Une infrastructure de simulation modulaire pour l'évaluation de performances de systèmes temps-réel

David Decotigny

N° d'ordre : **2831**
de la thèse

THÈSE

présentée

DEVANT L'UNIVERSITÉ DE RENNES 1

pour obtenir

le grade de : ***DOCTEUR DE L'UNIVERSITÉ DE RENNES 1***

Mention : Informatique

PAR

David Decotigny

Équipe d'accueil : Aces, Irisa
École Doctorale : Matisse
Composante universitaire : Ifsic/Irisa

TITRE DE LA THÈSE :

Une infrastructure de simulation modulaire pour l'évaluation de performances de systèmes temps-réel

Soutenue le 7 Avril 2003 devant la commission d'examen

COMPOSITION DU JURY :

MM. :	Françoise	Simonot-Lion	Rapporteurs
	Francis	Cottet	
MM. :	Jean	Camillerapp	Examinateurs
	Claude	Kaiser	
	Michel	Banâtre	
	Isabelle	Puaut	

Remerciements

J'ai pu réaliser cette thèse au sein de l'équipe du projet Solidor, devenu Aces (Ambient Computing and Embedded Systems), à l'Irisa (Institut de Recherche en Informatique et Systèmes Aléatoires) à Rennes.

À Mme Isabelