes (FPNP). En effet certains mécanismes induit par l'ordonnancement des chaînes de tâches sont également présent dans le cadre des ordonnanceurs FPNP.

Dans [12] le calcul d'un majorant sur le temps de réponse au pire cas d'une tâche τ_a se décompose en deux parties : (i) il faut calculer une fenêtre de temps maximal durant laquelle le processeur exécute des tâches de priorités supérieures ou égales à τ_a puis (ii) pour chaque activation dans cette fenêtre il faut calculer un temps réponse. Dans le cas des analyses pour tâches indépendantes allouées sur mono-processeur ordonnancé sous FPP, un unique calcul permet de calculer itérativement le temps de réponse de chaque job. De plus, les tâches de $\ell p(a)$ peuvent interférer avec τ_a compte tenu de la politique d'ordonnancement non préemptive. Dans [12], les auteurs introduisent et calculent un terme de *blocage* égal à l'interférence maximale des tâches moins prioritaires et un terme d'*interférence* égal à l'interférence maximale des tâches plus prioritaires.

La première étape pour calculer la latence au pire cas d'une chaîne est de calculer une fenêtre de temps maximal dans laquelle il faut ensuite analyser les latences des différents jobs. Pour une chaîne σ_a, cette fenêtre est appelée σ_a*-busy-window*.

3.2.1 Majorer la longueur des *busy-windows*

La plupart des analyses de latence pour systèmes mono-processeurs sont basées sur une notion de *busy-window* telles que [13], [11] ou encore [12]. Notre approche pour calculer les latences de chaînes utilise une notion similaire qui est identique à celle de [2], et qui elle-même étend le concept défini dans [11] aux chaînes de tâches.

Intuitivement, une *busy-window* pour une chaîne σ_a, appelée σ_a*-busy-window*, est un intervalle de temps durant lequel des tâches de priorité supérieure ou égale à π_a sont en instance. Commençons par définir la notion de σ_a*-busy-window* avant de présenter une méthode pour calculer un majorant sur la longueur de n'importe quelle σ_a*-busy-window*.

Définition 19 (σ_a-busy-window)**.** *Une σ_a**-busy-window** est un intervalle de temps maximum durant lequel il y a toujours au moins un job d'une tâche de priorité supérieure ou égal à π_a qui est en instance.*

En particulier, une σ_a*-busy-window* ne se ferme pas tant que toutes les exécutions des jobs de σ_a et des chaînes plus prioritaires ne sont pas terminées. Parmi les chaînes de priorité inférieure à σ_a, seules les tâches de priorité supérieure à σ_a font partie d'une σ_a*-busy-window*. Ainsi, une σ_a*-busy-window* :

1. s'ouvre avec l'activation d'une chaîne dans $hpe(a)$ de telle sorte que juste avant cette activation, aucune tâche dans $hpe(a)$ n'est en instance.
2. est un intervalle de temps durant lequel au moins une tâche de priorité supérieure ou égale à π_a est en instance.

3. se ferme avec la terminaison d'une tâche de priorité supérieure ou égale à π_a de telle sorte que juste après la terminaison de cette tâche aucune autre tâche de priorité supérieure ou égale à π_a n'est en instance.

Exemple 20. *Notre exemple figure 3.2 montre deux σ_a-busy-windows :*

1. *la première, $([t_1, t_2])$, s'ouvre avec l'activation de σ_a et se ferme avec la terminaison du deuxième job de σ_d.*
2. *la seconde, $([t_3, t_4])$, correspond à l'exécution de τ_b^4, elle s'ouvre avec l'activation de τ_b^4 et se ferme avec la terminaison de τ_b^4.*

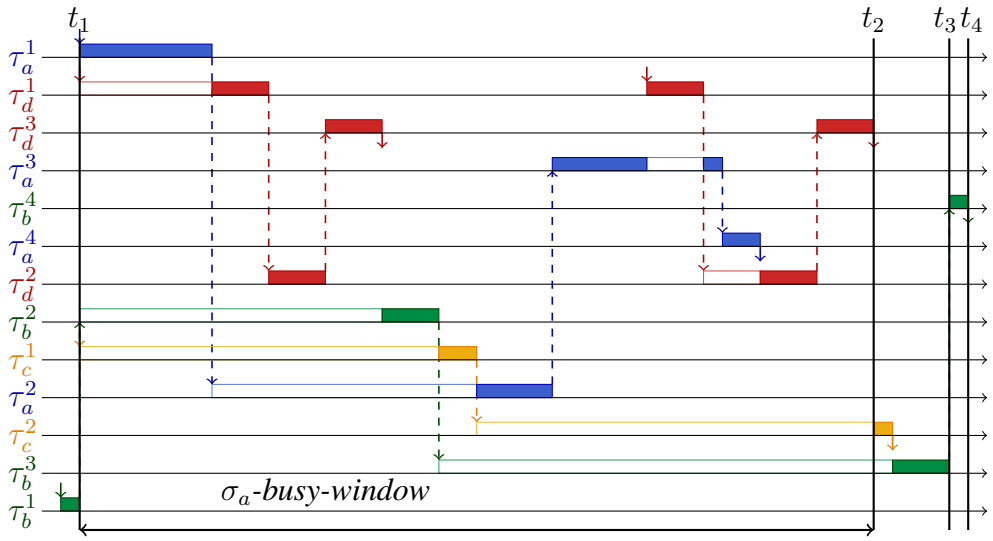

FIGURE 3.2 – Busy window : définition et calcul

Afin de calculer la latence au pire cas de σ_a il est nécessaire de considérer la plus longue σ_a-*busy-window* et on calculera un majorant sur la longueur des σ_a-*busy-windows*. Afin de calculer ce majorant, commençons par définir la notion d'interférence pour les chaînes moins prioritaires que σ_a.

Définition 21 (Interférence des chaînes de priorité inférieure)**.** *Soit une σ_a-busy-window. L'**interférence des chaînes de priorité inférieure** sur q événements d'une tâche τ_a^i, notée $\ell p\mathcal{I}_a^{i,q}()$, est la durée durant laquelle les chaînes de priorité inférieure à σ_a s'exécutent dans un préfixe de cette σ_a-busy-window qui termine à la terminaison du $q^{ème}$ job de τ_a^i.*

De la même manière qu'on calcule un majorant sur la longueur des σ_a-*busy-windows*, on calcule un majorant sur l'interférence des chaînes moins prioritaires qui est par définition supérieure ou égale à l'interférence maximum des chaînes moins prioritaires.

Définition 22 (Interférence maximum des chaînes de priorité inférieure)**.** *Soit une durée Δ. L'**interférence maximum des chaînes de priorité inférieure** sur q événements d'une tâche τ_a^i, notée $\ell p\mathcal{I}_a^{i,q}(\Delta)$, est la durée maximale durant laquelle les chaînes de priorité inférieure à σ_a peuvent s'exécuter dans n'importe quel préfixe de longueur Δ d'une σ_a-busy-window qui termine à la terminaison de la $q^{ème}$ exécution de τ_a^i.*

Le calcul de $\ell p\mathcal{I}_a$ est présenté dans les sections 3.4.1 (échéances contraintes) et 3.4.1 (échéances arbitraires).

Pour le calcul d'un majorant sur la longueur des σ_a-*busy-windows*, concernant la chaîne σ_a et les chaînes de $hp(a)$, chaque job activé s'exécute entièrement dans cette σ_a-*busy-window*. Il est à noter que l'interférence d'une chaîne de $hp(a)$ peut être partielle sur un job donné de σ_a dans cette σ_a-*busy-window* (cf sections 3.4.2 et 3.4.2).

Théorème 23. *La longueur de n'importe quelle σ_a-busy-window est majorée par le plus petit point fixe $\overline{BW_a}$ résultat de l'équation* (3.1) *:*

$$\overline{BW_a} = \ell p\mathcal{I}_a^{i,q}(\overline{BW_a}) + \sum_{\sigma_b \in hpe(a)} \eta_b^+(\overline{BW_a}) \times C_b \tag{3.1}$$

Démonstration. Selon la définition 21, les chaînes de plus basse priorité s'exécute au plus pendant $\ell p\mathcal{I}_a(\overline{BW_a})$ dans n'importe quel préfixe d'une σ_a-*busy-window* de longueur $\overline{BW_a}$. De plus, compte tenu de la définition 19, un job de σ_a ou d'une chaîne plus prioritaire ne peut pas être en instance au début d'une σ_a-*busy-window*. Toujours selon la définition 19, chaque job de σ_a ou d'une chaîne plus prioritaire, qui est activée dans une σ_a-*busy-window*, va s'exécuter entièrement avant la fin de cette σ_a-busy-window. Chaque chaîne $\sigma_b \in hpe(a)$ s'exécute donc pendant une durée au plus égale à C_b fois le nombre maximal de jobs de σ_b activés dans un préfixe de longueur $\overline{BW_a}$ (c'est à dire $\eta_b^+(\overline{BW_a})$. □

Pour une chaîne σ_a, l'équation (3.1) fournit un majorant sur la longueur des σ_a-*busy-windows*. En effet, il n'y a aucune garantie que l'interférence maximum des chaînes moins prioritaires et l'interférence maximum des chaînes de $hpe(a)$ puissent toutes les deux se réaliser simultanément, c'est à dire dans un même scénario d'exécution.

L'algorithme 24 permet de calculer, étant donnée la fonction $\ell p\mathcal{I}_a$, un majorant sur la longueur de n'importe quelle σ_a-*busy-window* en utilisant l'équation de point fixe (3.1). L'interférence maximale des chaînes moins prioritaires sur σ_a dépend du modèle d'exécution de chaque chaîne (synchrone ou asynchrone) et son calcul est détaillé dans les sections 3.4.1 (échéances contraintes) et 3.5.2 (échéances arbitraires).

Dans l'algorithme 24, le calcul du majorant sur la longueur des σ_a-*busy-windows* se déroule ainsi :

$2 \rightarrow 4$ - Initialisation : Le majorant sur la longueur des σ_a-*busy-windows* est au moins égal à C_a et dans cet intervalle de longueur C_a, l'interférence des chaînes moins prioritaires est $\ell p\mathcal{I}_a(C_a)$

$5 \rightarrow 12$ - Itération : A chaque itération, la longueur de la σ_a-*busy-window* est recalculée en prenant en compte : (i) l'interférence des chaînes moins prioritaires sur un préfixe de longueur BW_{prev}, (ii) le nombre d'activation de σ_a dans BW_{prev} et l'interférence de chaque chaîne plus prioritaire dans BW_{prev}. L'itération s'arrête quand le point fixe est atteint, c'est à dire quand $BW_{prev} = BW_{current}$.

Une fois calculé un majorant sur la longueur que peut atteindre une σ_a-*busy-window*, il nous faut un majorant sur le nombre de jobs de σ_a qui peuvent être activés dans n'importe quelle σ_a-*busy-window*.

Propriété 25. *Le nombre d'activations de σ_a dans une σ_a-busy-window est majoré par :*

$$\overline{K_a} = \eta_a^+(\overline{BW_a}) \tag{3.2}$$

Algorithme 24 Calcul d'un majorant sur la longueur des σ_a-*busy-windows* en utilisant l'équation (3.1)

function COMPUTE_UPPER_BOUND_BUSY_WINDOW(σ_a)
  $\ell p\mathcal{I}_a(C_a) = \text{COMPUTE\_LP\_INTERF}(\sigma_a, C_a)$
  $BW_{prev} = 0$
  $BW_{current} = \ell p\mathcal{I}_a(C_a) + C_a$
  while $BW_{prev} \neq BW_{current}$ **do**
    $BW_{prev} = BW_{current}$
    $\ell p\mathcal{I}_a(BW_{prev}) = \text{COMPUTE\_LP\_INTERF}(\sigma_a, BW_{prev})$
    $BW_{current} = \ell p\mathcal{I}_a(BW_{prev}) + \eta_a^+(BW_{prev}) \times C_a$
    for $\sigma_b \in hpe(a)$ **do**
      $BW_{current} = BW_{current} + \eta_b^+(BW_{prev}) \times C_b$
    end for
  end while
  return BW_{prev}
end function

Démonstration. Cette preuve découle du Théorème 23 qui nous donne un majorant sur la longueur des *busy-windows* et de la définition 4 selon laquelle $\eta_a^+(\Delta)$ est une borne supérieure sur le nombre d'activations de σ_a dans un intervalle de longueur Δ. La borne supérieure sur un majorant nous fournit un majorant sur le nombre d'activations de σ_a dans n'importe quelle σ_a-*busy-window*. □

Il est à noter qu'une σ_a-*busy-window* ne se ferme pas nécessairement à la fin d'un job de σ_a car il peut y avoir, par exemple, des chaînes plus prioritaires qui sont suspendues (cf figure 3.2). Ainsi pour calculer la latence au pire cas de σ_a il faut calculer un majorant sur la latence de chacun des $\overline{K_a}$ jobs de σ_a. Dans cet objectif, nous introduisons la notion de *busy-time*.

3.2.2 Majorer la longueur des *busy-times*

Pour chaque $q \in [1, \overline{K_a}]$, il faut calculer une valeur de ***busy-time*** qui correspond à la durée maximale depuis l'ouverture de la σ_a-*busy-window*, pour exécuter les q premiers jobs de σ_a dans cette σ_a-*busy-window*. La définition suivante est une adaptation aux chaînes de tâches du concept portant le même nom introduit dans [14].

Définition 26 (Busy-time sur q événements)**.** *Le **busy-time** sur q événements d'une chaîne σ_a (resp. une tâche τ_a^i), noté $b_a(q)$ (resp. $b_a^i(q)$), est la durée entre l'ouverture d'une σ_a-busy-window et la terminaison du $q^{ème}$ job de σ_a (resp. τ_a^i) dans cette σ_a-busy-window qui contient au moins q jobs de σ_a.*

Pour calculer la latence au pire cas d'une chaîne, il est nécessaire de calculer des majorants sur le temps mis par un job pour s'exécuter. Dans cet objectif, on a besoin de définir la notion de *busy-time* maximum sur q événements.

Définition 27 (Busy-time maximum sur q événements)**.** *Le **busy-time maximum** sur q événements d'une chaîne σ_a (resp. une tâche τ_a^i), noté $B_a(q)$ (resp. $B_a^i(q)$), est la durée*

maximale entre l'ouverture d'une σ_a-busy-window et la terminaison du $q^{ème}$ job de σ_a (resp. τ_a^i) dans n'importe quelle σ_a-busy-window qui contient au moins q jobs de σ_a.

Pour calculer un majorant sur le *busy-time* sur q événements de σ_a, nous aurons besoins du majorant sur le *busy-time* sur q événements de certaines tâches (les tâches concernées sont définies en section 3.4.1) de σ_a. Notre objectif est de calculer le *busy-time* sur q événements de la dernière tâche de la chaîne σ_a qui est, selon le théorème suivant, le *busy-time* sur q événements de σ_a.

Théorème 28. *Le busy-time sur q événements d'une chaîne σ_a est égal au busy-time sur q événements de sa dernière tâche :*

$$B_a(q) = B_a^{n_a}(q) \tag{3.3}$$

Démonstration. Cette preuve découle directement de la définition 3 de chaîne et de la définition 26 de *busy-time* sur q événements. Le *busy-time* sur q événements de τ_a^i est la durée maximale que cela peut prendre pour terminer l'exécution des q premiers jobs de τ_a^i. Compte tenu de la définition 3, la terminaison d'une chaîne correspond à l'instant où sa dernière tâche se termine (c'est à dire la terminaison de $\tau_a^{n_a}$) donc le *busy-time* du $q^{\text{ème}}$ job de $\tau_a^{n_a}$ est le *busy-time* du $q^{\text{ème}}$ job de σ_a. □

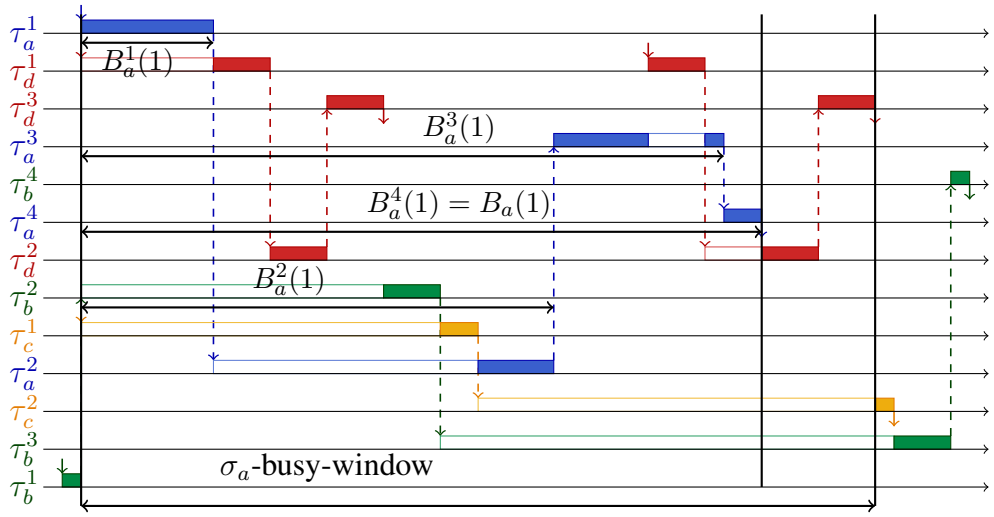

FIGURE 3.3 – Illustration des *busy-times* de σ_a

Exemple 29. *Sur la figure 3.3, comme la dernière tâche de σ_a qui a une priorité inférieure à toutes les tâches de σ_d est τ_a^2 alors nous n'avons pas besoins des busy-times de τ_a^1 pour calculer les busy-times de σ_a.*

Pour calculer le *busy-time* du $q^{\text{ème}}$ job de σ_a dans une σ_a-*busy-window*, il faut d'abord calculer le *busy-time* du $q^{\text{ème}}$ job des tâches τ_a^i. La première étape nécessaire au calcul du *busy-time* du $q^{\text{ème}}$ job de τ_a^i est de définir et calculer les interférences maximum sur les q premiers jobs de τ_a^i.

Comme énoncé précédemment les priorités sur les chaînes permettent de définir les interférences qui peuvent ainsi provenir des chaînes (i) moins prioritaires, (ii) plus prioritaires et (iii) de la chaîne σ_a.

Définition 30 (Interférence des chaînes de priorité supérieure)**.** *Soit une σ_a-busy-window. L'**interférence des chaînes plus prioritaires** sur q événements d'une tâche τ_a^i, notée $hp\mathfrak{I}_{b\to a}^{i,q}()$, est la durée durant laquelle une chaîne σ_b avec $\pi_a < \pi_b$ s'exécute dans un préfixe de cette σ_a-busy-window qui termine à la terminaison du $q^{ème}$ job de τ_a^i.*

Définition 31 (Auto-interférence des jobs suivants d'une chaîne)**.** *Soit une σ_a-busy-window. L'**auto-interférence** sur q événements d'une tâche τ_a^i, notée $self\mathfrak{I}_a^{i,q}()$, est la durée durant laquelle les jobs à venir de σ_a s'exécute dans un préfixe de cette σ_a-busy-window qui termine à la terminaison du $q^{ème}$ job de τ_a^i.*

De la même manière que pour les définitions 26 et 21 de *busy-time* et d'interférence des chaînes moins prioritaires, on calculera des majorants sur l'interférence des chaînes plus prioritaires et de la chaîne elle-même.

Définition 32 (Interférence maximum des chaînes de priorité supérieure)**.** *Soit une durée Δ. L'**interférence maximum des chaînes plus prioritaires** sur q événements d'une tâche τ_a^i, noté $hp\mathcal{I}_{b\to a}^{i,q}(\Delta)$, est la durée maximum durant laquelle une chaîne σ_b avec $\pi_a < \pi_b$ peut s'exécuter dans n'importe quel préfixe de longueur Δ d'une σ_a-busy-window qui termine à la terminaison du $q^{ème}$ job de τ_a^i.*

Définition 33 (Auto-interférence maximum des jobs suivants d'une chaîne)**.** *Soit une durée Δ. L'**auto-interférence maximum** sur q événements d'une tâche τ_a^i, noté $self\mathcal{I}_a^{i,q}(\Delta)$, est la durée maximum durant laquelle les jobs à venir de σ_a peuvent s'exécuter dans n'importe quel préfixe de longueur Δ d'une σ_a-busy-window qui termine à la terminaison de la $q^{ème}$ exécution de τ_a^i.*

L'interférence sur le *busy-time* du $q^{\text{ème}}$ job de τ_a^i est maintenant définie pour :

1. l'ensemble des chaînes moins prioritaires que σ_a ;
2. chaque chaîne σ_b plus prioritaire que σ_a ;
3. les jobs suivants de σ_a.

Il est à noter que l'auto-interférence sur le *busy-time* du $q^{\text{ème}}$ job de τ_a^i ne concerne que les jobs activés après le $q^{\text{ème}}$ car les autres sont inclus dans le calcul du *busy-time* du $q^{\text{ème}}$ job de τ_a^i.

La propriété suivante permet, en utilisant les interférences définies précédemment, de calculer un majorant sur le *busy-time* du $q^{\text{ème}}$ job de τ_a^i.

Propriété 34. *Le busy-time sur q événements de τ_a^i est majoré par :*

$$\overline{B_a^i(q)} = (q-1) \times C_a + C_{a[1..i]} + \ell p\mathcal{I}_a^{i,q}(\overline{B_a^i(q)}) + \sum_{\sigma_b \in hp(a)} hp\mathcal{I}_{b\to a}^{i,q}(\overline{B_a^i(q)}) + self\mathcal{I}_a^{i,q}(\overline{B_a^i(q)}) \quad (3.4)$$

Démonstration. Le temps maximum entre l'ouverture d'une σ_a-*busy-window* et la terminaison $q^{\text{ème}}$ job de τ_a^i est majoré par la somme des durées maximum nécessaires pour :

1. exécuter σ_a entièrement $q-1$ fois ;
2. exécuter le $q^{\text{ème}}$ job de σ_a jusque la tâche τ_a^i (c'est à dire, le WCET de la sous-chaîne $\sigma_{a[1..i]}$) ;

3. prendre en compte les interférences des
   (a) chaînes moins prioritaires que σ_a ;
   (b) chaînes plus prioritaires que σ_a ;
   (c) activations suivantes de σ_a.

□

De manière similaire au calcul du majorant sur la longueur des σ_a-*busy-windows*, l'équation (3.4) ne permet pas de calculer une valuer exacte (seulement un majorant) car il n'y a aucune garantie que les différentes interférences puissent se maximiser simultanément dans une même σ_a-*busy-window*.

Remarque 35. *Si les échéances sont contraintes ou si les chaînes sont synchrones alors il ne peut y avoir d'auto-interférence :*

1. *Si les échéances sont contraintes, l'existence d'auto-interférence implique que la chaîne n'est pas ordonnançable.*
2. *Par définition du modèle synchrone, les auto-interférences ne sont pas possibles.*

Exemple 36. *La figure 3.3 représente les différents busy-times pour différentes tâches de σ_a. En accord avec le théorème 28, le busy-time du premier job de τ_a^4 est le busy-time $B_a(1)$ du premier job de σ_a.*

Nous verrons par la suite que le *busy-time* des tâches de σ_a sert à calculer précisément l'interférence des chaînes de priorité supérieure à σ_a. C'est la raison pour laquelle nous n'avons pas besoin de calculer le *busy-time* de toutes les tâches d'une chaîne.

L'algorithme 37 est un calcul de point fixe qui permet d'obtenir un majorant sur le *busy-time* du $q^{\text{ème}}$ job de τ_a^i, il se déroule ainsi :

$2 \rightarrow 9$ - Initialisation : La longueur du préfixe Δ est initialisée à sa plus petite valeur possible. En effet, dans le *busy-time* du $q^{\text{ème}}$ job de τ_a^i, la chaîne σ_a s'exécute entièrement $q-1$ fois et le $q^{\text{ème}}$ job exécute ses tâches jusqu'à celle d'indice i. Ce préfixe permet également d'initialiser l'interférence des chaînes moins prioritaires. L'initialisation du *busy-time* $newB_a^i(q)$ dépend de l'indice de la tâche τ_a^i. Si pour ce $q^{\text{ème}}$ job, la tâche τ_a^i est la première pour laquelle il faut calculer un *busy-time* (ce qui signifie que $i = \ell t_a$) alors l'initialisation se fait avec le préfixe et les interférences initialisées précédemment. Sinon le *busy-time* du $q^{\text{ème}}$ job de τ_a^{i-1} a nécessairement déjà été calculé et il est utilisé pour initialiser $newB_a^i(q)$ qui est supérieur ou égal à $B_a^{i-1}(q) + C_a^i$.

$10 \rightarrow 20$ - Itération : Plusieurs valeurs du *busy-time* sont successivement calculées jusqu'à atteindre la plus petite valeur solution de l'équation (3.4). A chaque itération, sont ajoutées au *busy-time* $newB_a^i(q)$: (i) l'exécution des $q-1$ premiers jobs de la chaîne, (ii) l'exécution des tâches τ_a^1 à τ_a^i du $q^{\text{ème}}$ job de la chaîne et (iii) enfin les interférences en provenance des chaînes moins prioritaires, des chaînes plus prioritaires ou de la chaîne σ_a. Il est a noter que dans le cas des chaînes plus prioritaires notre définition concerne l'interférence d'une seule chaîne sur σ_a, il faut donc parcourir l'ensemble des chaînes plus prioritaires.

Algorithme 37 Calcul d'un majorant sur le *busy-time* du $q^{\text{ème}}$ job de τ_a^i en utilisant l'équation (3.4)

1: **function** COMPUTE_BUSY_TIME_TASK(σ_a, q, τ_a^i)
2: $\Delta = (q-1) \times C_a + C_{a[1..i]}$
3: $\ell p\mathcal{I}_a^{i,q}(\Delta) =$ COMPUTE_LP_INTERF(σ_a, Δ)
4: **if** $\tau_a^i = \tau_a^{\ell t_a}$ **then**
5: $newB_a^i(q) = (q-1) \times C_a + C_{a[1..i]} + \ell p\mathcal{I}_a^{i,q}(\Delta)$
6: **else**
7: $newB_a^i(q) = B_a^{i-1}(q) + C_a^i$
8: **end if**
9: $B_a^i(q) = 0$
10: **while** $B_a^i(q) \neq newB_a^i(q)$ **do**
11: $B_a^i(q) = newB_a^i(q)$
12: $\ell p\mathcal{I}_a^{i,q}(\overline{B_a^i(q)}) =$ COMPUTE_LP_INTERF$(\sigma_a, \overline{B_a^i(q)})$
13: $self\mathcal{I}_a^{i,q}(\overline{B_a^i(q)}) =$ COMPUTE_UB_SELF_INTERF$(\sigma_a, \overline{B_a^i(q)})$
14: $hp\mathcal{I}_a^{i,q}(\overline{B_a^i(q)}) = 0$
15: **for** $\sigma_b \in hp(a)$ **do**
16: $hp\mathcal{I}_{b\to a}^{i,q}(\overline{B_a^i(q)}) =$ COMPUTE_UB_HP_INTERF$(\sigma_a, \sigma_b, \tau_a^i, q, \overline{B_a^i(q)})$
17: $hp\mathcal{I}_a^{i,q}(\overline{B_a^i(q)}) += hp\mathcal{I}_{b\to a}^{i,q}(\overline{B_a^i(q)})$
18: **end for**
19: $newB_a^i(q) = (q-1) \times C_a + C_{a[1..i]} + \ell p\mathcal{I}_a^{i,q}(\overline{B_a^i(q)}) + self\mathcal{I}_a^{i,q}(\overline{B_a^i(q)})$
20: $+hp\mathcal{I}_a^{i,q}(\overline{B_a^i(q)})$
21: **end while**
22: **return** $B_a^i(q)$
23: **end function**

L'interférence des chaînes plus prioritaires sera calculée dans les sections 3.4.2 et 3.5.1 selon la politique des échéances.

L'algorithme 38 correspond à l'équation (3.4) et sert au calcul du *busy-time* du $q^{\text{ème}}$ job de σ_a dans une σ_a-*busy-window*.

L'algorithme se déroule ainsi :

2 - Initialisation : La variable q_{less} servant à parcourir les différents jobs de σ_a est initialisée à 1. En effet, le *busy-time* du $q^{\text{ème}}$ job de σ_a est calculé en traitant itérativement le *busy-time* de chaque job de σ_a dans une σ_a-*busy-window*, c'est à dire tous les *busy-times* de 1 jusqu'à q.

3 $\rightarrow$ 8 - Itération : Ensuite, pour chaque job q_{less} de la chaîne, il faut calculer itérativement le *busy-time* de chacune des tâches depuis celle d'indice ℓt_a (cf section 3.4.2) jusqu'à celle d'indice n_a (la dernière tâche) qui correspond au *busy-time* de la chaîne.

Le résultat est également valable si une ou plusieurs interférences sont majorées. Pour majorer le *busy-times* sur q événements, nous devons majorer les interférences correspondantes. Nous montrerons dans les section suivantes comme atteindre cet objectif. Avant cela, voyons comment les majorants sur les *busy-times* sur q événements sont utilisés pour majorer la latence d'une chaîne.

Algorithme 38 Calcul du *busy-time* du $q^{\text{ème}}$ job de σ_a en utilisant Eq. (3.3)

```
1: function COMPUTE_BUSY_TIME_CHAIN(σ_a, q)
2:     q_current = 1
3:     while q_current ≤ q do
4:         for i ∈ [ℓt_a, n_a] do
5:             B_a^i(q) = COMPUTE_BUSY_TIME_TASK(σ_a, q, τ_a^i)
6:         end for
7:         q_current = q_current + 1
8:     end while
9:     return B_a^{n_a}(q)
10: end function
```

3.2.3 Majorer la latence au pire cas

Une fois calculé un majorant sur les *busy-times* de chacun des q jobs de σ_a ($q \in [1, \overline{K_a}]$), un majorant sur la latence au pire cas peut être obtenu facilement via le théorème suivant.

Théorème 39 (Latence pire cas)**.** *La latence au pire cas de la chaîne σ_a est majorée par :*

$$\overline{L_a} = \max_{q \in [1, \overline{K_a}]} \{B_a(q) - \delta_a^-(q)\} \tag{3.5}$$

Démonstration. La propriété 23 permet de calculer un majorant sur la longueur des σ_a-*busy-windows* et la propriété 25 un majorant sur le nombre d'activations de σ_a dans une σ_a-*busy-window*. Il découle de la définition 26 que le $q^{\text{ème}}$ job de σ_a ne peut pas se terminer plus tard que $B_a(q)$ après l'ouverture d'une σ_a-*busy-window*. De plus, compte tenu de la définition 4, σ_a ne peut pas être activé plus tôt que $\delta_a^-(q)$ après l'ouverture d'une σ_a-*busy-window*. La différence entre la terminaison au plus tard et l'activation au plus tôt du $q^{\text{ème}}$ job est un majorant car il n'y a aucune garantie que les valeurs $B_a(q)$ et $\delta_a^-(q)$ puissent se réaliser simultanément. Pour conclure, selon la définition 10, la latence au pire cas est la latence maximum parmi toutes les latences possibles du système. D'où le résultat. □

L'équation 3.5 permet de calculer un majorant sur la latence au pire cas de σ_a même si les *busy-times* $B_a(q)$ des q jobs de la chaîne sont des majorants. En effet, la latence de la $q^{\text{ème}}$ job, calculée dans l'équation 3.5, est basée sur le plus long *busy-time* $B_a(q)$ et l'arrivée au plus tôt des instance de σ_a (définie par δ_a^-) mais nous n'avons aucune preuves que ces deux éléments sont compatibles et mènent à des scénarios réalisables.

L'algorithme 40 est utilisé pour calculer la latence au pire de la chaîne σ_a. Nous commençons par calculer un majorant sur la taille de la plus longue σ_a-*busy-window*, puis un majorant sur le nombre d'activations de σ_a dans cette σ_a-*busy-window*. Pour chaque activation, nous calculons une latence au pire cas en utilisant les *busy-times* de σ_a et son modèle d'activation δ_a^-. Nous gardons finalement, parmi toutes les latences calculées, la plus grande valeur.

Les résultats ci-dessus sont valables même si $B_a(q)$ est un majorant. Les sections suivantes concernent le calcul des majorants sur les interférences. Pour quantifier ces interférences le plus précisément possible, nous introduisons la notion de segment.

Algorithme 40 Calcul d'un, majorant sur la latence de σ_a en utilisant Eq. (3.5)

```
1: function COMPUTE_LATENCY(σ_a, lower_bound)
2:     BW_a = COMPUTE_LONGEST_BUSY_WINDOW(σ_a, lower_bound)
3:     K_a = η_a^+(BW_a)
4:     L_a = 0
5:     q = 1
6:     while q ≤ K_a do
7:         B_a(q) = COMPUTE_BUSY_TIME_CHAIN(σ_a, q, lower_bound)
8:         ℓ = B_a(q) − δ_a^−(q)
9:         if L_a < ℓ then
10:            L_a = ℓ
11:        end if
12:        q = q + 1
13:    end while
14:    return L_a
15: end function
```

3.3 Définition et utilisation du concept de segment

Dans cette section, nous nous intéressons à la définition et à l'exploitation du concept de segment. Cette notion permet de calculer l'interférence, sur σ_a, des chaînes moins prioritaires que σ_a, ainsi que l'interférence partielle d'une chaîne plus prioritaire que σ_a. La notion de segment est déjà présente dans l'état de l'art, mais l'utilisation que nous en faisons est plus formelle que [2] et nous permet d'avoir un majorant plus précis que [1].

Intuitivement, un **segment** d'une chaîne σ_b de priorité inférieure à σ_a, par rapport à σ_a, est une sous-chaîne maximale de σ_b qui peut suspendre une exécution de σ_a. La tâche immédiatement avant ou après un segment de σ_b par rapport à σ_a a une priorité inférieure à π_a (autrement dit cette tâche appartient à $\ell p_b(a)$ [1]). En connaissant les sous-chaînes maximales de σ_b qui peuvent interférer avec σ_a, il est possible de calculer la durée maximale pendant laquelle σ_b peut suspendre σ_a.

Commençons par introduire la notion de segment en utilisant l'exemple figure 3.4.

Exemple 41. *Dans notre exemple figure 3.4, les segments de σ_b qui peuvent interférer avec σ_a sont les suivants :*

1. *un segment de tête, [τ_b^1], au début de σ_b.*
2. *un segment de queue, [τ_b^5], à la fin de σ_b.*
3. *un segment interne, [τ_b^3], au milieu de σ_b.*
4. *un segment circulaire, [τ_b^3, τ_b^1], qui est la concaténation du segment de queue d'un job et du segment de tête du job suivant.*

Les tâches τ_b^2 et τ_b^4 ne font partie d'aucun segment car leur priorité est inférieure à celle de σ_a, elles ne peuvent donc pas interférer avec σ_a.

1. Pour rappel une tâche appartenant à $\ell p_b(a)$ est une tâche de σ_b qui à une priorité inférieure à celle de σ_a.

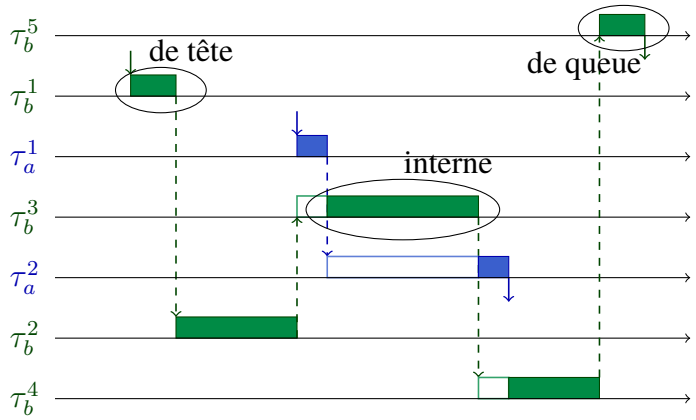

FIGURE 3.4 – Segments de σ_b sur σ_a

Il est à noter que les segments de tête, de queue et circulaire d'une chaîne σ_b sur σ_a, s'ils existent, sont uniques. Il peut cependant y avoir plusieurs segments internes de σ_b qui peuvent suspendre σ_a.

Pour calculer les interférences, il est nécessaire de définir la notion de segment interne mais, comme nous le verrons plus tard, tous les segments internes ne sont pas utiles au calcul des interférences.

Définition 42 (Segment interne $s_{b\to a}$)**.** *Un **segment interne** de σ_b par rapport à σ_a est une sous-chaîne $(\tau_b^i, \ldots, \tau_b^j)$ de σ_b tel que $1 < i \leq j < n_b$, $\tau_b^{i-1}, \tau_b^{j+1} \in \ell p_b(a)$ et $\forall k \in i..j$, $\tau_b^k \notin \ell p_b(a)$.*

Les définitions suivantes concernent les segments de tête et de queue.

Définition 43 (Segment de tête $s_{b\to a}^{head}$)**.** *Si $\tau_b^1 \notin \ell p_b(a)$, $(\tau_b^1, \ldots, \tau_b^{i-1})$ est le **segment d'en-tête** de σ_b par rapport à σ_a, où τ_b^i est la première tâche de σ_b qui appartient à $\ell p_b(a)$.*

Pour rappel, ces segments sont définis pour calculer les interférences des chaînes moins prioritaires. L'algorithme 44 sert à calculer le WCET du segment de tête de σ_b sur σ_a, noté $C_{s_{b\to a}^{head}}$ (plusieurs algorithmes sont possibles).

Algorithme 44 Calcul du WCET du segment de tête de σ_b sur σ_a

```
1: function COMPUTE_WCET_HEAD(σ_a, σ_b)
2:     C_{s^head_{b→a}} = 0
3:     j = 1
4:     while π_b^j > π_a do
5:         C_{s^head_{b→a}} = C_{s^head_{b→a}} + C_b^j
6:         j = j + 1
7:     end while
8:     return C_{s^head_{b→a}}
9: end function
```

L'algorithme se déroule ainsi :

2 → 3 - Initialisation : le WCET du segment d'en-tête est initialisé à 0.

$4 \rightarrow 7$ - Itération : les tâches de σ_b sont parcourues par ordre d'indice croissant en commençant par τ_b^1 (la première tâche). Si la priorité de la tâche est supérieure à π_a, alors elle appartient au segment et son WCET est ajouté à $C_{s_{b\rightarrow a}^{head}}$. Dans le cas contraire, la tâche n'est pas comprise dans le segment de tête et le calcul s'arrête.

Traitons maintenant le cas du segment de queue qui est à la fin de la chaîne.

Définition 45 (Segment de queue $s_{b\rightarrow a}^{tail}$)**.** *Si $\tau_b^{n_b} \notin \ell p_b(a)$, $(\tau_b^{j+1}, \ldots, \tau_b^{n_b})$ est le **segment de queue** de σ_b par rapport à σ_a, où τ_b^j est la dernière tâche de σ_b qui appartient à $\ell p_b(a)$.*

Pour calculer le WCET du segment de queue, il faut procéder de la même manière que pour le segment d'en-tête, mais en commençant par la fin de la chaîne.

Algorithme 46 Calcul du WCET du segment de queue de σ_b sur σ_a

```
1: function COMPUTE_WCET_TAIL(σ_a, σ_b)
2:     Cs^tail_{b→a} = 0
3:     j = n_b
4:     while π^j_b > π_a do
5:         Cs^tail_{b→a} = Cs^tail_{b→a} + C^j_b
6:         j = j − 1
7:     end while
8:     return Cs^tail_{b→a}
9: end function
```

L'algorithme se déroule ainsi :

$2 \rightarrow 3$ - Initialisation : le WCET du segment de queue est initialisé à 0.

$4 \rightarrow 7$ - Itération : les tâches de σ_b sont parcourues par ordre d'indice décroissant en commençant par $\tau_b^{n_b}$ (la dernière tâche). Le WCET de chaque tâche de σ_b qui est plus prioritaire que σ_a est ajouté à $C_{s_{b\rightarrow a}^{head}}$. Dès qu'une tâche a une priorité inférieure à celle de σ_a le calcul s'arrête.

Avec les définitions des segments de tête et de queue et les calculs de leur WCET, il est triviallement possible d'obtenir ces même informations pour le segment circulaire qui est la concaténation du segment de queue et du segment de tête du job suivant.

Définition 47 (Segment circulaire $s_{b\rightarrow a}^{circ}$)**.** *Si $\tau_b^1 \notin \ell p_b(a)$ et $\tau_b^{n_b} \notin \ell p_b(a)$ alors le **segment circulaire** de σ_b par rapport à σ_a est la concaténation des segments de queue et de tête de σ_b par rapport à σ_a.*

Le WCET du segment circulaire de σ_b sur σ_a est simplement la somme des WCETs des segments de tête et de queue de σ_b sur σ_a. Si σ_b n'a pas de segment de tête ou de queue alors le segment circulaire n'existe pas.

Pour finir, il faut identifier le segment de σ_b qui maximise les interférences sur σ_a.

Définition 48 (Segment critique $s_{b\rightarrow a}^{crit}$)**.** *Le **segment critique** de σ_b par rapport à σ_a est un des segments de σ_b par rapport à σ_a qui a le plus long temps d'exécution.*

Le segment critique peut être un segment de tête, interne, de queue ou circulaire. Comme présenté auparavant, les WCETs des segments internes ne sont pas nécessairement utiles au calcul des interférences. Ils permettent cependant de calculer le WCET du segment critique. L'algorithme 49 permet de calculer le WCET du segment critique de σ_b sur σ_a.

L'algorithme se déroule ainsi :

Algorithme 49 Calcul du WCET du segment critique de σ_b sur σ_a

1: **function** COMPUTE_WCET_CRITICAL(σ_a, σ_b)
2: $C_{s^{crit}_{b\to a}} = 0$
3: $C_{current} = 0$
4: **for** $j \in [1, n_b]$ **do**
5: **if** $\pi^j_b > \pi_a$ **then**
6: $C_{current} = C_{current} + C^j_b$
7: **if** $j = n_b$ **then**
8: $C_{current} = C_{current} +$ COMPUTE_WCET_HEAD(σ_a, σ_b)
9: **end if**
10: **if** $C_{current} > C_{s^{crit}_{b\to a}}$ **then**
11: $C_{s^{crit}_{b\to a}} = C_{current}$
12: **end if**
13: **else**
14: $C_{current} = 0$
15: **end if**
16: **end for**
17: **return** $C_{s^{crit}_{b\to a}}$
18: **end function**

$2 \to 3$ - Initialisation : Le WCET $C_{s^{crit}_{b\to a}}$ du segment est initialisé tout comme la variable $C_{current}$ permettant de parcourir les tâches.

$4 \to 16$ - Itération : L'ensemble de la chaîne σ_b est parcourue. Pour chaque tâche τ^j_b, si π^j_b est supérieure à π_a alors le WCET du segment est incrémenté de C^j_b. Si π^j_b est inférieure à π_a alors τ^j_b n'appartient pas à un segment et le WCET du segment est réinitialisé à 0. Parmi les WCETs des segments de la chaîne, seul le plus long est gardé.

En utilisant les segments, il est possible de préciser l'interférence des chaînes moins prioritaires sur σ_a.

Tout d'abord, si σ_b a plusieurs segments internes par rapport à σ_a, ils ne peuvent pas tous interférer avec σ_a car deux segments consécutifs sont séparés par une tâche qui appartient à $\ell p_b(a)$.

Exemple 50. *Sur la figure 3.5 par exemple, les tâches τ^3_b et τ^5_b ne pourront pas s'exécuter dans une même σ_a-busy-window car toutes les tâches s'exécutant dans la σ_a-busy-window sont de priorité supérieure ou égale à π_a et les tâches τ^3_b et τ^5_b sont activées à la terminaison de τ^2_b et τ^4_b qui sont de priorité inférieure à π_a.*

De manière générale, considérons deux chaînes σ_a et σ_b telles que $\pi_b < \pi_a$. Deux segments de σ_b qui ne sont pas des segments de tête ne peuvent pas s'exécuter (et donc suspendre σ_a) dans la même σ_a-*busy-window*. La raison est que, excepté le segment de tête, chacun des segments de σ_b sur σ_a est précédé d'une tâche de priorité inférieure à π_a, or toutes les tâches qui s'exécutent dans une σ_a-*busy-window* ont une priorité supérieure à σ_a. Dans le cas du modèle d'exécution asynchrone, si un autre job de σ_b est activé seul le segment de tête peut s'exécuter dans la même σ_a-*busy-window*.

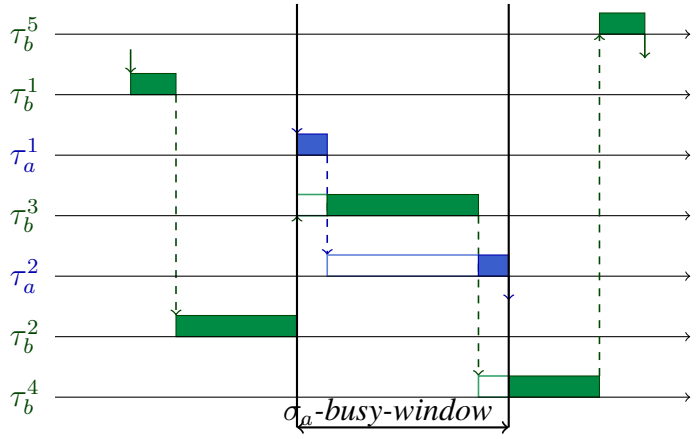

FIGURE 3.5 – Illustration de l'interférence d'un segment de σ_b sur σ_a

La propriété suivante n'est pas utilisé dans [1], ce qui mène à des résultats pessimistes. Au contraire, ce résultat est utilisé dans [2], mais sans preuve formelle, et uniquement dans le cas synchrone.

Propriété 51. *Soient σ_a, σ_b, et σ_c trois chaînes telles que $\pi_a > \pi_b > \pi_c$. Deux segments $s_{b\to a}$ et $s_{c\to a}$ qui ne sont pas des segments de tête ne peuvent pas s'exécuter (et donc suspendre σ_a) dans la même σ_a-busy-window.*

Démonstration. La preuve est par contradiction. Considérons une σ_a-*busy-window* dans laquelle deux segments $s_{b\to a}$ et $s_{c\to a}$ s'exécutent. Soit τ_b^k (resp. τ_c^j) la tâche qui précède $s_{b\to a}$ (resp. $s_{c\to a}$) dans σ_b (resp. σ_c). Suivant la définition 42, $\tau_b^k \in \ell p_b(a)$ et $\tau_c^j \in \ell p_c(a)$ car elles n'appartiennent pas à des segments pouvant interférer sur σ_a.

De plus, τ_b^k et τ_c^j ont une priorité plus basse que toutes les tâches de σ_a et également une priorité plus basse que toutes les tâches des chaînes de priorité supérieure à σ_a. Il en résulte que ni τ_b^k ni τ_c^j ne peuvent s'exécuter dans la σ_a-*busy-window* considérée (qui se ferme avant qu'ils puissent s'exécuter).

Pour que $s_{b\to a}$ et $s_{c\to a}$ puissent s'exécuter dans une même σ_a-*busy-window*, τ_b^k et τ_c^j doivent s'exécuter et terminer avant l'ouverture de cette σ_a-*busy-window*.

Rappelons que les tâches dans $s_{b\to a}$ et $s_{c\to a}$ ont une priorité plus haute que π_a, alors que τ_b^k et τ_c^j ont une priorité plus basse que π_a. Considérons que τ_b^k termine son exécution avant τ_c^j and donc active $s_{b\to a}$. Il résulte que $s_{b\to a}$ va s'exécuter avant que τ_c^j ne soit terminée. C'est une contradiction car τ_c^j doit s'exécuter avant la σ_a-*busy-window* considérée et $s_{b\to a}$ dans la même busy-window. Le même argument s'applique au cas où τ_c^k termine son exécution avant τ_b^j. □

Exemple 52. *Sur la figure 3.6 par exemple, selon la propriété 51, les tâches τ_b^3 et τ_c^3 ne peuvent pas s'exécuter dans une même σ_a-busy-window.*

Il est à noter que les résultats précédents n'utilisent aucune hypothèse concernant la politique d'échéance (contrainte ou arbitraire) ou le modèle d'exécution (synchrone or asynchrone). Les résultats suivants sont cependant plus simples à énoncer et à prouver si nous considérons des échéances contraintes. Pour cette raison, nous traitons d'abord le cas où toutes les échéances des chaînes du système sont contraintes avant de présenter le cas où les échéances sont arbitraires (section 3.5).

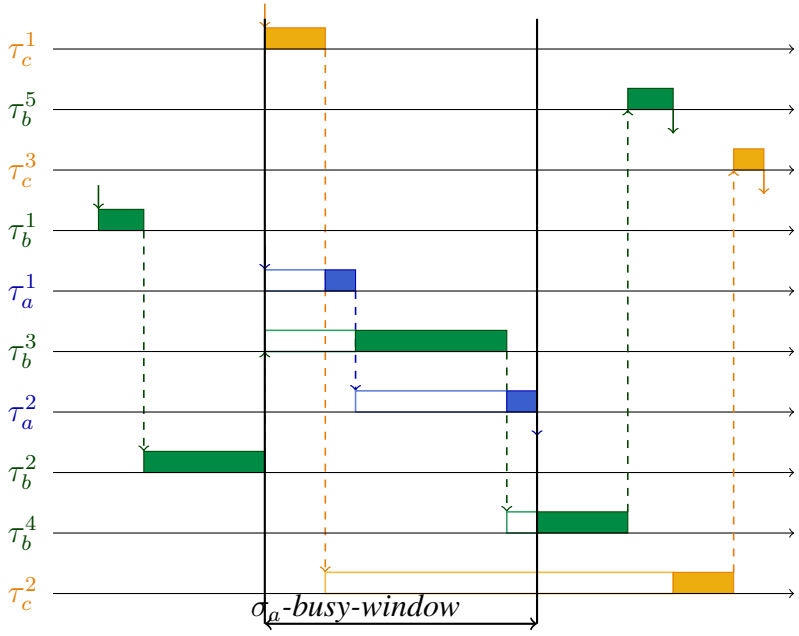

FIGURE 3.6 – Interférence des segments de σ_b et de σ_c sur σ_a

3.4 Cas particulier : échéances contraintes

Dans cette section, nous considérons que les échéances sont contraintes. Dans ce cas, si les chaînes sont ordonnançables, alors pour chaque chaîne σ_a, on a $L_a \leq \delta_a^-(2)$. Ainsi la distinction entre les modèles d'exécution synchrone et asynchrone n'est pas utile, car elle n'apparaît que dans le cas où deux jobs d'une même chaîne peuvent interférer l'un avec l'autre. Or, si les échéances sont contraintes et qu'un job d'une chaîne en suspend un autre alors la chaîne (et par conséquent le système) n'est pas ordonnançable. Il est à noter que si les échéances sont contraintes, les jobs d'une chaîne ne subissent pas d'auto-interférence (c'est à dire d'interférence des jobs ultérieurs de la même chaîne).

Si $L_a > \delta_a^-(2)$ alors la chaîne n'est pas ordonnançable et les interférences calculées dans cette section 3.4 peuvent ne pas permettre de calculer un majorant sur la latence au pire cas. Cependant un majorant peut être calculé en utilisant les équations de la section 3.5 développées dans le cas où les échéances sont arbitraires.

Nous présentons, dans cette section, les formules pour calculer des majorants sur les interférences dans le cas où les échéances sont contraintes, et nous étendons ces analyses aux échéances arbitraires dans la section suivante.

3.4.1 Interférence des chaînes de priorité inférieure

Tout d'abord, voyons comment calculer les interférences en provenance des chaînes moins prioritaires.

Les propriété 51 permet de préciser l'interférence sur σ_a des chaînes de priorités inférieures à π_a dans le cas général, elle est donc toujours valide. Cependant, si les échéances sont contraintes, il est possible d'établir une nouvelle propriété pour améliorer la précision du calcul des interférences. Cette propriété sur les segments exprime le fait qu'un unique segment par chaîne de plus basse priorité que π_a peut interférer avec une

σ_a-*busy-window* donnée.

Propriété 53. *Supposons que les échéances soient contraintes. Soient σ_a et σ_b des chaînes de tâches telles que $\pi_b < \pi_a$. Dans une σ_a-busy-window, σ_b exécute au plus un segment, potentiellement circulaire, sauf si le système n'est pas ordonnançable.*

Démonstration. Une tâche τ_b^k entre deux segments de σ_b est telle que $\pi_b^k < \pi_a$, donc après avoir exécuté un segment de σ_b, la tâche qui suit le segment est suspendue jusqu'à la fermeture de la σ_a-*busy-window*. De plus, si les échéances sont contraintes et si le système est ordonnançable alors, deux jobs de σ_b ne peuvent pas être activés dans une même σ_a-*busy-window*. □

La propriété 53 n'est valide que pour les échéances contraintes car, si les échéances sont arbitraires et les chaînes asynchrones, le segment de tête des jobs suivants de σ_b peuvent interférer avec σ_a. Basé sur la propriété 53, le théorème 54 permet de majorer l'interférence des chaînes de $\ell p(a)$ sur σ_a dans le cas des échéances contraintes.

Théorème 54. *Supposons que toutes les échéances soient contraintes et soit σ_a une chaîne. Dans n'importe quelle σ_a-busy-window, l'interférence des chaînes de priorité inférieure à π_a est majorée par :*

$$\overline{\ell p \mathcal{I}_a} = \max_{\sigma_b \in \ell p(a)} \left\{ C_{s_{b\to a}^{crit}} + \sum_{\sigma_c \in \ell p(a) \setminus \{\sigma_b\}} C_{s_{c\to a}^{head}} \right\} \tag{3.6}$$

Démonstration. Selon la propriété 53, une chaîne de priorité inférieure à σ_a peut exécuter au plus un segment dans une σ_a-*busy-window*. Selon la propriété 51, deux chaînes de priorité inférieure à σ_a ne peuvent pas toutes les deux exécuter des segments qui ne sont pas des segments de tête dans une même σ_a-*busy-window*. Il s'en suit que l'interférence des chaînes de $\ell p(a)$ est majorée par le maximum parmi toutes les combinaisons de 1 segment critique et $Card(\ell p(a)) - 1$ segments de tête, ce qui est formalisé par l'équation (3.6). La chaîne σ_b interfère à hauteur de son segment critique sur σ_a tandis que toutes les autres chaînes $\sigma_c \in \ell p(a) \setminus \{\sigma_b\}$ interfèrent à hauteur de leur segment de tête. □

Dans la définition 22, l'interférence maximum des chaînes de priorité inférieure à σ_a dépend d'un préfixe Δ d'une σ_a-*busy-window*. Il est à noter que, dans le cas où les échéances sont contraintes, le majorant sur l'interférence des chaînes de $\ell p(a)$ sur σ_a est une constante et ne dépend donc pas de la longueur d'un préfixe.

L'algorithme 55 permet de calculer un majorant sur l'interférence des chaînes de $\ell p(a)$ dans un préfixe de longueur $B_a^i(q)$ qui commence à l'ouverture d'une σ_a-*busy-window* et termine à la terminaison du $q^{\text{ème}}$ job de τ_a^i. Ce majorant est une des combinaison maximale entre 1 segment critique et $Card(\ell p(a)) - 1$ segments de tête et, pour la trouver, il faut tester successivement toutes les combinaisons possibles. L'algorithme se déroule ainsi :

2 - Initialisation : L'interférence est initialisée à 0.

$3 \to 14$ - Itération : Pour chacune des chaînes de $\ell p(a)$ on calcule le WCET du segment critique et le WCET du segment de tête des autres chaînes, ce qui nous donne une valeur possible pour le majorant. À chaque itération les interférences correspondant une combinaison sont calculées, et seule la plus grande valeur (notée $interf\_UB\_LP$) est conservée.

Algorithme 55 Calcul d'un majorant sur les interférences des chaînes de $\ell p(a)$ sur $B_a^i(q)$ en utilisant l'équation (3.6)

```
1: function COMPUTE_UB_LP_INTERF(σ_a, Δ)
2:     interf_UB_LP = 0
3:     for σ_b ∈ ℓp(a) do
4:         current_interf = COMPUTE_WCET_CRITICAL(σ_a, σ_b)
5:         for σ_c ∈ ℓp(a) do
6:             if σ_c ≠ σ_b then
7:                 C_{s^head_{c→a}} = COMPUTE_WCET_HEAD(σ_a, σ_c)
8:                 current_interf = current_interf + C_{s^head_{c→a}}
9:             end if
10:        end for
11:        if current_interf > interf_UB_LP then
12:            interf_UB_LP = current_interf
13:        end if
14:    end for
15:    return interf_UB_LP
16: end function
```

Exemple 56. *L'exécution figure 3.7 correspond à un scénario qui maximise l'interférence des chaînes de plus basse priorité sur σ_a. σ_b n'a pas de segment de tête tandis que σ_c n'a pas de segment interne. Les interférences maximales de σ_b et σ_c sur σ_a sont obtenues lorsque σ_b interfère à hauteur de son segment interne et σ_c à hauteur de son segment de tête.*

Nous avons maintenant un majorant sur l'interférence des chaînes de $\ell p(a)$ dans le cas où toutes les échéances sont contraintes.

3.4.2 Interférence des chaînes de priorité supérieure

Dans [1], la latence d'un job d'une chaîne σ_a est majorée en considérant que, si elles sont activées durant son exécution, toutes les chaînes de priorité supérieure suspendent ce job à hauteur de leur WCET. Cependant, l'interférence exacte des chaînes de priorité supérieure est plus complexe et, dans cette section, nous raffinons le majorant établi dans [1].

Commençons par comprendre comment les chaînes plus prioritaires interfèrent en utilisant l'exemple figure 3.7. Nous considérerons, dans cette section, deux chaînes σ_a et σ_d telles que $\pi_a < \pi_d$.

Exemple 57. *Considérons la figure 3.7, et intéressons nous à l'interférence de σ_d sur σ_a. La première activation de σ_d suspend l'exécution de σ_a pendant son WCET. En effet, σ_d est activée avant l'activation de τ_a^2 qui est la seule tâche de σ_a avec une priorité inférieure à σ_d, cela implique que τ_a^2 ne peut pas préempter l'exécution de σ_d. Compte tenu du modèle d'activation δ_a^-, une partie de la chaîne σ_a s'exécute avant la seconde activation de σ_d. De plus, lorsqu'elle est réactivée, la chaîne σ_d préempte σ_a mais ne la suspend pas pendant son WCET, elle interfère seulement partiellement à hauteur de C_d^1. Plus précisément,*

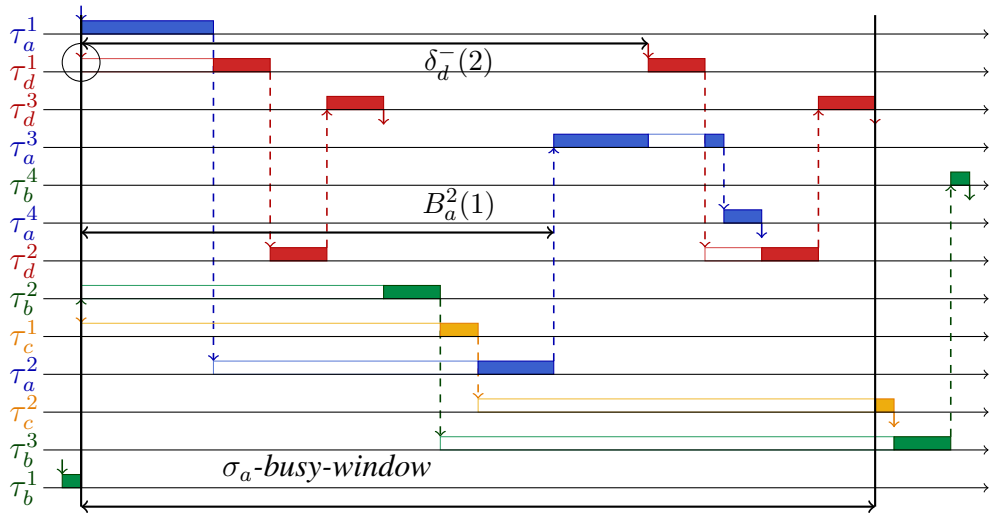

FIGURE 3.7 – Interférence des chaînes plus prioritaires

l'interférence maximum du second job de σ_d sur σ_a est le WCET du segment de tête de σ_d sur la sous-chaîne $[\tau_a^3, \tau_a^4]$ (qui correspond à τ_d^1).

Ces observations sont utiles pour proposer un majorant plus précis que dans [1] concernant le calcul de l'interférence des chaînes plus prioritaires. En effet, il est possible de raffiner ce calcul en utilisant le calcul des *busy-times* de certaines tâches de σ_a. Comme nous l'avons déjà mentionné en section 3.2.2, il n'est pas nécessaire de calculer le *busy-time* de toutes les tâches d'une chaîne.

Commençons par préciser les indices des tâches pour lesquelles on calcule le *busy-time*.

Notation 58. *Soit une chaîne $\sigma_d \in hp(a)$, $\tau_a^{\ell t_a(d)}$ est la dernière tâche de σ_a qui a une priorité inférieure à σ_d et $\tau_a^{\ell t_a}$ est la dernière tâche de la chaîne σ_a qui a une priorité inférieure à toutes les chaînes de $hp(a)$.*

Les tâches $\tau_a^{\ell t_a(d)}$ et $\tau_a^{\ell t_a}$ peuvent être identifiées via les algorithmes 59 et 60. Pour trouver $\tau_a^{\ell t_a(d)}$, les tâches de σ_a sont parcourues en commençant par la dernière ($\tau_a^{n_a}$) et, le parcours des tâches de la chaîne σ_a s'arrête dès qu'une tâche a une priorité inférieure à π_d (pour rappel, toutes les priorités sont distinctes).

Algorithme 59 Calcul de la dernière tâche de σ_a qui une priorité inférieure à π_d

```
1: function COMPUTE_LAST_TASK_WRT_CHAIN(σ_a, σ_d)
2:     i = n_a
3:     while π_a^i > π_d do
4:         i = i − 1
5:     end while
6:     return τ_a^i
7: end function
```

Pour calculer $\tau_a^{\ell t_a}$, il suffit de calculer successivement pour toutes les chaînes $\sigma_d \in hp(a)$ la valeur de $\tau_a^{\ell t_a(d)}$ et de garder la tâche $\tau_a^{\ell t_a(d)}$ qui a l'indice le plus bas.

Algorithme 60 Calcul de la dernière tâche de σ_a qui a une priorité inférieure à toutes les priorités des chaînes $\sigma_d \in S \backslash \{\sigma_a\}$

```
1: function COMPUTE_LAST_TASK_WRT_SYSTEM(σ_a)
2:     lt_a = n_a
3:     for σ_d ≠ σ_a do
4:         j = COMPUTE_LAST_TASK_WRT_CHAIN(σ_a, σ_d)
5:         if j < lt_a then
6:             lt_a = j
7:         end if
8:     end for
9:     return lt_a
10: end function
```

Par la suite, nous calculerons les *busy-times* des tâches τ_a^i pour tout $i \in [\ell t_a, n_a]$. Il est inutile de calculer les *busy-times* des tâches d'indices inférieurs à ℓt_a car tous les jobs des chaînes de $\sigma_d \in hp(a)$ activés avant la terminaison du $q^{\text{ème}}$ job de $\tau_a^{\ell t_a}$ suspendent le *busy-time* du $q^{\text{ème}}$ job des tâches τ_a^i pour tout $i \in [\ell t_a, n_a]$ pendant C_d.

Lors du calcul du *busy-time* de la chaîne σ_a, l'interférence des chaînes plus prioritaires est donc la suivante : (i) tous les jobs de σ_d qui arrivent avant la terminaison du $q^{\text{ème}}$ job de $\tau_a^{\ell t_a(d)}$ s'exécutent entièrement avant que σ_a ne puisse terminer son $q^{\text{ème}}$ job et (ii) si σ_d est activée après la terminaison du $q^{\text{ème}}$ job de $\tau_a^{\ell t_a(d)}$, alors l'interférence de ce job de σ_d sur σ_a est seulement *partielle*.

La propriété 61 permet de calculer l'interférence d'une chaîne plus prioritaire σ_d dans le cas où la chaîne est activée avant la terminaison de la tâche $\tau_a^{\ell t_a(d)}$.

Propriété 61. *Si $\ell t_a \leq i \leq \ell t_a(d)$ alors l'interférence de σ_d sur q événements de τ_a^i peut être majorée par*

$$\overline{hp\mathcal{I}_{d\to a}^{i,q}}(\Delta) = \eta_d^+(\Delta) \times C_d \tag{3.7}$$

Démonstration. Soit $[t_1, t_2]$ un intervalle de temps de longueur Δ qui commence à l'ouverture d'une σ_a-*busy-window* et termine à la terminaison du $q^{\text{ème}}$ job de τ_a^i. Selon la définition 19, il n'y a aucun job de σ_d en instance à l'instant t_1. De plus, il n'y a pas plus de $\eta_d^+(\Delta)$ jobs de σ_d qui peuvent être activés dans $[t_1, t_2]$, et ils ne peuvent pas suspendre σ_a pendant plus de C_d. D'où le résultat. □

La propriété 62 permet de prendre en compte les activations de σ_d après la terminaison du $q^{\text{ème}}$ job de $\tau_a^{\ell t_a(d)}$.

Propriété 62. *Si $i > \ell t_a(d)$ et les échéances sont contraintes, alors l'interférence de σ_d sur q événements de τ_a^i pour $\Delta > B_a^{i-1}(q)$ est majorée par :*

$$\overline{hp\mathcal{I}_{d\to a}^{i,q}}(\Delta) = \eta_d^+(B_a^{\ell t_a(d)}(q)) \times C_d + I_{a,b}^{i,q} \tag{3.8}$$

$$\text{où} \quad I_{a,b}^{i,q} = \begin{cases} 0 & \text{if } \eta_d^+(B_a^{\ell t_a(d)}(q)) = \eta_d^+(\Delta) \\ C_{s_{d\to a[k..i]}^{head}} & \text{sinon} \end{cases}$$

$$\text{et} \quad k = \min\left(i, \min_{u \in \{\ell t_a(d)+1,\ldots,i-1\}} \{\eta_d^+(B_a^{u-1}(q)) \neq \eta_d^+(B_a^u(q))\}\right)$$

Démonstration. Soit $[t_1, t_2]$ une σ_a-*busy-window* et soit un préfixe de longueur Δ d'une σ_a-*busy-window* qui finit à la terminaison du $q^{\text{ème}}$ job de τ_a^i. Pour calculer l'interférence d'une chaîne $\sigma_d \in hp(a)$, on différencie les cas selon l'indice de la tâche de σ_a préemptée par σ_d :

— **Toutes les jobs de σ_d qui sont activés dans l'intervalle $[t_1, t_1 + B_a^{\ell t_a(d)}(q)[$ interfèrent au plus à hauteur de C_d.** La propriété 61 permet de majorer l'interférence des jobs de σ_d activés avant la terminaison de $\tau_a^{\ell t_a(d)}$ et selon cette propriété chacun de ces jobs interfère au plus à hauteur de C_d. Si $\eta_d^+(B_a^{\ell t_a(d)}(q)) = \eta_d^+(\Delta)$ alors cela signifie que tous les jobs activés dans un préfixe de longueur Δ sont en réalités activés avant la terminaison du $q^{\text{ème}}$ job de $\tau_a^{\ell t_a(d)}$. Ainsi l'interférence de chaque job de σ_d est majorée par C_d. Cependant, si $\eta_d^+(B_a^{\ell t_a(d)}(q)) < \eta_d^+(\Delta)$ alors au moins un job de σ_d est activé dans l'intervalle $]t_1 + B_a^{\ell t_a(d)}(q), t_1 + \Delta[$.

— **La première activation de σ_d après $t_1 + B_a^{\ell t_a(d)}(q)$ interfère à hauteur de $C_{s_{d\to a[k..i]}^{head}}$** : Considérons le premier job de σ_d activé après $t_1 + B_a^{\ell t_a(d)}(q)$. Par définition, k est l'indice minimum strictement supérieure à $\ell t_a(d)$ tel que $\eta_d^+(B_a^{k-1}(q)) \neq \eta_d^+(B_a^k(q))$. Ce qui signifie que le nombre d'activations diffère dans les préfixes de longueur $B_a^{k-1}(q)$ ou $B_a^k(q)$ et que le job de σ_d ne peut pas être avant $t_1 + B_a^{k-1}(q)$ mais il peut être activé avant $t_1 + B_a^k(q)$. La tâche de σ_d qui est juste après le segment $s_{d\to a[k..i]}^{head}$ a, par définition, une plus basse priorité que toutes les tâches de la sous-chaîne $\sigma_a[k..i]$ et donc cette tâche sera suspendue jusqu'à la terminaison du $q^{\text{ème}}$ job de τ_a^i, c'est à dire, jusqu'à $t_1 + \Delta$.

— **Si σ_d est ordonnançable, alors il n'y a aucun autre job de σ_d qui peut arriver avant $t_1 + \Delta$** : Comme nous venons de le montrer, le premier job de σ_d activé après $t_1 + B_a^{\ell t_a(d)}(q)$ ne peut pas terminer son exécution avant $t_1 + \Delta$. D'où le résultat si les échéances sont contraintes.

□

Les algorithmes 63 et 64 servent à calculer un majorant sur l'interférence d'une chaîne plus prioritaire σ_d sur le $q^{\text{ème}}$ *busy-time* de τ_a^i. L'algorithme 63 permet de calculer l'interférence des jobs qui suspendent σ_a à hauteur de leur WCET, alors que l'algorithme 64 sert à calculer l'interférence des jobs dont l'interférence est seulement partielle.

L'algorithme 63 se déroule ainsi :

$2 \to 3$ - Initialisation : Les tâches $\tau_{\ell t_a}$ et $\tau_{\ell t_a(d)}$ sont identifiées.

$4 \to 12$ - Calcul de l'interférence : L'interférence de σ_d sur $B_a^i(q)$ est calculé selon l'indice i :

- $4 \to 5$ - si $\ell t_a \leq i \leq \ell t_a(d)$: Chacun des $\eta_d^+(\Delta)$ jobs de σ_d activés dans un préfixe de longueur Δ suspend σ_a pendant C_d.
- $6 \to 12$ - si $\ell t_a(d) < i \leq n_a$: Chacun des $\eta_d^+(B_a^{\ell t_a(d)}(q))$ jobs de σ_d activés avant la terminaison de $\tau_a^{\ell t_a(d)}$ suspendent σ_a pendant C_d. En comparant le nombre maximums de jobs de σ_d activés dans les préfixes de longueur $B_a^{\ell t_a(d)}(q)$ et Δ qui commencent simultanément à l'ouverture de la σ_a-*busy-window*, il est possible de savoir si un job de σ_d est activé après la terminaison de $\tau_a^{\ell t_a(d)}(q)$ et avant Δ. Si un tel job existe, il pourra interférer partiellement avec $B_a^i(q)$ et cette interférence est calculée avec l'algorithme 64.

L'algorithme 64 se déroule ainsi :

Algorithme 63 Calcul de l'interférence de $\sigma_d \in hp(a)$ sur $B_a^i(q)$ en utilisant les équations (3.7) et (3.8)

1: **function** COMPUTE_UB_HP_INTERF(σ_a, σ_d, τ_a^i, q, Δ)
2: $\tau_{\ell t_a} =$ COMPUTE_LAST_TASK_WRT_SYSTEM(σ_a)
3: $\tau_{\ell t_a(d)} =$ COMPUTE_LAST_TASK_WRT_CHAIN(σ_a, σ_d)
4: **if** $\ell t_a \leq i \leq \ell t_a(d)$ **then**
5: $interf = \eta_d^+(\Delta) \times C_d$
6: **else**
7: $interf = \eta_d^+(B_a^{\ell t_a(d)}(q)) \times C_d$
8: **if** $\eta_d^+(\Delta) \neq \eta_d^+(B_a^{\ell t_a(d)}(q))$ **then**
9: $interf =$ COMPUTE_UB_SC_HP_INTERF($\sigma_a, \sigma_d, \tau_a^i, q, \Delta$)
10: **end if**
11: **end if**
12: **return** $interf$
13: **end function**

Algorithme 64 Calcul de l'interférence partielle de $\sigma_d \in hp(a)$ sur $B_a^i(q)$ en utilisant les équations (3.7) et (3.8)

1: **function** COMPUTE_UB_SC_HP_INTERF(σ_a, σ_d, τ_a^i, q, Δ)
2: $\tau_a^{parser} = \tau_a^{\ell t_a(d)}$
3: $v = 1$
4: **while** $\eta_d^+(B_a^{parser}(q)) = \eta_d^+(B_a^{\ell t_a(d)}(q))$ and $\tau_a^{parser} \neq \tau_a^i$ **do**
5: $\tau_a^{parser} = \tau_a^{\ell t_a(d)+v}$
6: $v = v + 1$
7: **end while**
8: $interf = interf +$ COMPUTE_WCET_HEAD($\sigma_{a[parser..i]}, \sigma_d$)
9: **return** $interf$
10: **end function**

$2 \rightarrow 3$ - Initialisation : La tâche τ_a^{parser} permettant de parcourir σ_a est initialisée à $\tau_a^{\ell t_a(d)}$.

$4 \rightarrow 7$ - Itération : Pour calculer l'interférence partielle de σ_d sur σ_a, il faut parcourir les tâches de σ_a en commençant par celle d'indice $\ell t_a(d)$ jusqu'à celle d'indice i. Comparer le nombre d'activations de σ_d des préfixes de longueur $B_a^{\ell t_a(d)}(q)$ et B_a^{parser} permet d'identifier la tâche de plus bas indice de σ_a qui peut être préemptée par σ_d car l'indice de τ_a^{parser} est incrémenté jusqu'à atteindre τ_a^i. L'interférence de ce job de σ_d sur σ_a est le segment de tête de σ_d sur la sous-chaîne de σ_a qu'il reste à exécuter.

Il est à noter que le calcul successifs des *busy-times* des tâches τ_a^i ($i \in [\ell t_a, n_a]$) du $q^{\text{ème}}$ job de σ_a permet de calculer précisément l'interférence partielle de σ_d sur σ_a et donc de raffiner la borne établie dans [1].

Nous savons désormais calculer un majorant sur la latence d'une chaîne si les échéances du système sont contraintes. Nous allons maintenant étendre l'analyse au cas où les chaînes sont arbitraires.

3.5 Cas général : échéances arbitraires

Dans cette section, les calculs des majorants sur les interférences sont étendus au cas où les chaînes ont des échéances arbitraires. Pour cela, il faut prendre en compte :

1. le modèle d'exécution de chaque chaîne (synchrone ou asynchrone cf définition 6) ;
2. l'auto-interférence d'une chaîne, c'est à dire l'interférence d'une chaîne sur elle-même (cf définition 31).

Les chronogrammes 3.8 et 3.9 illustrent la différence d'exécution entre chaînes synchrones et asynchrones pour un système composé de trois chaînes σ_a, σ_d et σ_c. Sur le chronogramme 3.8 toutes les chaînes sont synchrones tandis que sur le chronogramme 3.9, toutes les chaînes sont asynchrones.

Dans cette section, nous traitons dans cet ordre le calcul des interférences en provenance (i) des chaînes de priorité supérieure, (ii) des chaînes de priorité inférieure et (iii) de la chaîne elle-même.

3.5.1 Interférence des chaînes de priorité supérieure

Considérons d'abord le calcul d'un majorant sur l'interférence d'une chaîne σ_d plus prioritaire que σ_a et synchrone.

Cas 1 : σ_d est synchrone

Commençons par étudier sur l'exemple figure 3.8 comment la chaîne σ_d interfère sur σ_a.

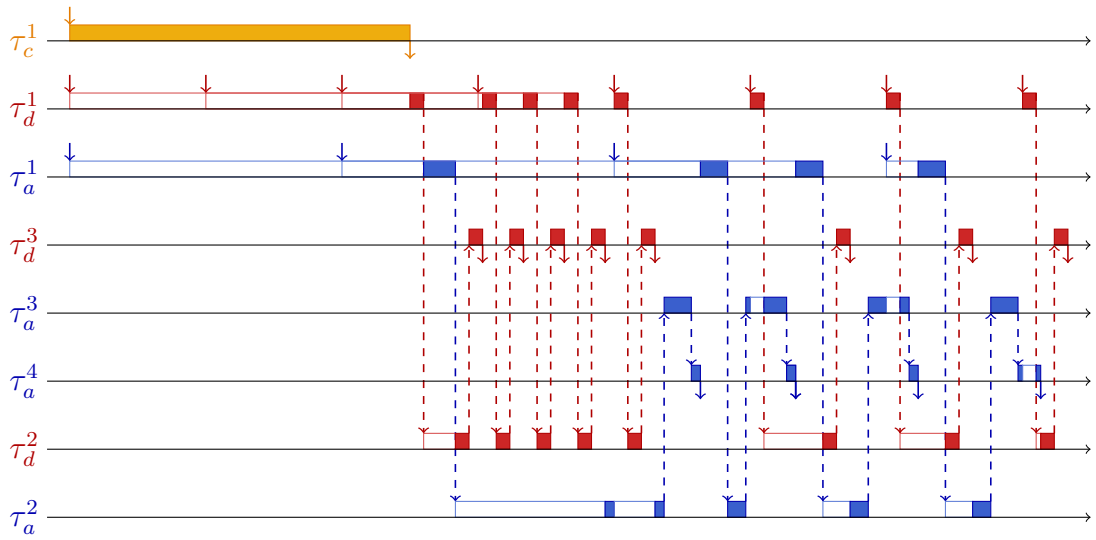

FIGURE 3.8 – Chronogramme avec σ_a et σ_d synchrones

Exemple 65. *De la même manière que si les échéances sont contraintes, tous les jobs de σ_d activés avant la terminaison de τ_a^2 (qui correspond à $\tau_a^{\ell t_a(d)}$) peuvent suspendre σ_a pendant C_d. C'est le cas des 5 premiers jobs de σ_d qui sont activés avant la terminaison du*

1^{er} job de τ_a^2 et qui interfèrent sur σ_a à hauteur de C_d. Si un job de σ_d est activé après la terminaison du $q^{ème}$ job de τ_a^2 alors l'interférence de σ_d sur σ_a est partielle, c'est le cas du $6^{ème}$ job de σ_d qui interfère partiellement avec le $2^{ème}$ job de τ_a^3. De plus, il est à remarquer qu'une fois terminée l'exécution du segment de tête de σ_d sur la sous-chaîne $[\tau_a^3, \tau_a^4]$ alors, compte tenu des priorités, l'exécution de σ_d est suspendue jusqu'à la terminaison du job de σ_a.

Compte tenu de ces observations, l'interférence d'une chaîne plus prioritaire, si elle est synchrone, est la même que si son échéance est contrainte, ce qui est établi par le théorème suivant.

Théorème 66. *Si σ_d est synchrone alors l'interférence de σ_d sur q événements de τ_a^i peut être majorée en utilisant le même majorant que pour les échéances contraintes :*

$$\overline{hp\mathcal{I}^{i,q}_{\substack{d\to a\\ sync}}} = \overline{hp\mathcal{I}^{i,q}_{d\to a}} \tag{3.9}$$

Démonstration. La preuve de la propriété 62 établit que $\overline{hp\mathcal{I}^{i,q}_{d\to a}}$ est un majorant sur les interférences de σ_d sur σ_a. L'hypothèse sur les échéances contraintes est uniquement utilisée pour justifier qu'un unique job de σ_d activé après la terminaison du $q^{\text{ème}}$ job de $\tau_a^{\ell t_a(d)}$ peut interférer avec σ_a. Par ailleurs, cette interférence est partielle, c'est à dire que σ_d ne peut pas terminer son exécution avant la terminaison du $q^{\text{ème}}$ job de τ_a^i. Si σ_d est synchrone alors, une activation ne peut pas commencer à s'exécuter avant que le job précédent soit terminé. Il s'en suit la même conclusion que pour les échéances contraintes. □

L'algorithme 64 permet de calculer l'interférence partielle d'une chaîne σ_d sur σ_a. Cette interférence partielle est intégrée par la suite dans l'algorithme 70.

Les algorithmes 64 et 63 servant à calculer l'interférence d'une chaîne σ_d plus prioritaire sur σ_a permettent également de calculer l'interférence d'une chaîne plus prioritaire synchrone.

Considérons maintenant le cas où la chaîne de priorité supérieure est asynchrone.

Cas 2 : σ_d est asynchrone

Si la chaîne σ_d est asynchrone, alors un job de σ_d activé durant l'exécution d'un job de σ_a peut suspendre ce job même si le job précédent de σ_d n'est pas terminé. Alors que dans le cas synchrone, une seule interférence partielle de σ_d sur σ_a est possible, dans le cas asynchrone plusieurs interférences partielles sont possibles.

Exemple 67. *La figure 3.9 représente l'exécution d'un système dans lequel toutes les chaînes sont asynchrones*[2]*. Les 4 premiers jobs de τ_d^1 se terminent avant que le premier job de τ_d^2 ne préempte σ_a. Autrement dit, comme σ_d (resp. σ_a) est asynchrone, un job de σ_d (resp. σ_a) peut en suspendre d'autres.*

Alors que pour les chaînes synchrones le calcul des interférences s'arrête dès qu'un job de σ_d ne peut interférer que partiellement, si les chaînes sont asynchrones, alors chaque job

2. Pour une meilleure lisibilité, un job sur deux est de couleur plus claire.

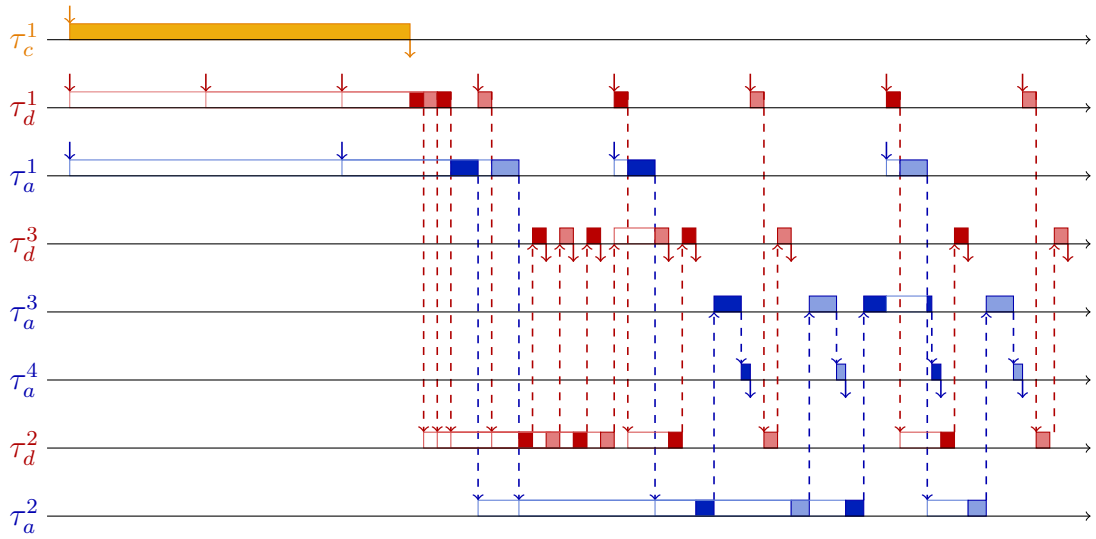

FIGURE 3.9 – Chronogramme avec σ_a et σ_d asynchrones

de σ_d qui s'active lors de la $q^{\text{ème}}$ exécution de τ_a^k, $k > \ell t_a(d)$, peut suspendre $B_a^i(q)$ pendant $C_{s^{head}_{d\to a[k..i]}}$. Le même raisonnement sur chaque job de σ_d permet de calculer l'interférence de $\sigma_d \in hp(a)$ sur σ_a. Ce calcul est formalisé via le théorème suivant.

Théorème 68. *Si $i > \ell t_a(d)$ et σ_d est asynchrone, alors l'interférence de σ_d sur q événements sur τ_a^i pour $\Delta > B_a^{i-1}(q)$ peut être majorée par :*

$$\overline{hp\mathcal{I}^{i,q}_{\substack{d\to a\\ async}}}(\Delta) = \eta_d^+(B_a^{\ell t_a(d)}(q)) \times C_d + I^{i,q}_{a,d} \tag{3.10}$$

$$\text{où} \quad I^{i,q}_{a,d} = \sum_{\substack{k\in\{\ell t_a(d)+1,\dots,i-1\}\\ n=\eta_d^+(B_a^k(q))-\eta_d^+(B_a^{k-1}(q))}} n \times C_{s^{head}_{d\to a[k..i]}} + \left(\eta_d^+(\Delta) - \eta_d^+\left(B_a^{i-1}(q)\right)\right) \times C_{s^{head}_{d\to a[i..i]}}$$

Démonstration. La différence avec la propriété 62 est que, dans ce cas, tous les jobs de σ_d qui sont activés après la terminaison de $\tau_a^{\ell t_a(d)}$ peuvent interférer pour une durée obtenue comme dans l'équation (3.8). Le terme $I^{i,q}_{a,d}$ de l'équation (3.10) permet de calculer l'interférence partielle de chaque job de σ_d activé après la terminaison de $\tau_a^{\ell t_a(d)}$. □

La propriété 68 permet de calculer l'interférence sur σ_a d'une chaîne σ_d plus prioritaire et asynchrone. Le calcul de l'interférence des jobs de σ_d activés avant la terminaison du $q^{\text{ème}}$ job de $\tau_a^{\ell t_a(d)}$ est similaire à celui dans le cas où les échéances sont contraintes. L'interférence des jobs σ_d dont l'interférence n'est que partielle est calculée via l'algorithme 77.

L'algorithme se déroule ainsi :

$2 \to 4$ - Initialisation : La tâche permettant de parcourir σ_a est initialisé à $\tau_a^{\ell t_a(d)}$ car tous les jobs de σ_d activés avant la terminaison du $q^{\text{ème}}$ job de $\tau_a^{\ell t_a(d)}$ interfèrent à hauteur de C_d. Le nombre maximum de jobs de σ_d qui sont activées avant la terminaison du $q^{\text{ème}}$ job de $\tau_a^{\ell t_a(d)}$ est $\eta_d^+(B_a^{\ell t_a(d)}(q))$.

$5 \to 20$ - Itération : L'interférence des jobs de σ_d dont l'interférence n'est que partielle est calculée ainsi :

Algorithme 69 Calcul de l'interférence de σ_d dans $hp(a)$ sur $B_a^i(q)$ en utilisant l'équation (3.8)

1: **function** COMPUTE_UB_AC_HP_INTERF(σ_a, σ_d, τ_a^i, q, Δ)
2: $\tau_a^{parser} = \tau_a^{\ell t_a(d)}$
3: $nb\_act\_chain = \eta_d^+(B_a^{\ell t_a(d)}(q))$
4: $v = 0$
5: $interf = 0$
6: **while** $\eta_d^+(B_a^{parser}(q)) < \eta_d^+(\Delta)$ and $\tau_a^i \neq \tau_a^{parser}$ **do**
7: $\tau_a^{parser} = \tau_a^{\ell t_a(d)+v}$
8: $current\_nb\_act = \eta_d^+(B_a^{parser}(q)) - nb\_act\_chain$
9: $v = v + 1$
10: **if** $current\_nb\_act \neq 0$ **then**
11: $nb\_act\_chain = nb\_act\_chain + current\_nb\_act$
12: $s_{d\to a[parser..i]}^{head} = \text{COMPUTE\_WCET\_HEAD}(\sigma_{a[parser..i]}, \sigma_d)$
13: $interf = interf + current\_nb\_act \times s_{d\to a[parser..i]}^{head}$
14: **end if**
15: **end while**
16: **if** $\tau_a^i = \tau_a^{parser}$ **then**
17: $current\_nb\_act = \eta_d^+(\Delta) - nb\_act\_chain$
18: $s_{d\to a[parser..i]}^{head} = \text{COMPUTE\_WCET\_HEAD}(\sigma_{a[parser..i]}, \sigma_d)$
19: $interf = interf + current\_nb\_act \times s_{d\to a[parser..i]}^{head}$
20: **end if**
21: **return** $interf$
22: **end function**

$5 \to 14$ - $\ell t_a(d) < parser < i$: On sait qu'il y au plus $\eta_d^+(\Delta)$ (resp. $\eta_d^+(B_a^{parser})$) dans un préfixe de longueur Δ (resp. B_a^{parser}) d'une σ_a-*busy-window*. Tant que le nombre d'activations de σ_d considérés est inférieur à $\eta_d^+(\Delta)$ et que le parcours des tâches n'a pas atteint τ_a^i. Pour chaque tâche parcourue, on vérifie si un job de σ_d peut être activé. Si c'est le cas $current\_nb\_act \neq 0$ alors, on actualise le nombre nb_act_chain d'activations de σ_b pris en compte pour le calcul de l'interférence et on ajoute l'interférence de ces jobs à l'interférence de σ_d sur σ_a.

$15 \to 20$ - $parser = i$: Une fois la tâche τ_a^i atteinte, si le nombre $\eta_d^+(\Delta)$ de jobs de σ_d à activer n'est pas atteint alors les jobs restant peuvent être activés entre la terminaison de τ_a^{i-1} et celle de τ_a^i.

Maintenant que nous pouvons calculer les interférences partielles des chaînes de priorité supérieure, voyons comment intégrer ces calculs à l'algorithme global.

Calcul de l'interférence d'une chaîne de priorité supérieure

L'algorithme 70 est l'approche globale qui permet de calculer l'interférence d'une chaîne plus prioritaire σ_d sur σ_a.

L'algorithme se déroule ainsi :

$2 \to 3$ - Initialisation : Les tâches $\tau_{\ell t_a}$ et $\tau_{\ell t_a(d)}$ sont identifiées.

Algorithme 70 Calcul de l'interférence de $\sigma_d \in hp(a)$ sur $B_a^i(q)$ en utilisant les équations (3.7) et (3.8)

```
1: function COMPUTE_UB_HP_INTERF(σ_a, σ_d, τ_a^i, q, Δ)
2:     τ_{ℓt_a} = COMPUTE_LAST_TASK_WRT_SYSTEM(σ_a)
3:     τ_{ℓt_a(d)} = COMPUTE_LAST_TASK_WRT_CHAIN(σ_a, σ_d)
4:     if ℓt_a ≤ i ≤ ℓt_a(d) then
5:         interf = η_d^+(Δ) × C_d
6:     else
7:         interf = η_d^+(B_a^{ℓt_a(d)}()) × C_d
8:         if η_d^+(Δ) ≠ η_d^+(B_a^{ℓt_a(d)}()) then
9:             if σ_d is synchronous then
10:                COMPUTE_UB_SC_HP_INTERF(σ_a, σ_d, τ_a^i, q, Δ)
11:            else
12:                COMPUTE_UB_AS_HP_INTERF(σ_a, σ_d, τ_a^i, q, Δ)
13:            end if
14:        end if
15:    end if
16:    return interf
17: end function
```

4 → 15 - Calcul de l'interférence : Si $\ell t_a \leq i \leq \ell t_a(d)$ alors chaque job de σ_d activé avant la terminaison de τ_a^i interfère à hauteur de C_d. Sinon certains jobs de σ_d peuvent interférer partiellement. Le calcul de l'interférence partielle se fait via l'algorithme 64 si σ_d est synchrone ou 77 si σ_d est asynchrone.

3.5.2 Interférence des chaînes de priorité inférieure

Dans le cas où les échéances sont contraintes, un majorant sur l'interférence des chaînes moins prioritaires est donné par l'équation (3.6). Ce majorant peut facilement être étendu pour prendre en compte les chaînes avec des échéances arbitraires. Dans cette section, deux majorants sont proposés pour calculer l'interférence des chaînes de priorité inférieure. Le second est plus précis mais il est plus difficile à calculer.

Tout comme dans la section 3.4.1, le calcul des interférences des chaînes moins prioritaires se fait en cherchant la combinaison de différents segments qui maximise ces interférences. Compte tenu de la propriété 51 qui est valable peu importe l'échéance et le modèle d'exécution, parmi toutes les chaînes de priorité inférieure à σ_a, un unique segment qui n'est pas un segment de tête peut préempter, et donc suspendre σ_a.

Si la chaîne σ_b qui interfère avec σ_a est synchrone alors selon la définition 6, σ_b ne peut exécuter qu'un unique segment (potentiellement circulaire) dans une σ_a-*busy-window*. Cependant si la chaîne σ_b est asynchrone, alors elle peut suspendre σ_a pendant l'exécution d'un segment, et chacun des jobs suivants de σ_b peut également suspendre σ_a pendant l'exécution d'un segment de tête.

Le calcul des interférences des chaînes moins prioritaires dans le cas général (i) échéances arbitraires et (ii) modèle d'exécution synchrone ou asynchrone est donné par le théorème suivant.

Théorème 71. *L'interférence des chaînes moins prioritaires sur une chaîne σ_a dans tout préfixe de longueur Δ d'une σ_a-busy-window est majorée par :*

$$\overline{\ell p \mathcal{I}_a}(\Delta) = \max_{\sigma_b \in \ell p(a)} \left\{ C_{s^*_{b\to a}} + \sum_{\sigma_c \in \ell p(a) \cap \mathcal{SC} \wedge c \neq b} C_{s^{head}_{c\to a}} + \sum_{\sigma_c \in \ell p(a) \cap \mathcal{AC}} \eta^+_c(\Delta) \times C_{s^{head}_{c\to a}} \right\} \quad (3.11)$$

où

$$s^*_{b\to a} = \begin{cases} s^{crit}_{b\to a} & \textit{si le segment critique n'est pas circulaire} \\ s^{crit}_{b\to a} & \textit{si le segment critique est circulaire et } \sigma_b \in \mathcal{SC} \\ s^{tail}_{b\to a} & \textit{sinon} \end{cases}$$

Démonstration. Nous avons ici la même formule que l'équation (3.6), sauf que les chaînes asynchrones peuvent exécuter plusieurs segments de tête. Il est à noter que si le segment critique est circulaire pour une chaîne asynchrone σ_b, alors le segment de tête est inclus dans le dernier terme de la somme. □

L'algorithme 72 permet de calculer un majorant sur les interférences des chaînes dans le cas général en utilisant l'équation 3.11. L'algorithme se déroule ainsi :

2 - Initialisation : L'interférence des chaînes de priorité inférieure est initialisé à 0.

$3 \to 22$ - Itération : Il faut tester toutes les combinaisons possible d'interférence des chaînes de priorités inférieure à σ_a et garder la combinaison qui maximise cette interférence. Soit σ_b la chaîne qui interfère à hauteur de son segment critique.

- $4 \to 8$ - Interférence de σ_b : Si σ_b est asynchrone et que son segment critique est circulaire alors on considère l'interférence du segment de queue sinon on considère l'interférence du segment critique au complet.
- $9 \to 18$ - Interférence des autres chaînes σ_c : Une chaîne synchrone interfère à hauteur d'un unique segment ainsi chaque chaîne synchrone exceptée σ_b interfère à hauteur de $C_{s^{head}_{c\to a}}$. Si la chaîne est asynchrone alors plusieurs jobs de σ_c peuvent interférer.

Ce premier calcul fournit un majorant sur l'interférence des chaînes moins prioritaires que σ_a cependant, il est possible de réduire les approximations lors du calcul de l'interférence des chaînes asynchrones. Les approximations apparaissent lors du calcul des interférences provoquées par les activations de σ_c si la chaîne est asynchrone car, dans l'équation (3.11), l'interférence est égale à $C_{s^{head}_{c\to a}}$. Or, pour chaque activation, selon la tâche de σ_a qui est préemptée par l'activation σ_c l'interférence de σ_c sur σ_a peut être inférieure à $C_{s^{head}_{c\to a}}$.

Le calcul précis de cette interférence se fait de la même manière que celle des chaînes plus prioritaires et pour chaque activation de σ_c, il faut calculer le segment de tête de σ_c sur σ_a en fonction de la tâche de σ_a qui est préemptée. Ce calcul est possible en utilisant le calcul des *busy-times* des tâches de σ_a et les activations au plus tôt de σ_c. Cette approche plus précise est formalisée via le théorème suivant.

Théorème 73. *L'interférence des chaînes moins prioritaire sur q événements de la tâche τ^i_a pour $\Delta > B^{i-1}_a(q)$ est majorée par :*

$$\overline{\ell p \mathcal{I}^{i,q}_a}(\Delta) = \max_{\sigma_b \in \ell p(a)} \left\{ C_{s^*_{b\to a}} + \sum_{\sigma_c \in \ell p(a) \cap \mathcal{SC} \wedge c \neq b} C_{s^{head}_{c\to a}} + \sum_{\sigma_c \in \ell p(a) \cap \mathcal{AC}} hp\mathcal{I}^{i,q}_{\substack{s^{head}_{c\to a} \to a \\ async}}(\Delta) \right\} \quad (3.12)$$

Algorithme 72 Calcul d'un majorant sur l'interférence des chaînes de $\ell p(a)$ sur $B_a^i(q)$ en utilisant les équations (3.11) et (3.12)

1: **function** COMPUTE_UB_LP_INTERF(σ_a, Δ)
2: $interf\_UB\_LP = 0$
3: **for** $\sigma_b \in \ell p(a)$ **do**
4: **if** σ_b is synchronous or $s_{b\to a}^{crit}$ is not circular **then**
5: $current\_interf =$ COMPUTE_WCET_CRITICAL(σ_a, σ_b)
6: **else** ▷ σ_b is not synchronous and $s_{b\to a}^{crit}$ is circular
7: $current\_interf =$ COMPUTE_WCET_TAIL(σ_a, σ_b)
8: **end if**
9: **for** $\sigma_c \in \ell p(a)$ **do**
10: **if** σ_c is synchronous and $\sigma_c \neq \sigma_b$ **then**
11: $C_{s_{c\to a}^{head}} =$ COMPUTE_WCET_HEAD(σ_a, σ_c)
12: $current\_interf = current\_interf + C_{s_{c\to a}^{head}}$
13: **end if**
14: **if** σ_c is asynchronous **then**
15: $C_{s_{c\to a}^{head}} =$ COMPUTE_WCET_HEAD(σ_a, σ_c)
16: $current\_interf = current\_interf + \eta_c^+(\Delta) \times C_{s_{c\to a}^{head}}$
17: **end if**
18: **end for**
19: **if** $current\_interf > interf\_UB\_LP$ **then**
20: $interf\_UB\_LP = current\_interf$
21: **end if**
22: **end for**
23: **return** $interf\_UB\_LP$
24: **end function**

Démonstration. Comme dans l'équation (3.11), une unique chaîne dans $hp(a)$ peut interférer avec un segment interne et toutes les autres chaînes interfèrent avec leur segment de tête. Pour les chaînes asynchrones, plusieurs segments de tête peuvent interférer. Le segment de tête $s_{c\to a}^{head}$ d'une chaîne asynchrone de priorité inférieure σ_c interfère avec σ_a exactement comme une chaîne de priorité supérieure. □

L'algorithme 72 permet de calculer un majorant sur les interférences des chaînes dans le cas général cependant, il est possible d'en avoir un plus précis via l'équation (3.12).

Pour calculer un majorant plus précis, il suffit de changer les lignes 15 et 16 de l'algorithme 72 par l'algorithme 74.

Algorithme 74 Calcul de l'interférence d'une chaîne synchrone de $\ell p(a)$ sur $B_a^i(q)$ en utilisant l'équation (3.10)

1: $hp\mathcal{I}^{i,q}_{\substack{s_{c\to a}^{head}\to a \\ async}}(\Delta) =$ COMPUTE_UB_AC_HP_INTERF($\sigma_a, s_{c\to a}^{head}, \tau_a^i, q, \Delta$)
2: $current\_interf = current\_interf + hp\mathcal{I}^{i,q}_{\substack{s_{c\to a}^{head}\to a \\ async}}(\Delta)$

Le calcul de l'interférence est réalisé en considérant l'interférence de la sous-chaîne

$s^{head}_{c\to a}$ qui est de priorité supérieure à σ_a. Si les échéances sont arbitraires et qu'une chaîne est asynchrone alors cette chaîne peut interférer avec elle même.

3.5.3 Auto-interférence

Dans le cas où la chaîne σ_a est asynchrone, l'auto-interférence $self\mathcal{I}^{i,q}_a(\Delta)$ est l'interférence des jobs suivants (c'est à dire des jobs d'indices strictement supérieurs à q) de σ_a sur le *busy-time* sur q événements de τ^i_a. Pour calculer l'auto-interférence, on utilisera le segment de tête de σ_a sur une sous-chaîne de σ_a. Ainsi pour calculer l'auto-interférence, les principes utilisés sont ceux nécessaires au calcul de l'interférence des chaînes plus prioritaires ou des chaînes moins prioritaires asynchrones. Si deux jobs d'une même chaîne peuvent se suspendre mutuellement alors deux jobs de la même tâche τ^i_a peuvent être suspendus en même temps. Dans ce cas, pour déterminer lequel de ces jobs s'éxecutera en premier, nous appliquons une politique de FIFO.

Exemple 75. *Sur la figure 3.9, le second job de σ_a interfère avec le premier. Compte tenu de la politique FIFO, le premier job (bleu foncé) de τ^1_a s'exécute avant le second (bleu clair). Cependant comme $\pi^1_a > \pi^2_a$ le second job de τ^1_a préempte le premier job de τ^2_a. Après l'exécution du premier job de τ^2_a comme $\pi^3_a > \pi^4_a > \pi^2_a$, alors le premier job de σ_a ne subit plus d'interférences en provenance du second job. Compte des priorités, le troisième job de σ_a pourrait interférer mais $\delta^-_a(3)$ est trop grand pour que ce job de σ_a puisse interférer avec le premier.*

Ces observations sont mises en équation par la propriété 76.

Théorème 76. *Si σ_a est asynchrone, alors l'auto-interférence de σ_a sur q événements de τ^i_a pour tout $i \leq \ell t_a$ et $\Delta > B^{i-1}_a(q)$, noté $self\mathcal{I}^{i,q}_a(\Delta)$, peut être majorée par :*

$$\overline{self\mathcal{I}^{i,q}_a}(\Delta) = \left(\eta^+_a(B^{\ell t_a}_a(q)) - q\right) \times C_{s^{head}_{a\to a[1..\ell t_a]}} + I^{i,q}_a \tag{3.13}$$

$$\text{où } I^{i,q}_a = \left(\eta^+_a(\Delta) - \eta^+_a(B^{i-1}_a(q))\right) \times C_{s^{head}_{a\to a[i..i]}} + \sum_{\substack{k\in\{\ell t_a+1,\ldots,i-1\}\\ n=\eta^+_a(B^k_a(q))-\eta^+_a(B^{k-1}_a(q))}} n \times C_{s^{head}_{a\to a[k..i]}}$$

Démonstration. Tous les jobs de σ_a activés après le $q^{\text{ème}}$, qui peut être activé avant que $\tau^{\ell t_a}_a$ soit terminée, peuvent interférer à hauteur du segment de tête de σ_a sur elle-même. Les activations suivantes peuvent uniquement interférer moins, en fonction du modèle d'activation de σ_a par rapport à la terminaison des tâches au pire cas. □

Il est à noter que l'algorithme servant à calculer l'auto-interférence de σ_a est similaire à celui permettant de calculer l'interférence d'une chaîne asynchrone de priorité supérieure à σ_a.

L'algorithme pour calculer l'auto-interférence de σ_a se déroule ainsi :

$2 \to 5$ - Initialisation : La tâche permettant de parcourir σ_a est initialisé à $\tau^{\ell t_a(a)}_a$ car tous les jobs de σ_a activés avant la terminaison du $q^{\text{ème}}$ job de $\tau^{\ell t_a(a)}_a$ interfèrent à hauteur de C_a. Le nombre maximum de jobs de σ_a qui sont activés avant la terminaison du $q^{\text{ème}}$ job de $\tau^{\ell t_a(a)}_a$ est $\eta^+_a(B^{\ell t_a(a)}_a(q))$.

Algorithme 77 Calcul de l'auto-interférence de σ_a sur $B_a^i(q)$ en utilisant l'équation (3.13)

1: **function** COMPUTE_UB_AC_SELF_INTERF(σ_a, τ_a^i, q, Δ)
2: $\quad \tau_a^{parser} = \tau_a^{\ell t_a(a)}$
3: $\quad nb\_act\_chain = \eta_a^+(B_a^{\ell t_a(a)}(q))$
4: $\quad v = 0$
5: $\quad interf = 0$
6: $\quad$ **while** $\eta_a^+(B_a^{parser}(q)) < \eta_a^+(\Delta)$ and $\tau_a^i \neq \tau_a^{parser}$ **do**
7: $\quad\quad \tau_a^{parser} = \tau_a^{\ell t_a(d)+v}$
8: $\quad\quad current\_nb\_act = \eta_a^+(B_a^{parser}(q)) - nb\_act\_chain$
9: $\quad\quad v = v + 1$
10: $\quad\quad$ **if** $current\_nb\_act \neq 0$ **then**
11: $\quad\quad\quad nb\_act\_chain = nb\_act\_chain + current\_nb\_act$
12: $\quad\quad\quad s_{d\rightarrow a[parser..i]}^{head} = \text{COMPUTE\_WCET\_HEAD}(\sigma_{a[parser..i]}, \sigma_a)$
13: $\quad\quad\quad interf = interf + current\_nb\_act \times s_{d\rightarrow a[parser..i]}^{head}$
14: $\quad\quad$ **end if**
15: $\quad$ **end while**
16: $\quad$ **if** $\tau_a^i = \tau_a^{parser}$ **then**
17: $\quad\quad current\_nb\_act = \eta_a^+(\Delta) - nb\_act\_chain$
18: $\quad\quad s_{d\rightarrow a[parser..i]}^{head} = \text{COMPUTE\_WCET\_HEAD}(\sigma_{a[parser..i]}, \sigma_a)$
19: $\quad\quad interf = interf + current\_nb\_act \times s_{d\rightarrow a[parser..i]}^{head}$
20: $\quad$ **end if**
21: $\quad$ **return** $interf$
22: **end function**

$6 \rightarrow 20$ - Itération : L'interférence des jobs de σ_a dont l'interférence n'est que partielle est calculée ainsi :

- $6 \rightarrow 14$ - $\ell t_a(a) < parser < i$: On sait qu'il y au plus $\eta_a^+(\Delta)$ (resp. $\eta_a^+(B_a^{parser})$) dans un préfixe de longueur Δ (resp. B_a^{parser}) d'une σ_a-*busy-window*. Tant que le nombre d'activations de σ_a considérés est inférieur à $\eta_a^+(\Delta)$ et que le parcours des tâches n'a pas atteint τ_a^i. Pour chaque tâche parcourue, on vérifie si un job de σ_a peut être activé. Si c'est le cas $current\_nb\_act \neq 0$ alors, on actualise le nombre nb_act_chain d'activations de σ_a pris en compte pour le calcul de l'interférence et on ajoute l'interférence de ces jobs à l'interférence de σ_a sur σ_a.
- $15 \rightarrow 20$ - $parser = i$: Une fois la tâche τ_a^i atteinte, si le nombre $\eta_a^+(\Delta)$ de jobs de σ_a à activer n'est pas atteint alors les jobs restant peuvent être activés entre la terminaison de τ_a^{i-1} et celle de τ_a^i.

Nous avons présenté les équations et les algorithmes nécessaires pour calculer les interférences et nous pouvons donc maintenant calculer la latence d'une chaîne dans le cas général où les échéances sont arbitraires.

3.6 Comparaison théorique avec l'état de l'art

Dans cette section, nous comparons notre approche avec les deux principales approches de l'état de l'art [1] et [2].

3.6.1 Comparaison avec [1]

Le modèle utilisé dans l'article [1] est similaire à celui que nous utilisons, ainsi chaque chaîne est composée d'un modèle d'activation, un modèle d'exécution (synchrone ou asynchrone), une échéance et chaque tâche a un WCET et une priorité.

Pour calculer la latence d'une chaîne de tâche σ_a, la méthodologie utilisée dans l'article [1] consiste à calculer un majorant sur la *q-event-busy-window* de σ_a. La *q-event-busy-window* de σ_a correspond à la durée maximum nécessaire pour exécuter entièrement q jobs de σ_a. Il est à noter que cette notion de *q-event-busy-window* de σ_a correspond à notre notion de *busy-time* sur q événements de σ_a. Pour calculer la *q-event-busy-window* de σ_a, les auteurs calculent des majorants sur les interférences qui peuvent suspendre l'exécution de q jobs consécutifs de σ_a. Il est important de remarquer que pour calculer la latence au pire cas d'une chaîne, on utilise deux notions de σ_a*-busy-window* et de *busy-time* ce qui n'est pas le cas dans l'article [1].

Dans l'article [1], pour calculer des majorants sur les interférences que peut subir σ_a, deux cas sont différenciés selon que σ_a soit synchrone ou asynchrone puis les tâches du système sont réparties dans plusieurs ensembles. Les interférences sont soit *intra-chain* et correspondent à notre auto-interférence soit *inter-chain* et correspondent à l'interférence des autres chaînes sur σ_a.

La chaîne à analyser est synchrone Si σ_a est une **chaîne synchrone** alors, elle ne subit pas d'interférence *intra-chain* et les ensembles utilisés pour répartir les tâches du système sont les suivants :

— Le **premier ensemble** contient toutes les tâches du système qui ont une priorité supérieure ou égale à $\min_i\{\pi_a^i\}$. Ces tâches peuvent interférer avec σ_a.

— Le **second ensemble** contient, pour chaque chaîne du système, toutes les tâches qui ont une priorité supérieure ou égale à $\min_i\{\pi_a^i\}$ et au moins une tâche de la chaîne a une priorité inférieure à $\min_i\{\pi_a^i\}$. En relation avec notre approche, cet ensemble inclut toutes les tâches de priorité supérieure à π_a mais qui appartiennent à une chaîne de priorité inférieure à π_a.

— Le **troisième ensemble** contient toutes les tâches qui sont dans le premier ensemble et qui ne sont pas dans le deuxième. En relation avec notre approche, c'est l'ensemble des tâches des chaînes de priorités supérieures à σ_a.

En utilisant les tâches du second ensemble sont définis :

— les segments d'une chaîne σ_b sur σ_a ;

— la notion de segment circulaire ;

— le segment critique de σ_b sur σ_a.

Il est à noter que dans l'article [1] les segments internes, de tête ou de queue ne sont pas différenciés.

Une fois les segments définis, le calcul de la plus longue durée temporelle, noté $B_a(q)$, durant laquelle s'exécutent q jobs de σ_a est obtenue en additionnant :

— Le WCET de chacun des q jobs de σ_a à prendre en compte c'est à dire $q \times C_a$.

— Le WCET de chaque job de chaque chaîne plus prioritaire que σ_a, activée pendant l'exécution des q jobs de σ_a. L'interférence d'une chaîne σ_d plus prioritaire que σ_a est de $\eta_d^+(B_a(q)) \times C_b$.

— Le WCET du segment critique de chaque chaîne de priorité inférieure à σ_a (que le segment critique soit circulaire ou non).

Ces segments servent à calculer l'interférence des chaînes moins prioritaires. Concernant le calcul de l'interférence des chaînes moins prioritaires que σ_a, la propriété 51 permet de réduire le nombre de segments qui peuvent interférer. Ainsi, un unique segment qui n'est pas un segment de tête peut interférer avec σ_a ce qui n'est pas pris en compte dans l'article [1].

Lors du calcul des interférences des chaînes plus prioritaires, nous tenons compte du modèle pour (i) calculer l'interférence partielle d'un job de $\sigma_d \in hp(a)$ et (ii) réduire le nombre de jobs de σ_d susceptibles d'interférer avec σ_a. Ainsi la propriété 66 permet de raffiner le majorant sur l'interférence des chaînes plus prioritaires par rapport à l'article [1].

La chaîne à analyser est asynchrone Si σ_a est une **chaîne asynchrone** alors, elle peut subir des interférences *intra-chain* et, les second et troisième ensembles utilisés pour répartir les tâches changent par rapport au cas précédent :

— Le **second ensemble** contient toutes les tâches du premier ensemble qui sont précédés d'une tâche de priorité inférieure à $\min_i\{\pi_a^i\}$. Autrement dit, cet ensemble contient toutes les tâches des segments des chaînes moins prioritaires excepté les tâches appartenant à un segment de tête.

— Le **troisième ensemble** contient toutes les tâches du premier ensemble qui ne sont pas dans le second. Cet ensemble contient toutes les tâches appartenant aux chaînes de priorités supérieures ainsi que toutes les tâches appartenant aux différents segments de tête des chaînes de priorités inférieures.

Une fois les segments définis, le calcul de la plus longue durée temporelle, noté $B_a(q)$, durant laquelle s'exécutent q jobs de σ_a est obtenue en additionnant :

— le maximum entre le WCET de q jobs de σ_a (c'est à dire $q \times C_a$) et le WCET de $\eta_a^+(B_a(q))$ jobs de σ_a (c'est à dire $\eta_a^+(B_a(q)) \times C_a$) s'il y plus de q activations de σ_a entre l'ouverture de la σ_a-*busy-window* et la terminaison du $q^{\text{ème}}$ job de σ_a. Cela permet de prendre en compte les interférences *intra-chain* que nous appelons auto-interférences. Tous les jobs de σ_a activés après le $q^{\text{ème}}$ interfèrent à hauteur de C_a, ce qui est sur-approximé.

— Le WCET de chaque job de chaque chaîne σ_d plus prioritaire que σ_a qui est activé durant $B_a(q)$, cette interférence est donc de $\eta_d^+(B_a(q)) \times C_d$. De plus, chaque chaîne de priorité inférieure à σ_a interfère à hauteur de son segment de tête sur σ_a.

— Le WCET du segment critique de chaque chaîne de priorité inférieure à σ_a que ce segment critique soit circulaire ou non.

Dans le cas de chaînes asynchrones nos améliorations sont multiples. Tout d'abord dans le cadre du calcul d'un majorant sur l'auto-interférence, l'interférence d'un job de σ_a sur un job précédent dépend de la tâche qui est préemptée. Ainsi l'interférence d'un job suivant de σ_a n'est pas de C_a et la propriété 76 permet de raffiner le majorant sur l'auto-interférence.

Lors du calcul des interférences des chaînes plus prioritaires, chaque job peut interférer mais l'interférence peut être partielle. Nous tenons compte de l'interférence partielle via la propriété 68 ce qui n'est pas le cas de l'article [1].

L'utilisation de la propriété 51 permet, comme pour le cas des chaînes synchrones, de réduire le nombre de segments qui peuvent interférer. Et l'interférence des segments de tête des chaînes moins prioritaires change selon le job et la tâche de σ_a qui est préemptée. Notre calcul, via la propriété 73, de l'interférence d'une chaîne moins prioritaire asynchrone est donc plus précis.

3.6.2 Comparaison avec [2]

Le modèle de système utilisé dans l'article [2] est composé de tâches indépendantes, chacune composée d'une séquence de sous-tâches qui sont définies par un WCET et une priorité. Ainsi, même si la terminologie est différente ce modèle correspond au notre, composé de chaînes de tâches. Par la suite, nous utiliserons les termes de chaînes de tâches pour présenter l'article [2]. Dans l'article [2], chaque chaîne a un modèle d'activation qui est périodique, le notre est donc plus expressif car il permet de prendre en compte les activations périodiques et sporadiques. La définition 6 nous permet de définir pour chaque chaîne un modèle d'exécution (synchrone ou asynchrone) alors que, selon l'*Assumption 3* de l'article [2] toutes les chaînes sont synchrones.

Dans l'article [2], afin d'analyser la latence, les auteurs transforment le modèle en utilisant la forme canonique (en anglais *canonical form*). Dans la forme canonique, les tâches dans les chaînes ont des priorités croissantes (non strictement). Afin d'obtenir ce modèle les auteurs considèrent la priorité de la dernière tâche de la chaîne ($\pi_a^{n_a}$), si $\pi_a^{n_a-1} > \pi_a^{n_a}$ alors $\pi_a^{n_a-1} = \pi_a^{n_a}$ et ils procèdent de la même manière jusqu'à la première tâche de la chaîne. Ils démontrent ensuite que la latence au pire cas d'une chaîne est identique qu'elle soit sous sa forme canonique ou sous sa forme initiale.

Après cette transformation, l'approche pour calculer un majorant sur la latence au pire cas est similaire à la notre, elle se déroule ainsi :

1. Calcul d'un majorant sur la longueur des σ_a-*busy-windows*.
2. Calcul d'un majorant sur le nombre de jobs de σ_a activés dans n'importe quelle σ_a-*busy-window*.
3. Pour chaque job, calcul d'un majorant sur la durée entre l'ouverture de la σ_a-*busy-window* et la terminaison du job.
4. Calcul d'un majorant sur la latence au pire cas de σ_a

Vient ensuite le calcul des interférences qui peuvent suspendre σ_a. Pour cela, les auteurs distinguent 5 différents types de chaînes qui peuvent interférer sur σ_a :

— *Chaîne de type 1 :* Toutes les tâches de la chaîne ont une priorité supérieure à $\min_i\{\pi_a^i\}$. Ces chaînes correspondent, dans notre cas, aux chaînes de priorité supérieure à σ_a.

— *Chaîne de type 2 :* Chaque tâche ou segment de priorité supérieure à $\min_i\{\pi_a^i\}$ est suivi d'une ou plusieurs tâches de priorité inférieure à $\min_i\{\pi_a^i\}$. Ces chaînes correspondent, dans notre cas, aux chaînes ayant un segment de tête et pouvant avoir un ou plusieurs segments internes mais aucun segment de queue.

— *Chaîne de type 3 :* Ces chaînes sont similaires à celles de type 2 mais la ou les tâches en fin de chaîne ont une priorité supérieure à $\min_i\{\pi_a^i\}$. Ces chaînes possèdent donc un segment de tête, un segment de queue et peuvent également avoir un ou plusieurs segments internes.

— *Chaîne de type 4 :* Chaque segment qui peut interférer avec σ_a est précédé d'une tâche de priorité inférieure à $\min_i\{\pi_a^i\}$. Cette chaîne est composée de un ou plusieurs segments internes et peut également avoir un segment de queue.

— *Chaîne de type 5 :* Toutes les tâches de la chaîne ont une priorité inférieure à $\min_i\{\pi_a^i\}$.

Dans le cadre de notre analyse, nous séparons simplement les chaînes en fonction de leur priorité.

Dans l'article [2], pour calculer l'interférence des chaînes moins prioritaires, appelé *blocking*, plusieurs segments sont définis :

— Pour les chaînes de type 4, B' est le plus long segment pouvant interférer avec σ_a.

— Pour les chaînes de types 2 et 3, W est le segment de tête, V est le plus segment de queue et U est le plus long segment interne.

Ensuite, pour chaque chaîne est calculé $\max\{U - W - B'; V - B'\}$ ainsi :

— Pour les chaînes de type 4, $U = V = W = 0$ et ainsi valeur retournée est celle de $-B'$ où B' est la longueur du segment critique.

— Pour les chaînes de types 2 et 3, $B' = 0$ et $\max\{U - W; V\}$. $V + W$ (resp. U) est la longueur du segment circulaire (resp. du plus long segment interne).

Si le segment qui maximise les interférences sur σ_i est de type 2 ou 3 alors les auteurs précisent qu'une nouvelle valeur du *blocking* doit être recalculée. Dans ce cas, aucune méthodologie précise n'est fournie dans l'article [2] alors que la propriété 3.6 permet de calculer l'interférence des chaînes moins prioritaires en recherchant la combinaison maximum.

Les chaînes sont également séparées en fonction du nombre de jobs de la chaîne pouvant interférer :

1. MP_{ij} est l'ensemble des chaînes $\sigma_p \neq \sigma_i$ tel que $\pi_p \geq \pi_i^j$. Ce qui signifie que si σ_p est activée pendant l'exécution de τ_i^j alors σ_p interférera entièrement avec σ_i.
2. SP_{ij} est l'ensemble des chaînes du système qui peuvent interférer une unique fois avec σ_i si elles sont activées pendant l'exécution de τ_i^j.

Lors du calcul de la latence au pire cas d'une chaîne σ_i, ces deux ensembles sont définis pour chacune des tâches de σ_i ce qui, comme nous l'avons vu en utilisant la notation 58, n'est pas nécessaire.

Le calcul de l'interférence des chaînes de priorité supérieure à σ_i est aussi précis dans l'article [2] que dans le cadre de notre approche.

Le principal inconvénient de l'approche dans l'article [2] est qu'il est complexe, en utilisant une approche peu formelle, de développer les algorithmes du calcul du majorant et d'identifier les sources de pessimisme, ce qui est nécessaire pour calculer des minorants sur la latence au pire cas.

Pour finir voici un tableau récapitulatif présentant les principale différences entre les articles [1], [2] par rapport à notre approche.

	approche [1]	approche [2]
modèle synchrone	oui	oui
modèle asynchrone	oui	non
interférence $\ell p(a)$	moins précis	identique
interférence $hp(a)$	moins précis	identique
auto-interférence	moins précis	absent

3.7 Evaluation expérimentale

Dans cette section, nous évaluons la précision de notre approche par rapport à celle présentée dans l'article [1]. Pour cela, on a besoin de générer des systèmes avant de les analyser via les alorithmes développés dans ce chapitre. Nous commençons donc par décrire la méthode pour générer les cas de tests puis nous présentons les comparaisons effectuées entre notre approche ou celle de l'article [1].

3.7.1 Génération des cas de test

Commençons par décrire comment sont générés les systèmes que nous utilisons pour évaluer notre calcul du majorant sur la latence. Le modèle utilisé est celui de l'outil d'analyse *python Compositional Performance Analysis* ou *pyCPA*. *pyCPA* est un outil développé en Python permettant d'analyser le comportement temporel de systèmes. Il est à noter que les analyses développées dans l'article [1] sont également développées en utilisant *pyCPA*. Le langage Python est un langage objet, ainsi *pyCPA* est composé de plusieurs *class* qui rassemblent un certains nombre de *method* qui permettent de manipuler des informations liées aux *class*.

Les *class* de *pyCPA* que nous utilisons pour définir notre modèle sont :

— *System* qui permet de créer l'entité globale qui contient les ressources (dans notre cas le processeur).

— *Resource* pour générer les ressources matérielles (dans notre cas le processeur).

— *Path* pour créer des séquences de tâches, c'est à dire des chaînes.

— *Task* pour créer des tâches et leurs affecter des propriétés temporelles.

— *EventModel* pour de décrire les modèles d'activation de tâches, on l'utilise pour gérer les chaînes activées de manière sporadique.

— *PJdEventModel* pour décrire les modèles d'activation périodiques.

En utilisant ces *class*, on peut générer des systèmes qu'on analysera ensuite. On génère et stocke, via le document *generator.py*, une liste de systèmes dans un fichier *csv*. Les paramètres à fixer dans *generator.py* pour définir les systèmes à générer sont les suivants :

— Un entier $nb\_systems\_to\_generate$ pour définir le nombre de systèmes à générer.

— Un booléen $sporadic\_only$ pour définir les modèles d'activation. Dans un système, soit toutes les chaînes sont sporadiques soit il y a des chaînes sporadiques et périodiques.

— Un booléen $full\_synchronous$ permettant de fixer le modèle d'exécution de toutes les chaînes à synchrone.

— Un booléen $full\_asynchronous$ permettant de fixer le modèle d'exécution de toutes les chaînes à asynchrone.

— Un tuple $utilization\_range$ contenant l'ensemble des pourcentages d'utilisation possibles pour un processeur.

— Un tuple $sporadic\_utilization\_range$ contenant l'ensemble des pourcentages d'utilisation possibles pour les chaînes sporadiques.

— Un entier $max\_nb\_task\_chains$ pour fixer le nombre maximum de chaînes dans un système.

— Un entier $max\_nb\_tasks\_per\_chain$ pour fixer le nombre maximum de tâches par chaîne.

— Un tuple $periods\_range$ contenant l'ensemble des périodes possibles pour une chaîne périodique.

— Une variable flottante $min\_sporadic\_wcet$ pour fixer le WCET minimum des tâches d'une chaîne.

— Une variable flottante $max\_sporadic\_wcet$ pour fixer le WCET maximum des tâches d'une chaîne.

De ces variables il est possible de générer des systèmes.

Nous décrivons maintenant l'algorithme permettant de générer les systèmes. La charge du processeur est choisie aléatoirement parmi les valeurs du tuple $utilization\_range$ défini par $[0.4, 0.5, 0.6, 0.7]$. On différencie ensuite deux cas selon les modèles d'activation des chaînes :

— S'il n'y a que des chaînes sporadiques dans le système alors le pourcentage d'utilisation $utilization\_range$ est également le pourcentage d'utilisation du processeur par les chaînes sporadiques $sporadic\_utilization\_range$.

— S'il y a des chaînes périodiques et sporadiques alors on choisit aléatoirement le pourcentage d'utilisation $sporadic\_utilization\_range$ alloué au chaînes sporadique parmi les valeurs $[0.001, 0.01, 0.1]$. Dans ce cas, le pourcentage d'utilisation alloué aux chaînes périodiques $periodic\_utilization$ est la différence $utilization - sporadic\_utilization$.

Ensuite on génère aléatoirement le nombre de chaînes de tâches qui est compris entre 2 et 9. Ces chaînes sont ensuite réparties entre les chaînes sporadiques et périodiques :

— Si toutes les chaînes sont sporadiques alors, il y a $max\_nb\_task\_chains$ chaînes sporadiques.

— S'il y a des chaînes sporadiques et périodiques alors on choisit aléatoirement le nombre de chaînes périodiques $nb\_periodic\_task\_chains$ qui est compris entre 1 et 9, et les autres chaînes sont sporadiques.

On fixe ensuite un nombre de tâches pour chacune des chaînes du système en commençant par les chaînes périodiques s'il y en a. Le nombre de tâches dans une chaîne est choisi aléatoirement entre 1 et 9. On crée ensuite une liste de priorités $possible\_priorities$ telle qu'il y ait autant de priorités que de tâches dans le système et on range ces priorités de manière aléatoire dans la liste. Pour finir chaque chaîne a un modèle d'exécution et on différencie 3 cas :

— Si $full\_synchronous = True$ alors toutes les chaînes sont synchrones.

— Si $full\_asynchronous = True$ alors toutes les chaînes sont asynchrones.

— Si $full\_synchronous = False$ et $full\_asynchronous = False$ alors le système est composé de chaînes synchrones et asynchrones et le modèle d'exécution de chaque chaîne est choisi aléatoirement.

La génération des paramètres restant dépend du modèle d'activation de la chaîne.

Cas 1 : La chaîne est périodique

La période de chacune des chaînes est choisie aléatoirement dans le tuple $periods\_range$ qui contient les valeurs $[10, 20, 50, 100, 200, 500, 1000]$. Ensuite, en fonction du nombre de chaînes périodiques $nb\_periodic\_task\_chains$ et de la charge allouée $periodic\_utilization$ aux chaînes périodiques, on répartit et on stocke l'utilisation de la ressource entre les chaînes périodiques dans la structure $periodic\_task\_chain\_utilizations$. Le WCET $chain\_wcets$ de chaque chaîne peut être réparti en utilisant le pourcentage d'utilisation $periodic\_task\_chain\_utilizations$ et la période $periods$ de la chaîne. Ensuite, pour chaque chaîne, le pourcentage d'utilisation $periodic\_task\_chain\_utilizations$ de la chaîne est réparti entre les $nb\_tasks\_per\_chain$ différentes tâches et cette répartition est stockée dans $task\_utilizations$. Le pourcentage d'utilisation d'une tâche et le modèle d'activation de la chaîne permettent de déduire le WCET $task\_wcets$ de chaque tâche dans cette chaîne. Pour chaque chaîne et pour chaque tâche dans cette chaîne, la dernière priorité de la liste $possible\_priorities$ est assignée à une tâche avant d'être retirée de cette liste.

Cas 2 : La chaîne est sporadique

Pour générer les $nb\_sporadic\_task\_chains$ chaînes sporadiques, on réparti la charge $sporadic\_utilization$ parmi l'ensemble des chaînes sporadiques. Le WCET $chain\_wcets$ de chaque chaîne sporadique est choisi de manière aléatoire entre $min\_sporadic\_wcet$ et $max\_sporadic\_wcet$. Les paramètres de WCET et d'utilisation de la charge permettent maintenant de définir un modèle d'activation $deltas$ des chaînes sporadiques. Pour chaque chaîne, on choisit de générer des valeur de δ^- pour de 2 à 100 activations consécutives. Ainsi, les $deltas$ sont établis de manière à ce que l'utilisation de la charge pour une chaîne sporadique soit respectée sur 100 activations consécutives. De la même manière que pour les chaînes périodiques, via le modèle d'activation d'une chaîne et le pourcentage d'utilisation de la charge qui lui est alloué, il est possible de fournir des WCETs $task\_wcets$ pour chaque tâche. Pour finir chaque tâche à également une priorité.

Les différentes charges pour les chaînes et les tâches sont générées utilisant U-Unifast [15] et les paramètres ont été choisis car ils sont considérés comme pertinents pour plusieurs cas d'études industriels [16].

Nous souhaitons ensuite comparer notre analyse à celle développée dans [1], qui fait plusieurs sur-approximations dans le calcul des des interférences. Nous ne comparons pas notre analyse à [2] car l'approche coïncide avec notre approche le cas où les chaînes sont synchrones et elle ne permet pas de prendre en compte les chaînes asynchrones.

3.7.2 Exploitation des résultats

Nous avons généré deux séries de systèmes. Les chaînes de ces systèmes ont toutes le même modèle d'exécution (synchrone ou asynchrone). La différence entre ces deux séries est que toutes les chaînes des systèmes de la première série sont sporadiques, ce qui n'est pas le cas de la seconde série. Nous différencions les deux séries selon le modèle d'activation des chaînes du système car, comme nous le verrons au chapitre 4, le modèle d'activation des chaînes influe sur l'évaluation de la précision du majorant sur la latence.

Les 1000 systèmes générés contenant uniquement des chaînes sporadiques rassemblent un total de 5581 chaînes qui sont également toutes synchrones. Sur les graphiques de cette section, pour chaque chaîne, nous calculons le majorant sur la latence en utilisant (i) les algorithmes développés dans l'article [1] (représenté en bleu) puis (ii) les algorithmes développés dans ce chapitre (représenté en rouge). Dans les différentes expérimentations, nous avons vu que le nombre de tâches par chaînes, le nombre de chaînes dans le système, la longueur des chaînes ou l'utilisation du processeur n'influence pas la précision du majorant de l'article [1] par rapport au notre. Ainsi, nous avons trié les chaînes par valeur croissante du majorant sur la latence et fixé leur identifiant selon la valeur de ce majorant.

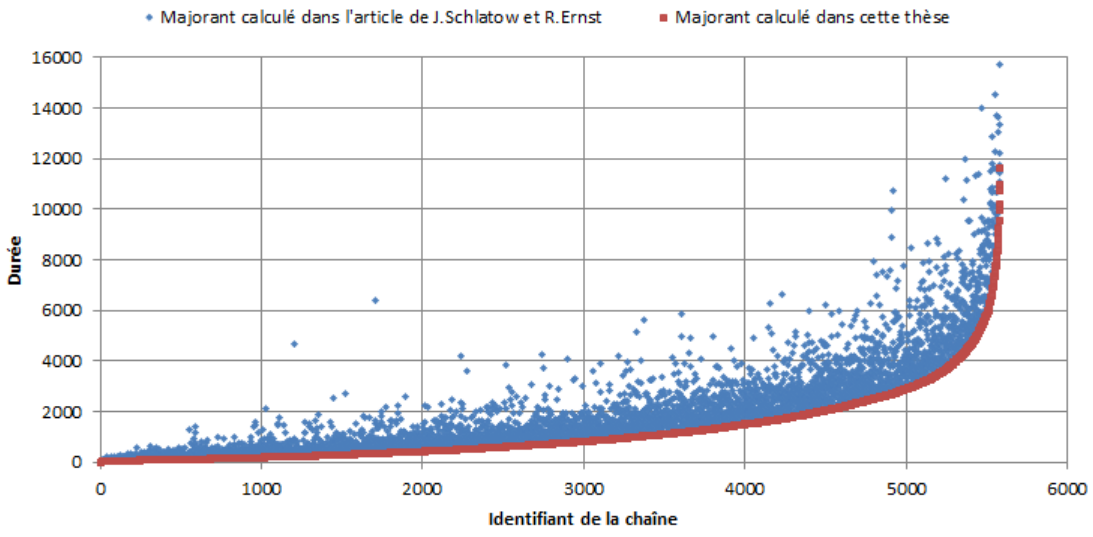

FIGURE 3.10 – Comparaison des majorants sur la latence au pire cas quand toutes les chaînes sont sporadiques et synchrones. En abscisse les identifiants des chaînes ordonnées selon la valeur du majorant calculé dans le chapitre 3.

La forte croissance observée au-dessus d'une valeur de 5000 pour la latence au pire cas est due à la génération des systèmes. La génération aléatoire des modèles d'activation peut créer des *burst*, c'est à dire que les valeurs de $\delta^-(n)$ pour les premières valeurs de n

sont petites par rapport à la latence. Cela crée donc beaucoup d'interférences et donc des latences au pire cas plus grandes.

Pour l'ensemble des 5581 chaînes analysées, nous calculons l'écart (en pourcentage) entre notre majorant dénoté $\overline{L_a}$ et celui calculé dans l'article [1] dénoté L_a^{SotA} en utilisant la formule :

$$\frac{L_a^{SotA} - \overline{L_a}}{L_a^{SotA}} \times 100 \tag{3.14}$$

Les résultats sont synthétisés dans la figure 3.11.

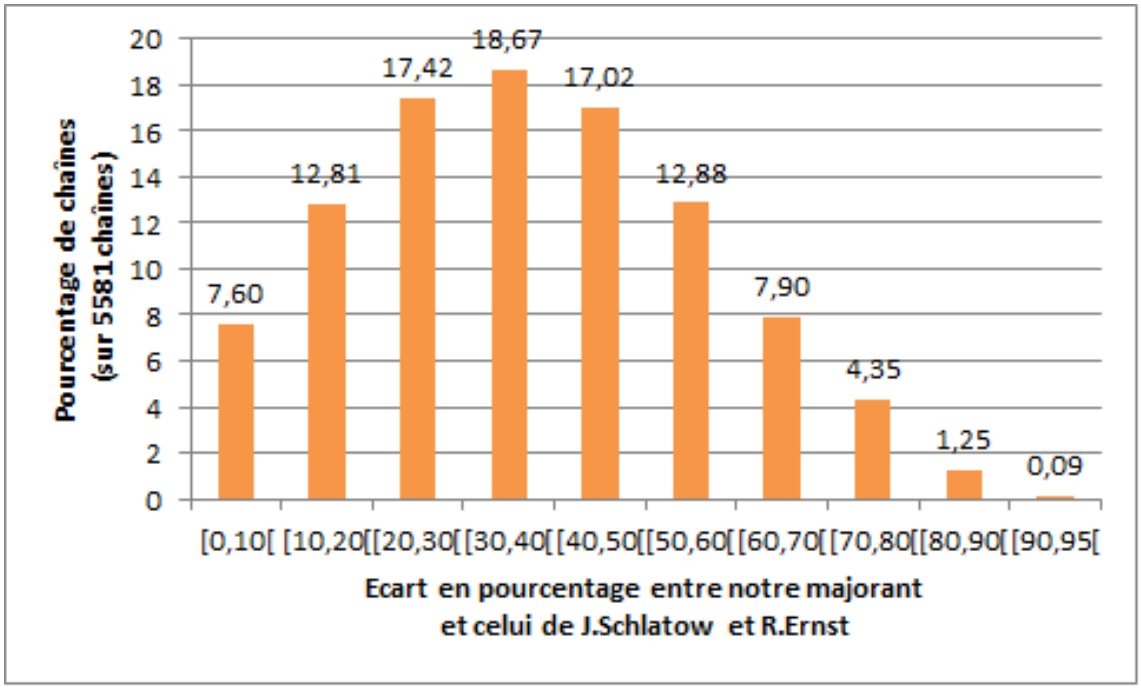

FIGURE 3.11 – Écart en pourcentages entre notre calcul du majorant et celui calculé dans l'article [1] dans le cas où toutes les 5581 chaînes sont sporadiques et synchrones.

Nous remarquons que d'une part le majorant de [1] est systématiquement supérieur au notre, et d'autre part, pour plus de 60% des chaînes, l'écart entre le majorant de [1] et le notre est supérieur à 30%.

Les 1000 systèmes générés contenant des chaînes sporadiques et périodiques rassemblent un total de 5538 chaînes qui ont été analysées en considérant successivement que (i) toutes les chaînes sont synchrones et (ii) toutes les chaînes sont asynchrones.

En comparant les résultats obtenus dans le cas de chaînes synchrones, il y une différence si les chaînes sont toutes sporadiques par rapport au cas où les chaînes sont sporadiques et périodiques. Dans le cas de chaînes uniquement sporadiques la différence entre notre majorant et celui de l'article [1] est plus importante que s'il y a des chaînes périodiques. En effet, la période minimum d'une chaîne périodique est de 10 tandis que $\delta^-(2)$ peut être très petit ce qui augmente le nombre de jobs pouvant interférer et, si l'interférence d'un job est sur-approximé, plus il y a de jobs qui interfèrent plus il y aura de sur-approximations.

Pour l'ensemble des 5538 chaînes des systèmes contenant des chaînes périodiques et sporadiques, nous avons utilisé l'équation (3.14) pour calculer l'écart (en poucentage)

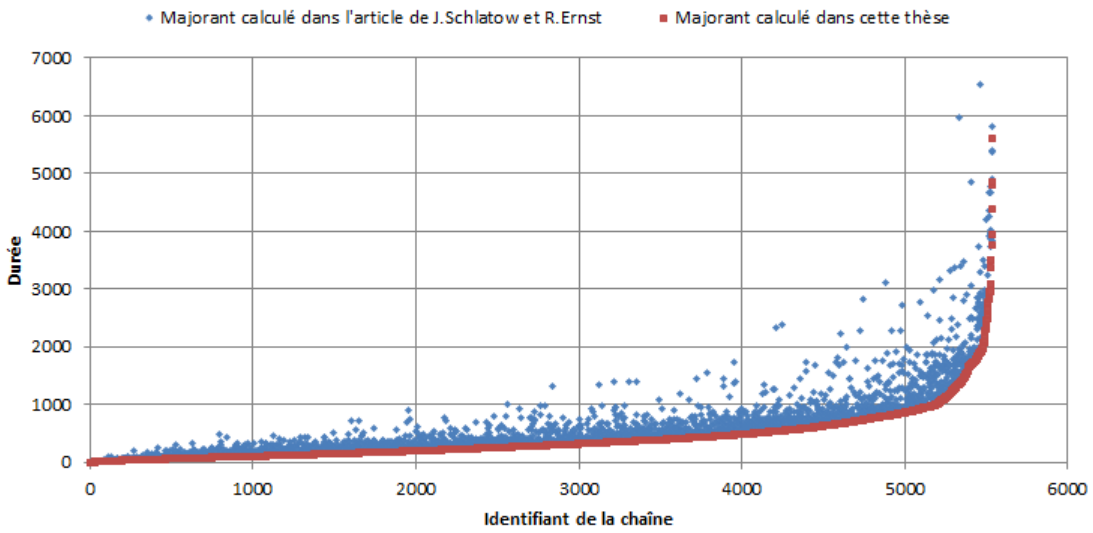

FIGURE 3.12 – Comparaison des majorants sur la latence au pire cas quand les chaînes sont sporadiques ou périodiques, et synchrones. En abscisse les identifiants des chaînes ordonnées selon la valeur du majorant calculé dans le chapitre 3.

entre notre majorant et celui calculé dans l'article [1].

Comme dans le cas où toutes les chaînes sont sporadiques, le majorant de [1] est supérieur au notre. Mais cette fais il n'y a plus que 25% des chaînes pour lesquelles l'écart entre le majorant de [1] et le notre est supérieur à 30%.

Il est à noter que dans le cas ou les chaînes sont asynchrones, les interférences sur une chaîne σ_a sont plus importantes que si les chaînes sont synchrones. Cela implique que les σ_a-*busy-window* sont plus longues tout comme les *busy-times*. Dans ce cas, le temps nécessaire pour que l'analyse converge est plus long. Dans le cas synchrone, sur la figure 3.12, certaines chaînes ont un long temps de latence (supérieure à 2000) et pour ces chaînes, le temps nécessaire pour que l'analyse converge peut être long. Si toutes les chaîne sont synchrones, il faut moins d'une heure pour analyser toutes les 5538 chaînes. Si les chaînes sont asynchrones, les interférences et donc le temps nécessaire pour converger augmentent. Pour cette raison, nous avons choisit de ne pas considérer les chaînes pour lesquelles le temps de convergence est trop long. Il n'y a donc que 4128 chaînes analysées dans le cas où toutes les chaînes du système sont asynchrones.

On remarque que parmi les 4128 chaînes, la plus grande valeur du majorant sur la latence est de environ 1000 alors que dans le cas où les chaînes sont synchrones le majorant de la $4128^{\text{ème}}$ chaîne est de environ 500. Les chaînes asynchrones génèrent donc plus d'interférences.

Sur l'ensemble de ces graphiques 3.10, 3.12 et 4.17, on remarque que notre majorant est plus précis que celui calculé dans [1].

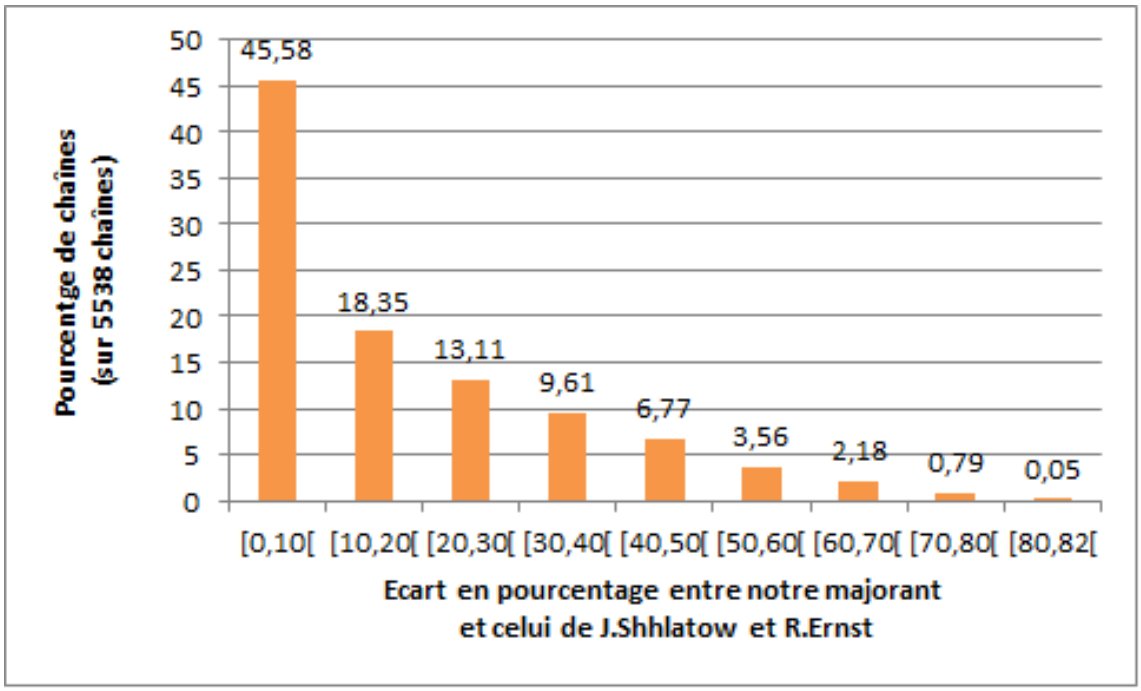

FIGURE 3.13 – Écart en pourcentages entre notre calcul du majorant et celui calculé dans l'article [1] dans le cas où au moins une chaîne par système est périodique.

3.8 Conclusion

Dans ce chapitre nous avons calculé des majorants sur la latence au pire cas de chaînes de tâches dans le cas général où les échéances des chaînes sont arbitraires en tenant compte des modèles d'exécution synchrones et asynchrones. Par rapport à l'article [2], notre approche est plus formelle, et plus générale car nous considérons le cas des chaînes asynchrones. Par rapport à l'article [1], comme nous l'avons montré par expérimentations, notre approche est plus précise.

Maintenant que nous avons calculé des majorants sur la latence au pire des chaînes, nous souhaitons maintenant évaluer la précision de ces majorants. Pour cela, nous allons calculer des minorants.

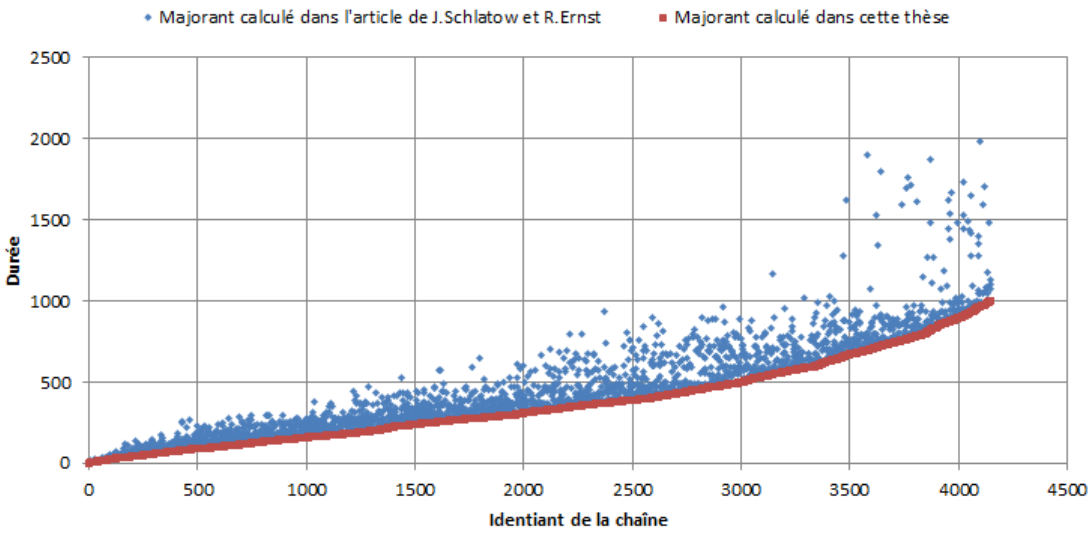

FIGURE 3.14 – Comparaison des majorants sur la latence au pire cas quand les chaînes sont sporadiques ou périodiques, et asynchrones. En abscisse les identifiants des chaînes ordonnées selon la valeur du majorant calculé dans le chapitre 3.

Chapitre 4

Minorer sur la latence au pire cas d'une chaîne

Sommaire

Dans ce chapitre, nous souhaitons évaluer la précision du majorant sur la latence au pire cas calculé au chapitre 3. Comme présenté dans l'introduction (section 1.3.1), notre objectif est d'adapter l'analyse développée au chapitre 3 pour calculer, pour une chaîne σ_a, un minorant sur sa latence pire cas, noté $\underline{L_a}$. La précision du calcul d'un majorant sur la latence au pire cas de σ_a peut être calculée ainsi :

$$precison(\overline{L_a}) = \frac{\overline{L_a} - \underline{L_a}}{\underline{L_a}} \tag{4.1}$$

Pour rappel, il ne faut pas confondre un minorant sur la latence au pire cas et la latence au meilleur cas (cf figure 4.1). La latence au meilleur cas (ou latence minimum) correspond à la latence de σ_a dans le cas où (i) toutes les tâches du système exécutent leur BCET et (ii) les interférences sur σ_a sont minimales. La latence des jobs de σ_a est nécessairement supérieure ou égale à la latence au meilleur cas. Un minorant sur la latence au pire cas est une valeur de latence inférieure ou égale à la latence au pire et il n'est pas nécessairement possible d'exhiber un scénario d'exécution qui réalise cette latence. Cependant, dans ce chapitre, nous nous intéressons uniquement aux minorants sur la latence au pire cas qui correspondent à une valeur de latence réalisable d'après le modèle du système donné [1].

1. Rappelons néanmoins que le minorant sur la latence au pire cas est calculé sur un modèle qui est une abstraction du système réel. L'exécution qui réalise ce minorant peut donc ne pas être possible en pratique sur la plate-forme finale.

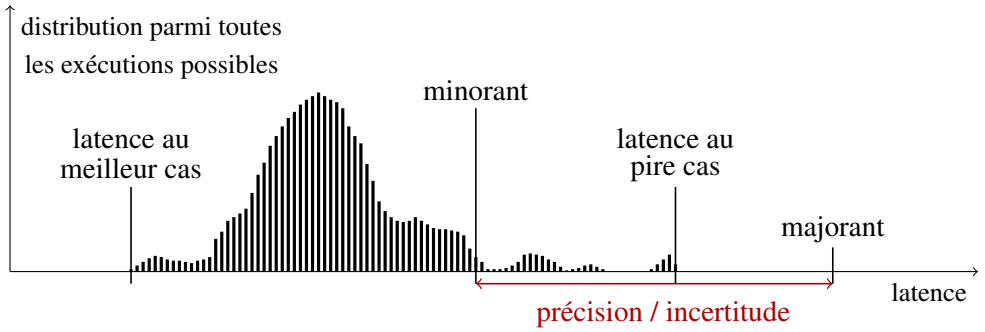

FIGURE 4.1 – Illustration du problème d'incertitude sur la latence au pire cas

Autrement dit, il existe un scénario d'exécution conforme au modèle tel que le minorant sur la latence au pire cas est atteint. Intuitivement, un **scénario d'exécution** est une séquence d'activations des chaînes du système.

Pour que le minorant soit utile pour évaluer la précision du majorant sur la latence au pire cas, il faut trouver le plus grand minorant possible. Il est à noter que si $\underline{L_a} = \overline{L_a}$ alors le majorant est exact, et donc il existe un scénario d'exécution qui réalise le majorant. Plus l'écart entre $\underline{L_a}$ et $\overline{L_a}$ est important, plus les approximations sont importantes. Ces dernières peuvent provenir du calcul du majorant ou de celui du minorant.

Les analyses du chapitre 3 peuvent introduire du pessimisme. Par rapport à une chaîne σ_a, les calculs développés au chapitre 3 fournissent des majorants sur les interférences provenant des chaînes plus prioritaires, moins prioritaires et de la chaîne elle-même. Ces interférences sont ensuite additionnées pour calculer un majorant sur le *busy-time* du $q^{\text{ème}}$ job de σ_a (équation (3.4)) qui sert directement au calcul d'un majorant sur la latence du $q^{\text{ème}}$ job de σ_a (équation (3.5)).

Le calcul du *busy-time* est majoré car la fonction $max()$ est subadditive [2]. Ainsi, lors du calcul d'un *busy-time*, calculer séparément des majorants sur les interférences, même si ces majorants sont les plus petits possibles, pour ensuite les additionner peut conduire à un résultat supérieur à l'interférence maximum réelle. C'est pour cette raison que l'équation (3.4) introduit potentiellement du pessimisme. Pour obtenir la latence maximum, il faudrait calculer les interférences maximum causées par l'ensemble des chaînes sans les séparer selon leur priorité, ce qui augmenterait beaucoup la complexité de l'analyse. Nous n'avons donc pour l'instant pas de garantie que la valeur calculée via l'équation (3.4) soit atteignable dans une σ_a-*busy-window*.

De même, l'équation (3.5) fournit un majorant sur la latence au pire et il n'y a pas de garantie que le plus long *busy-time* $B_a(q)$ soit compatible avec l'arrivée au plus tôt δ_a^- des q premier jobs de σ_a dans une σ_a-*busy-window*.

4.1 Etat de l'art et contribution

L'analyse d'ordonnançabilité est souvent utilisée pour majorer la latence au pire cas mais pas pour minorer cette latence. Pour minorer la latence au pire cas d'une chaîne, la

2. Ce qui signifie que pour deux fonctions $f(x)$ et $g(x)$, $x \in \mathbb{N}^+$, on a $max_{x\in\mathbb{N}^+}\{f(x) + g(x)\} \leq max_{x\in\mathbb{N}^+}\{f(x)\} + max_{x\in\mathbb{N}^+}\{g(x)\}$.

simulation est souvent utilisée. Pour obtenir un minorant par simulation, il faut simuler le système sur une ou plusieurs traces puis récupérer la latence maximum mesurée sur ces traces. Dans ce chapitre, nous proposons d'utiliser l'analyse d'ordonnançabilité pour calculer un minorant sur la latence au pire cas d'une chaîne.

Comme vu au chapitre 3, l'analyse d'ordonnançabilité permet de majorer la latence au pire cas d'une chaîne en majorant les différentes interférences qui peuvent suspendre l'exécution de cette chaîne. Il est à noter que, dans ce cas et comme nous le verrons plus tard dans ce chapitre, il n'est pas toujours possible de trouver un scénario d'exécution qui réalise ce majorant sur la latence au pire cas. Cependant, il est toujours possible d'exhiber un scénario d'exécution telle que la latence obtenue par simulation est atteinte.

La simulation est une approche basée sur les modèles et le modèle utilisé en simulation est identique à celui de l'analyse d'ordonnançabilité. Cependant, l'approche pour évaluer la latence au pire cas est différente dans le cas de la simulation. Lors d'une simulation, plusieurs scénarios d'exécution sont générés et exécutés. Parmi tous ces scénarios, il est possible d'approcher la latence au pire cas en gardant la plus grande valeur de latence trouvée parmi toutes les exécutions. La latence obtenue par simulation est inférieure ou égale à cette latence au pire cas. Une valeur de latence obtenue par simulation n'est pas basée sur les principes de l'analyse statique, et pour l'obtenir, il faut choisir aléatoirement certains paramètres temporels pour exécuter différents scénarios. Il faudra choisir le nombre de simulations à exécuter, la longueur des traces qui composent cette simulation, et pour chaque simulation, le scénario d'activation. Le temps d'exécution de chaque job est également choisi aléatoirement dans le cas général. Dans le cas de chaînes de tâches, une meilleure comparaison avec l'analyse statique sera obtenue en configurant la simulation de façon à ce que chaque job s'exécute à hauteur de son WCET. Plusieurs outils permettent de simuler le comportement temporel de systèmes temps-réel, par exemple *SimSo* [17] [3] ou *Storm* [18] [4].

La simulation est rapide à mettre en oeuvre et ne nécessite pas le développement d'analyses d'ordonnançabilité. Cependant, la précision de la latence obtenue par simulation dépend (i) du nombre de simulations lancées et (ii) de la longueur des traces. Ainsi la précision de cette latence dépend de temps passé à simuler le système. En utilisant l'analyse d'ordonnançabilité pour minorer la latence au pire cas, le calcul du minorant sera effectué par un calcul de point fixe.

Dans ce chapitre, nous montrons comment calculer, pour un système composé de chaînes de tâches allouées sur un processeur, des minorants sur la latence au pire cas d'une chaîne. Pour rappel, un minorant sur la latence au pire cas d'une chaîne permet d'évaluer la précision du majorant sur la latence au pire cas de cette même chaîne.

4.2 Approche générale

Pour calculer un minorant sur la latence au pire cas, on souhaite construire une σ_a-*busy-window* réalisable compte tenu du modèle du système et dans cette σ_a-*busy-window* la latence du $q^{\text{ème}}$ job est $\ell_a(q)$. Plus un minorant est proche du pire cas plus il est pertinent pour calculer la précision du majorant et, dans cet objectif, on cherche la plus grande valeur

3. http ://projects.laas.fr/simso/
4. http ://www.rts-software.org/

possible de latence parmi les jobs de σ_a activés dans la σ_a-*busy-window*. Cette latence maximale est un minorant, notée $\underline{L_a}$, sur la latence au pire cas de σ_a.

Pour obtenir un minorant sur la latence au pire cas d'une chaîne σ_a, on construit un scénario d'exécution des chaînes du système et donc une σ_a-*busy-window* telle que le modèle du système soit respecté. On peut ensuite déterminer le nombre d'activations de σ_a dans cette σ_a-*busy-window*. Pour chacun des q jobs activés dans cette σ_a-*busy-window*, on calcule un *busy-time* $b_a(q)$ qui est réalisable compte tenu du modèle du système. Calculer ce *busy-time* implique de calculer les interférences réelles sur σ_a. Ces interférences et donc les *busy-times* dépendent de la σ_a-*busy-window* construite. Compte tenu du modèle d'activation de σ_a dans cette σ_a-*busy-window*, il est ensuite possible de calculer les latences $\ell_a(q)$ des jobs de σ_a activés dans cette σ_a-*busy-window*.

Ainsi, notre approche pour obtenir un minorant sur la latence au pire cas est similaire à celle permettant de calculer des majorants sur la latence au pire cas. Les différentes étapes pour obtenir un minorant sur la latence au pire cas de σ_a sont :

1. Construction d'une σ_a-*busy-window* réalisable et calcul de la longueur de cette σ_a-*busy-window*.
2. Calcul du nombre d'activations de σ_a dans cette σ_a-*busy-window*.
3. Pour chacun des jobs de σ_a dans cette σ_a-*busy-window* :
   (a) Calcul de la longueur du *busy-time* du $q^{\text{ème}}$ job de σ_a.
   (b) Calcul de la latence du $q^{\text{ème}}$ job de σ_a.
4. Calcul de la latence maximum parmi tous les jobs de σ_a activés dans la σ_a-*busy-window*.

Il est à noter que pour calculer un minorant sur la latence au pire cas d'une chaîne, il est possible de ne considérer qu'un unique job de σ_a. Le calcul du minorant sera cependant plus précis en considérant le nombre maximum de jobs qui peuvent être activés dans une σ_a-*busy-window*. Il est également à noter que le calcul du minorant dépend de la σ_a-*busy-window* construite.

Par la suite, nous souhaitons calculer un minorant sur la latence au pire cas en utilisant les concepts et les équations développées au chapitre 3. Le *busy-time* du $q^{\text{ème}}$ job de σ_a introduit à la définition 26 ainsi que les interférences sur σ_a introduites aux définitions 21, 30 et 31 sont définis à partir de l'ouverture d'une σ_a-*busy-window*. Pour réutiliser les concepts développés au chapitre 3, nous nous restreignons à construire une σ_a-*busy-window* qui s'ouvre avec l'activation d'un job de σ_a. Ainsi, même si le scénario figure 4.2 est réalisable, nous ne l'utiliserons pas pour calculer un minorant sur la latence au cas de σ_a.

Les contraintes sur la σ_a-*busy-window* que nous construisons sont (i) qu'elle est réalisable compte tenu du modèle du système et (ii) qu'elle s'ouvre avec l'activation d'un job de σ_a. Ainsi, l'hypothèse suivante permet de réutiliser le calcul du majorant sur la latence au pire cas pour calculer un minorant sur la latence au pire cas.

Hypothèse 78. *La σ_a-busy-window s'ouvre avec l'activation d'un job de σ_a.*

Dans la σ_a-*busy-window* construite, il est possible, via le théorème suivant, de calculer la latence de chaque job de σ_a activé dans cette σ_a-*busy-window*.

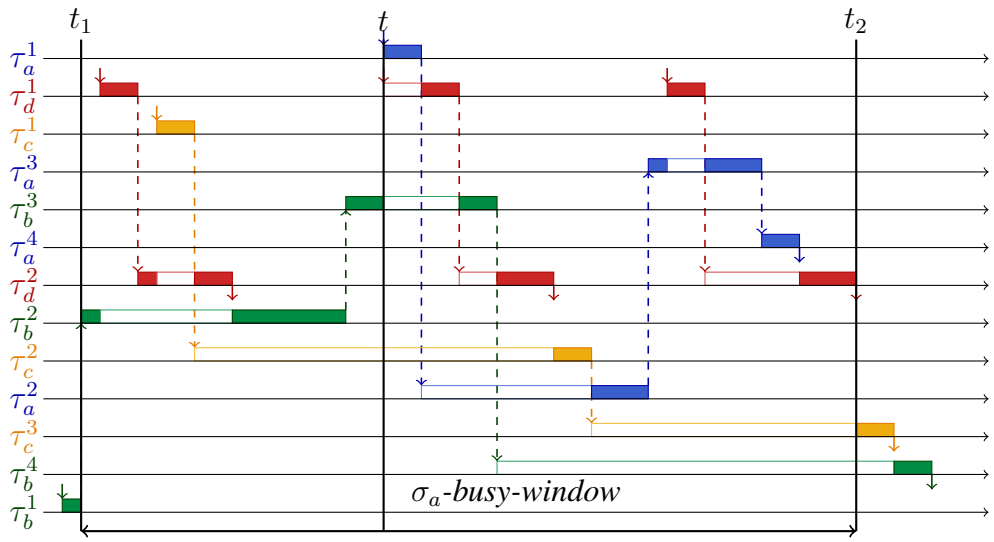

FIGURE 4.2 – Illustration d'une σ_a-*busy-window* qui ne s'ouvre pas avec une activation de σ_a

Théorème 79 (Minorant sur la latence au pire cas)**.** *Soit une σ_a-busy-window $[t_1, t_2]$ construite selon l'hypothèse 78 et dans laquelle σ_a est activée à t_1 et ensuite au plus tôt selon son modèle d'activation. La latence au pire cas de σ_a est minorée par :*

$$\underline{L_a} = \max_{q \in [1, \eta_a^+(t_2 - t_1)]} \{b_a(q) - \delta_a^-(q)\} \tag{4.2}$$

Démonstration. Soit une σ_a-*busy-window* qui s'ouvre à t_1 avec l'activation d'un job de σ_a et se ferme t_2. Dans cette σ_a-*busy-window*, le $q^{\text{ème}}$ job de σ_a est activé à $t_1 + \delta_a^-(q)$ et termine à $t_1 + b_a(q)$. Ainsi $b_a(q) - \delta_a^-(q)$ est la latence du $q^{\text{ème}}$ job de σ_a dans cette σ_a-*busy-window*. La latence maximum parmi les q activations [5] correspond bien à un minorant sur la latence au pire cas. □

Pour obtenir un minorant sur la latence au pire cas en utilisant l'équation 4.2, il reste à calculer les valeurs $b_a(q)$ des q jobs de σ_a ($q \in [1, \eta_a^+(t_2 - t_1)]$) activés dans la σ_a-*busy-window*. Nous avons déjà restreint le scénario d'exécution en considérant une σ_a-*busy-window* construite selon l'hypothèse 78, ce qui permet de réutiliser les analyses développées au chapitre 3. Pour calculer le plus grand minorant possible sur la latence au pire cas, on maximise les *busy-times* des q jobs de σ_a activés dans la σ_a-*busy-window* pour obtenir, pour chacun des q jobs de σ_a un minorant $\underline{B_a(q)}$ le plus proche possible du *busy-time* maximum $B_a(q)$.

Pour calculer un minorant le plus proche possible du pire cas, on rajoute une contrainte supplémentaire lors de la construction de la σ_a-*busy-window*. On active, comme présenté dans l'hypothèse 80, les chaînes de $hpe(a)$ à l'ouverture de la σ_a-*busy-window* et ensuite le plus tôt possible compte tenu de leur modèle d'activation.

Hypothèse 80. *La chaîne σ_a et l'ensemble des chaînes de priorité supérieure à π_a sont activées à l'ouverture de la σ_a-busy-window et ensuite le plus tôt possible compte tenu de leur modèle d'activation.*

5. Il y a au plus $\eta_a^+(t_2 - t_1)$ activations de σ_a dans un intervalle de temps de longueur $t_2 - t_1$.

On rajoute une seconde hypothèse concernant les WCETs des chaînes.

Hypothèse 81. *La chaîne σ_a et l'ensemble des chaînes de priorité supérieure à π_a s'exécutent à hauteur de leur WCET.*

Rajouter ces contraintes supplémentaires permet d'utiliser le calcul des majorants sur les interférences des chaînes de $hpe(a)$ pour calculer un minorant sur la latence au pire cas. Un exemple de σ_a-*busy-window* que nous utilisons pour le calcul d'un minorant sur la latence au pire cas est donné par la figure 4.3.

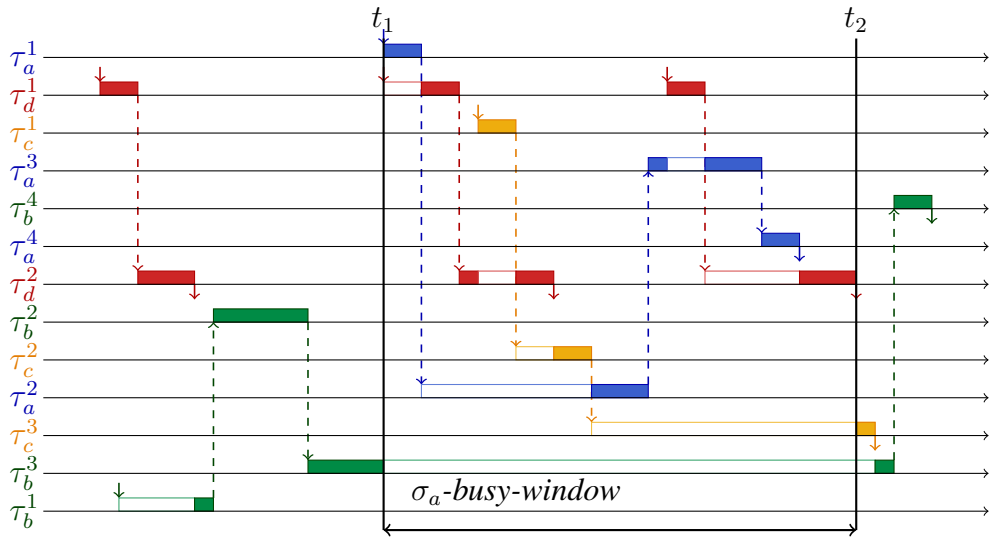

FIGURE 4.3 – Les chaînes de $hpe(a)$ sont activées à l'ouverture de la σ_a-*busy-window*

On remarque que sur la figure 4.3, l'interférence de σ_b sur σ_a n'est pas maximale car son segment interne $[\tau_b^3]$ n'interfère que partiellement avec σ_a.

La propriété suivante établit qu'il est possible d'adapter la formule pour calculer des majorants sur l'interférence des chaînes plus prioritaires et sur l'auto-interférence dans le but de calculer un minorant sur le *busy-time* $b_a^i(q)$ du $q^{\text{ème}}$ job de τ_a^i.

Propriété 82. *Soit une σ_a-busy-window $[t_1, t_2]$ construite en accord avec les hypothèses 78, 80 et 81. Soit Δ un préfixe de la σ_a-busy-window et $i > \ell t_a(d)$:*

Cas A : L'interférence de $\sigma_d \in hp(a)$ sur q jobs de τ_a^i pour $\Delta > b_a^{i-1}(q)$ est égale à :

Cas A.1 : Si σ_d est synchrone :

$$hp\mathfrak{I}^{i,q}_{\substack{d\to a\\ sync}}(\Delta) = \eta_b^+(\mathbf{b_a^{\ell t_a(d)}(q)}) \times C_d + sync\mathfrak{I}^{i,q}_{a,d} \tag{4.3}$$

$$\text{où} \quad sync\mathfrak{I}^{i,q}_{a,d} = \begin{cases} 0 & \text{si } \eta_d^+(\mathbf{b_a^{\ell t_a(d)}(q)}) = \eta_d^+(\Delta) \\ C_{s^{head}_{d\to a[k..i]}} & \text{sinon} \end{cases}$$

$$\text{et} \quad k = \min\left(i, \min_{u\in\{\ell t_a(d)+1,\dots,i-1\}} \{\eta_d^+(\mathbf{b_a^{u-1}(q)}) \neq \eta_d^+(\mathbf{b_a^u(q)})\}\right)$$

Les principales différences avec l'équation 3.8 sont représentées en rouge.

Cas A.2 : Si σ_d est asynchrone :

$$hp\mathfrak{I}^{i,q}_{\substack{d\to a\\ async}}(\Delta) = \eta^+_d(\mathbf{b}^{\ell t_a(d)}_{\mathbf{a}}(\mathbf{q})) \times C_d + async\mathfrak{I}^{i,q}_{a,d} \tag{4.4}$$

$$où \quad async\mathfrak{I}^{i,q}_{a,d} = \sum_{k=\ell t_a(d)+1}^{i-1} \left(\eta^+_d(\mathbf{b}^{\mathbf{k}}_{\mathbf{a}}(\mathbf{q})) - \eta^+_d(\mathbf{b}^{\mathbf{k-1}}_{\mathbf{a}}(\mathbf{q}))\right) \times C_{s^{head}_{d\to a[k..i]}} + \left(\eta^+_d(\Delta) - \eta^+_d\left(\mathbf{b}^{\mathbf{i-1}}_{\mathbf{a}}(\mathbf{q})\right)\right) \times C_{s^{head}_{d\to a[i..i]}}$$

Les principales différences avec l'équation 3.10 sont représentées en rouge.

Cas B : l'auto-interférence sur q événements de τ^i_a des jobs suivants de σ_a pour $\Delta > b^{i-1}_a(q)$ est égale à :

$$self\mathfrak{I}^{i,q}_a(\Delta) = \left(\eta^+_a(\mathbf{b}^{\ell t_a}_{\mathbf{a}}(\mathbf{q})) - q\right) \times C_{s^{head}_{a\to a[1..\ell t_a]}} + self\mathfrak{I}^{i,q}_a \tag{4.5}$$

$$où \quad self\mathfrak{I}^{i,q}_a = \left(\eta^+_a(\Delta) - \eta^+_a(\mathbf{b}^{\mathbf{i-1}}_{\mathbf{a}}(\mathbf{q}))\right) \times C_{s^{head}_{a\to a[i..i]}} + \sum_{k=\ell t_a+1}^{i-1} \left(\eta^+_a(\mathbf{b}^{\mathbf{k}}_{\mathbf{a}}(\mathbf{q})) - \eta^+_a(\mathbf{b}^{\mathbf{k-1}}_{\mathbf{a}}(\mathbf{q}))\right) \times C_{s^{head}_{a\to a[k..i]}}$$

Les principales différences avec l'équation 3.13 sont représentées en rouge.

Démonstration. Cette preuve se déroule en deux parties, nous considérons successivement l'interférence des chaînes de priorité supérieure puis l'auto-interférence.

Cas A : Concernant les chaînes de priorité supérieure à σ_a, nous différencions deux cas selon l'indice de la tâche pour laquelle le *busy-time* est calculé :

Si $\ell t_a \leq i \leq \ell t_a(d)$ alors, compte tenu de notre scénario et de manière similaire à la propriété 61, la chaîne σ_d est activée exactement $\eta^+(\Delta)$ fois entre t_1 et $t_1 + \Delta$. Tant que ces jobs sont activés avant la terminaison de $\tau^{\ell t_a(d)}_a$, ils s'exécutent entièrement avant la terminaison du $q^{\text{ème}}$ job de τ^i_a peu importe le modèle d'exécution de la chaîne σ_d.

Si $i > \ell t_a(d)$ alors, de manière similaire à la propriété 62, il faut distinguer deux cas :

Si $\eta^+(\Delta) = \eta^+_d(\underline{b^{\ell t_a(d)}_a(q)})$ alors les arguments précédents sont toujours valables car il n'y a pas de nouveaux jobs de σ_d activés entre la terminaison du $q^{\text{ème}}$ job de $\tau^{\ell t_a(d)}_a$ et la terminaison du $q^{\text{ème}}$ job de τ^i_a.

Si $\eta^+_d(b^{\ell t_a(d)}_a(q)) < \eta^+_d(\Delta)$ alors un nouveau job de σ_d est activé entre la terminaison du $q^{\text{ème}}$ job de $\tau^{\ell t_a(d)}_a$ et la terminaison du $q^{\text{ème}}$ job de τ^i_a. Son interférence dépend du modèle d'exécution de σ_d :

Cas A.1 : Si σ_d est synchrone, l'interférence d'une chaîne plus prioritaire dans le cas où les échéances du système sont arbitraires peut être calculée via le calcul des interférences dans le cas des échéances contraintes. τ^k_a est la première tâche de σ_a qui ne peut pas terminer sa $q^{\text{ème}}$ exécution avant que le premier job de σ_d après $t_1 + \underline{b^{\ell t_a(d)}_a(q)}$ soit activé. Ce job de σ_d interfère pour $C_{s^{head}_{d\to a[k..i]}}$ et les jobs suivants sont suspendus dès leur activation car σ_d est synchrone.

Cas A.2 : Si σ_d est asynchrone, alors chaque activation de σ_d peut interférer avec σ_a. Un job de σ_d activé pendant l'exécution de τ_a^k interfère pour $C_{s^{head}_{d\to a[k..i]}}$.

Cas B : Concernant l'auto-interférence, de la même manière que pour une chaîne plus prioritaire, si σ_a est synchrone alors les jobs suivants de σ_a ne peuvent pas interférer. Si la chaîne est asynchrone alors les segments de tête des jobs suivants peuvent interférer. σ_a est activée à l'instant t_1 puis le plus tôt possible selon son modèle d'activation ainsi l'interférence des segments de tête peut facilement être calculée en utilisant la même approche que pour les chaînes plus prioritaires.

□

La propriété 82 permet d'établir une stratégie pour calculer un minorant sur la latence au pire cas s'il est possible de trouver un minorant sur les interférences des chaînes de priorité inférieure à σ_a.

Corollaire 83. *Soit une σ_a-busy-window $[t_1, t_2]$ construite en accord avec les hypothèses 78, 80 et 81 et soit un préfixe de longueur Δ de cette σ_a-busy-window ; si, dans cette σ_a-busy-window, l'interférence des chaînes de priorité inférieure sur q jobs de τ_a^i est $\ell p\mathfrak{I}_a^{i,q}(\Delta)$, alors le busy-time sur q événements de τ_a^i est :*

$$b_a^i(q) = (q-1) \times C_a + C_{a[1..i]} + \ell p\mathfrak{I}_a^{i,q}\left(b_a^i(q)\right) + \sum_{\sigma_d \in hp(a)} hp\mathfrak{I}_{d\to a}^{i,q}\left(b_a^i(q)\right) + self\mathfrak{I}_a^{i,q}\left(b_a^i(q)\right) \quad (4.6)$$

Démonstration. Considérons une *σ_a-busy-window* $[t_1, t_2]$ construite selon les hypothèses 78, 80 et 81. Le $q^{\text{ème}}$ job de τ_a^i se termine à $t_1 + b_a^i(q)$ car, dans cette *σ_a-busy-window* :

1. L'interférence des chaînes moins prioritaires est de $\ell p\mathfrak{I}_a\left(b_a^i(q)\right)$.
2. D'après la propriété 82, l'interférence d'une chaîne σ_d plus prioritaire est $hp\mathfrak{I}_{d\to a}^{i,q}\left(b_a^i(q)\right)$
3. D'après la propriété 82, l'auto-interférence est $self\mathfrak{I}_a^{i,q}\left(b_a^i(q)\right)$.

□

Compte tenu du corollaire 83, pour calculer un minorant $\underline{B_a(q)}$, il nous reste à calculer un minorant sur l'interférence maximum des chaînes de priorité inférieure à σ_a dans une *σ_a-busy-window* $[t_1, t_2]$ satisfaisant les hypothèses 78, 80 et 81.

Dans le reste de ce chapitre, nous proposons différentes méthodes de calcul de minorant en fonction du modèle du système.

Nous utilisons la notation $\underline{\ell p\mathcal{I}_a}^{i,q}_{(n)}$ pour différencier les minorants proposés. Pour un système donné, on obtiendra une meilleure précision en utilisant le maximum sur les minorants disponibles.

$$\underline{\ell p\mathcal{I}_a}^{i,q} = \max_n \underline{\ell p\mathcal{I}_a}^{i,q}_{(n)} \quad (4.7)$$

4.3 Minorer l'interférence des chaînes de priorité inférieure

De manière similaire au calcul du majorant sur les interférences, le calcul d'un minorant sur les interférences est plus simple si l'on considère que toutes les échéances du système sont contraintes.

4.3.1 Cas I : Les échéances sont contraintes

Pour rappel, si les échéances sont contraintes et si le système est ordonnançable, alors deux jobs d'une même chaîne ne peuvent pas être en instance simultanément et par conséquent la distinction entre les modèles d'exécutions synchrones et asynchrones ne se manifeste pas pour les systèmes ordonnançables.

Nous commençons par traiter le cas où toutes les échéances sont contraintes et le segment critique qui maximise l'équation (3.6) n'est pas circulaire. L'équation (3.6), rappelée ci-dessous, permet de calculer un majorant sur l'interférence des chaînes moins prioritaires si les échéances sont contraintes.

$$\overline{\ell p \mathcal{I}_a} = \max_{\sigma_b \in \ell p(a)} \left\{ C_{s^{crit}_{b \to a}} + \sum_{\sigma_c \in \ell p(a) \wedge c \neq b} C_{s^{head}_{c \to a}} \right\}$$

Cas I.A : Le segment critique qui maximise l'équation (3.6) **n'est pas circulaire**

Il est à noter que valider le minorant sur la latence au pire cas est plus simple si toutes les chaînes du système sont sporadiques. En effet, si une chaîne est sporadique, il n'y a pas de distance maximum entre plusieurs activations consécutives (c'est à dire que la fonction δ^+ n'est pas définie) et par conséquent deux activations peuvent être arbitrairement éloignées l'une de l'autre. Il est ainsi possible de construire un scénario réalisable dans une σ_a-*busy-window* en supposant qu'aucun job n'est activé avant l'ouverture de cette σ_a-*busy-window*. Ce n'est plus le cas s'il y a une chaîne périodique dans le système.

Théorème 84. *Dans un système ordonnançable, supposons que toutes les échéances sont contraintes et que le segment critique qui maximise l'équation* (3.6) *n'est pas circulaire alors, $\overline{\ell p \mathcal{I}_a}$ tel que calculé dans l'équation* (3.6) *est un minorant sur les interférences au pire cas des chaînes moins prioritaires que σ_a pour les scénarios d'exécution compatibles avec les hypothèses 78, 80 et 81.*

Démonstration. Soit σ_{crit} la chaîne [6] dont le segment critique maximise $\overline{\ell p \mathcal{I}_a}$ dans l'équation (3.6) et soit τ^j_{crit} la première tâche du segment critique de σ_{crit} sur σ_a. Supposons que le segment critique n'est pas circulaire. Considérons une σ_a-*busy-window* $[t_1, t_2]$ compatible avec les hypothèses 78, 80 et 81 et telle que s'ouvrant à un instant t_1 et le scénario d'exécution suivant dans lequel toutes les tâches s'exécutent durant leur WCET :

1. toutes les chaînes de $\ell p(a)$ sauf σ_{crit} sont activées à t_1 ;
2. σ_{crit} est activée de telle sorte que τ^j_{crit} est activée à l'instant t_1 ;
3. toutes les chaînes de $\ell p(a)$ s'exécutent à hauteur de leur WCET.

6. S'il y en a plusieurs, on en choisit une aléatoirement.

Compte tenu de ce scénario d'exécution, il faut prouver :

A : qu'il est possible d'activer σ_{crit} de telle sorte que τ_{crit}^{j} soit activée à l'instant t_1 ;

B : que l'interférence des chaînes de priorité inférieure à σ_a correspond à celle fournie par l'équation (3.6).

Commençons par considérer le cas simple dans lequel **toutes les chaînes du système sont sporadiques**.

Toutes les chaînes du système sont sporadiques - Preuve de A : Comme l'objectif est de maximiser les interférences sur la σ_a-*busy-window* et que les chaînes sporadiques n'ont pas de distance maximale δ^+ alors nous considérons également qu'il n'y a aucune activation avant t_1 sauf celle de σ_{crit}. Ainsi, si σ_{crit} est activée à $t_1 - C_{crit[1..j[}$ alors τ_{crit}^{j} est activée exactement à l'instant t_1 (cf figure 4.4).

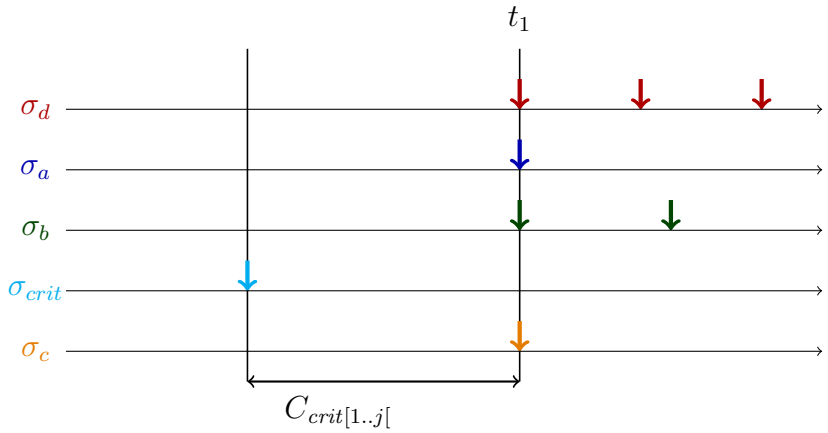

FIGURE 4.4 – Scénario d'exécution de chaînes sporadiques

Toutes les chaînes du système sont sporadiques - Preuve de B : Dans un tel scénario d'exécution, le segment de tête de chaque chaîne dans $\ell p(a) \backslash \{\sigma_{crit}\}$ est activé à t_1, en même temps que le segment critique de σ_{crit}. Toutes les tâches de ces segments ont, par définition, une priorité plus grande que celle de $\tau_a^{\ell t_a}$ et donc ces segments termineront leur exécution avant la terminaison du premier job de $\tau_a^{\ell t_a}$ et donc, avant la terminaison du $q^{\text{ème}}$ job de τ_a^i pour $i > \ell t_a$. D'où le résultat.

Considérons maintenant qu'il y a **au moins une chaîne périodique** dans le système.

Au moins une chaîne du système est périodique - Preuve de A : Dans ce cas, on ne peut pas supposer qu'aucune chaîne n'est activée avant l'ouverture de la σ_a-*busy-window*. On considère toujours que toutes les chaînes sauf σ_{crit} sont activées à l'instant t_1, mais pour les chaînes périodiques il faut prendre en compte les activations précédentes. Commençons donc par montrer qu'il existe une activation de σ_{crit} telle que τ_{crit}^{j} soit activée à l'instant t_1. Pour illustrer notre construction nous utilisons la figure 4.5. On note $t_0 = t_1 - \overline{BW_{crit}}$. Puisque $\overline{BW_{crit}}$ est un majorant sur la longueur des σ_{crit}-*busy-windows* alors, par définition, il existe dans l'intervalle $[t_0, t_1]$ au moins un instant pendant lequel aucun job d'une chaîne de priorité supérieure ou égale à σ_{crit} n'est en instance. Nous notons t_{idle} le dernier de ces instants dans $[t_0, t_1]$ et, à t_{idle} s'ouvre une σ_{crit}-*busy-window*. Pour quantifier l'interférence des chaînes du système sur le segment de σ_{crit} qui s'exécute avant t_1, nous séparons les chaînes du système en trois catégories selon leur priorité :

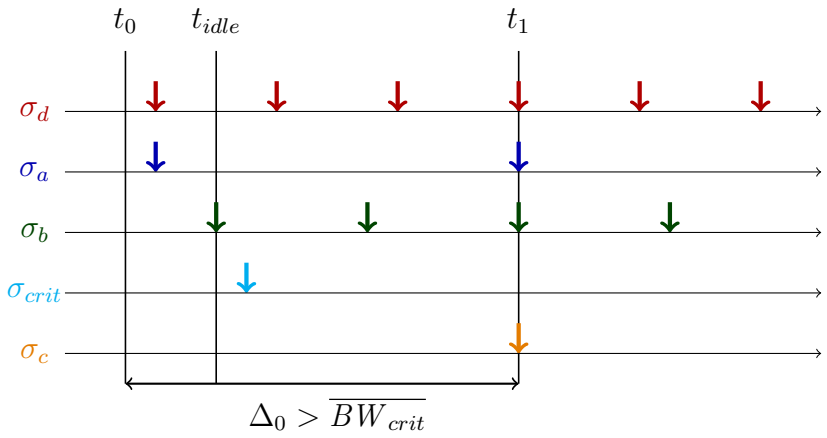

FIGURE 4.5 – Scénario d'exécution de chaînes périodique

1. Chaque job des **chaînes de** $hpe(a)$ (σ_d et σ_a sur la figure 4.5) activé avant t_1 suspend σ_{crit} et termine son exécution entre t_{idle} et t_1 car pour toute chaîne σ_d de $hpe(a)$, $\pi^j_{crit} < \pi_d$.
2. Si le système est ordonnançable, alors chaque job d'une **chaîne de** $\ell p(a)$ **et** $hp(crit)$ (σ_b sur la figure 4.5), activé dans l'intervalle $[t_{idle}, t_1[$, interfère entièrement avec σ_{crit}. Dans le cas où un job de σ_b ne serait pas terminé à t_1, le système ne serait pas ordonnançable. Ainsi, si le système est ordonnançable, chaque chaîne σ_b interfère entièrement et donc, il est possible de quantifier les interférences de ces chaînes sur σ_{crit}.
3. Si le système est ordonnançable, les **chaînes de** $\ell p(crit)$ (σ_c sur la figure 4.5) ne peuvent pas être activées deux fois dans l'intervalle $[t_{idle}, t_2]$. En effet, une chaîne de $\ell p(crit)$ peut s'exécuter dans une σ_{crit}-*busy-window* à hauteur d'un unique segment (cette chaîne ne peut donc pas s'exécuter entièrement) et si deux jobs peuvent être activés dans une même σ_{crit}-*busy-window* alors un des jobs manquera son échéance.

Pour rappel nous cherchons l'instant d'activation de σ_{crit} tel que τ^j_{crit} termine à t_1 et donc tel que le segment critique de σ_{crit} commence à t_1. On a montré comment calculer l'interférence de chaque job activé pendant l'exécution de la sous-chaîne $\sigma_{crit[1...j]}$. Il faut maintenant considérer le cas où un job activé avant σ_{crit} peut interférer avec σ_{crit}. Si σ_{crit} est activée pendant l'exécution d'une tâche τ^k_f (peu importe la priorité de σ_f) alors, l'interférence de ce job de σ_f est le segment de tête de la sous-chaîne $\sigma_{crit[1...j]}$ sur la sous-chaîne $\sigma_{f[k...n_f]}$, c'est à dire $s^{head}_{crit[1...j] \to f[k...n_f]}$.

Au moins une chaîne du système est périodique - Preuve de B : Toutes les chaînes de priorité inférieure à π_a sauf σ_{crit} peuvent être activées à l'ouverture de la σ_a-*busy-window* et ensuite au plus tôt compte tenu de leur modèle d'activation. Dans ce cas, le segment de tête de chacune de ces chaînes est composé de tâches de priorité inférieure à π_a. Comme $\pi_a < \pi^{\ell t_a}_a$, alors les segments de tête de chacune de ces chaînes termineront leur exécution avant la terminaison du premier job de $\tau^{\ell t_a}_a$. Ces chaînes seront ensuite suspendues jusqu'à la fermeture de la σ_a-*busy-window*. L'interférence de ces chaînes est donc calculée via l'équation (3.6). On a également montré qu'il existe un scénario d'exécution possible pour σ_{crit} tel que la tâche τ^j_{crit} soit activée à l'instant t_1. Pour rappel, τ^j_{crit} est la première tâche du segment critique de σ_{crit} sur σ_a. Ce segment terminera son

exécution avant la terminaison du premier job de $\tau_a^{\ell t_a}$ avant d'être suspendu jusqu'à la fermeture de la σ_a-*busy-window*. □

Ainsi, si les échéances sont contraintes et si le segment qui maximise l'équation (3.6) n'est pas circulaire alors il est possible d'exhiber un scénario d'exécution qui réalise le majorant sur la latence pire cas et donc le calcul du majorant su la latence au pire est exact.

Considérons maintenant que le segment qui maximise l'équation (3.6) est circulaire.

Cas I.B : Le segment critique qui maximise l'équation (3.6) **est circulaire**

Si le segment critique est circulaire, c'est à dire composé du segment de queue d'un job de σ_{crit} et du segment de tête du job suivant, trouver un scénario réalisable est plus complexe. En effet, il y a une condition supplémentaire pour que le segment circulaire s'exécute entièrement avant la fermeture de la σ_a-*busy-window* : le second job de σ_{crit} doit être activé avant la terminaison de $\tau_a^{\ell t_a(crit)}$. En effet, $\tau_a^{\ell t_a(crit)}$ est la dernière tâche de σ_a dont la priorité est inférieure à toutes les tâches des segments de σ_{crit}. Si le segment de queue de σ_{crit} est activé après la terminaison de $\tau_a^{\ell t_a(crit)}$ alors ce segment pourra n'interférera pas entièrement.

Pour le calcul du majorant sur les interférences des chaînes moins prioritaires, nous considèrons que le segment critique de la chaîne σ_{crit} interfère à hauteur de son WCET avec σ_a (qu'il soit circulaire ou pas). Or, l'interférence d'un segment critique circulaire de σ_{crit} sur σ_a dépend du modèle d'activation de σ_{crit}. En effet, si $b_a^{\ell t_a(crit)}(q) > \delta_{crit}^-(2)$ alors le segment circulaire interfère entièrement avec le *busy-time* du $q^{\text{ème}}$ job de σ_a. Nous utilisons l'exemple suivant pour illustrer notre propos.

Exemple 85. *Trois modèles d'activation différents sont proposés pour σ_{crit} dans le tableau 4.6. Selon le modèle choisi, et en considérant que le segment critique circulaire $[\tau_a^4, \tau_a^1, \tau_a^2]$ est activé à l'ouverture de la σ_a-busy-window, on constate une variation de l'interférence du segment de tête du job suivant (à savoir $[\tau_a^1, \tau_a^2]$). Nous distinguons trois cas selon l'interférence du segment de tête :*

chaîne	activation ($\delta_a^-(2)$)	tâche	priorité	WCET
σ_d	6.5	τ_d^1	1	0.5
σ_a	20	τ_a^1	2	1.5
		τ_a^2	7	2
		τ_a^3	4	1
σ_{crit}	6.5 (figure 4.7) 9 (figure 4.8) 8 (figures 4.9 et 4.10)	τ_{crit}^1	3	1
		τ_{crit}^2	5	1
		τ_{crit}^3	8	1
		τ_{crit}^4	6	1

FIGURE 4.6 – Caractéristiques des tâches

Compte tenu de l'exemple de la figure 4.6 et selon le modèle d'activation de σ_{crit}, l'interférence du segment circulaire de σ_{crit} sur σ_a change. En considérant le scénario dans lequel σ_d, σ_a et le segment de queue de σ_{crit} sont activés à l'ouverture de la σ_a-busy-window et ensuite au plus tôt compte tenu de leur modèle d'activation, alors nous distinguons 3 cas :

1. ***Le segment de tête de σ_{crit} interfère entièrement avec σ_a.*** *Comme sur la figure 4.7, si $\sum_{j\in[1,3]} C^j_{crit} + B^2_a(1) > \delta^-_{crit}(2)$ alors l'interférence de σ_{crit} sur σ_a est $C^4_{crit} + C^1_{crit} + C^2_{crit}$.*

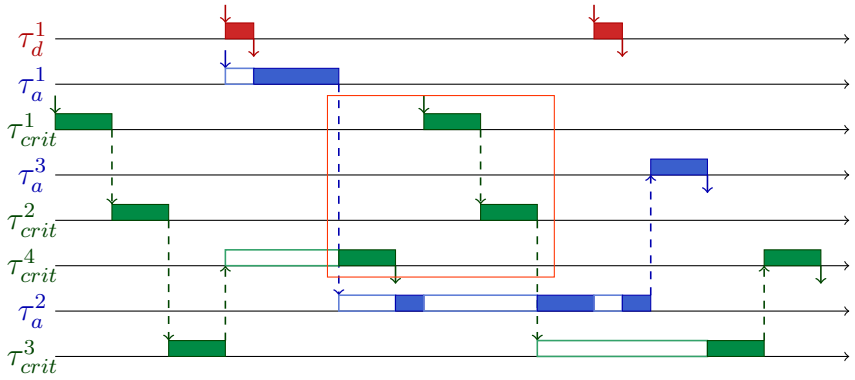

FIGURE 4.7 – Interférence complète de la partie tête du segment circulaire : $C^4_{crit} + C^1_{crit} + C^2_{crit}$

2. ***Le segment de tête de σ_{crit} n'interfère pas avec σ_a.*** *Comme sur la figure 4.8, si $\sum_{j\in[1,3]} C^j_{crit} + B_a(1) < \delta^-_{crit}(2)$ alors, le segment de tête du job suivant ne peut pas interférer avec σ_a et l'interférence subit par σ_a de la part de σ_{crit} est C^4_{crit}.*

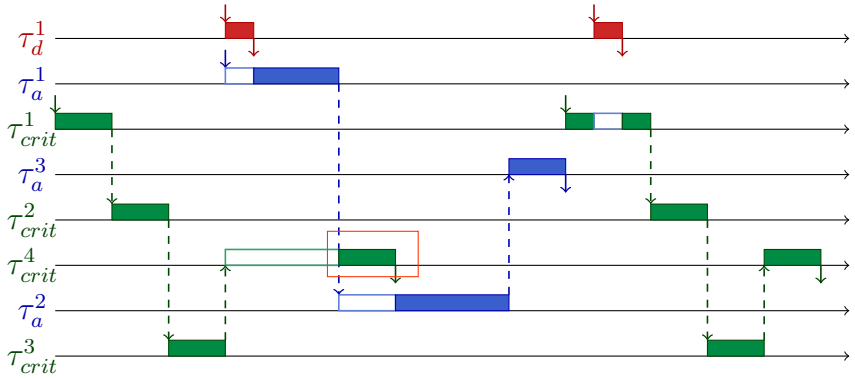

FIGURE 4.8 – Interférence nulle de la partie tête du segment circulaire : C^4_{crit}

3. ***Le segment de tête de σ_{crit} interfère partiellement avec σ_a.*** *Comme sur la figure 4.9, si $\sum_{j\in[1,3]} C^j_{crit} + B^2_a(1) < \delta^-_{crit}(2)$ et $\sum_{j\in[1,3]} C^j_{crit} + B_a(1) > \delta^-_{crit}(2)$ alors, le segment de tête du job suivant n'interfère que partiellement avec σ_a et l'interférence de σ_{crit} sur σ_a est $C^4_{crit} + C^1_{crit}$.*

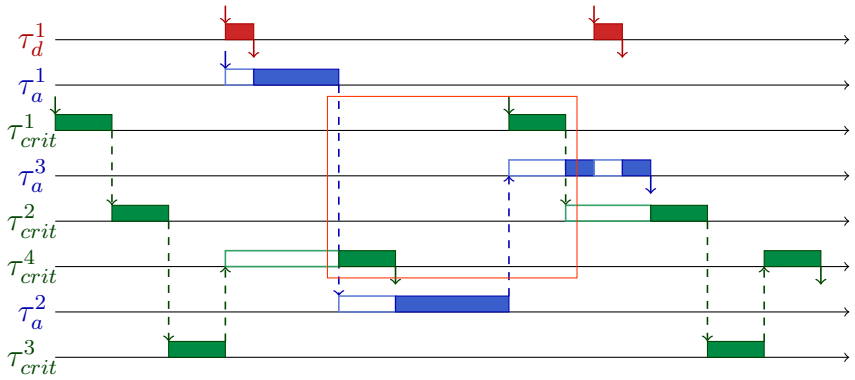

FIGURE 4.9 – Interférence partielle de la partie tête du segment circulaire : $C_{crit}^4 + C_{crit}^1$

De manière générale, comme nous l'avons vu dans la preuve du théorème 84, si le système est ordonnançable et les échéances sont contraintes alors il existe un scénario d'exécution des chaînes du système tel que le segment de queue de σ_{crit} s'active à l'ouverture de la σ_a-*busy-window* et le job suivant de σ_{crit} s'active $\delta_{crit}^-(2)$ après le premier. Pour rappel, la dernière tâche de σ_a qui a une priorité inférieure à toutes les tâches du segment de tête de σ_{crit} sur σ_a est $\tau_a^{\ell t_a(crit)}$ (τ_a^2 sur l'exemple 85). Ainsi, si le second job de σ_{crit} s'active avant la terminaison de $\tau_a^{\ell t_a(crit)}$ alors le segment circulaire interfère à hauteur de son WCET avec σ_a. Sinon l'interférence du segment de tête du job suivant est soit partielle soit nulle. Il est à noter que même si le second job de σ_{crit} est activé avant la terminaison de σ_a, il n'interfère pas nécessairement car selon la tâche de σ_a préemptée par σ_{crit}, le WCET du segment de tête de σ_{crit} sur σ_a peut être nul.

Le problème vient du fait que le segment critique est circulaire, et un minorant peut être trouvé en utilisant le plus long segment qui n'est pas circulaire.

Définition 86 (Segment critique non circulaire $s_{b\to a}^{nocirc}$)**.** *Le segment critique non circulaire de σ_b par rapport à σ_a, est le segment non circulaire de σ_b par rapport à σ_a qui a le plus long WCET.*

Ce segment peut ensuite être utilisé pour trouver la combinaison des chaînes moins prioritaires qui maximise les interférences sur σ_a.

Théorème 87. *Lorsque le segment critique est circulaire, l'interférence des chaînes moins prioritaires sur σ_a est minorée par :*

$$\underline{\ell p \mathcal{I}_a}(1) = \max_{\sigma_b \in \ell p(a)} \left\{ \mathbf{C}_{\mathbf{s}_{\mathbf{b\to a}}^{nocirc}} + \sum_{\sigma_c \in \ell p(a) \wedge c \neq b} C_{s_{c\to a}^{head}} \right\} \tag{4.8}$$

Les différences avec l'équation 3.6 sont représentées en rouge.

Démonstration. Étant donné que nous ne considérons pas les segments critiques qui sont circulaires, alors le raisonnement utilisé dans la preuve du théorème 84 est toujours valable. □

Les expériences réalisées montrent qu'en général la différence relative entre le minorant $\underline{\ell p \mathcal{I}_a}(1)$ et le majorant $\overline{\ell p \mathcal{I}_a}$ est faible. Dans le cas où les échénaces sont contraintes et

le segment qui maximise les interférences des chaînes moins prioritaires est circulaire, c'est ce qui explique que l'écart entre le minorant et le majorant soit inférieur à 10% pour environ 90% des chaînes.

Algorithme 88 Calcul d'un minorant sur l'interférence des chaînes de $\ell p(a)$ sur $B_a^i(q)$ en utilisant l'équation (4.8)

```
1:  function COMPUTE_LB_LP_INTERFERENCE_SPORADIC(σ_a)
2:      interference_LB_LP_sporadic = 0
3:      for σ_b ∈ ℓp(a) do
4:          interf = COMPUTE_WCET_CRSEG_NOTCIRC(σ_a, σ_b)
5:          for σ_c ∈ ℓp(a) do
6:              if σ_c ≠ σ_b then
7:                  interf = interf + COMPUTE_WCET_HEAD(σ_a, σ_c)
8:              end if
9:          end for
10:         if interf > interference_LB_LP_sporadic then
11:             interference_LB_LP_sporadic = interf
12:         end if
13:     end for
14:     return interference_UB_LP
15: end function
```

En utilisant le scénario d'exécution défini dans la propriété 82 (réutilisé pour l'exemple 85), il est possible de calculer l'interférence exacte du segment circulaire de σ_{crit} sur σ_a. Cependant, calculer, compte tenu de ce scénario d'exécution, l'interférence exacte de σ_{crit} sur σ_a ne permet pas nécessairement de calculer l'interférence maximum sur σ_a. En effet, le scénario défini dans la propriété 82 n'est pas nécessairement compatible avec l'interférence maximum subi par σ_a. Illustrons ce problème via l'exemple suivant :

Exemple 89. *Considérons l'exemple de la figure 4.6 avec $\delta^-_{crit}(2) = 8$. Une exécution de ce système est présentée figure 4.9. Sur cette figure, la chaîne σ_d de priorité supérieure à σ_a est activée à l'ouverture de la σ_a-busy-window et ensuite le plus tôt possible. Compte tenu de ce scénario, pour maximiser l'interférence de σ_{crit}, son segment de queue doit être activé à l'ouverture de la σ_a-busy-window. De plus, il est possible de calculer l'interférence exacte du segment critique circulaire de σ_{crit} sur σ_a. Les interférences sont les suivantes : l'interférence de $\sigma_d \in hp(a)$ est de $2 \times C_d^1 = 1$, celle de $\sigma_{crit} \in \ell p(a)$ est de $C_{crit}^4 + C_{crit}^1 = 2$ et la latence de σ_a est de $C_a + 2 \times C_d^1 + C_{crit}^4 + C_{crit}^1 = 7.5$. En réalité, ce scénario d'activation ne maximise pas nécessairement les interférences sur σ_a.*

Un autre scénario d'activation, non conforme aux hypothèses de la propriété 82, est présenté sur la figure 4.10 et la latence obtenue via ce scénario est supérieure à celle obtenue avec le scénario précédent. La latence maximum présentée sur la figure 4.10 est obtenue en n'alignant pas les activations de σ_d avec l'ouverture de la σ_a-busy-window (c'est à dire l'activation de σ_a) et par conséquent, le scénario d'activation ne correspond pas à celui présenté dans la propriété 82. Décaler les activations de σ_d comme sur la figure 4.10 implique qu'une seule activation de σ_d interfère avec σ_a au lieu de deux, cependant cela permet à σ_{crit} d'être réactivée plus tôt dans la σ_a-busy-window et donc le segment critique circulaire interfère entièrement avec σ_a. Sur ce scénario d'exécution :

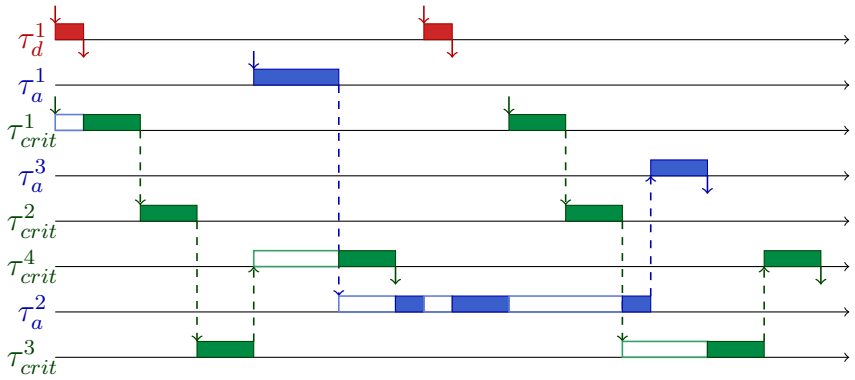

FIGURE 4.10 – Scénario alternatif à celui utilisé pour le calcul du majorant

l'interférence de $\sigma_d \in hp(a)$ *est de* 0.5*, celle de* $\sigma_{crit} \in \ell p(a)$ *est de* 3 *et la latence de* σ_a *est de* 8*.*

Dans l'exemple 89, si le segment qui maximise l'équation (3.6) est circulaire alors le scénario qui maximise les interférences sur σ_a n'est pas nécessairement compatible avec celui fixé dans la propriété 82.

Voyons maintenant comment calculer des minorants sur la latence au pire cas si les échéances sont arbitraires.

4.3.2 Cas II : Les échéances sont arbitraires

Voici un tableau récapitulatif permettant de lier le calcul chaque minorant au modèle du système pour lequel le minorant est calculé :

<table>
<tr><td>Échéances</td><td colspan="2">les échéances sont containtes</td><td colspan="4">au moins une échéance est arbitraire</td></tr>
<tr><td>Modèles d'activation des chaînes</td><td colspan="2">non pertinent</td><td colspan="3">tous sporadiques</td><td>au moins un est périodique</td></tr>
<tr><td>Modèle d'exécution de σ_{crit}</td><td colspan="2">non pertinent</td><td colspan="2">synchrone</td><td>asynchrone</td><td>non pertinent</td></tr>
<tr><td>le segment critique de σ_{crit} est circulaire</td><td>non</td><td>oui</td><td>non</td><td>oui</td><td>non pertinent</td><td>non pertinent</td></tr>
<tr><td>Équation pour calculer un minorant</td><td>(3.6)</td><td>(4.8)</td><td>(3.12)</td><td>(4.9)</td><td colspan="2">(4.10)</td></tr>
</table>

FIGURE 4.11 – Équations permettant de calculer un minorant sur les interférences des chaînes moins prioritaires selon le modèle du système

Il est à noter que même si dans la figure 4.11 les équations diffèrent, certains minorants sont calculés en utilisant des arguments similaires et ainsi :

— Les minorants obtenus via les équations (3.6) et (3.12) sont égaux aux majorants.

— Les minorants obtenus via les équations (4.8) et (4.9) sont obtenus en considérant le WCET du plus long segment non circulaire.

Dans la section 4.3.1, nous avons calculé des minorants sur l'interférence des chaînes moins prioritaires dans le cas où les échéances sont contraintes et, dans cette section, nous présentons le calcul des minorants sur l'interférence des chaînes moins prioritaires dans le cas où au moins une échéance d'une chaîne est arbitraire.

Si les échéances sont arbitraires alors un majorant sur l'interférence des chaînes moins prioritaires est calculée via l'équation (3.12) qui, pour rappel, est :

$$\overline{\ell p \mathcal{I}_a^{i,q}}(\Delta) = \max_{\sigma_b \in \ell p(a)} \left\{ C_{s^*_{b\to a}} + \sum_{\sigma_c \in \ell p(a) \cap \mathcal{SC} \wedge c \neq b} C_{s^{head}_{c\to a}} + \sum_{\sigma_c \in \ell p(a) \cap \mathcal{AC}} hp\mathcal{I}^{i,q}_{\substack{s^{head}_{c\to a} \to a \\ async}}(\Delta) \right\}$$

où

$$s^*_{b\to a} = \begin{cases} s^{crit}_{b\to a} & \text{si le segment critique n'est pas circulaire} \\ s^{crit}_{b\to a} & \text{si le segment critique est circulaire et } \sigma_b \in \mathcal{SC} \\ s^{tail}_{b\to a} & \text{sinon} \end{cases}$$

et où Δ est la longueur d'un préfixe d'une σ_a-*busy-window*.

Pour calculer des minorants sur l'interférence des chaînes moins prioritaires, nous différencions les systèmes selon les modèles d'activation des chaînes du système. Considérons d'abord que toutes les chaînes du système sont sporadiques.

Cas II.A : Toutes les chaînes sont sporadiques

Le calcul du minorant change selon le modèle d'exécution de la chaîne dont le segment critique maximise l'équation (3.12). Considérons que la chaîne dont le segment critique maximise l'équation (3.12) est synchrone.

Cas II.A.1 : La chaîne dont le segment critique maximise l'équation (3.12) est synchrone

Si les chaînes sont sporadiques et la chaîne dont le segment critique maximise l'équation (3.12) est synchrone, alors notre approche pour minorer l'interférence des chaînes moins prioritaires est similaire à celle utilisée dans le cas où les échéances sont contraintes. Commençons par considérer que le segment critique qui maximise l'équation (3.12) n'est pas circulaire.

Théorème 90. *Supposons que toutes les chaînes sont sporadiques, que la chaîne dont le segment critique maximise l'équation* (3.12) *est synchrone et que ce segment n'est pas circulaire. Soit une σ_a-busy-window $[t_1, t_2]$ construite en accord avec les hypothèses 78, 80 et 81 alors, l'équation* (3.12) *permet de calculer un minorant sur l'interférence maximum des chaînes moins prioritaires que σ_a sur q événements de la tâche τ_a^i.*

Démonstration. Comme toutes les chaînes sont sporadiques et que le segment qui maximise l'interférence des chaînes moins prioritaire n'est pas circulaire alors de la même

manière que pour le théorème 84 il est possible de créer un scénario d'exécution tel que le segment critique de σ_{crit} et les autres chaînes de $\ell p(a)$ sont activés simultanément à t_1. σ_{crit} et les autres chaînes synchrones de $\ell p(a)$ interfèrent à hauteur de exactement un segment. L'interférence des chaînes asynchrones de $\ell p(a)$ (qui sont activées à t_1 et ensuite au plus tôt selon leur modèle d'activation) est exacte. En effet cette interférence est calculée en utilisant l'équation (3.10) qui permet de calculer l'interférence des chaînes plus prioritaires asynchrones et selon le théorème 84, compte tenu des hypothèses 78, 80 et 81, cette interférence est exacte.

□

Ainsi si le segment critique qui maximise l'équation (3.12) n'est pas circulaire alors un minorant peut être calculé via le théorème 90. Si le segment est circulaire alors un minorant peut être calculé, comme dans le théorème 87, en utilisant le plus long segment qui n'est pas circulaire.

Théorème 91. *Supposons que toutes les chaînes sont sporadiques, que la chaîne σ_b dont le segment critique maximise l'équation* (3.12) *est synchrone et que ce segment est circulaire. Soit une σ_a-busy-window $[t_1, t_2]$ construite en accord avec les hypothèses 78, 80 et 81 et soit Δ la longueur d'un préfixe de la σ_a-busy-window, alors l'interférence au pire cas des chaînes moins prioritaires que σ_a sur q événements de la tâche τ_a^i est minorée par :*

$$\overline{\ell p \mathcal{I}_a^{i,q}}(\Delta) = \max_{\sigma_b \in \ell p(a)} \left\{ \mathbf{C}_{\mathbf{s}_{\mathbf{b}\to\mathbf{a}}^{nocirc}} + \sum_{\sigma_c \in \ell p(a) \cap \mathcal{SC} \wedge c \neq b} C_{s_{c\to a}^{head}} + \sum_{\sigma_c \in \ell p(a) \cap \mathcal{AC}} hp\mathcal{I}^{i,q}_{\substack{s_{c\to a}^{head} \to a \\ async}}(\Delta) \right\} \quad (4.9)$$

Démonstration. Pour calculer ce minorant, nous considérons le plus long segment critique qui n'est pas circulaire. Ainsi, le raisonnement utilisé dans la preuve du théorème 90 est toujours valable. □

Nous avons traité le cas où la chaîne dont le segment critique maximise l'équation (3.12) est synchrone, passons maintenant au cas où la chaîne est asynchrone.

Cas II.A.2 : La chaîne dont le segment critique maximise l'équation (3.12) est asynchrone

Si la chaîne dont le segment critique maximise l'équation (3.12) est asynchrone alors, peu importe la nature du segment critique (circulaire ou non), l'interférence calculée via l'équation (3.12) peut être supérieure à l'interférence maximum. Plus précisément, le calcul de l'interférence de la chaîne σ_{crit} qui interfère à hauteur de son segment critique peut être sur-approximé.

Voyons d'où peuvent provenir ces approximations. Soit σ_{crit} la chaîne asynchrone dont le segment critique maximise l'équation (3.12), son interférence sur σ_a est de :

$$C_{s^*_{crit\to a}} + hp\mathcal{I}^{i,q}_{\substack{s_{crit\to a}^{head} \to a \\ async}}(\Delta)$$

avec

$$s^*_{crit\to a} = \begin{cases} s^{crit}_{crit\to a} & \text{si le segment critique n'est pas circulaire} \\ s^{tail}_{crit\to a} & \text{sinon} \end{cases}$$

Le terme $hp\mathcal{I}^{i,q}_{\substack{s^{head}_{crit\to a}\to a\\ async}}(\Delta)$ implique que $\eta^+_{crit}(\Delta)$ jobs de σ_{crit} peuvent suspendre l'exécution de σ_a dans un préfixe de longueur Δ d'une σ_a-*busy-window*. Par définition, $\eta^+_{crit}(\Delta)$ est le nombre maximum de jobs de σ_{crit} activés dans un préfixe de longueur Δ et le scénario d'exécution qui permet d'atteindre cette quantité d'interférence nécessite d'activer σ_{crit} à l'ouverture de la σ_a-*busy-window* et ensuite le plus tôt possible. Pour que l'interférence calculée par $C_{s^*_{crit\to a}}$ se réalise, il faut que la tâche précédent le segment critique se termine à l'ouverture de la σ_a-*busy-window* ce qui implique qu'un job de σ_{crit} doit être activé avant l'ouverture de cette σ_a-*busy-window*. Nous n'avons aucune garantie que l'activation de σ_{crit} qui interfère à hauteur de $C_{s^*_{crit\to a}}$ soit compatible avec les activations de σ_{crit} qui interfèrent à hauteur de $hp\mathcal{I}^{i,q}_{\substack{s^{head}_{crit\to a}\to a\\ async}}(\Delta)$.

Ainsi, dans le cas où la chaîne qui maximise l'équation (3.12) est asynchrone alors le calcul du minorant n'est pas nécessairement compatible avec le scénario présenté à la propriété 82.

Théorème 92. *Supposons que toutes les chaînes sont sporadiques et que la chaîne dont le segment critique maximise l'équation* (3.12) *est asynchrone. Soit une σ_a-busy-window $[t_1, t_2]$ construite en accord avec les hypothèses 78, 80 et 81, et Δ la longueur d'un préfixe d'une σ_a-busy-window, alors un minorant sur les interférences des chaînes moins prioritaires sur σ_a est :*

$$\overline{\ell p\mathcal{I}^{i,q}_a}(\Delta) = \sum_{\sigma_c \in \ell p(a) \cap \mathcal{SC}} C_{s^{head}_{c\to a}} + \sum_{\sigma_c \in \ell p(a) \cap \mathcal{AC}} hp\mathcal{I}^{i,q}_{\substack{s^{head}_{c\to a}\to a\\ async}}(\Delta) \tag{4.10}$$

Démonstration. Dans l'équation 4.10 le segment critique est ignoré et toutes les chaînes sont activées à l'ouverture de la σ_a-*busy-window*. Comme toutes les chaînes sont sporadiques, il est possible d'activer toutes les chaînes à l'ouverture de la σ_a-*busy-window*. Comme présenté dans l'équation 4.10, l'interférence de chaque chaîne est calculée en utilisant le scénario similaire à celui des chaînes plus prioritaires et il a été prouvé (propriété 82) que ce scénario est réalisable. □

Il est important de noter que le cas où toutes les chaînes sont sporadiques est plus simple car, selon le modèle défini en section 2.3, les fonctions $\delta^+_a()$ et $\eta^-_a()$ ne sont pas définies pour une chaîne sporadique σ_a. Ces fonctions ne sont pas définies dans le cas des chaînes sporadiques pour être cohérent avec les informations disponibles dans un cadre industriel. Cependant si elles sont définies, alors les minorants fournis dans le cadre des chaînes sporadiques ne sont plus valables. Autrement dit, les analyses développées en section 4.3.1, dans le cas où les chaînes d'un système sont sporadiques, sont valides tant qu'il n'y a pas de distance minimum entre deux activations.

Cas II.B : Au moins une chaîne est périodique

Si au moins une chaîne est périodique alors il faut tenir compte des jobs des chaînes périodiques qui sont activés avant l'ouverture de la σ_a-*busy-window*. Cela implique que, même si la chaîne dont le segment critique maximise l'équation (3.12) est synchrone, l'interférence calculée via l'équation (3.12) peut être sur-appoximée. Commençons par comprendre sur un exemple le problème que pose le calcul de l'interférence d'une chaîne périodique synchrone.

Exemple 93. *Soit un système avec trois chaînes synchrones σ_a, σ_b et σ_c telles que $\pi_c < \pi_b < \pi_a$ et considérons que σ_c est périodique. On souhaite calculer la latence de σ_a. Le scénario présenté sur la figure 4.12 est obtenu en ignorant les jobs de σ_c activés avant l'ouverture de la σ_a-busy-window qui s'ouvre avec l'activation de τ_a^1. Sur ce job de σ_a, la combinaison qui maximise l'interférence des chaînes moins prioritaires est lorsque le segment de tête de σ_c et le segment critique de σ_d interfèrent avec σ_a. Ce scénario correspond à celui du calcul du majorant.*

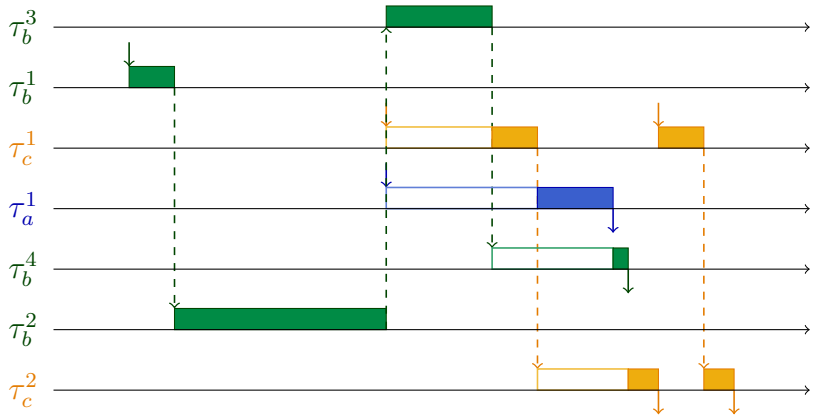

FIGURE 4.12 – Interférence sur σ_a du segment critique de σ_b et du segment de tête de σ_c

Sur la figure 4.13, on tient compte des jobs de σ_c qui sont activés avant l'ouverture de la σ_a-busy-window. Remarquons qu'aucun scénario d'activations ne permet de réaliser le majorant sur l'interférence des chaînes moins prioritaires. En effet, comme σ_c est synchrone et périodique alors l'activation qui se produit avant la σ_a-busy-window empêche σ_c d'interférer avec σ_a.

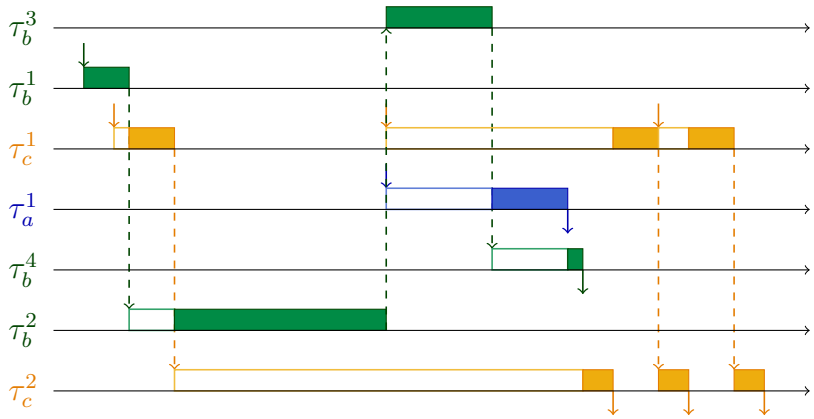

FIGURE 4.13 – Latence si une chaîne est périodique (sous-approximation)

Ainsi, si une chaîne du système est périodique et synchrone alors, qu'elle exécute son segment critique ou non, on ne peut plus garantir que l'interférence calculée via l'équation (3.12) est atteinte. Il est cependant possible de calculer un minorant en considérant un scénario dans lequel toutes les chaînes sont activées en même temps à l'ouverture de la σ_a-*busy-window*. Le minorant est donc égal à celui calculé au théorème 92.

Théorème 94. *Soit une σ_a-busy-window $[t_1, t_2]$ construite en accord avec les hypothèses 78, 80 et 81. Si au moins une chaîne est périodique, l'interférence des chaînes moins prioritaires sur σ_a est minorée par l'équation* (4.10)*.*

Démonstration. Soit une *σ_a-busy-window* s'ouvrant à l'instant t_1. Toutes les chaînes de $hpe(a)$ sont activées à l'instant t_1 et ensuite le plus tôt possible de manière à maximiser les interférences sur la *σ_a-busy-window*. De la même manière les chaînes de $\ell p(a)$ sont toutes activées à t_1 et ensuite le plus tôt possible. Si juste avant t_1, la ressource n'est pas libre alors, cela signifie qu'elle est chargée à plus de 100%. Sinon il est possible d'activer toutes les chaînes du système à l'instant t_1 et l'interférence des chaînes moins prioritaire peut être calculée par l'équation (4.10) □

Nous avons maintenant calculé des minorants sur l'interférence des chaînes moins prioritaires dans le cas général. Ces minorants nous permettent de calculer, via le corollaire 83, des minorants sur la latence au pire cas des chaînes.

Dans la suite de ce chapitre, nous utilisons le calcul des minorants sur la latence au pire cas pour évaluer expérimentalement la précision des majorants calculés au chapitre 3.

4.4 Evaluation expérimentale

4.4.1 Évaluation de la précision des majorants

Dans cette section, on évalue, par expérimentations, la précision du majorant sur la latence au pire cas. Pour ce faire, pour plusieurs systèmes, on calcule un majorant et un minorant sur la latence au pire cas et, on calcule l'écart entre les deux bornes. Plus cet écart est grand plus le calcul du minorant (resp. majorant) est sous-approximé (resp. surapproximé).

On utilise les systèmes temps-réel décris dans la section 3.7.1. De la même manière que pour les expérimentations concernant le majorant sur la latence au pire cas, les chaînes sont ordonnées selon la valeur de leur majorant sur la latence au pire cas.

Nous avons vu que, si toutes les chaînes d'un système sont sporadiques et si le segment critique qui maximise l'interférence des chaînes moins prioritaires sur σ_a n'est pas circulaire, alors le majorant sur la latence de σ_a est exact. Il est donc possible d'exhiber un scénario qui le réalise. Cependant, si le segment critique qui maximise l'interférence des chaînes moins prioritaires est circulaire alors le calcul du majorant sur la latence peut comporter des sur-approximations. Parmi les 5581 chaînes de la première série de systèmes, dans lesquels les chaînes sont toutes sporadiques, l'interférence des chaînes moins prioritaires de 2023 chaînes est maximisée par un segment critique circulaire. Ainsi, le majorant sur la latence de ces chaînes est sur-approximé et, l'écart entre le minorant et le majorant est représenté dans la figure 4.14.

Nous remarquons, sur la figure 4.14, que l'écart entre les deux bornes est faible. Cet écart est dû au fait que (i) pour le calcul du majorant nous utilisons le segment critique circulaire et (ii) pour le calcul du minorant nous utilisons le plus long segment non circulaire. L'écart entre les deux bornes est une conséquence de la différence entre les WCET de ces deux segments.

Sur l'ensemble des 2023 chaînes dont l'interférence des chaînes moins prioritaires est maximisée par un segment critique circulaire, nous calculons l'écart (en pourcentage) entre

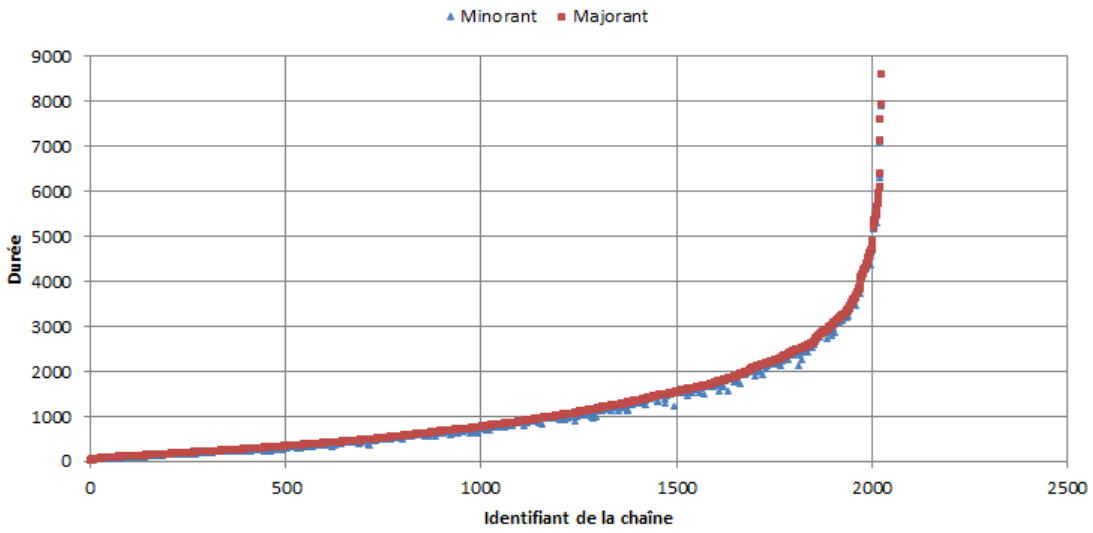

FIGURE 4.14 – Évaluation de la précision du majorant sur la latence au pire cas quand les chaînes sont sporadiques et synchrones. En abscisse les identifiants des chaînes ordonnées selon la valeur du majorant calculé dans le chapitre 3.

notre minorant dénoté $\underline{L_a}$ et notre majorant dénoté $\overline{L_a}$ en utilisant la formule :

$$\frac{\overline{L_a} - \underline{L_a}}{\overline{L_a}} \times 100 \tag{4.11}$$

Les résultats sont synthétisés dans la figure 4.15.

Même si un segment circulaire interfère, dans plus de 90% des cas, l'écart entre le minorant et le majorant sur la latence au pire cas de la chaîne est inférieur à 10%. Cela signifie que, dans la majorité des cas, nous pouvons évaluer precisément la valeur de la latence au pire cas des chaînes même si un segment circulaire interfère avec ces chaînes.

Dans le cas de systèmes contenant des chaînes périodiques et sporadiques, et dans lequel les échéances sont contraintes, nous aurions des résultats similaires. En effet comme nous l'avons présenté précédemment, la précision de l'analyse dans le cas d'échéances contraintes est similaire à celle obtenue dans le cas de chaînes sporadiques, synchrones avec des échéances arbitraires.

Nous avons également testé les systèmes dans le cas où ces derniers comportent des chaînes sporadiques et périodiques, et où ces chaînes sont soit synchrones soit asynchrones. La figure 4.16 représente le cas où toutes les chaînes sont synchrones tandis que la figure 4.17 représente le cas où les chaînes sont asynchrones.

Les tendances sur les figures 4.16 et 4.17 sont similaires car les scénarios utilisés pour le calcul des minorants sont identiques. En effet, peu importe les modèles d'exécution des chaînes, si les échénaces sont arbitraires et qu'au moins une chaîne est périodique un minorant sur l'interférence des chaînes moins prioritaires est calculé via l'équation (4.10). Cependant, l'écart entre les bornes est plus important dans le cas où une chaîne du système est périodique (figures 4.16 et 4.17) que dans le cas où toutes les chaînes sont sporadiques (figure 4.14).

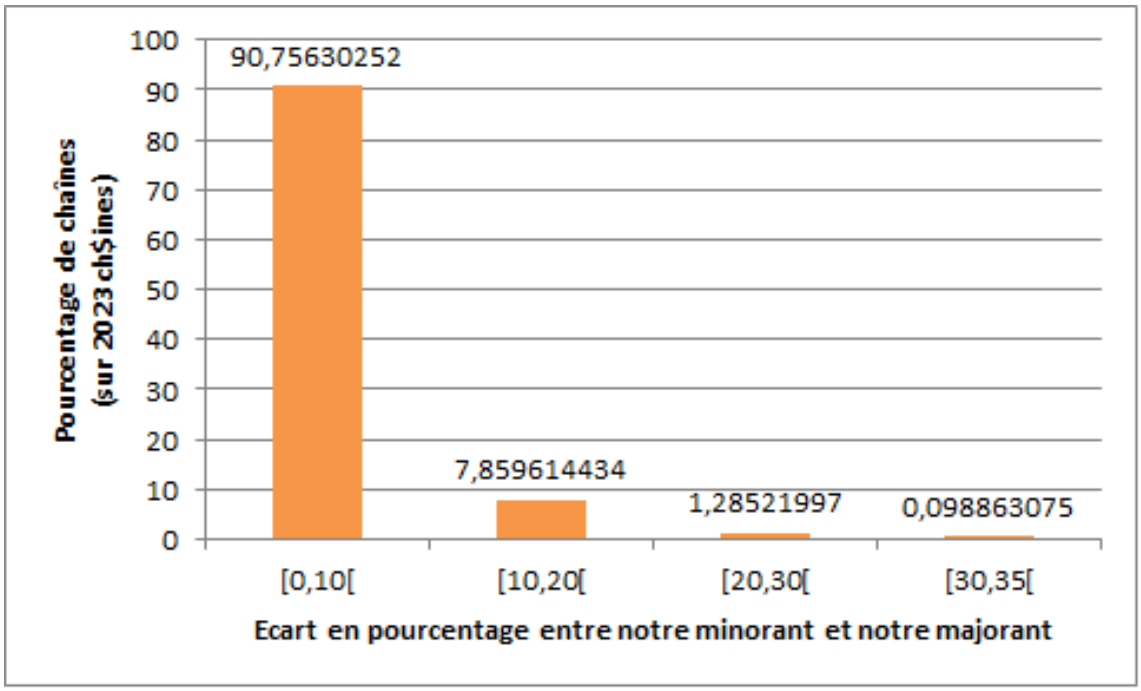

FIGURE 4.15 – Écart en pourcentages entre notre calcul du majorant et le calcul du minorant dans le cas où toutes les chaînes sont sporadiques et l'interférence des chaînes moins prioritaires est maximisée par un segment critique circulaire.

Pour l'ensemble des 5538 chaînes des systèmes contenant des chaînes synchrones, périodiques et sporadiques, nous avons utilisé l'équation (4.11) pour calculer l'écart entre notre majorant et notre minorant.

Dans environ 63% des cas, l'écart entre le minorant et le majorant sur la latence au pire cas de la chaîne est inférieur à 10%. Ainsi l'écart entre les bornes est plus important que dans le cas où toutes les chaînes sont sporadiques. Dans 78% des cas, cet écart est inférieur à 20%. Une précision de 20% est acceptable dans un contexte industriel, ainsi dans le cas général (chaînes sporadiques ou périodiques, et échéances arbitraires) les calculs des majorants et des minorants sont suffisement précis. Concernant les 22% restant il serait utile d'identifier plus précisément les sources de pessimise et/ou de tenter de calculer par simulation un plus grand minorant. Ceci constitue une piste pour des travaux futurs.

4.4.2 Comparaison des minorants calculés et simulés

Nous avons également comparé nos minorants calculés avec des minorants obtenus par simulation. Pour cela, nous avons utilisé l'outil *SimSo*. *SimSo* est un outil qui permet de générer des scénarios d'activations et de les exécuter. Le modèle utilisé par cet outil est le même que celui utilisé pour l'analyse d'ordonnançabilité et, pour simuler un système, il faut spécifier certaines informations.

Pour décrire un système dans *SimSo*, il faut spécifier chacune des tâches qui compose ce système. Pour chaque tâche τ_a^i, il faut spécifier :

— Son nom $name$.

— Son identifiant $identifier$ qui est unique.

— Son modèle d'activation $task\_type$.

— Un booléen $abort\_on\_miss$ pour définir si l'exécution de la tâche doit continuer une fois l'échéance dépassée.

— Un WCET $wcet$.

— Une échéance $deadline$.

— L'ensemble des tâches $followed\_by$ qui sont activées à la terminaison de la tâche courante.

— Une liste d'activations $activation\_dates$ de τ_a^i qui dépend de $task\_type$.

— Une donnée $data$ qui dépend de l'ordonnanceur et qui correspond dans notre cas à la priorité.

Dans notre cas l'ensemble $followed\_by$ contient au maximum une tâche car nous considérons des chaînes.

La variable $task\_type$ permet de définir l'activation de la première tâche de chacune des chaînes et peut être de trois types :

— *Periodic* : Le modèle d'activation est périodique et il faut spécifier, dans $activation\_dates$, la période d'activation ainsi que l'instant de la première activation.

— *Sporadic* : Le modèle d'activation est sporadique et il faut spécifier l'ensemble des instants d'activations dans $activation\_dates$.

— *Aperiodic* : Ce modèle correspond à celui des tâches qui sont activées à la terminaison d'une autre, ce qui permet de modéliser des chaînes.

Il existe différents ordonnanceurs qui sont spécifiés dans le logiciel dont *FP.py* pour un ordonnanceur préemptif à priorités fixes. Dans le cadre des chaînes de tâches cet ordonnanceur permet de modéliser les chaînes asynchrones si les échéances sont arbitraires ou les chaînes avec des échéances contraintes. Pour rappel, si les échéances sont contraintes, la distinction entre les modèles d'exécution synchrones et asynchrones est inutile car, si le système est ordonnançable, un job se termine nécessairement avant que le suivant ne soit activé.

Nous avons modélisé l'exemple figure 2.13 et paramétré le simulateur pour utiliser le WCET de chaque tâche et, ensuite nous avons généré aléatoirement $1.000.000$ scénarios d'activations pour les chaînes de ce système. Nous avons mesuré la plus longue latence de chacune des chaînes parmi toutes les itérations de la simulation, ce qui permet d'obtenir un minorant pour chaque chaîne.

Les minorants obtenus par simulation pour chacune des chaînes du système sont présentés dans le tableau 4.1. On remarque que le minorant obtenu par le calcul est plus proche de la latence au pire cas pour les deux chaînes σ_c et σ_d. L'écart en pourcentage entre les minorants calculés et simulés est obtenu via l'équation suivante :

$$\frac{\text{computed } \underline{L_i} - \text{simulated } \underline{L_i}}{\text{simulated } \underline{L_i}} \times 100$$

Le tableau 4.2 contient les résultats pour un système plus complexe composé de 6 chaînes. De la même manière que pour le système précédent on a généré aléatoirement $1.000.000$ scénarios d'activation. Le minorant obtenu par simulation est parfois identique

chaîne i	$\overline{L_i}$	computed $\underline{L_i}$	simulated $\underline{L_i}$	diff.
σ_a	24	16	16	0%
σ_b	35	34	34	0%
σ_c	42	41	21	95%
σ_d	48	48	28	71%

TABLE 4.1 – Minorants calculés et simulés pour les chaînes du système figure 2.13, sur 1.000.000 scénarios d'activation

chaîne i	$\overline{L_i}$	computed $\underline{L_i}$	simulated $\underline{L_i}$	diff.
σ_a	55	55	55	0%
σ_b	178	178	38	368%
σ_c	124	124	75	65%
σ_d	176	176	122	44%
σ_e	286	286	286	0%
σ_f	37	37	14	164%

TABLE 4.2 – Minorants calculés et simulés pour les chaînes d'un système sur 1.000.000 scénarios d'activation

à celui obtenu par l'analyse. Cependant, dans certains cas l'écart entre les deux minorants est important et le minorant calculé est plus précis.

Par manque de temps, nous n'avons pas testé plus de systèmes. Nous n'avons pas non plus connecté le générateur de systèmes à l'outils *SimSo*. Nous avons donc choisi deux systèmes pour lesquels nous avons évalué notre approche.

Une fois l'analyse développé obtenir un minorant calculé est instantané. On a vu, par expérimentation que dans beaucoup de cas, l'écart entre le minorant et le majorant est faible. Cela signifie que le minorant et le majorant sont proches de la latence au pire cas. Dans ce cas où la précision est satisfaisante, il est inutile d'utiliser l'approche par simulation. Cependant, si la précision est mauvaise alors un minorant obtenu par simulation peut être plus proche du pire cas. Parmi les systèmes que nous avons testé le minorant obtenu par l'analyse était toujours supérieur ou égal à celui obtenu par simulation cependant nous avons testé trop peu de systèmes pour conclure sur la fréquence de ce cas.

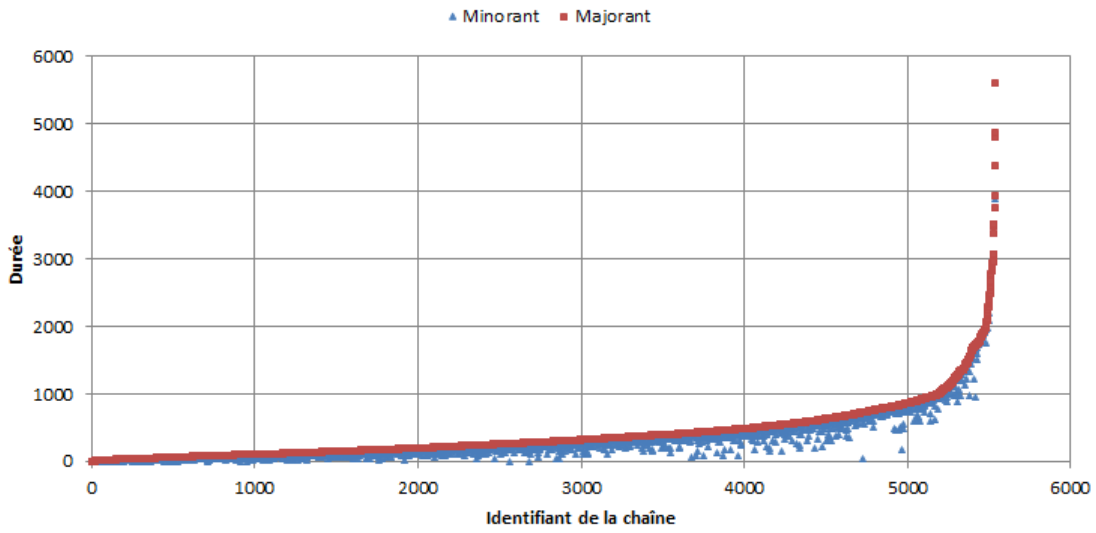

FIGURE 4.16 – Évaluation de la précision du majorant sur la latence au pire cas quand les chaînes sont périodiques ou sporadiques, et synchrones. En abscisse les identifiants des chaînes ordonnées selon la valeur du majorant calculé dans le chapitre 3.

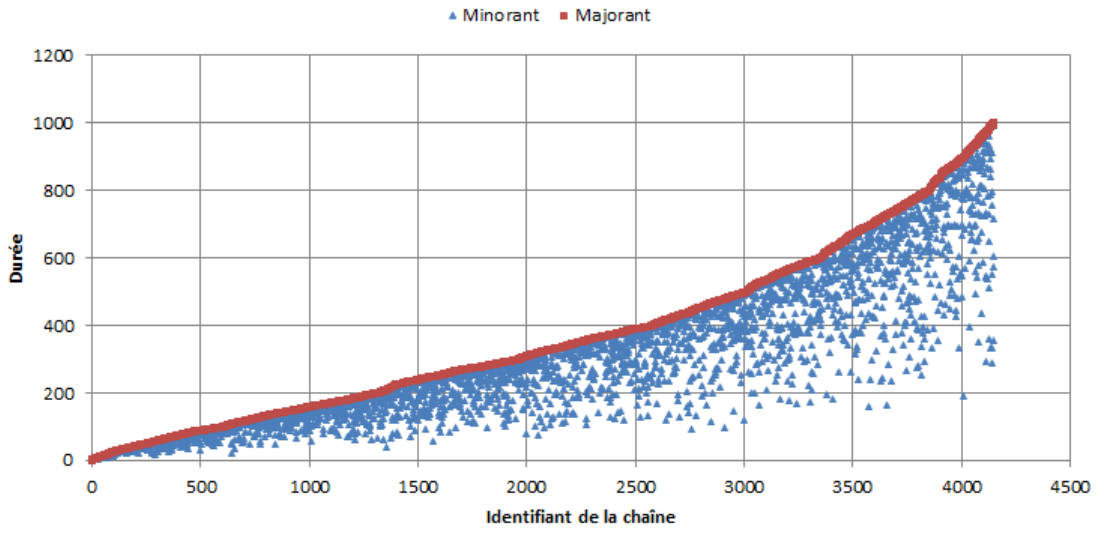

FIGURE 4.17 – Évaluation de la précision du majorant sur la latence au pire cas quand les chaînes sont périodiques ou sporadiques, et asynchrones. En abscisse les identifiants des chaînes ordonnées selon la valeur du majorant calculé dans le chapitre 3.

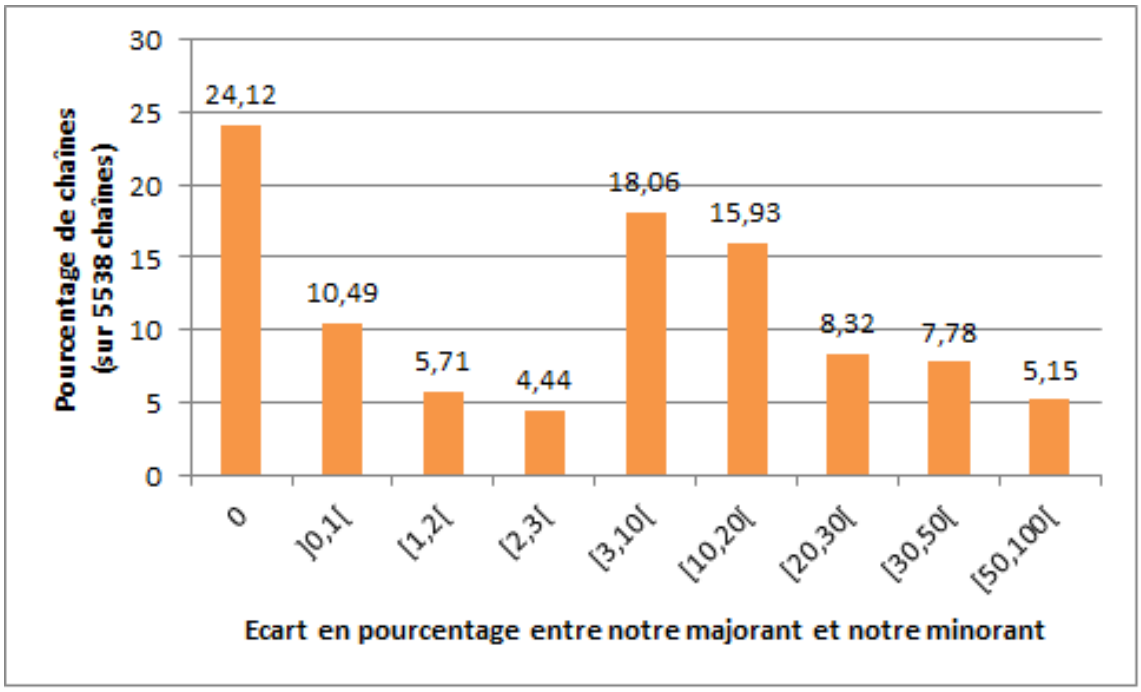

FIGURE 4.18 – Écart en pourcentages entre notre calcul du majorant et le calcul du minorant dans le cas où au moins une chaîne est périodique.

Chapitre 5

Analyse de flexibilité

Sommaire

Nous avons présenté, en section 1.3, les différents problèmes lié à l'évolution des systèmes temps-réels. Pour rappel, un système est amené à évoluer pour plusieurs raisons :

1. Un paramètre temporel d'une chaîne du système, par exemple le WCET d'une tâche ou son modèle d'activation, est **modifié**. En phase de conception les paramètres temporels du système sont évalués pour anticiper le comportement du système mais ne correspondent pas nécessairement à la réalité. Après le développement du système, certains paramètres peuvent être modifiés pour améliorer les performances ou les fonctionnalités du système. Dans l'état de l'art, la modification de paramètres temporels concerne l'analyse de **sensibilité** d'un système ou la **robustesse** face au changement.
2. Une chaîne est **ajoutée** à un système afin d'étendre les fonctionnalités réalisées par le système. Ce cas n'est pas traité dans l'état de l'art et c'est l'objet de l'analyse de la **flexibilité** d'un système.

Dans ce chapitre, nous souhaitons évaluer les possibilités d'évolution d'un système. Pour un système donné, on veut identifier, au plus tôt dans le processus de conception, les évolutions qui pourront être intégrées tout en garantissant l'ordonnançabilité de ce système. Cela implique de pouvoir garantir l'ordonnançabilité d'un système sans refaire l'analyse d'ordonnançabilité à l'intégration de l'évolution. Pour commencer, nous présentons une définition de la notion d'évolution et du problème de flexibilité qui correspondent au problème industriel que nous avons identifié.

5.1 Définition du problème

Dans cette section, en nous basant sur les besoins des concepteurs de systèmes, nous formalisons les problèmes liés à l'évolution des systèmes temps-réels. Basé sur le modèle temporel d'un système et compte tenu des besoin des concepteurs nous commençons par définir la notion d'évolution qui permet d'anticiper l'impact d'un changement sur un système. La définition 95 explicite le problème que nous souhaitons résoudre.

Définition 95 (Évolution)**.** *Soit S un système ordonnançable. Une **évolution** de S est la modification d'au moins un paramètre temporel de S, ou l'addition d'une nouvelle chaîne σ_{new} dans S. Faire évoluer un système signifie :*

- *modifier le modèle d'activation δ_a^- d'une chaîne σ_a, son échéance ou, pour une tâche $\tau_a^i \in S$, son WCET C_a^i ou sa priorité π_a^i.*
- *ajouter une nouvelle chaîne $\sigma_{new} = (\delta_{new}^-, \{(\pi_{new}^i, C_{new}^i)\}_{1 \leq i \leq n_{new}}, D_{new})$ au système S.*

Compte tenu de la définition 95, une évolution peut concerner la modification de paramètres temporels ou l'ajout d'une chaîne de tâches. Comme vu précédemment, nous associons la modification d'un paramètre temporel à la notion de sensibilité/robustesse d'un système, et l'ajout d'une chaîne à la notion de flexibilité. Tandis que les notions de sensibilité et de robustesse sont déjà présentes dans l'état de l'art, la flexibilité est définie spécifiquement pour notre problème.

Définition 96 (Flexibilité)**.** *Soit S un système ordonnançable. La **flexibilité de** S est l'ensemble des valeurs des paramètres temporels d'une nouvelle chaîne σ_{new}, telle que le système résultant $S_{new} = S \cup \{\sigma_{new}\}$ soit ordonnançable.*

Définition 97 (Sensibilité/robustesse)**.** *Soit S un système ordonnançable et x un de ses paramètres temporels. La **sensibilité (ou robustesse) du système** S **par rapport à** x est l'ensemble des valeurs de x tel que S reste ordonnançable.*

Nous verrons plus tard que l'approche que nous proposons dans ce chapitre pour évaluer la flexibilité d'un système permet également d'évaluer la sensibilité/robustesse de ce système.

Commençons par comprendre en quoi consiste l'analyse de flexibilité d'un système sur un exemple.

Exemple 98. *Considérons le cas particulier d'un système S dans lequel nous souhaitons rajouter une chaîne σ_{new} qui est composée d'une unique tâche τ_{new}^1. La chaîne σ_{new} est modélisée via trois paramètres temporels qui sont :*

1. *le modèle d'activation η_{new}^+ (ou de manière équivalente δ_{new}^-)*
2. *le WCET C_{new}*
3. *la priorité π_{new}*

Dans ce cas la flexibilité de S correspond à toutes les combinaisons possibles de $\eta_{new}^+/\delta_{new}^-$, C_{new}, et π_{new} telles que $S_{new} = S \cup \{\sigma_{new}\}$ soit ordonnançable. Pour évaluer la flexibilité de S, il faut identifier les combinaisons de valeurs des trois paramètres $\eta_{new}^+/\delta_{new}^-$, C_{new}, et π_{new} qui mènent à un système ordonnançable. Si les valeurs des deux paramètres $\eta_{new}^+/\delta_{new}^-$

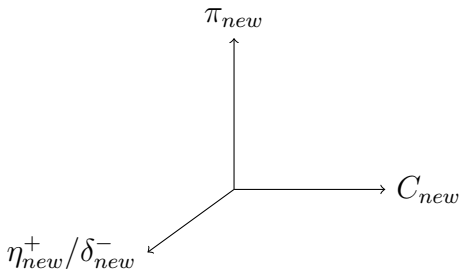

FIGURE 5.1 – Représentation de la flexibilité de S

et π_{new} sont fixées (comme c'est souvant le cas en pratique), alors la flexibilité peut être déterminée en cherchant la plus grande valeur de C_{new} telle que S_{new} soit ordonnançable. Il est à noter que si S_{new} est ordonnançable pour une combinaison de valeurs alors S_{new} sera également ordonnançable si C_{new} diminue ou si $\eta^+_{new}/\delta^-_{new}$ augmente.

Pour une chaîne σ_{new}, les paramètres temporels à considérer sont donc :

1. les BCETs des n_{new} tâches de σ_{new} ;
2. les WCETs C^i_{new} des n_{new} tâches de σ_{new} ;
3. les priorités π^i_{new} des n_{new} tâches de σ_{new} ;
4. le modèle d'activation $\delta^-_{new}/\eta^+_{new}$ de σ_{new} ;
5. l'échéance D_{new} de σ_{new}.

L'analyse de la flexibilité d'un système nécessite d'analyser le système au pire cas pour garantir son ordonnançabilité et ainsi, il est possible d'ignorer les BCETs, en les supposant égaux à 0 au besoin.

Dans le contexte de développement de systèmes à Thales, les paramètres d'une évolution ne sont pas tous inconnus car certains sont restreints à l'avance par les concepteurs. Une nouvelle fonctionnalité (c'est à dire une nouvelle chaîne de tâches) est souvent anticipée en phase de conception de telle sorte que le nombre de tâches dans la chaîne est connu. De plus, comme la priorité d'une tâche dépend de la fonction qu'elle réalise, alors la priorité de chacune des tâches dans la chaîne est également connue. Le modèle d'activation de la chaîne est défini par le concepteur en fonction de son cahier des charges et de la fonctionnalité à réaliser, il peut y en avoir plusieurs mais le nombre de possibilités est faible. Par exemple, pour une chaîne donnée, si un modèle d'activation est fixé à l'avance, il peut être utile de doubler la fréquence d'activation. Ainsi, lors de l'analyse de flexibilité de ce système, il faut prendre en compte seulement deux modèles d'activations.

Il reste à considérer les WCETs des tâches qui sont les principales inconnues. En effet avant la phase de développement d'une fonctionnalité, il est difficile de connaître les WCETs des différentes tâches et pour éviter les pertes de performances après développement, il est nécessaire de déterminer les WCETs maximum des tâches de σ_{new} tels que le système reste ordonnançable.

Par la suite, la flexibilité d'un système est calculée en fonction des WCETs des tâches de σ_{new} et si plusieurs modèles d'activation sont possibles alors les WCETs peuvent être calculés en considérant successivement tous les modèles d'activation possibles.

Dans cette thèse, nous choisissons donc de définir et de calculer la flexibilité par rapport aux C^i_{new}.

Définition 99 (Template d'analyse de flexibilité)**.** *Soit S un système ordonnançable et $\sigma_{new} = (\{(\pi^i_{new}, \_)\}_{1 \leq i \leq n_{new}}, \delta^-_{new}, D_{new})$ une chaîne. L'analyse de flexibilité consiste à déterminer quelles sont les valeurs maximum des WCETs C^i_{new}, pour tout $1 \leq i \leq n_{new}$, telle que le système résultant $S_{new} = S \cup \{\sigma_{new}\}$ soit ordonnançable.*

Le terme de flexibilité a déjà été utilisé dans l'état de l'art et nous précisons ici en quoi notre notion est différente.

5.2 Etat de l'art

Le domaine de recherche le plus proche du nôtre est celui de l'analyse de sensibilité [19, 20, 21] cependant il est également possible d'établir des liens avec d'autres sujets tels que le *slack stealing protocol* ou diverses notions d'évolutions.

L'analyse de sensibilité (ou de robustesse) est utilisée (i) pour donner des garanties sur l'ordonnançabilité d'un système dans le cas où certains paramètres sont susceptibles de changer, et (ii) dans le cas d'un système non ordonnançable, pour trouver les changements de paramètres qui permettent d'avoir un système ordonnançable. Plusieurs méthodes ont été développées afin d'analyser la sensibilité des systèmes. La première méthode est développée dans l'article [22] puis étendue dans [20, 23, 24]. Ces analyses utilisent un modèle de systèmes appelé *Linear Compute Model* qui permet, dans le cadre de ces articles, d'analyser les systèmes mono-processeurs avec des tâches indépendantes. Ces analyses permettent de déterminer la sensibilité d'un système par rapport à un type de paramètre (modèles d'activation, WCET). Les systèmes considérés sont composés de tâches indépendantes et certains concepts, comme les *scheduling points*, sont difficiles à étendre aux chaînes de tâches. Les analyses développées sont également complexes car, compte tenu du modèle des systèmes, le WCET d'une tâche peut changer à chaque job. Considérer cet aspect n'est pas pertinent dans le cadre de notre analyse de flexibilité.

Une autre approche développée dans [25, 26, 27, 28, 19, 29, 30] permet d'analyser la sensibilité de systèmes via des algorithmes dichotomiques. Plusieurs paramètres, dont on souhaite évaluer la sensibilité et qui peuvent être de différents types, sont testés de manière à définir les combinaisons qui mènent à un système ordonnançable tout en optimisant l'utilisation des ressources. Cette approche a l'avantage de permettre l'analyse de sensibilité de plusieurs types de paramètres simultanément. De plus, elle permet de prendre en compte des systèmes complexes. Cependant, l'approche étant dichotomique, pour obtenir la sensibilité d'un système via cette approche, il faut exécuter successivement plusieurs tests d'ordonnançabilité. Ainsi pour obtenir un résultat précis sur un système complexe, cette approche peut être coûteuse en temps. Nos calculs développés au chapitre 3 nous permettent d'utiliser uniquement des calculs de points fixes (et pas d'algorithmes dichotomiques).

L'analyse statique d'ordonnançabilité est utilisée dans l'article [21] pour analyser la sensibilité de systèmes mono-processeur composés de tâches indépendantes. Cette analyse est donc trop simple pour prendre en compte les systèmes qui nous intéressent.

Contrairement au problème d'analyse de sensibilité, notre problème de flexibilité ne concerne pas l'incertitude sur les paramètres temporels mais les parmètres temporelles d'un nouvelle chaîne (cf définitions 97 et 96). Nous verrons dans la section 5.5 le lien entre les analyses de flexibilité et de sensibilité. Dans le cadre de l'analyse de flexibilité et

compte tenu des besoins industriels nous considérons que certains paramètres temporels d'un système sont fixes ce qui diminue la complexité de l'analyse. Il est inutile dans un cadre industriel de modifier l'ensemble d'un système.

Le *slack stealing protocol* est un protocole permettant, dans un système mono-processeur composé de tâches périodiques ordonnancées sous FPP, de garantir l'ordonnançabilité de tâches sporadiques, moins prioritaires. Pour mettre en place ce protocole, il faut caractériser la disponibilité de la ressource pour savoir si les tâches sporadiques peuvent être ordonnancées. Ce protocole est développé dans les articles [31, 32, 33, 34, 35]. Le lien entre cette approche et notre calcul de flexibilité est que, dans les deux cas, pour fournir des garanties il faut caractériser la disponibilité de la ressource. Cependant le *slack stealing protocol* répond à un objectif précis qui est de garantir l'ordonnançabilité de tâches sporadiques de basse priorité et il n'est développé que pour des systèmes avec des tâches indépendantes.

Dans les articles [36, 37], les auteurs analysent la flexibilité de système dans le cadre précis de l'allocation de tâches ou de l'assignement de priorités. Dans [37], les auteurs utilisent des concepts similaires aux nôtres pour définir des assignements de priorités. Dans [36], les auteurs allouent les tâches et assignent les priorités pour maximiser l'*extensibilité* de chaque chaîne. L'extensibilité est définie comme le WCET maximum d'une chaîne tel que cette chaîne reste ordonnançable. Dans l'article [38], les auteurs fournissent une méthodologie pour faire évoluer des systèmes, ils se basent sur des scénarios d'évolutions mais ne développent pas les analyses permettant de garantir l'ordonnançabilité d'un système lors de son évolution.

Dans le cas de certains ordonnanceurs, il n'est pas nécessaire de calculer des marges temporelles. C'est le cas, par exemple, pour l'ordonnanceur *Rate Monotonic* [39]. Pour un système mono-processeur composé de tâches indépendantes ordonnancées sous *Rate Monotonic*, tant que l'utilisation du processeur est au-dessous d'une certaine valeur, il est possible de rajouter des tâches tout en fournissant des garanties sur l'ordonnançabilité du système. Cette approche permet de faire évoluer un système mais le modèle est très restreint. De plus, il n'est pas possible d'approcher les 100% de charge CPU en utilisant cette méthode car, comme présenté dans l'article [39], pour un nombre quelconque de tâches la charge du processeur doit rester inférieure à environ 69% (83% dans le cas particulier de deux tâches).

Dans ce chapitre, nous nous intéressons au problème non résolu de l'analyse de flexibilité et nous montrons que la même approche peut être utilisée pour analyser la sensibilité d'un système. Cette approche repose entièrement sur le calcul d'une marge temporelle appelée *slack* que nous définissons et calculons en section 5.3.

Dans ce chapitre, notre objectif est de calculer la flexibilité d'un système. Pour cela nous devons tout d'abord étendre la notion de *marge temporelle* aux systèmes mono-processeurs avec des chaînes de tâches. Ces marges temporelles permettent ensuite d'évaluer l'interférence qu'il est possible de rajouter à chaque chaîne. Cela nous permet d'établir des contraintes sur les paramètres temporels de σ_{new} de telle sorte que le système reste ordonnançable après évolution. Ces contraintes permettent donc de calculer la flexibilité du système et pour finir, nous montrons que ces contraintes permettent d'analyser la sensibilité du système. Via l'analyse de flexibilité, nous étendons notre analyse statique de sensibilité aux chaînes de tâches.

5.3 La marge temporelle d'une chaîne

La première étape pour calculer la capacité d'un système à évoluer (flexibilité ou sensibilité) est de calculer la marge temporelle de chacune des chaînes de ce système avant de rajouter de la charge.

Comme présenté dans la section 5.1, la latence au pire cas d'une chaîne de tâches peut augmenter suite à l'ajout d'une nouvelle chaîne dans le système ou la modification d'une chaîne existante. Il est nécessaire, pour garantir l'ordonnançabilité d'un système après son évolution, de calculer l'interférence maximum supplémentaire que chaque chaîne peut subir.

L'objectif de cette section est de définir la notion de **marge temporelle** pour une chaîne σ_a et de présenter une méthode pour calculer un minorant sur cette marge. Intuitivement, la marge temporelle d'une chaîne σ_a correspond à l'interférence maximum supplémentaire qui peut suspendre σ_a de telle sorte que cette chaîne reste ordonnançable.

La notion de marge temporelle, déjà présente dans l'état de l'art, doit être adaptée à notre modèle temporel. Nous n'avons pas identifié, dans l'état de l'art, de définition formelle de marge temporelle. La marge temporelle et son calcul sont associés à l'augmentation du WCET de la tâche/chaîne pour laquelle on calcule la marge. Nous étendons le calcul de marge temporelle développé dans l'article **??**.

Nous utilisons le scénario suivant et le calcul du majorant sur la latence au pire cas pour ensuite formaliser la notion de marge temporelle. Considérons un système S dans lequel est ajoutée une chaîne σ_∞. Compte tenu des analyses développées pour le calcul du majorant sur la latence au pire cas, présentées dans le chapitre 3, si $\delta^-_\infty(2)$ tend vers $+\infty$, alors pour toute chaîne σ_a de S, l'interférence de σ_∞ sur σ_a dans une σ_a*-busy-window* est maximale si la chaîne σ_∞ :

— est de priorité supérieure à toutes les autres tâches du système,

— est activée à l'ouverture de la σ_a*-busy-window* et

— a un WCET C_∞ aussi grand que possible.

Concernant le dernier point, nous voulons le plus grand C_∞ tel que la chaîne σ_a soit garantie ordonnançable. Ce scénario nous permet de définir la notion de marge temporelle pour une chaîne et un job d'une chaîne. Commençons d'abord par comprendre comment calculer la marge temporelle en utilisant un exemple.

Exemple 100. *Considérons un système S avec deux chaînes σ_a et σ_d telles que $\sigma_a = \{\delta^-_a(2) = 200, \tau^1_a = (4,8), \tau^2_a = (1,10), \tau^3_a = (3,10)\}$ et $\sigma_d = \{\delta^-_d(2) = 50, \tau^1_d = (5,12), \tau^2_d = (2,14)\}$. L'ordonnancement de S qui mène à la latence pire cas pour σ_a est présenté dans la figure 5.2. Comme présenté précédemment, les interférences sur σ_a sont maximales si toutes les chaînes plus prioritaires commencent à l'ouverture de la σ_a-busy-window. Sur la figure 5.2, il n'y a qu'un job de σ_a activé dans la σ_a-busy-window et on va calculer la marge temporelle de ce job.*

Pour calculer la marge temporelle de σ_a, on considère, comme présenté dans la définition 101, une tâche τ_∞ tel que (i) $\delta^-_\infty(2) > \infty$ et (ii) $\pi_\infty < \pi^1_b$.

Le temps d'exécution de τ_∞ est ensuite augmenté comme sur la figure 5.3 et la marge $S\ell_a(1)$ est obtenue en récupérant la plus grande valeur de C_∞ telle que σ_a respecte son échéance. En augmentant C_∞, l'interférence de σ_d sur σ_a augmente.

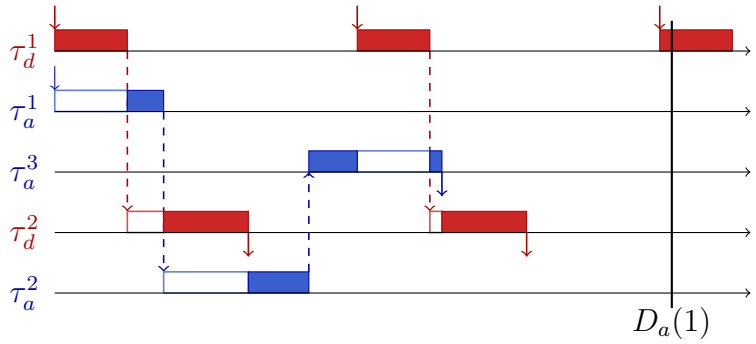

FIGURE 5.2 – Système S avec $C_\infty = 0$ and $B_a(1) = 66$

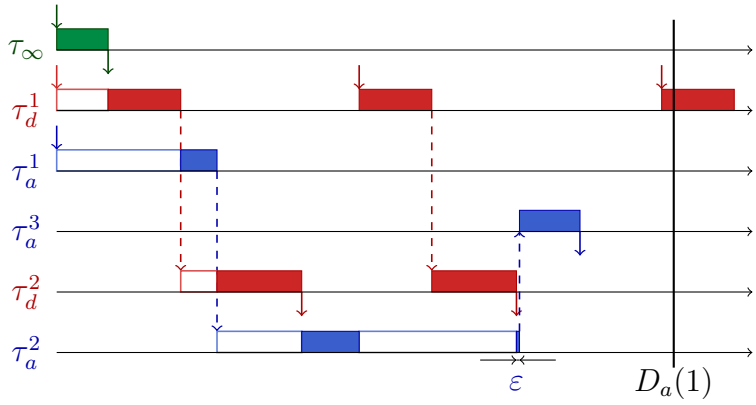

FIGURE 5.3 – Système S avec $C_\infty = C_1 + \varepsilon$ et $B_a(1) = 80$

Cette augmentation de l'interférence est représentée sur le graphique 5.4 par l'augmentation du busy-time de la chaîne. Sur la figure 5.4, l'état du système présenté sur la figure 5.2 correspond au point $C_\infty = 0$ et $B_a(1) = 66$, et l'état du système présenté sur la figure 5.3 correspond au point $C_\infty = C_1 + \varepsilon$ [1] *et $B_a(1) = 66 + C_1$.*

En augmentant la valeur de C_∞ de 0 à C_1 alors le busy-time $B_a(1)$ augmente linéairement, si $C_\infty = C_1 + \varepsilon$ alors l'interférence sur σ_a du second job de σ_d augmente. L'interférence du second job de σ_d sur σ_a passe de C_d^1 si $C_\infty \leq C_1$ (cf figure 5.2) à $C_d^1 + C_d^2$ si $C_\infty \geq C_1$ (cf figure 5.3).

La notion de marge temporelle d'une chaîne ou d'un job d'une chaîne est formalisée par la définition suivante :

Définition 101. *Soit S un système, soit $\sigma_a \in S$, et soit $\sigma_\infty \notin S$ une chaîne ayant une unique activation et telle que :*

$$\forall \sigma_b \in S, \forall \tau_b^i \in \sigma_b \text{ alors } \pi_b^i < \pi_\infty \tag{5.1}$$

Alors la marge temporelle *$S\ell_a$ de la chaîne σ_a (resp. $S\ell_a(q)$ du $q^{ème}$ job de σ_a) est définie comme le temps d'exécution maximum de σ_∞ (c'est à dire C_∞) tel que tous les jobs de σ_a (resp. le $q^{ème}$ job de σ_a) sont ordonnançables dans $S \cup \{\sigma_\infty\}$.*

1. ε dénote la plus petite valeur positive possible

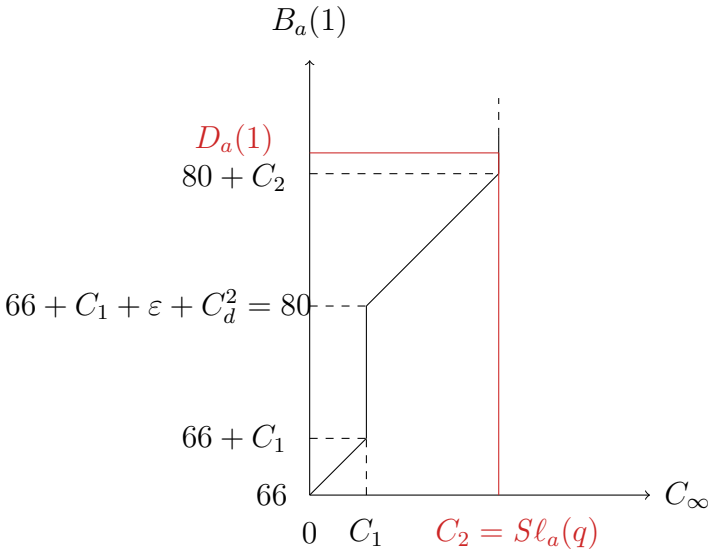

FIGURE 5.4 – Évolution de $B_a(1)$ en fonction de C_∞

Voyons maintenant comment calculer un minorant sur la marge temporelle d'une chaîne σ_a en utilisant les analyses pire cas présentées au chapitre 3.

On va montrer que le calcul du minorant sur la marge temporelle d'une chaîne se fait de manière similaire au majorant sur la latence au pire cas, c'est à dire :

1. Calcul d'un majorant sur la longueur de n'importe quelle σ_a-*busy-window*.
2. Calcul d'un majorant K_a sur le nombre de jobs de σ_a dans n'importe quelle σ_a-*busy-window*.
3. Pour chacun des q jobs de σ_a, $q \in [1, K_a]$, calcul d'un minorant $S\ell_a(q)$ sur la marge temporelle du $q^{\text{ème}}$ job de σ_a.
4. Calcul de la marge temporelle de la chaîne σ_a.

La marge d'une chaîne σ_a est la marge minimum parmi tous les jobs de σ_a. Ceci est formalisé par la propriété suivante.

Propriété 102. *La marge temporelle de σ_a dans n'importe quelle σ_a-busy-window est minorée par :*

$$S\ell_a = \min_{q \in [1, K_a]} \{S\ell_a(q)\} \tag{5.2}$$

Démonstration. Cette propriété découle de la définition 101. La marge $S\ell_a(q)$ est calculée pour garantir l'ordonnançabilité du $q^{\text{ème}}$ job de σ_a dans une σ_a-*busy-window* alors que $S\ell_a$ est calculée pour garantir que tous les jobs de σ_a dans une σ_a-*busy-window* sont ordonnançables. □

L'algorithme 103 sert à calculer la marge temporelle d'une chaîne σ_a à partir de la marge $S\ell_a(q)$ du $q^{\text{ème}}$ job de σ_a dans n'importe quelle σ_a-*busy-window*. L'algorithme se déroule ainsi :

$1 \rightarrow 4$ - Initialisation : Le système est initialisé en calculant un majorant sur la longueur des σ_a-*busy-windows* et un majorant sur le nombre de jobs de σ_a dans n'importe

quelle σ_a-*busy-window*. Comme on veut un minorant sur la marge temporelle, on l'initialise avec la plus grande valeur possible pour la diminuer à chaque itération. On commence par calculer la marge du premier job (c'est à dire $q = 1$).

$5 \rightarrow 11$ - Itération : La marge temporelle de chacun des jobs est calculée et à chaque itération, si la marge temporelle est inférieure à la valeur courante cette dernière est remplacée. L'objectif est de trouver la valeur minimum parmi tous les jobs.

Algorithme 103 Calcul de la marge temporelle de σ_a en utilisant l'équation (5.2)

1: $\overline{BW_a} = \text{COMPTE\_UPPER\_BOUND\_BUSY\_WINDOW}(\sigma_a)$
2: $\overline{K_a} = \eta_a^+(\overline{BW_a})$
3: $S\ell_a = +\infty$
4: $q = 1$
5: **while** $q \leq \overline{K_a}$ **do**
6: $\quad S\ell_a(q) = \text{COMPTE\_SLACK\_JOB}(\sigma_a, q)$
7: $\quad$ **if** $S\ell_a(q) < S\ell_a$ **then**
8: $\quad\quad S\ell_a = S\ell_a(q)$
9: $\quad$ **end if**
10: $\quad q = q + 1$
11: **end while**
12: **return** $S\ell_a$

Considérons un exemple pour ensuite étendre ce calcul au cas général.

Dans le cas général, pour calculer la marge temporelle d'une chaîne σ_a, une première idée est d'incrémenter la valeur de C_∞ et de recalculer la latence de σ_a jusqu'à ce qu'elle manque son échéance. Incrémenter C_∞ de 1 mène à deux différents cas :

1. Le *busy-time* du $q^{\text{ème}}$ job de σ_a est augmenté de 1. Cela arrive, comme présenté sur la figure 5.4, si $C_\infty \in [0, C_1[$ ou si $C_\infty \in]C_1, C_2[$.
2. Le *busy-time* du $q^{\text{ème}}$ job de τ_a est augmenté de plus de 1. Cela arrive, comme présenté sur les figures 5.3 and 5.4 pour $C_\infty = C_1$ ou $C_\infty = C_2$.

Il est à noter que l'augmentation de C_∞ modifie l'interférence sur σ_a soit de manière directe soit en augmentant les interférences provoquées par les autres chaînes.

Calculer la marge en augmentant itérativement C_∞ de 1 peut être long. Une autre méthode possible, et également plus rapide, est la dichotomie qui a déjà été utilisée dans les articles [27] et [30]. Pour calculer la marge temporelle d'une chaîne en utilisant la dichotomie, il faut établir un minorant m et majorant M sur la marge. En séparant l'intervalle $[m, M]$ en deux parties égales, on obtient une valeur $n = m + \frac{M-m}{2}$ pour la marge et on vérifie si l'utilisation d'une marge n résulte en une chaîne ordonnançable ou non. Si la chaîne est ordonnançable alors la marge réelle est supérieure ou égale n sinon elle est inférieure à n. On obtient alors un nouvel intervalle plus restreint et on réitère en réduisant les bornes de l'intervalle jusqu'à obtenir la marge. Selon la complexité du système la méthode par dichotomie peut être coûteuse en temps. De plus, l'augmentation linéaire de la marge temporelle n'augmente pas linéairement la latence d'une chaîne à cause des préemptions.

Par la suite nous proposons une approche plus efficace pour calculer C_∞ en calculant l'augmentation maximum de C_∞ qui ne change pas l'interférence des chaînes de S sur σ_a

(c'est à dire C_1 et C_2 sur le graphique 5.4). Pour ce faire, commençons par comprendre comment varient les interférences d'une chaîne $\sigma_d \in S$ sur σ_a si C_∞ augmente.

Considérons l'exemple 100, la figure 5.4 représente l'évolution de $B_a(1)$ [2] en fonction de C_∞. Sur la figure 5.4, le point $(C_\infty = 0, B_a(1) = 66)$ correspond au chronogramme figure 5.2 et le point $(C_\infty = C_1, B_a(1) = 80)$ correspond au chronogramme figure 5.3.

Le premier job de σ_d est activé avant la terminaison de $\tau_a^{\ell t_a(d)}$, ainsi ce job interfère à hauteur de son WCET. Le second job de σ_d n'interfère que partiellement avec σ_a. Sur la figure 5.2 $C_\infty = 0$ et l'interférence du second job de σ_d est de C_d^1. Sur la figure 5.3 $C_\infty = C_1 + \varepsilon$ et l'interférence du second job de σ_d est de $C_d^1 + C_d^2$. Les activations de σ_d sont fixées par δ_d^- mais augmenter C_∞ va retarder l'exécution de σ_a et de σ_d, et donc changer l'interférence de σ_d sur σ_a.

Dans le cas général, les chaînes dont l'interférence sur σ_a ne change pas en augmentant C_∞ sont les chaînes synchrones de priorité inférieure à π_a, ainsi que la chaîne σ_a si elle est synchrone. En effet, si σ_a est synchrone alors elle ne peut pas préempter un de ses jobs précédents, ainsi un job de σ_a est suspendu jusqu'à la terminaison des jobs précédents. Si une chaîne $\sigma_b \in \ell p(a)$ est synchrone, elle ne peut exécuter qu'un unique segment dans une σ_a*-busy-window*.

Par déduction, en augmentant C_∞ les interférences d'une chaîne $\sigma_b \in S$ sur σ_a peuvent changer :

1. si σ_b est de priorité supérieure à σ_a ou
2. si σ_b est asynchrone (inclut le cas où σ_a est asynchrone).

Pour calculer la marge temporelle d'une chaîne, nous commençons par identifier les conditions pour que la latence au pire cas de σ_a augmente linéairement avec l'augmentation de C_∞. Ensuite, nous calculons la plus grande valeur Δ_{max} telle qu'en augmentant C_∞ de Δ_{max} les interférences sur σ_a augmentent de Δ_{max}.

L'interférence d'un job de σ_b sur σ_a dépend de la tâche de σ_a qui s'exécute lorsque σ_b est activée. Ainsi, nous identifions les tâches τ_a^i de la chaîne σ_a telles que l'interférence d'une chaîne σ_b sur σ_a change si σ_b est activée pendant l'exécution de τ_a^i ou τ_a^{i+1}, autrement dit telles que le segment de tête de σ_b sur la sous-chaîne $\sigma_{a[i+1..n_a]}$ est différent de celui sur $\sigma_{a[i..n_a]}$. Nous appelons ces tâches de σ_a les *tâches échelons* et notons $\mathcal{T}_a^b$ l'ensemble des tâches échelon de σ_a par rapport à σ_b.

Définition 104 (Tâche échelon de σ_a par rapport à σ_b)**.** *Soient σ_a et σ_b deux chaînes. Pour tout $i \in [\ell t_a, n_a[$, la tâche τ_a^i est une tâche échelon par rapport à σ_b si :*

$$s^{head}_{b\to a[i..n_a]} \neq s^{head}_{b\to a[i+1..n_a]} \tag{5.3}$$

et $\tau_a^{n_a}$ est une tâche échelon si $C_{s^{head}_{b\to a[n_a..n_a]}} \neq 0$.

Pour rappel, $s^{head}_{b\to a[i..n_a]}$ est le segment de tête de la chaîne σ_b sur la sous-chaîne $\sigma_{a[i..n_a]}$. Illustrons maintenant cette notion de tâches échelons via un exemple.

Exemple 105. *Soit un système composé de trois chaînes σ_a, σ_d et σ_e telles que $\pi_a < \pi_d < \pi_e$. Le scénario d'exécution figure 5.5 maximise les interférences sur σ_a. Considérons les deux chaînes σ_a et σ_d. L'interférence sur σ_a d'un job de σ_d est de :*

2. dans cas $B_a(1)$ correspond à la latence au pire cas de σ_a

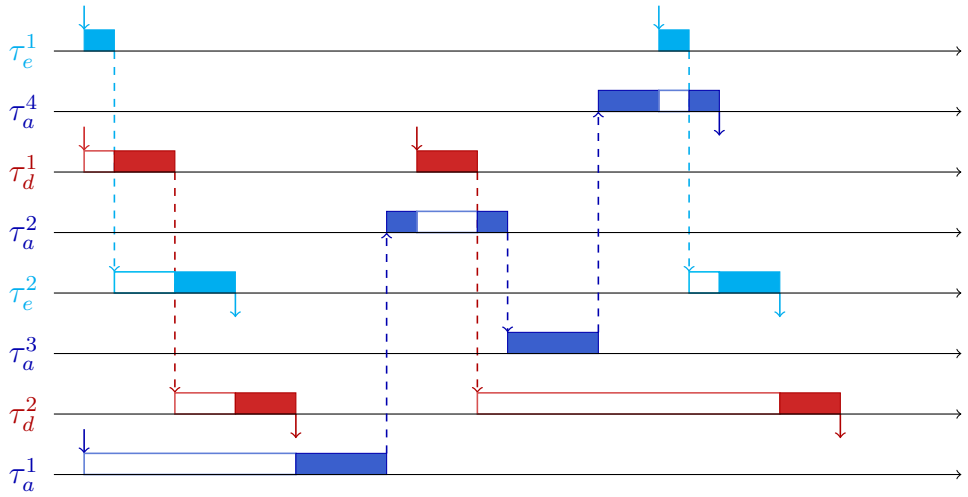

FIGURE 5.5 – Tâches échelons de σ_a

— C_d *s'il est activé pendant l'exécution de* τ_a^1.
— C_d^1 *s'il est activé pendant l'exécution de* τ_a^2.
— C_d^1 *s'il est activé pendant l'exécution de* τ_a^3.
— 0 *s'il est activé pendant l'exécution de* τ_a^4.

Les tâches τ_a^1 *et* τ_a^3 *sont les tâches échelons de* σ_a *par rapport à* σ_d. *Les tâches échelons de* σ_a *par rapport à* σ_e *sont* τ_a^3 *et* τ_a^4.

Algorithme 106 Identification des tâches τ_a^i de l'ensemble $\mathcal{T}_a^b$

```
1:  function COMPUTE_STEP_TASKS(σ_a, σ_b)
2:      T_a^b = ∅
3:      for i ∈ [ℓt_a, n_a[ do
4:          C_{s^head_{b→a[i...n_a]}} = COMPUTE_WCET_HEAD(σ_{a[i...n_a]}, σ_b)
5:          C_{s^head_{b→a[i+1...n_a]}} = COMPUTE_WCET_HEAD(σ_{a[i+1...n_a]}, σ_b)
6:          if C_{s^head_{b→a[i...n_a]}} ≠ C_{s^head_{b→a[i+1...n_a]}} then
7:              T_a^b = T_a^b ∪ τ_a^i
8:          end if
9:      end for
10:     C_{s^head_{b→a[n_a...n_a]}} = COMPUTE_WCET_HEAD(σ_{a[n_a...n_a]}, σ_b)
11:     if C_{s^head_{b→a[n_a]}} ≠ 0 then
12:         T_a^b = T_a^b ∪ τ_a^{n_a}
13:     end if
14:     return T_a^b
15: end function
```

L'algorithme 106 permet d'identifier toutes les tâches qui appartiennent à l'ensemble $\mathcal{T}_a^b$, il se déroule ainsi :

2 - Initialisation : L'ensemble $\mathcal{T}_a^b$ est vide.

$3 \rightarrow 13$ - Itération : Pour chaque tâche τ_a^i de σ_a, en comparant les WCETs des segments de tête de σ_b sur la sous-chaîne $\sigma_{a[i..n_a]}$, il est possible d'identifier les tâches échelons de σ_a par rapport à σ_b.

L'interférence des chaînes synchrones de $\ell p(a)$ et l'auto-interfèrence si σ_a est synchrone ne changent pas avec l'augmentation de C_∞. En effet une chaîne synchrone de $\ell p(a)$ interfère à hauteur d'au plus un segment dans une σ_a-*busy-window*, et si σ_a est synchrone alors l'auto-interférence est nulle. Ainsi, l'ensemble des tâches échelons de σ_a par rapport à ces chaînes est donc vide.

Propriété 107. *L'ensemble des tâches échelons de σ_a par rapport à toutes les chaînes de S est :*

$$\mathcal{T}_a = \bigcup_{\sigma_b \in [hp(a) \cup (\ell p(a) \cap \mathcal{AC}) \cup (\sigma_a \cap \mathcal{AC})]} \mathcal{T}_a^b \tag{5.4}$$

Démonstration. Pour définir les tâches de σ_a appartenant à $\mathcal{T}_a$, il suffit de calculer les ensembles $\mathcal{T}_a^b$ pour chaque chaîne σ_b dans $[hp(a) \cup (\ell p(a) \cap \mathcal{AC}) \cup (\sigma_a \cap \mathcal{AC})]$. Cet ensemble inclut (i) les chaînes plus prioritaires (synchrone ou asynchrone), (ii) les chaînes asynchrones moins prioritaires et (iii) la chaîne σ_a si elle est asynchrone. L'union de ces ensembles correspond à correspond aux chaînes de S dont l'interférence sur σ_a augmente avec l'augmentation des *busy-time* $B_a^i(q)$. □

Les tâches échelons permettent de calculer la plus grande augmentation Δ_{max} de C_∞ telle que les interférences sur σ_a, et donc sa latence, augmentent de Δ_{max}.

Voyons sur un exemple comment calculer l'augmentation maximum de C_∞ qui augmente la latence de σ_a de manière continue.

Exemple 108. *Considérons le système défini dans l'exemple 105 dont une exécution est représentée sur la figure 5.6. Pour rappel, les tâches τ_a^1 et τ_a^3 sont les tâches échelons de σ_a par rapport à σ_d et celles de σ_a par rapport à σ_e sont τ_a^3 et τ_a^4. Les seconds jobs de σ_d et σ_e interfèrent partiellement avec σ_a, et avec l'augmentation de C_∞ l'interférence de ces jobs augmente. Compte tenu de l'exécution figure 5.6, si on considère que le second job de σ_d (resp. σ_e) est activé pendant l'exécution de τ_a^1 (resp. τ_a^3), l'interférence de ce job sur σ_a augmente. Ainsi augmenter C_∞ de $\delta_d^-(2) - B_a^1(1)$ (resp. $\delta_e^-(2) - B_a^3(1)$) ne change pas l'interférence de σ_d (resp. σ_e) sur σ_a.*

Les propriétés suivantes permettent de définir l'augmentation maximum de C_∞ telle que l'interférence d'une chaîne σ_b sur σ_a ne change pas. Pour cela, nous considèrons trois cas selon la chaîne qui interfère avec σ_a et qui peut être :

1. une chaîne de priorité supérieure à π_a,
2. une chaîne asynchrone de priorité inférieure à π_a,
3. les jobs suivants de σ_a si elle est asynchrone.

Propriété 109. *Pour toutes les tâches τ_a^i de $\mathcal{T}_a^d$, si le nombre d'activations de σ_d (avec $\pi_a < \pi_d$) dans un intervalle $B_a^i(q)$ est égal au nombre d'activations de σ_d dans un intervalle $B_a^i(q) + \Delta_{max}$ alors l'interférence de σ_d est la même pour les deux intervalles. Formellement :*

$$\forall \tau_a^i \in \mathcal{T}_a^d \text{ si } \eta_d^+(B_a^i(q)) = \eta_d^+(B_a^i(q) + \Delta_{max}) \\ \implies \overline{hp\mathcal{I}_{d\rightarrow a}^{i,q}}(B_a^i(q) + \Delta_{max}) = \overline{hp\mathcal{I}_{d\rightarrow a}^{i,q}}(B_a^i(q)) \tag{5.5}$$

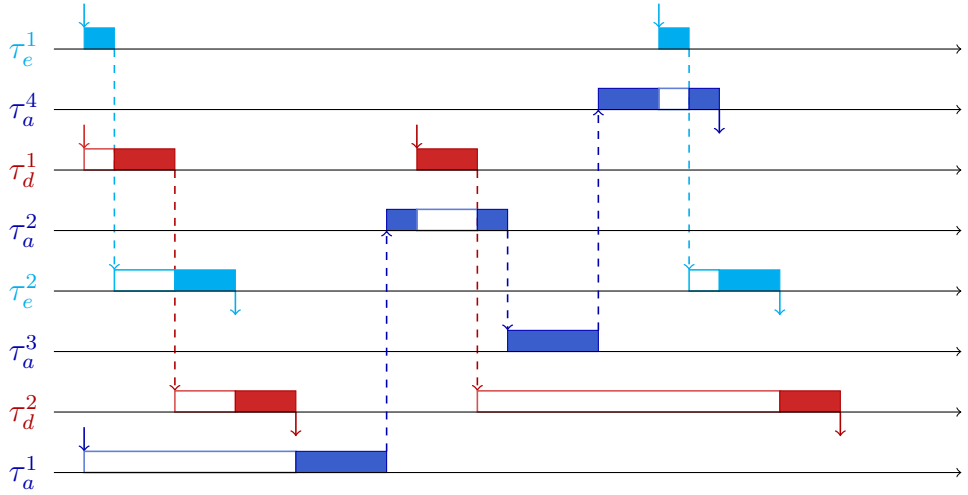

FIGURE 5.6 – Les tâches échelons de σ_a par rapport à σ_d sont τ_a^1 et τ_a^3 (resp. σ_e sont τ_a^3 et τ_a^4)

Démonstration. L'ensemble $\mathcal{T}_a^d$ est l'ensemble des tâches échelons de σ_a tel que l'interférence d'une chaîne $\sigma_d \in hp(a)$ change si σ_d est activée avant ou après la terminaison d'une de ces tâches. Mais si le nombre d'activations de σ_d dans le *busy-time* $B_a^i(q)$ ($\tau_a^i \in \mathcal{T}_a^d$) ne change pas en augmentant C_∞ de Δ_{max}, alors selon les équations 3.9 (cas synchrone) ou 3.10 (cas asynchrone), l'interférence de σ_d sur $B_a^i(q)$ ne change pas non plus car aucun job de σ_d n'arrive plus tôt pour augmenter les interférences sur σ_a. □

La propriété 109 indique de combien il est possible d'augmenter le WCET C_∞ de Δ sans modifier l'interférence d'une chaîne plus prioritaire sur σ_a. De la même manière, il est possible d'établir des propriétés similaires concernant l'interférence d'une chaîne moins prioritaire ainsi que l'auto-interférence dans le cas de chaînes asynchrones.

Propriété 110. *Pour toutes les tâches τ_a^i de $\mathcal{T}_a^b$, si le nombre d'activations d'une chaîne asynchrone σ_b (avec$\pi_b < \pi_a$) dans un intervalle $B_a^i(q)$ est égal au nombre d'activations de σ_b dans un intervalle $B_a^i(q) + \Delta_{max}$ alors l'interférence de σ_b est la même pour les deux intervalles. Formellement :*

$$\forall \tau_a^i \in \mathcal{T}_a,\ \sigma_b \in \mathcal{AC} \text{ si } \eta_b^+(B_a^i(q)) = \eta_b^+(B_a^i(q) + \Delta_{max}) \\ \implies \overline{\ell p \mathcal{I}_a^{i,q}}(B_a^i(q) + \Delta_{max}) = \overline{\ell p \mathcal{I}_a^{i,q}}(B_a^i(q)) \quad (5.6)$$

Démonstration. La preuve est similaire à celle du théorème 109. De manière trivial, si aucun job de σ_b n'est activé plus tôt lors de l'exécution du $q^{\text{ème}}$ job de σ_a alors l'interférence de σ_b ne change pas. □

Les propriétés 109 et 110 permettent de traiter l'interférence d'une unique chaîne sur σ_a. Une tâche τ_a^i peut être une tâche échelon par rapport à une chaîne mais pas par rapport à d'autres.

Propriété 111. *Si σ_a est asynchrone, pour toutes les tâches τ_a^i de $\mathcal{T}_a^a$, si le nombre d'activations de σ_a (avec $\pi_a < \pi_b$) dans un intervalle $B_a^i(q)$ est égal au nombre d'activations de*

σ_a dans un intervalle $B_a^i(q) + \Delta_{max}$ alors l'auto-interférence est la même pour les deux intervalles. Formellement :

$$\forall \tau_a^i \in \mathcal{T}_a,\ \sigma_b \in \mathcal{AC} \text{ si } \eta_a^+(B_a^i(q)) = \eta_a^+(B_a^i(q) + \Delta_{max}) \\ \implies \overline{self\mathcal{I}_a^{i,q}}(B_a^i(q) + \Delta_{max}) = \overline{self\mathcal{I}_a^{i,q}}(B_a^i(q)) \quad (5.7)$$

Démonstration. De manière similaire aux propriétés précédentes, si les jobs suivants de la chaîne σ_a sont activés plus tôt lors de l'exécution du $q^{\text{ème}}$ job de σ_a alors leur interférence augmentera. □

Maintenant, il reste à calculer la valeur maximum de Δ_{max} telle que les propriétés 109, 110 et 111 soient valides. Plusieurs jobs de plusieurs chaînes sont susceptibles de changer les interférences sur σ_a, cependant dans la liste des jobs qui peuvent modifier l'interférence sur σ_a, il faut d'abord identifier le premier job qui modifiera cette interférence. Dans cet objectif, pour une chaîne σ_b, nous cherchons tout d'abord l'activation de cette chaîne qui arrive après $B_a^i(q)$ (c'est à dire la terminaison du $q^{\text{ème}}$ job de $\tau_a^i \in \mathcal{T}_a^b$).

Dans le cas général, l'instant d'activation au plus tôt d'une chaîne σ_b qui arrive après la terminaison du $q^{\text{ème}}$ job de τ_a^i (c'est à dire après $B_a^i(q)$) avec $\tau_a^i \in \mathcal{T}_a^b$ est donné par :

$$\mathcal{A}_b^i = \delta_b^-(\eta_b^+(B_a^i(q)) + 1) \quad (5.8)$$

Ainsi pour chaque tâche $\tau_a^i \in \mathcal{T}_a^b$, il est possible de calculer la durée entre la terminaison du $q^{\text{ème}}$ job de τ_a^i et l'activation d'une chaîne σ_b dont l'interférence sur σ_a peut augmenter. La durée maximum Δ_{max} entre $B_a^i(q)$ et l'activation suivante d'une chaîne σ_b correspond à une valeur de Δ_{max} telle que les propriétés 109, 110 et 111 sont valides.

Propriété 112. *La durée maximum pendant laquelle σ_a peut être suspendue sans que l'interférence sur σ_a change est :*

$$\lambda_a = \min_{\sigma_b \in S, \tau_a^i \in \mathcal{T}_a^b} \{\delta_b^-(\eta_b^+(B_a^i(q)) + 1) - B_a^i(q)\} \quad (5.9)$$

Démonstration. En utilisant l'activation au plus tôt de toutes les chaînes par rapport à toutes les tâches dans $\mathcal{T}_a^b$ nous obtenons la durée maximum qui peut être ajoutée à σ_a sans changer les interférences sur cette chaîne. □

Voyons maintenant sur un exemple comment s'applique la propriété 112 qui permet de trouver l'augmentation maximum de C_∞ telle que L_a augmente de manière continue.

Exemple 113. *On a vu que pour trouver l'augmentation maximum de C_∞ telle que $B_a(1)$ augmente de manière continue, il faut comparer les busy-times des tâches échelons, c'est à dire $B_a^1(1)$, $B_a^3(1)$, $B_a^4(1)$, aux modèles d'activations $\delta_d^-(2)$ et $\delta_e^-(2)$. Sur la trace d'exécution de la figure 5.5, en augmentant C_∞, le premier job dont l'interférence sur σ_a augmente est le second job de σ_d. Le second job de σ_d sera le premier dont l'interférence augmentera car $\delta_d^-(\eta_d^+(B_a^1(1)) + 1) - B_a^1(1) < \delta_e^-(\eta_e^+(B_a^3(1)) + 1) - B_a^3(1)$. La figure 5.7 représente l'augmentation maximum de C_∞ telle que $B_a(1)$ augmente de manière continue. Sur l'exécution figure 5.7, augmenter C_∞ de ε augmente l'interférence de σ_d sur σ_a.*

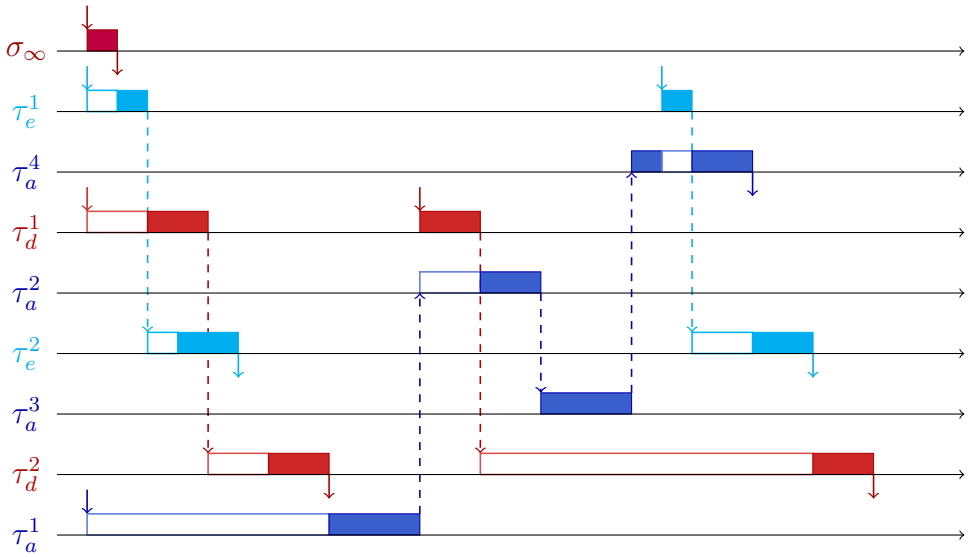

FIGURE 5.7 – Augmentation de C_∞ telle que l'interférence augmente de manière continue

Pour calculer la valeur maximum de C_∞, en accord avec la définition 101 de marge temporelle, il faut garantir que la chaîne σ_a ne manque pas son échéance temporelle. En effet en agmentant C_∞, il est possible que la chaîne σ_a manque son écheance et il faut rajouter une condition à la propriété 112 afin de garantir que le $q^{\text{ème}}$ job de σ_a respecte son échéance. Si σ_a manque son échéance à cause de l'augmentation de l'interférence alors la marge temporelle est obtenue en gardant la valeur maximum de C_∞ telle que la chaîne σ_a est ordonnançable.

Propriété 114. *La durée de préemption maximum qui peut être ajoutée à σ_a tout en garantissant l'ordonnançabilité de σ_a est :*

$$\lambda_a^{min} = \min\{\lambda_a, D_a(q) - B_a(q)\} \tag{5.10}$$

Démonstration. Nous ajoutons simplement une condition pour garantir que le $q^{\text{ème}}$ job de σ_a respecte son échéance. □

λ_a^{min} est à la durée maximum qui peut être ajoutée à C_∞ sans changer les interférences du système sur σ_a tout en garantissant l'ordonnançabilité de σ_a. Augmenter C_∞ de $\lambda_a^{min}+\varepsilon$ change les interférences sur σ_a et, dans ce cas il faut recalculer les majorants sur les *busy-times* et itérer jusqu'à ce qu'un job de σ_a atteigne son échéance. Pour cela, nous avons besoin d'initialiser le *busy-time* avec C_∞. Pour rappel, afin de calculer la marge temporelle d'un job de σ_a, on doit calculer plusieurs fois λ_a^{min} et utiliser cette valeur pour augmenter le WCET C_∞ jusqu'à ce que ce job atteigne son échéance. Le WCET C_∞ est donc une somme de un ou plusieurs λ_a^{min}. Ensuite, on étend simplement l'équation (3.4) en initialisant le calcul du *busy-time* avec une valeur initiale que l'on note p et qui est une valeur de C_∞.

Notation 116. *Le busy-time sur q événements de τ_a^i est majoré par :*

$$\overline{B_a^i(q,p)} = p + (q-1)\times C_a + C_{a[1..i]} + \ell p\mathcal{I}_a^{i,q}(\overline{B_a^i(q,p)}) + \sum_{\sigma_b \in hp(a)} hp\mathcal{I}_{b\to a}^{i,q}(\overline{B_a^i(q,p)}) + self\mathcal{I}_a^{i,q}(\overline{B_a^i(q,p)}) \tag{5.11}$$

Algorithme 115 Calcul de λ_a

function COMPUTE_MAX_PREEMPTION_DURATION(σ_a)
  $\lambda_a^{min} = D_a(q) - B_a(q)$
  for $\sigma_b \in S$ **do**
    for $\tau_a^i \in \mathcal{T}_a^b$ **do**
      $\lambda_a = \delta_b^-(\eta_b^+(B_a^i(q)) + 1) - B_a^i(q)$
      if $\lambda_a < \lambda_a^{min}$ **then**
        $\lambda_a^{min} = \lambda_a$
      end if
    end for
  end for
  return λ_a^{min}
end function

L'équation est différente de l'équation (3.4) car elle comprend un terme supplémentaire, noté p, qui correspond à la valeur de la marge temporelle.

Il est à noter que $\overline{B_a^i(q, p = 0)} = \overline{B_a^i(q)}$, ce qui signifie que si $p = 0$ alors l'équation (5.3) permet de calculer le *busy-time* du $q^{\text{ème}}$ job de τ_a^i.

L'algorithme suivant permet de calculer un majorant sur le *busy-time* du $q^{\text{ème}}$ job de σ_a dans une σ_a-*busy-window*. La différence avec l'algortihme 37 calculant le *busy-time* d'une tâche est que le calcul est initialisé avec p.

Algorithme 117 Calcul d'un majorant sur le *busy-time* du $q^{\text{ème}}$ job de σ_a

function COMPUTE_BUSY_TIME(σ_a, τ_a^i, q, p)
  $B_a^i(q,p) = p + (q-1) \times C_a + C_{a[1..i]}$
  $newB_a^i(q,p) = 0$
  while $B_a^i(q,p) \neq newB_a^i(q,p)$ **do**
    $B_a^i(q,p) = newB_a^i(q,p)$
    $\ell p\mathcal{I}_a^{i,q}(\overline{B_a^i(q,p)}) =$ COMPUTE_UB_LP_INTERF($\sigma_a, B_a^i(q,p)$)
    $self\mathcal{I}_a^{i,q}(\overline{B_a^i(q,p)}) =$ COMPUTE_UB_SELF_INTERF($B_a^i(q,p)$)
    $hp\mathcal{I}_a^{i,q}(\overline{B_a^i(q,p)}) = 0$
    for $\sigma_b \in hp(a)$ **do**
      $hp\mathcal{I}_{b\to a}^{i,q}(\overline{B_a^i(q,p)}) =$ COMPUTE_UB_HP_INTERF($\sigma_a, \sigma_b, \tau_a^i, q, B_a^i(q,p)$)
      $hp\mathcal{I}_a^{i,q}(\overline{B_a^i(q)}) = hp\mathcal{I}_a^{i,q}(\overline{B_a^i(q)}) + hp\mathcal{I}_{b\to a}^{i,q}(\overline{B_a^i(q,p)})$
    end for
    $newB_a^i(q,p) = p + (q-1) \times C_a + C_{a[1..i]} + \ell p\mathcal{I}_a^{i,q}(\overline{B_a^i(q,p)}) + self\mathcal{I}_a^{i,q}(\overline{B_a^i(q,p)}) + hp\mathcal{I}_a^{i,q}(\overline{B_a^i(q,p)})$
  end while
  return $B_a^i(q,p)$
end function

Notation 118 (Maximum busy time)**.** *Pour tout $i \in [\ell t_a, n_a]$, on dénote $\mathfrak{B}_a^i(q)$, la valeur maximum du busy-time des tâches τ_a^i, $\forall i \in [\ell t_a, n_a]$ tel que le système est ordonnançable. Par définition, on a $\mathfrak{B}_a^i(q) = B_a^i(q,p) = S\ell_a(q)$).*

L'algorithme 119 calcule la marge temporelle d'un job de la chaîne σ_a.

Algorithme 119 Calcul de la marge temporelle du $q^{\text{ème}}$ job de σ_a en utilisant l'équation (5.2)

```
1: function COMPUTE_SLACK_JOB(σ_a, q)
2:     B_a(q, 0) = B_a(q)
3:     Sℓ_a(q) = 0
4:     while B_a(q, p) ≤ D_a(q) do
5:         Sℓ_a(q) = Sℓ_a(q) + COMPUTE_MAX_PREEMPTION_DURATION(σ_a)
6:         B_a(q, Sℓ_a(q)) = COMPUTE_BUSY_TIME(σ_a, τ_a^i, q, p)
7:     end while
8:     Sℓ_a(q) = Sℓ_a(q) − ε
9:     return Sℓ_a(q)
10: end function
```

L'algorithme 119 qui permet de calculer la marge du $q^{\text{ème}}$ job de σ_a se déroule ainsi :

$2 \rightarrow 3$ - Initialisation : La marge temporelle et le *busy-time* du $q^{\text{ème}}$ job de σ_a sont initialisés.

$4 \rightarrow 7$ - Itération : La marge temporelle $S\ell_a(q)$ est augmentée itérativment et à chaque itération, le *busy-time* est calculé grâce à l'équation (5.3).

Dans cette section, nous avons montré comment calculer la marge temporelle d'une chaîne. Nous allons montrer comment cette marge peut être utilisée pour rajouter des chaînes tout en garantissant l'ordonnançabilité du système.

5.4 Analyse de flexibilité

Dans cette section, considérons le système temps-réel ordonnançable [3] S pour lequel nous avons calculé la marge temporelle de chaque chaîne. Comme présenté dans la section 5.1, nous voulons ajouter une chaîne σ_{new} et calculer des conditions sur les paramètres de σ_{new} telles que le système résultant $S_{new} = S \cup \{\sigma_{new}\}$ soit ordonnançable. Plus précisément, nous souhaitons identifier, pour un ensemble de priorités et un modèle d'activation de σ_{new} donnés, des contraintes sur les WCETs telles que les chaînes de S_{new} soient ordonnançables.

Dans le cas d'un système mono-processeur avec des tâches indépendantes, pour garantir l'ordonnançabilité de ce système après avoir rajouté une tâche τ_{new}, il faut calculer :

1. l'interférence des chaînes de priorité supérieure à τ_{new} sur cette tâche pour garantir son ordonnançabilité,
2. l'interférence de τ_{new} sur les tâches de priorité inférieure pour garantir leur ordonnançabilité.

Dans le cas de chaînes de tâches, les interférences sont plus complexes et, afin de garantir l'ordonnançabilite de S_{new}, nous devons garantir que :

3. Si le système S n'est pas ordonnançable, alors au moins une chaîne du système ne respecte pas son échéance et dans ce cas il n'est pas possible de rajouter une nouvelle chaîne σ_{new} tout en garantissant l'ordonnançablité de S_{new}. Ainsi, si S n'est pas ordonnançable, alors la flexibilité de ce système est nulle.

1. les interférences causées par σ_{new} sur les chaînes moins prioritaires ne causent aucun dépassement d'échéance,
2. les interférences causées par σ_{new} sur les chaînes plus prioritaires ne causent aucun dépassement d'échéance,
3. la chaîne σ_{new} est elle-même ordonnançable.

Dans cette section, le calcul des majorants sur les interférences, présenté au chapitre 3, permet d'établir des contraintes sur σ_{new} pour garantir l'ordonnançabilité de S_{new} et par conséquent, de calculer la flexibilité d'un système.

Commençons par établir des contraintes pour garantir l'ordonnançabilité des chaînes de priorité inférieure à σ_{new}.

5.4.1 Ordonnançabilité des chaînes dans $\ell p(new)$

Pour garantir l'ordonnançabilité des chaînes de priorité inférieure à σ_{new} dans S_{new}, nous avons besoin de quantifier l'interférence de σ_{new} sur chaque chaîne $\sigma_a \in \ell p(new)$. Pour cela, nous utilisons les propriétés 66 et 68 qui permettent de calculer l'interférence d'une chaîne $\sigma_{new} \in hp(a)$ sur σ_a dans les cas où σ_{new} est synchrone (propriété 66) ou asynchrone (propriété 68). Nous souhaitons établir, en utilisant ces deux propriétés, des contraintes sur le WCET des tâches de σ_{new} afin de garantir l'ordonnançabilité des chaînes de priorité inférieure à π_{new}.

Pour chaque chaîne $\sigma_a \in \ell p(new)$, l'interférence de σ_{new} sur le *busy-time* sur q événements de τ_a^i ($i > \ell t_a(new)$), pour $\Delta > B_a^{i-1}(q)$, est majorée par :

— $\overline{hp\mathcal{I}^{i,q}_{\substack{new\to a\\ sync}}}(\Delta)$ si σ_{new} est synchrone, selon la propriété 66,

— $\overline{hp\mathcal{I}^{i,q}_{\substack{new\to a\\ async}}}(\Delta)$ si σ_{new} est asynchrone selon la propriété 68.

Les propriétés 120 et 121 permettent de garantir l'ordonnançabilité dans S_{new} de $\sigma_a \in \ell p(new)$ que σ_a soit synchrone ou asynchrone. Pour établir ces propriétés, nous utilisons :

1. le calcul de l'interférence de σ_{new} sur σ_a dont les équations ont été établies au chapitre 3,
2. le plus long *busy-time* $\mathfrak{B}_a^{n_a}(q)$ tel que σ_a soit ordonnançable qui a été déterminé lors du calcul de la marge temporelle du $q^{\text{ème}}$ job de σ_a (cf 118).

Propriété 120. *Soit S un système ordonnançable et σ_{new} une chaîne synchrone. La chaîne $\sigma_a \in \ell p(new)$ est garantie ordonnançable dans S_{new} si :*

$$\forall q \in [1, K_a], \overline{hp\mathcal{I}^{n_a,q}_{\substack{new\to a\\ sync}}}(\mathfrak{B}_a^{n_a}(q)) \leq S\ell_a(q) \tag{5.12}$$

Démonstration. $\overline{hp\mathcal{I}^{n_a,q}_{\substack{new\to a\\ sync}}}(\mathfrak{B}_a^{n_a}(q))$ est un majorant sur l'interférence causée par une chaîne plus prioritaire σ_{new} sur le *busy-time* du $q^{\text{ème}}$ job de σ_a de telle sorte que ce job soit ordonnançable. De plus, $S\ell_a(q)$ est, par définition, un minorant sur la quantité d'interférence additionnelle que peut subir le $q^{\text{ème}}$ job de σ_a tout en restant ordonnançable. D'où le résultat. □

Nous établissons maintenant une propriété similaire pour gérer le cas où σ_a est asynchrone.

Propriété 121. *Soit S un système ordonnançable et σ_{new} une chaîne. Une chaîne asynchrone $\sigma_a \in \ell p(new)$ est garantie ordonnançable dans S_{new} si :*

$$\forall q \in [1, K_a], \overline{hp\mathcal{I}^{n_a,q}_{new\to a}}_{async}(\mathfrak{B}_a^{n_a}(q)) \leq S\ell_a(q) \tag{5.13}$$

Démonstration. $\mathfrak{B}_a^{n_a}(q)$ est la terminaison au plus tard du $q^{\text{ème}}$ job de σ_a de telle sorte que ce job soit ordonnançable. $\overline{hp\mathcal{I}^{n_a,q}_{new\to a}}_{async}(\mathfrak{B}_a^{n_a}(q))$ est un majorant sur l'interférence causée par une chaîne plus prioritaire σ_{new} sur le *busy-time* du $q^{\text{ème}}$ job de σ_a dans un préfixe de longueur $\mathfrak{B}_a^{n_a}(q)$. De plus, $S\ell_a(q)$ est un minorant sur la quantité d'interférence additionnelle que peut subir le $q^{\text{ème}}$ job de σ_a tout en restant ordonnançable. □

Les propriétés 120 et 121 permettent de borner l'interférence de σ_{new} sur σ_a. On utilise ces équations pour contraindre les WCETs C^i_{new} ($i \in [1, n_{new})$. Dans le cas où σ_a atteint son échéance, l'interférence du $p^{\text{ème}}$ job de σ_{new} sur le $q^{\text{ème}}$ job de σ_a est :

1. C_{new} si $\delta^-_{new}(p) < \mathfrak{B}_a^{\ell t_a(new)}(q)$.
2. $C_{s^{head}_{new\to a[i..n_a]}}$ si $\mathfrak{B}_a^{i-1}(q) < \delta^-_{new}(p) < \mathfrak{B}_a^{i}(q)$ avec $\ell t_a(new) < i - 1 < i \leq n_a$. Si σ_{new} est synchrone un unique job de σ_{new} est activé après $\mathfrak{B}_a^{\ell t_a(new)}(q)$. Si σ_{new} est asynchrone chaque job de σ_{new} activé après la terminaison du $q^{\text{ème}}$ job de σ_a interfère.
3. Nulle si $\mathfrak{B}_a^{n_a}(q) < \delta^-_{new}(p)$.

Intéressons-nous maintenant à la répartition du WCET parmi les tâches de σ_{new}. L'interférence de σ_{new} sur σ_a dépend de la répartition du WCET parmi les C^i_{new}, avec i dans $[1, n_{new}]$. Nous considèrons, compte tenu du contexte industriel, que la répartition du WCET est inconnue et donc, nous n'avons aucune contrainte sur cette répartition.

Si la répartition du WCET est inconnue alors utiliser la répartition de WCET qui maximise les interférences sur chaque chaîne σ_a permet de valider l'ensemble des répartitions possibles. Ainsi, pour garantir l'ordonnançabilité de σ_a, nous utilisons la répartition des WCETs C^i_{new} telle que l'interférence de σ_{new} sur σ_a est maximum. Cette répartition est obtenue en maximisant la valeur de C^1_{new} et en fixant tous les autres C^i_{new} avec $i > 1$ à ε, où ε dénote la plus petite valeur positive possible.

Compte tenu de ce scénario qui permet de maximiser l'interférence de σ_{new}, il est maintenant possible de borner C_{new} de manière à garantir l'ordonnançabilité de σ_a.

En utilisant l'équation 5.13 qui permet de contraindre l'interférence de la chaîne synchrone σ_{new} sur σ_a, nous obtenons la propriété 122 pour établir une contrainte sur le WCET.

Propriété 122. *Soit σ_{new} une chaîne synchrone alors, la condition sur C_{new} pour que σ_a soit garantie ordonnançable est :*

$$\forall q \in [1; K_a], C_{new} \leq \begin{cases} \frac{S\ell_a(q)}{nb_{new}} & \textit{si } nb_{new} = \eta^+_{new}(\mathfrak{B}_a(q)) \\ \frac{S\ell_a(q)}{nb_{new}+1} & \textit{sinon} \end{cases} \tag{5.14}$$

où $nb_{new} = \eta^+_{new}(\mathfrak{B}_a^{\ell t_a(new)}(q))$.

Démonstration. Comme σ_{new} est synchrone, un unique job activé après la terminaison de $\tau_a^{\ell t_a(new)}$ peut interférer avec σ_a. Pour le $q^{\text{ème}}$ job, s'il n'y a aucun job de σ_{new} activé entre la terminaison de $\tau_a^{\ell t_a(new)}$ et celle de $\tau_a^{n_a}$ alors il y a $\eta_{new}^+(\mathfrak{B}_a^{\ell t_a(new)}(q))$ job de σ_{new} qui interfèrent avec $\mathfrak{B}_a^{\ell t_a(new)}(q)$. Sinon, un job supplémentaire de σ_{new} peut interférer avec le $q^{\text{ème}}$ job de σ_a. □

Nous établissons maintenant une propriété similaire pour gérer le cas où σ_{new} est asynchrone.

Propriété 123. *Soit σ_{new} une chaîne asynchrone et soit τ_a^k la dernière tâche de σ_a telle que $s_{new\to a[k..n_a]}^{head} \neq \emptyset$ alors, la condition sur C_{new} pour que σ_a soit garantie ordonnançable est :*

$$\forall q[1; K_a],\ C_{new} \leq \frac{S\ell_a(q)}{\eta_{new}^+(\mathfrak{B}_a^k(q))} \tag{5.15}$$

Démonstration. τ_a^k est la dernière tâche de σ_a telle que l'interférence d'un job de σ_{new} activé pendant l'exécution de τ_a^k n'est pas nulle. Ainsi tous les job de σ_{new} activés avant la terminaison du $q^{\text{ème}}$ job de τ_a^k pourront interférer avec σ_a. Or nous avons choisit de maximiser le WCET de la première tâche de σ_{new} ainsi chaque job de σ_{new} interfère à hauteur de son WCET. □

Les deux propriétés 122 et 123 permettent de trouver pour chaque chaîne σ_a de priorité inférieure à π_{new}, le WCET maximum de σ_{new} tel que σ_a reste ordonnançable. Ainsi, pour garantir l'ordonnançabilité de l'ensemble des chaînes de priorité inférieure à π_{new}, il faut prendre le WCET minimum.

Propriété 124. *Soit $C_{new}(\sigma_a)$ le WCET de σ_{new} tel que la chaîne de priorité inférieure σ_a soit garantie ordonnançable, alors la valeur C_{new} qui permet de garantir l'ordonnaçabilité de toutes les chaînes de priorité inférieure à π_{new} est :*

$$C_{new} = \min_{\sigma_a \in \ell p(a)} \{C_{new}(a)\} \tag{5.16}$$

Démonstration. Le WCET $C_{new}(a)$ permet de garantir l'ordonnançabilité de σ_a. Ainsi utiliser le WCET minimum permet de garantir l'ordonnançabilité de l'ensemble des chaînes de $\ell p(new)$. □

Il est à noter qu'une répartition différente des WCETs C_{new}^i ne change pas l'interférence des jobs de σ_{new} activés avant la terminaison de $\tau_a^{\ell t_a(new)}$ car chaque job de σ_{new} interfère entièrement avec σ_a. Ainsi, si pour une évolution le système n'est pas garanti ordonnançable, il faut raffiner le calcul de l'interférence des jobs de σ_{new} activés après la terminaison de $\tau_a^{\ell t_a(new)}$. Pour cela il faut rajouter des contraintes sur les possibles répartitions des WCETs pour identifier celles qui sont impossibles.

5.4.2 Ordonnançabilité des chaînes dans $hp(new)$

Nous souhaitons maintenant garantir l'ordonnançabilité des chaînes de priorité supérieure à σ_{new}. Nous commençons par garantir l'ordonnançabilité d'une chaîne $\sigma_a \in hp(new)$ et dans cet objectif, nous procédons ainsi :

1. Calcul de l'interférence de chaque chaîne $\sigma_b \in \ell p(a)$ sur σ_a.
2. Calcul de l'interférence de $\sigma_{new} \in \ell p(a)$ sur σ_a.
3. Calcul de la différence entre l'interférence subie par σ_a dans S et dans S_{new}.
4. Calcul de l'interférence additionnelle maximum de manière à garantir l'ordonnançabilité de σ_a dans S_{new}.

L'interférence de σ_{new} sur $\sigma_a \in hp(new)$ est calculée en utilisant l'équation (3.12). Pour rappel, l'interférence calculée via l'équation (3.12) est la combinaison des interférences de chacune des chaînes de $\ell p(a)$ telle que cette interférence sur σ_a soit maximum. En rajoutant une chaîne σ_{new} le segment critique qui maximise l'équation (3.12) peut changer. Une approche pour calculer l'interférence, dans S_{new}, des chaînes moins prioritaires sur σ_a est de recalculer la combinaison de l'interférence des chaînes de $\ell p(a)$ qui maximise l'interférence sur σ_a. Nous proposons cependant une autre approche qui consiste à calculer les changements qu'implique l'addition de σ_{new} au système. Pour cela, nous commençons par caractériser l'interférence d'une chaîne $\sigma_b \in \ell p(a)$ sur σ_a.

Pour rappel, l'équation (3.12) permet de majorer l'interférence de l'ensemble des chaînes moins prioritaires sur σ_a. Nous commençons par identifier l'interférence d'une chaîne σ_b moins prioritaire sur σ_a dans un préfixe de longueur Δ d'une σ_a-*busy-window*. Dans cet objectif, nous différencions plusieurs cas :

1. Si σ_b est synchrone et :
   (a) si σ_b est la chaîne [4] dont l'exécution du segment critique maximise l'équation (3.12) alors son interférence est :
   $$\overline{crit\mathcal{I}^{i,q}_{\substack{b\to a\\ sync}}}(\Delta) = C_{s^{crit}_{b\to a}} \tag{5.17}$$
   (b) sinon son interférence est :
   $$\overline{head\mathcal{I}^{i,q}_{\substack{b\to a\\ sync}}}(\Delta) = C_{s^{head}_{b\to a}} \tag{5.18}$$
2. Si σ_b est asynchrone :
   (a) si σ_b est la chaîne dont l'exécution du segment critique maximise l'équation (3.12) alors son interférence est :
   $$\overline{crit\mathcal{I}^{i,q}_{\substack{b\to a\\ async}}}(\Delta) = C_{s^{*}_{b\to a}} + hp\mathcal{I}^{i,q}_{\substack{s^{head}_{b\to a}\to a\\ async}}(\Delta) \tag{5.19}$$
   où
   $$s^{*}_{b\to a} = \begin{cases} s^{crit}_{b\to a} & \text{si le segment critique n'est pas circulaire} \\ s^{tail}_{b\to a} & \text{sinon} \end{cases}$$
   (b) sinon son interférence est :
   $$\overline{head\mathcal{I}^{i,q}_{\substack{b\to a\\ async}}}(\Delta) = hp\mathcal{I}^{i,q}_{\substack{s^{head}_{b\to a}\to a\\ async}}(\Delta) \tag{5.20}$$

4. S'il y a plusieurs chaînes dont l'exécution du segment critique maximise l'équation (3.12) alors il faut en choisir une.

Si σ_b est asynchrone l'interférence de σ_b sur σ_a dépend de la longueur d'un préfixe d'une σ_a-*busy-window*, et l'interférence maximum de σ_b sur σ_a telle que σ_a est ordonnançable peut être calculée en utilisant les *busy-times* $\mathfrak{B}_a^i(q)$. Si $\sigma_b \in \ell p(a)$ est synchrone alors l'interférence de cette chaîne sur σ_a ne dépend pas d'un intervalle de temps.

Contrairement aux calculs des interférences présentés dans les sections 3.4.1 et 3.5.2, nous ne souhaitons pas calculer une interférence qui prend en compte toutes les chaînes dans $\ell p(a)$, mais l'interférence d'une unique chaîne $\sigma_b \in \ell p(a)$ sur σ_a. Chaque chaîne $\sigma_b \in \ell p(a)$ interfère sur σ_a avec son segment critique (potentiellement circulaire) ou son segment de tête. Basé sur l'analyse de latence, nous identifions tout d'abord si σ_{new} interfère avec son segment critique ou son en-tête sur σ_a.

Propriété 125. *Soient σ_a, σ_b, et σ_{new} trois chaînes de S_{new} telles que $\pi_a > \pi_b$ et $\pi_a > \pi_{new}$. Supposons que l'interférence des chaînes dans $\ell p(a)$ sur σ_a soit maximale si σ_b interfère sur σ_a avec son segment critique. Alors, dans S_{new}, le segment critique de σ_b maximise les interférences sur σ_a si :*

$$\overline{crit\mathcal{I}^{i,q}_{\substack{b\to a\\ M_b}}}(\Delta) + \overline{head\mathcal{I}^{i,q}_{\substack{new\to a\\ M_a}}}(\Delta) \leq \overline{crit\mathcal{I}^{i,q}_{\substack{new\to a\\ M_a}}}(\Delta) + \overline{head\mathcal{I}^{i,q}_{\substack{b\to a\\ M_b}}}(\Delta) \tag{5.21}$$

avec M_b (resp. M_a) le modèle d'exécution (synchrone ou asynchrone) de la chaîne σ_b (resp. σ_a).

Démonstration. Dans S, la propriété 73 permet de calculer un majorant concernant l'interférence des chaînes moins prioritaires sur σ_a. Dans S_{new}, il y a une chaîne supplémentaire et dans ce cas, il faut vérifier si la combinaison qui maximise les interférences des chaînes moins prioritaires sur σ_a change. L'équation (5.21) permet de vérifier si l'interférence est toujours maximale si σ_b interfère à hauteur de son segment critique ou si l'interférence est plus grande lorsque σ_{new} interfère à hauteur de son segment critique. La chaîne σ_b maximise les interférences dans S et l'équation (5.21) permet de calculer le maximum entre σ_b et σ_{new} dans S_{new}. □

En fonction du segment de σ_{new} qui maximise les interférences sur la chaîne σ_a dans $hp(new)$, nous établissons un majorant sur le plus grand WCET du segment de tête ou du segment critique de σ_{new} sur σ_a.

Propriété 126. *Soient σ_a, σ_b et σ_{new} trois chaînes telles que $\pi_{new} < \pi_a$. L'interférence de σ_{new} sur le busy-time du $q^{ème}$ job de τ_a^i est majorée par :*

$$\overline{head\mathcal{I}^{i,q}_{\substack{new\to a\\ M_a}}}(\Delta) \textit{ si l'inégalité } (5.21) \textit{ est vérifiée} \tag{5.22}$$

$$\overline{crit\mathcal{I}^{i,q}_{\substack{new\to a\\ M_a}}}(\Delta) - \overline{crit\mathcal{I}^{i,q}_{\substack{b\to a\\ M_b}}}(\Delta) + \overline{head\mathcal{I}^{i,q}_{\substack{b\to a\\ M_b}}}(\Delta) \textit{ sinon} \tag{5.23}$$

avec M_b (resp. M_a) le modèle d'exécution (synchrone ou asynchrone) de la chaîne σ_b (resp. σ_a).

Démonstration. L'équation (5.21) permet d'identifier le segment de σ_{new} qui interfère sur σ_a tel que l'interférence des chaînes moins prioritaires sur σ_a soit maximum. Si l'inégalité (5.21) est vérifiée alors σ_{new} interfère avec son segment de tête. Sinon, σ_b interfère sur σ_a avec son segment de tête au lieu de son segment critique et σ_{new} interfère sur σ_a avec son segment critique. □

Nous utilisons l'interférence établie dans la propriété 126 pour garantir l'oronnançabilité de chaque chaîne dans $hp(new)$.

Propriété 127. *Soit le système $S_{new} = S \cup \{\sigma_{new}\}$. Toutes les chaînes dans $hp(new)$ sont garanties ordonnançable si $\forall \sigma_a \in hp(new)$:*

$$\overline{head\mathcal{I}^{i,q}_{\substack{new\to a\\ M_a}}}(\Delta) \leq S\ell_a \textit{ si l'inégalité (5.21) est vérifiée} \tag{5.24}$$

$$\overline{crit\mathcal{I}^{i,q}_{\substack{new\to a\\ M_a}}}(\Delta) - \overline{crit\mathcal{I}^{i,q}_{\substack{b\to a\\ M_b}}}(\Delta) + \overline{head\mathcal{I}^{i,q}_{\substack{b\to a\\ M_b}}}(\Delta) \leq S\ell_a \textit{ sinon} \tag{5.25}$$

avec σ_b la chaîne dont l'exécution du segment critique maximise les interférence sur σ_a dans S et M_b (resp. M_a) le modèle d'exécution (synchrone ou asynchrone) de la chaîne σ_b (resp. σ_a).

Démonstration. Ceci est une conséquence direct de la propriété 126. Si l'interférence additionnelle causée par l'ajout de σ_{new} est inférieure à la marge temporelle de chacune des chaînes alors les chaînes sont toutes ordonnançables dans σ_{new}. □

Il ne reste maintenant qu'à garantir l'ordonnançabilité de la chaîne σ_{new}.

5.4.3 Ordonnançabilité de la chaîne σ_{new}

Finalement, nous établissons une condition telle que σ_{new} soit ordonnançable. Premièrement, nous calculons la marge temporelle de σ_{new} en supposant que le WCET de chaque tâche de σ_{new} est infiniment petit (c'est à dire égal à ε). Le marge temporelle $S\ell_{new}(q)$ nous donne le temps additionnel maximum que le $q^{\text{ème}}$ job de σ_{new} peut subir tout en restant ordonnançable. Cela nous donne également la terminaison au plus tard de chaque tâche τ^i_{new} pour tout i dans $[\ell t_{new}, n_{new}]$, noté $\mathfrak{B}^i_{new}(q)$. Ensuite, comme pour les chaînes dans $\ell p(new)$ ou dans $hp(new)$, nous établissons des contraintes pour garantir l'ordonnançabilité de σ_{new}.

Propriété 128. *Soit S un système temps-réel ordonnançable et σ_{new} une chaîne. σ_{new} est garantie ordonnançable dans $S_{new} = S \cup \{\sigma_{new}\}$ si :*

$$\forall q \in [1, K_{new}], C_{new} \leq \frac{S\ell_{new}(q)}{\eta^+_{new}(\mathfrak{B}_{new}(q))} \tag{5.26}$$

Démonstration. Le WCET C_{new} est calculé de manière à répartir la marge temporelle entre tous les jobs de σ_{new}. □

Il est maintenant possible de garantir l'ordonnançabilité des chaînes de S_{new} en calculant le WCET C_{new} tel que les propriétés 124, 127 et 128 soient respectées.

5.5 Analyse de sensibilité

Dans cette section, nous souhaitons faire le lien entre l'analyse de flexibilité et l'analyse de sensibilité. Cela permet de comprendre comment utiliser l'analyse de flexibilité précédemment développée pour analyser la sensibilité d'un système. Cependant, par manque de temps nous ne développons pas formellement cette analyse de sensibilité.

L'analyse de flexibilité développée précédemment permet de résoudre certains problèmes concernant l'analyse de sensibilité des systèmes. Nous aurions également pu calculer la sensibilité d'un système pour en déduire sa flexibilité. En développant l'analyse de sensibilité d'abord, les contributions sont identiques car cette analyse n'existe pas pour les chaînes de tâches.

Considérons un système ordonnançable S et soit σ_a une chaîne de ce système pour laquelle nous souhaitons calculer la sensibilité. Cela signifie que nous souhaitons calculer les modifications des paramètres temporels de σ_a tel que le système reste ordonnançable.

Commençons par quelques rappels sur l'**analyse de la flexibilité** :

1. Elle prend en entrée un système ordonnançable et une chaîne σ_{new} définie par

$$\sigma_{new} = (\{(\pi^i_{new}, C^i_{new})\}_{1 \leq i \leq n_{new}}, \delta^-_{new}, D_{new})$$

2. Lors du calcul de la flexibilité de S, tous les paramètres sont fixes exceptés les WCETs $\{C^i_{new}\}_{1 \leq i \leq n_{new}}$ et nous calculons des contraintes sur ces WCETs pour garantir l'ordonnançabilité de S_{new}

Pour calculer la **sensibilité d'une chaîne $\sigma_a \in S$ par rapport à des variations de son WCET**, nous pouvons :

1. analyser la flexibilité du système $S \setminus \{\sigma_a\}$ pour obtenir le WCET maximum d'une chaîne σ_{new} tel que $S \setminus \{\sigma_a\} \cup \sigma_{new}$ soit ordonnançable ;
2. déduire le WCET maximum que peut avoir σ_a dans S tel que le système reste ordonnançable. En effet, le WCET maximum d'une chaîne σ_{new} tel que $S \setminus \{\sigma_a\} \cup \sigma_{new}$ soit ordonnançable est le WCET maximum de σ_a tel que S soit ordonnançable.

Nous pouvons également étendre les notions développées dans la section 5.4 pour calculer la sensibilité d'un système en considérant la modification simultanée de plusieurs paramètres temporels. Pour calculer la sensibilité d'un système par rapport à un ensemble de paramètres temporels $x_1, \ldots, x_n$:

1. Nous calculons la marge temporelle de chacune des chaînes du système.
2. De la même manière qu'on a calculé l'interférence additionnelle causée par l'ajout d'une chaîne, nous calculons l'interférence additionnelle causée par la modification de chacun des paramètres x_i, $i \in [1, n]$.
3. Pour chaque chaîne σ_a du système, nous établissons des contraintes pour que l'interférence causée par l'ensemble des modifications ne dépasse pas la marge $S\ell_a$.

Via l'analyse de flexibilité, nous pouvons, sans modification supplémentaire, calculer la sensibilité d'une chaîne par rapport à des modifications de son WCET. Et avec certaines adaptations, nous pouvons calculer la sensibilité d'un système dans le cas général où plusieurs paramètres de différents types sont modifiés simultanément dans un même système.

5.6 Analyse de flexibilité sur un cas d'étude

Dans cette section, nous illustrons sur un exemple comment l'analyse de flexibilité peut être utilisée. Pour cela, nous utilisons l'exemple 105 dans lequel toutes les chaînes sont considérées synchrones et dont les paramètres temporels sont définis dans le tableau 5.8.

chaîne	activation	Échéance relative	tâche	priorité	WCET
σ_e	$\delta_e^-(2) = 19$	18	τ_e^1	11	1
			τ_e^2	5	2
σ_d	$\delta_d^-(2) = 11$ $\delta_d^-(3) = 27$	17	τ_d^1	8	2
			τ_a^2	3	2
σ_a	$\delta_a^-(2) = 45$	36	τ_a^1	1	3
			τ_a^2	6	2
			τ_a^3	4	3
			τ_a^4	10	3
σ_{new}	$\delta_d^-(2) = 20$	23	τ_{new}^1	7	C_{new}^1
			τ_{new}^2	9	C_{new}^2
			τ_{new}^3	2	C_{new}^3

FIGURE 5.8 – Caractéristiques des tâches

Le chronogramme figure 5.9 représente une exécution du système défini dans le tableau 5.8. Sur cette figure, le WCET de chacune des tâches de σ_{new} est égal à 0 et on souhaite calculer la flexibilité de ce système, c'est à dire des contraintes sur les WCETs des tâches de σ_{new} telles que le système S_{new} soit ordonnançable.

Pour rappel, le calcul de la flexibilité d'un système se déroule en plusieurs étapes qui sont :

1. Calcul de la marge temporelle de chacun des jobs des chaînes du système dans un intervalle de temps donné.
2. Calcul de contraintes sur les WCETs des tâches de σ_{new} pour garantir l'ordonnançabilité :
   (a) des chaînes de priorité inférieure à σ_{new}, c'est à dire σ_a ;
   (b) des chaînes de priorité supérieure à σ_{new}, c'est à dire σ_d et σ_e ;
   (c) de la chaîne σ_{new}.

Analyse de la flexibilité de σ_a : Considérons le cas de la chaîne σ_a. On commence par **calculer la marge temporelle de** σ_a. Pour calculer la marge temporelle de σ_a, nous procèdons comme dans la section 5.3 et nous calculons le WCET maximum d'une chaîne σ_∞ qui est composée d'une unique tâche de priorité supérieure à toutes les autres tâches du système.

Pour calculer la marge temporelle de σ_a, nous utilisons le scénario et les équations qui permettent de calculer le majorant sur la latence au pire cas de σ_a. Ainsi, nous pouvons utiliser la figure 5.9 pour représenter graphiquement la marge temporelle de σ_a.

Dans n'importe quelle σ_a-*busy-window* il y a au maximum un job de σ_a et nous allons calculer la marge temporelle de ce job.

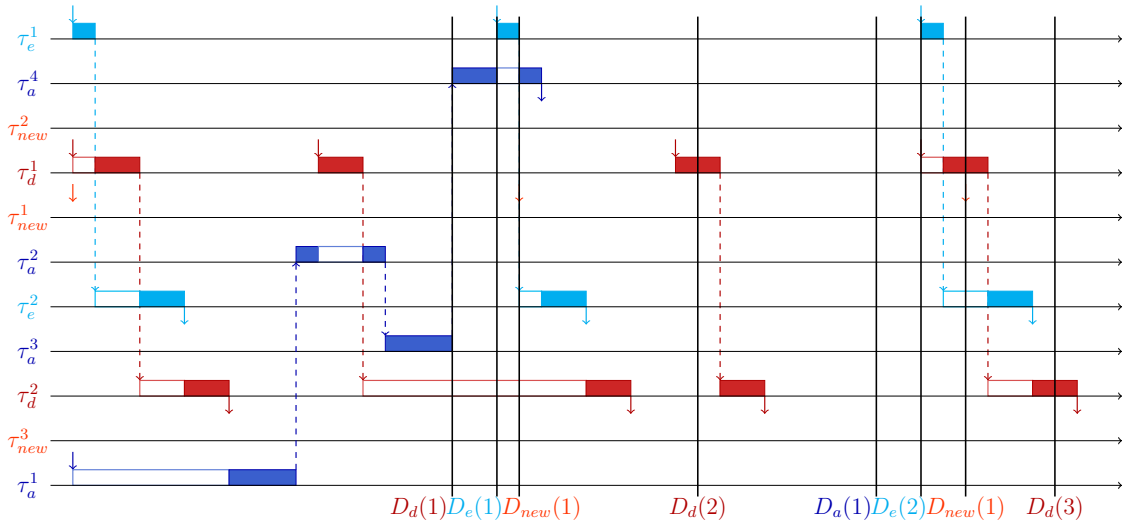

FIGURE 5.9 – Exemple d'une exécution du système S_{new} avec $C_{new} = 0$

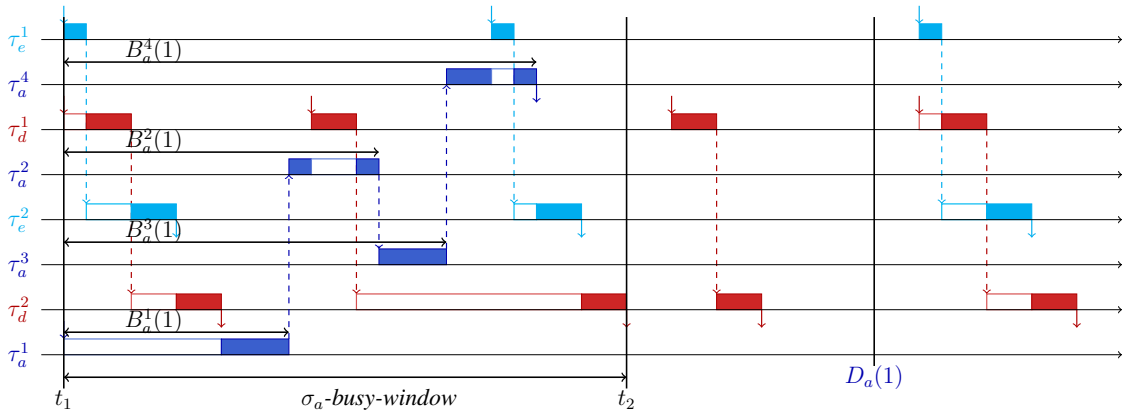

FIGURE 5.10 – Calcul d'un majorant sur la latence au pire cas de σ_a

Pour calculer la marge temporelle de σ_a, nous commençons par identifier les tâches échelons de σ_a :

1. Les tâches échelons de σ_a par rapport à σ_e sont τ_a^3 et τ_a^4. Autrement dit $\mathcal{T}_a^e = \{\tau_a^3, \tau_a^4\}$.
2. Les tâches échelons de σ_a par rapport à σ_d sont τ_a^1 et τ_a^3. Autrement dit $\mathcal{T}_a^d = \{\tau_a^1, \tau_a^3\}$.

Ainsi, l'ensemble des tâches échelons par rapport à σ_a est : $\mathcal{T}_a = \{\tau_a^1, \tau_a^3, \tau_a^4\}$

Pour calculer l'augmentation maximum de C_∞ telle que les interférences sur σ_a ne changent pas, nous calculons, via la propriété 112, les différences :

1. $\delta_e^-(2) - B_a^1(1) = 9$
2. $\delta_e^-(2) - B_a^3(1) = 2$
3. $\delta_d^-(2) - B_a^1(1) = 1$
4. $\delta_d^-(3) - B_a^3(1) = 10$
5. $\delta_d^-(3) - B_a^4(1) = 6$

La différence qui mène à la valeur minimum est $\delta_d^-(2) - B_a^1(1)$. Ainsi, nous pouvons augmenter C_∞ de $\delta_d^-(2) - B_a^1(1) = 1$ sans que l'interférence de σ_d et σ_e sur σ_a change. L'exécution correspondante est représentée sur la figure 5.11.

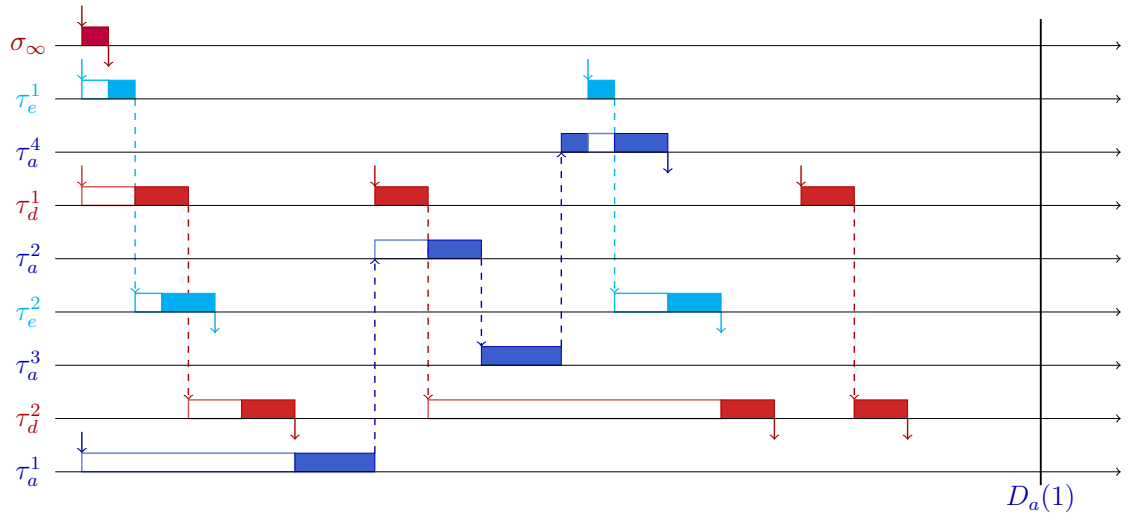

FIGURE 5.11 – Première itération pour calculer la marge temporelle du 1er job de σ_a avec $C_\infty = 1$

Sur la figure 5.11, on remarque que le second job de σ_d est activé à la terminaison de la première tâche de σ_a. Ce job de σ_d interfère sur σ_a à hauteur de C_d^1 et, en augmentant C_∞ de ε l'interférence de ce job de σ_d augmente pour atteindre $C_d = C_d^1 + C_d^2$. Il faut ré-analyser le système en calculant les *busy-times* avec l'équation (5.3) et en fixant $p = C_\infty + \varepsilon$.

Nous obtenons alors un nouvel ordonnançement du système et nous recalculons l'augmentation maximum de C_∞ telle que les interférences sur σ_a ne changent pas, via les différences :

1. $\delta_e^-(2) - B_a^1(1) = 8$
2. $\delta_e^-(2) - B_a^3(1) = 1$
3. $\delta_d^-(3) - B_a^1(1) = 16$
4. $\delta_d^-(3) - B_a^3(1) = 9$
5. $\delta_d^-(3) - B_a^4(1) = 5$

Dans ce cas, l'augmentation maximum de C_∞ qui ne change pas l'interférence sur σ_a est $\delta_e^-(2) - B_a^3(1)$.

On procède ainsi jusqu'à ce que la chaîne σ_a atteigne son échéance temporelle comme présenté sur la figure 5.12.

La figure 5.12 représente l'exécution du système telle que la marge temporelle de σ_a corresponde au WCET de σ_∞. Si on augmente encore C_∞ alors la chaîne σ_a n'est plus ordonnançable.

On utilise ensuite la marge temporelle pour **calculer des contraintes pour garantir l'ordonnançabilité de** σ_a.

Étant donné que σ_a est de priorité inférieure à π_{new}, c'est la propriété 123 qui permet de contraindre C_{new} pour garantir l'ordonnançabilité de σ_a. Dans notre exemple, nous

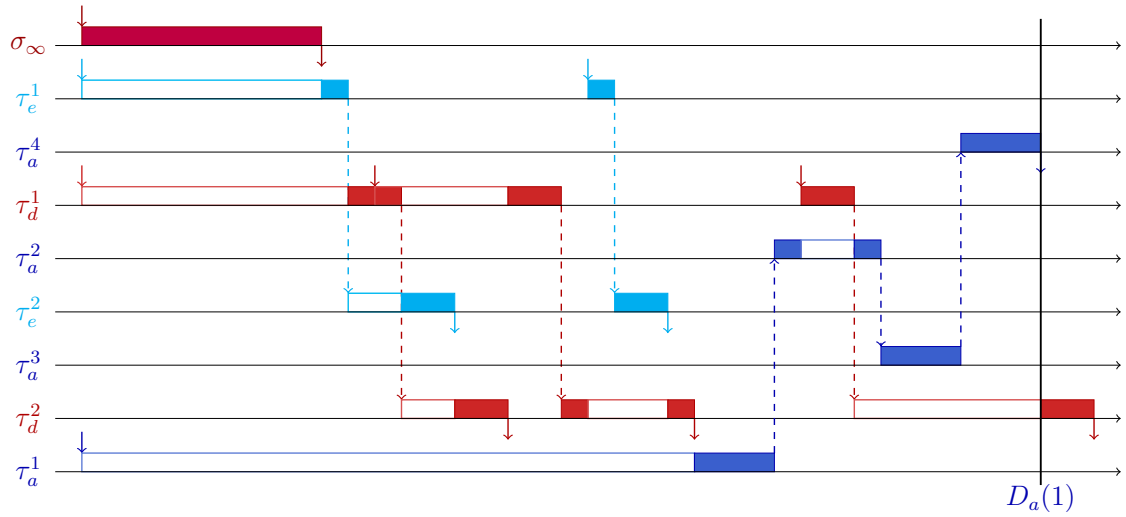

FIGURE 5.12 – Dernière itération pour calculer la marge temporelle du 1er job de σ_a avec $C_\infty = 9$

obtenons la contrainte suivante :

$$C_{new} \leq \frac{9}{2} \tag{5.27}$$

Cela signifie que si C_{new} est inférieur ou égal à 4.5 alors la chaîne σ_a est garantie ordonnançable dans S_{new}. Ce C_{new} peut être réparti librement sur toutes les tâches de σ_{new}.

Par exemple, si on choisit la répartition suivante $C_{new}^1 = 2$, $C_{new}^2 = 2$ et $C_{new}^3 = 0.5$, alors le scénario qui réalise le pire cas pour σ_a, dans ce cas particulier, est représenté sur la figure 5.13. Sur cette figure la latence de σ_a est inférieure à son échéance temporelle.

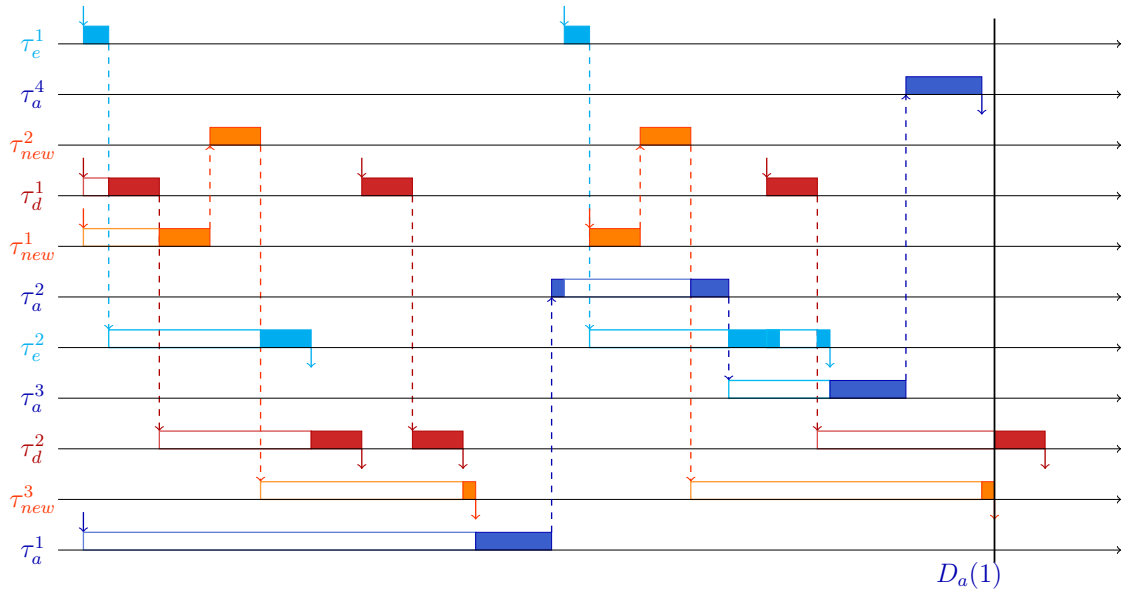

FIGURE 5.13 – Exemple d'une exécution du système S_{new} avec $C_{new} = 4.5$

De plus, dans le cas particulier de cette répartition de C_{new} sur les trois tâches de σ_{new}, nous constatons que C_{new} pourrait être plus grand. En effet, la terminaison du job de σ_a ne

coïncide pas avec son échéance temporelle. Ceci est dû au fait que la priorité de τ^3_{new} est inférieure à celle de τ^2_a, τ^3_a et τ^4_a, donc τ^3_{new} n'interfère pas autant sur σ_a que ce que faisait σ_a dans la figure 5.12. La durée entre la terminaison de σ_a et son échéance temporelle est de 0.5 et, comme il y a deux jobs de σ_{new} qui interfèrent avec σ_a alors C_{new} peut être augmenté de $\frac{0.5}{2}$.

Analyse de la flexibilité des autres chaînes Pour les autres chaînes (y compris σ_{new}), nous devons établir des contraintes sur C_{new} pour garantir leur ordonnançabilité dans S_{new}.

Le calcul de la marge temporelle des jobs de ces chaînes donne les résultats suivants :

1. Marge temporelle des jobs de σ_d :
   (a) $S\ell_d(1) = 2$
   (b) $S\ell_d(2) = 6$
   (c) $S\ell_d(3) = 15$
2. Marge temporelle des jobs de σ_e :
   (a) $S\ell_e(1) = 11$
   (b) $S\ell_e(2) = 27$
3. Marge temporelle des jobs de σ_{new} :
   (a) $S\ell_{new}(1) = 1$
   (b) $S\ell_{new}(2) = 15$

Via la propriété 125 nous identifions le segment de σ_{new} qui permet de majorer les interférence sur σ_d (resp. σ_e). Comme le segment critique de σ_{new} sur σ_d (resp. σ_e) est nul et que le segment de tête de σ_{new} sur σ_d (resp. σ_e) est également nul alors σ_{new} interfère sur σ_d (resp. σ_e) à hauteur de son segment de tête.

Ainsi c'est l'équation de la propriété 127 qui permet de majorer le WCET de σ_{new}. Les deux contraintes sont les suivantes :

1. $C^1_{new} + C^2_{new} \leq 2$
2. $C^1_{new} + C^2_{new} \leq 11$

Nous remarquons que dans ce cas une des contraintes ne peut pas être atteinte si on veut que S_{new} soit ordonnançable. En effet, la première contrainte est plus forte que la seconde.

Pour finir, pour que σ_{new} soit garantie ordonnançable, il faut que : $C_{new} \leq 1$ et $C_{new} \leq \frac{15}{2}$.

Dans le cas particulier de ce système, pour garantir l'ordonnançabilité des chaînes de S_{new}, il faut donc que C_{new} soit inférieur à 1. Il n'y a en définitive qu'une contrainte car si $C_{new} < 1$ alors les autres contraintes sont nécessairement respectées. Dans d'autres cas, plusieurs contraintes peuvent être à respecter simultanément. Considérons par exemple les deux contraintes suivantes : $C^1_{new} + C^2_{new} \leq 2$ et $C_{new} \leq 3$ alors les deux contraintes sont utiles pour répartir le WCET.

Si le système ne permet pas d'intégrer la fonctionnalité voulue, plusieurs options peuvent être à explorer. Tout d'abord, il peut être utile d'intégrer plus d'éléments du modèle lors de l'analyse comme, par exemple, les *offsets*. D'autre part, il est à noter que la contrainte la plus forte permet de garantir l'ordonnançabilité de σ_{new}. C'est donc une

chaîne qui limite la flexibilité du système. Ainsi, si on veut augmenter la flexibilité du système il faut changer le système (et donc son modèle) en diminuant les interférences sur la chaîne qui limite la flexibilité (σ_{new} sur cet exemple).

Chapitre 6

Chaîne d'outils

Dans ce chapitre, nous détaillons comment les contributions présentées dans cette thèse sont intégrées dans la chaîne d'outils utilisée pour le développement de systèmes à *Thales*.

La figure 6.1 représente la chaîne d'outils dans lequel s'inscrit cette thèse. Elle est composée de trois types d'outils.

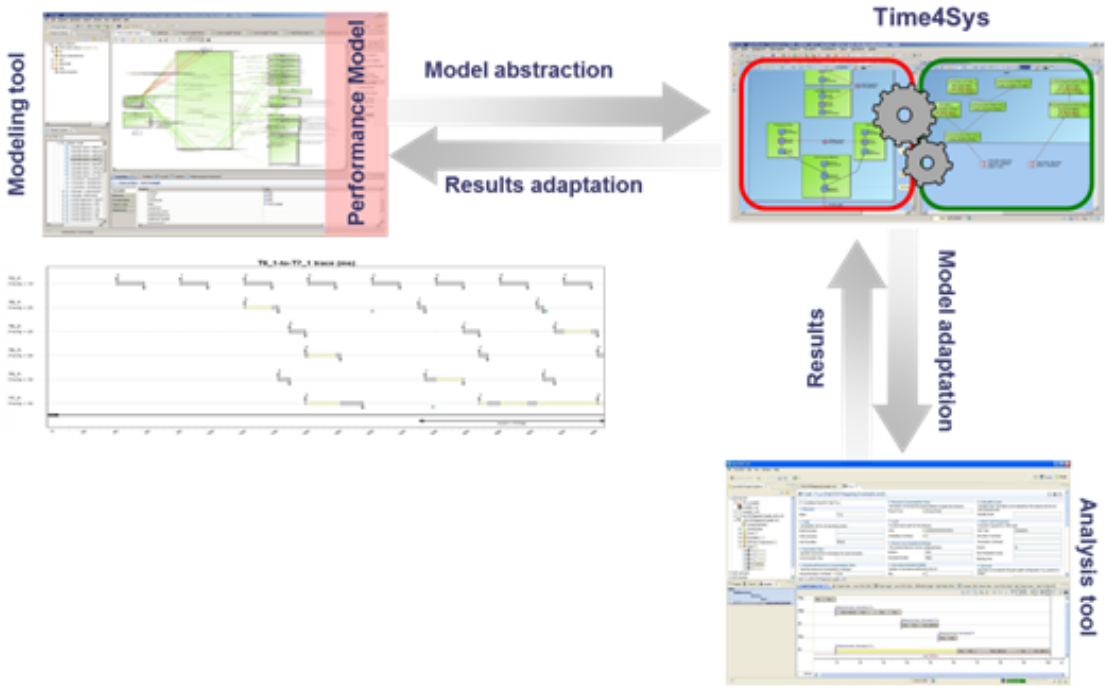

FIGURE 6.1 – Chaîne d'outils : modélisation fonctionnelle ↔ transformation ↔ analyse

Les différents outils permettant la modélisation des fonctionnalités d'un système et l'analyse de ce système sont les suivants :

1. Un outil de modélisation des fonctionnalités du système. Dans le cadre de cette thèse on a utilisé *Capella* cependant plusieurs outils peuvent être utilisés comme, par exemple, *Papyrus*. Via cet outil, les éléments fonctionnels sont modélisés en utilisant des standards comme *UML* dédié à la modélisation de logiciel, *SysML*

dédié à la modélisation de système ou encore *UML MARTE* qui permet de modéliser les éléments fonctionnels d'un système et en les enrichissant avec des paramètres temporels. On utilise ce dernier standard qui permet de modéliser les paramètres temporels qui sont utiles à l'analyse.

2. Un outil de transformation de systèmes. On utilise l'outil *Time4Sys* développé à *Thales* [1]. Le modèle fonctionnel peut ne pas être analysable et dans ce cas il est nécessaire de le transformer pour qu'il puisse être pris en charge par un outil d'analyse. De plus, si des transformations permettant de lier les outils de modélisation aux outils d'analyses existent et si ces transformations sont spécifiques aux deux modèles (de modélisation et d'analyse) alors, pour changer un des outils, il faut redévelopper toute la transformation, ce qui est contraignant. L'outil *Time4Sys* a pour avantages d'automatiser et de faciliter la transformation de manière à pouvoir changer les différents outils de modélisations et d'analyses.
3. Un outil d'analyse qui utilise le modèle fournit par *Time4Sys*. De nombreux outils existent tels que *Romeo*, *Cheddar*, *SymTA/S* ou *MAST*. Dans le cadre de cette thèse, on a développé des analyses dans l'outil académique *pyCPA* pour les tester puis certaines ont été implémentées dans l'outil d'analyse intégré à *Time4Sys* pour servir dans un cadre industriel.

Il est à noter que différentes fonctionnalités, autre que la transformation de modèles sont développées dans *Time4Sys* comme, par exemple, (i) des fonctions permettant de guider le concepteur dans le choix de l'outil d'analyse à utiliser selon les garanties souhaitées ou (ii) un outils d'analyse, déjà cité précédemment, dans lequel ont été implémentées différentes contributions de cette thèse.

L'outil de modélisation *Capella* que nous utilisons est inclut dans *PolarSys* qui est un ensemble de solutions, basés sur *Eclipse*, permettant de gérer le développement de systèmes durant leur cycle de vie. Quatre outils sont inclut dans *PolarSys* dont deux outils, *Kitalpha* et *Sirius*, pour définir les environnements et les représentations graphiques et deux outils de modélisation *Capella* et *Papyrus*.

Il est important que le modèle d'analyse permette de donner des garanties sur le modèle fonctionnel de départ. Si dans un outils d'analyse un système est garantie ordonnançable, il est nécessaire que le modèle fonctionnel le soit également.

Nos différentes contributions ont pour objectif d'étendre les outils d'analyses de manière à répondre à des besoins des concepteurs au niveau du modèle fonctionnel. Dans cet objectif, nous avons développé dans *pyCPA*, les calculs des majorants et des minorants sur la latence au pire cas dans le but d'évaluer ces analyses par l'expérimentation. Dans l'outil industriel intégré à *Time4Sys*, nous avons développé le calcul des minorants et des majorants dans le cas où les échéances sont contraintes et l'analyse de flexibilité pour les systèmes composés de tâches indépendantes. Ce dernier point fait parti de l'état l'art cependant, son implémentation à permis de quantifier l'interêt des concepteurs pour cette technologie.

Il est à noter que les développements sous *Time4Sys* ont été proportionnellement plus long que ceux sous *pyCPA* compte tenu des différentes interfaces à développer pour présenter les résultats.

1. Également développé dans le cadre du projet *Waruna*

Chapitre 7

Conclusion

Cette thèse a été initiée pour répondre à des problèmes rencontrés au sein de l'entreprise Thales. Les systèmes considérés sont des systèmes mono-processeurs sur lesquels sont allouées des chaînes de tâches. Sur les processeurs, les tâches sont ordonnancées sous ordonnanceur non péemptif à priorités fixes.

Pour ces systèmes, compte tenu des problèmes identifiés dans l'industrie, nous avons défini trois problèmes.

1. Dans le cadre de l'analyse d'ordonnançabilité, comment calculer le plus petit majorant possible sur la latence au pire cas des différentes chaînes d'un système ? Les majorants obtenus peuvent être sur-approximés et donc éloignés de la latence au pire cas. Dans le cas où les approximations sur les majorants sont importantes, il peut être impossible de garantir l'ordonnançabilité d'un système.
2. Une fois le majorant calculé, il est utile pour le concepteur d'évaluer la précision de ce majorant. Pour cela il faut calculer un minorant sur la latence au pire cas. De façon similaire au calcul du majorant, le calcul du minorant peut comporter d'importantes sous-approximations. De plus, il n'existe aucune autre méthode que la simulation pour calculer des minorants, et la précision d'un minorant obtenu par simulation dépend du temps passé à simuler le système.
3. Pour finir, les systèmes considérés sont sujets à de nombreuses évolutions. Afin de les faire évoluer, il est nécessaire d'anticiper au plus tôt en phase de conception l'impact qu'aura une évolution pour éviter les développements et les configurations qui mènent à un système non ordonnançable.

Contributions Dans cette thèse, compte tenu des problèmes soulevés, nos contributions sont les suivantes :

1. Nous avons développé une analyse de latence au pire cas de chaînes qui est plus précise et/ou plus générale que celles existantes dans l'état de l'art. Nous avons présenté l'approche générale que nous utilisons pour calculer un majorant sur la latence au pire cas d'une chaîne σ_a, puis nous avons calculé les interférences en provenances des différentes chaînes sur σ_a. Tout d'abord, nous avons séparé les calculs des interférences en fonction des échéances (contraintes ou arbitraires) car le cas où les échéances sont contraintes est plus simple. Pour finir, en fixant une priorité sur chaque chaîne d'un système, nous avons séparé le calcul des interférences en

fonction de la priorité de la chaîne qui interfère. Fixer une priorité pour chaque à pour avantage de rendre l'analyse plus facile à comprendre.

2. Nous avons présenté une méthode pour calculer des minorants sur la latence au pire cas en utilisant l'analyse d'ordonnançabilité. Les minorants que nous avons calculés sont définis comme des scénarios d'exécution réalisables compte tenu. Nous avons développé une approche qui nous permet de réutiliser les concepts développés pour le calcul des majorants. Pour calculer des minorants sur la latence au pire cas nous avons calculé des minorants uniquement sur l'interférence des chaînes moins prioritaires. Les minorant calculés nous ont permis d'évaluer la précision de nos majorants. Nous avons également comparer les minorants calculés et simulés sur des exemples cependant plus de tests sont nécessaires pour comparer les minorants calculés aux minorants simulés.
3. Nous avons définit la notion d'évolution compte tenu des besoins industriels. Lors de l'évolution d'un système, une nouvelle chaîne de tâches est rajoutée à ce système. Ensuite nous avons calculé la flexibilité, c'est à dire le WCET maximum de la nouvelle chaîne tel que le système après évolution reste ordonnançable. Pour cela nous avons étendu la notion de marge temporelle aux chaînes de tâches. Les nouveaux concepts développés pour l'analyse de flexibilité sont utiles pour étendre l'analyse de sensibilité au systèmes composés de chaînes de tâches.

Perspectives Compte tenu des objectifs réalisés plusieurs perspectives peuvent être envisagées. Tout d'abord, les analyses peuvent être étendues à des systèmes plus complexes avec :

1. des graphes de tâches,
2. plusieurs processeurs,
3. des dépendances de données (*cause-effect chains*),
4. des modèles d'activations incluant du *jitter* et des *offsets*.

Il serait intéressant de comparer les minorants simulés et calculés sur un grand nombre de systèmes. En effet, nous avons comparé les deux types de minorants sur deux exemples de systèmes. Dans ce cas il est impossible de conclure sur les avantages de l'analyse d'ordonnançabilité par rapport à la simulation pour obtenir des minorants sur la latence au pire cas.

Les minorants peuvent également être étendus à d'autres analyses. Il serait intéressant de calculer des minorants lorsque la précision des majorants est médiocre.

Concernant la partie flexibilité, il serait intéressant de considérer des problèmes plus complexes comme l'ajout de plusieurs chaînes à un système. Il faudrait également développer et formaliser l'analyse de sensibilité.

Pour finir, les analyses développées sont complexes et donc difficile à vérifier formellement. Il serait intéressant d'utiliser un outil de preuve interactif afin de prouver l'exactitude des équations développées dans cette thèse.

Bibliographie

[1] J. Schlatow and R. Ernst, "Response-time analysis for task chains in communicating threads," in *Real-Time Embedded Technology and Applications Symposium, RTAS'16*. IEEE, 2016, pp. 245–254. [Online]. Available : http://dx.doi.org/10.1109/RTAS.2016.7461359

[2] M. González Harbour, M. Klein, and J. Lehoczky, "Fixed priority scheduling periodic tasks with varying execution priority," in *Real-Time Systems Symposium, RTSS'91*. IEEE, Dec. 1991, pp. 116–128. [Online]. Available : https://doi.org/10.1109/REAL.1991.160365

[3] "Nextgen," https://www.faa.gov/nextgen/.

[4] "Sesar," https://www.sesarju.eu/.

[5] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis — the SymTA/S approach," *IEEE Proceedings Computers and Digital Techniques*, vol. 152, no. 2, pp. 148–166, Apr. 2005.

[6] K. Tindell, "Adding time-offsets to schedulability analysis," p. 28, 1994. [Online]. Available : http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.58.8815&rep=rep1&type=pdf

[7] J. Palencia and M. G. Harbour, "Schedulability analysis for tasks with static and dynamic offsets," in *Real-Time Systems Symposium, RTSS'98*. Madrid, Spain : IEEE, Dec. 1998, pp. 23–37. [Online]. Available : http://dx.doi.org/10.1109/REAL.1998.739728

[8] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *Proceedings of the 31st IEEE Real-Time Systems Symposium, RTSS 2010, San Diego, California, USA, November 30 - December 3, 2010*, 2010, pp. 259–268. [Online]. Available : https://doi.org/10.1109/RTSS.2010.42

[9] A. Saifullah, K. Agrawal, C. Lu, and C. D. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *Proceedings of the 32nd IEEE Real-Time Systems Symposium, RTSS 2011, Vienna, Austria, November 29 - December 2, 2011*, 2011, pp. 217–226. [Online]. Available : https://doi.org/10.1109/RTSS.2011.27

[10] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "End-to-end timing analysis of cause-effect chains in automotive embedded systems," *Journal of Systems Architecture - Embedded Systems Design*, vol. 80, pp. 104–113, 2017. [Online]. Available : https://doi.org/10.1016/j.sysarc.2017.09.004

[11] J. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines," in *Real-Time Systems Symposium, RTSS'90*. Lake Buena Vista (FL), USA : IEEE,

1990, pp. 201–209. [Online]. Available : http://dx.doi.org/10.1109/REAL.1990.128748

[12] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller area network (CAN) schedulability analysis : Refuted, revisited and revised," *Real-Time Systems*, vol. 35, no. 3, pp. 239–272, 2007. [Online]. Available : http://dx.doi.org/10.1007/s11241-007-9012-7

[13] M. Spuri, "Analysis of deadline scheduled real-time systems," 1996.

[14] S. Schliecker, "Performance analysis of multiprocessor real-time systems with shared resources," PhD Thesis, TU Braunschweig, Braunschweig, Germany, Jan. 2011. [Online]. Available : http://goo.gl/AwxhDS

[15] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, 2005. [Online]. Available : https://doi.org/10.1007/s11241-005-0507-9

[16] S. Kramer, D. Ziegenbein, and A. Hamann, "Real world automotive benchmarks for free," in *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015. [Online]. Available : https://www.ecrts.org/forum/viewtopic.php?f=20&t=23

[17] M. Chéramy, P.-E. Hladik, and A.-M. Déplanche, "SimSo : A Simulation Tool to Evaluate Real-Time Multiprocessor Scheduling Algorithms," in *5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2014, p. 6 p. [Online]. Available : https://hal.archives-ouvertes.fr/hal-01052651

[18] Y. T. R. Urunuela, A. Déplanche, "Storm a simulation tool for real-time multiprocessor scheduling evaluation," in *2010 IEEE 15th Conference on Emerging Technologies Factory Automation (ETFA 2010)*, 2010, pp. 1–8.

[19] R. Racu, M. Jersak, and R. Ernst, "Applying sensitivity analysis in real-time distributed systems," in *Real-Time Embedded Technology and Applications Symposium, RTAS'05*. IEEE, 2005, pp. 160–169.

[20] E. Bini, M. Di Natale, and G. Buttazzo, "Sensitivity analysis for fixed-priority real-time systems," in *ECRTS'06*. IEEE, 2006, pp. 13–22.

[21] S. Punnekkat, R. Davis, and A. Burns, "Sensitivity analysis of real-time task sets," in *ASIAN'97*, ser. LNCS, vol. 1345. Springer-Verlag, 1997, pp. 72–82.

[22] S. Vestal, "Fixed-priority sensitivity analysis for linear compute time models," pp. 308–317, 1994. [Online]. Available : http://dx.doi.org/10.1109/32.277577

[23] E. Bini, M. D. Natale, and G. Buttazzo, "Sensitivity analysis for fixed-priority real-time systems," pp. 5–30, 2007. [Online]. Available : http://dx.doi.org/10.1007/s11241-006-9010-1

[24] E. Bini and G. Buttazzo, "Schedulability analysis of periodic fixed priority systems," pp. 1462–1473, 2004. [Online]. Available : http://dx.doi.org/10.1109/TC.2004.103

[25] T. M. P. A. R. E. Moritz Neukirchner, Sophie Quinton, "Sensitivity analysis for arbitrary activation patterns in real-time systems," pp. 135–140, 2013. [Online]. Available : http://dx.doi.org/

[26] A. Hamann, R. Racu, and R. Ernst, “A formal approach to robustness maximization of complex heterogeneous embedded systems,” pp. 40–45, 2006. [Online]. Available : http://doi.acm.org/10.1145/1176254.1176267

[27] ——, “Multi-dimensional robustness optimization in heterogeneous distributed embedded systems,” pp. 269–280, 2007. [Online]. Available : http://dx.doi.org/10.1109/RTAS.2007.19

[28] A. Hamann, M. Jersak, K. Richter, and R. Ernst, “A framework for modular analysis and exploration of heterogeneous embedded systems,” pp. 101–137, 2006. [Online]. Available : http://dx.doi.org/10.1007/s11241-006-6884-x

[29] R. Racu, A. Hamann, and R. Ernst, “Sensitivity analysis of complex embedded real-time systems,” *Real-Time Syst.*, vol. 39, no. 1-3, pp. 31–72, 2008.

[30] ——, “A formal approach to multi-dimensional sensitivity analysis of embedded real-time systems,” pp. 2–12, 2006. [Online]. Available : http://dx.doi.org/10.1109/ECRTS.2006.5

[31] R. I. Davis, K. Tindell, and A. Burns, “Scheduling slack time in fixed priority pre-emptive systems,” in *Real-Time Systems Symposium, RTSS’93*, Dec. 1993, pp. 222–231. [Online]. Available : http://dx.doi.org/10.1109/REAL.1993.393496

[32] J. P. Lehoczky and S. R. Thuel, “Scheduling periodic and aperiodic tasks using the slack stealing algorithm,” pp. 175–198, 1995. [Online]. Available : http://dl.acm.org/citation.cfm?id=207721.207729

[33] S. Ramos-Thuel and J. P. Lehoczky, “On-line scheduling of hard deadline aperiodic tasks in fixed-priority systems,” in *Proceedings of the Real-Time Systems Symposium*. IEEE, 1993, pp. 160–171. [Online]. Available : http://dx.doi.org/10.1109/REAL.1993.393504

[34] S. R. Thuel and J. P. Lehoczky, “Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing,” in *Proceedings of the 15th Real-Time Systems Symposium*. IEEE, 1994, pp. 22–33. [Online]. Available : http://dx.doi.org/10.1109/REAL.1994.342733

[35] T.-S. Tia, J. W. S. Liu, and M. Shankar, “Algorithms and optimality of scheduling soft aperiodic requests in fixed-priority preemptive systems,” pp. 22–43, 1996. [Online]. Available : http://dx.doi.org/10.1007/BF00357882

[36] Q. Zhu, Y. Yang, E. Scholte, M. Di Natale, and A. Sangiovanni-Vincentelli, “Optimizing extensibility in hard real-time distributed systems,” in *Real-Time Embedded Technology and Applications Symposium, RTAS’09*. IEEE, 2009, pp. 275–284.

[37] R. Davis and A. Burns, “Robust priority assignment for fixed priority real-time systems,” in *Real-Time Systems Symposium, RTSS’07*. IEEE, Dec. 2007, pp. 3–14.

[38] I. Bate and P. Emberson, “Incorporating scenarios and heuristics to improve flexibility in real-time embedded systems,” in *Real-Time Embedded Technology and Applications Symposium, RTAS’06*. San Jose (CA), USA : IEEE, 2006, pp. 221–230.

[39] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” pp. 46–61, 1973. [Online]. Available : http://doi.acm.org/10.1145/321738.321743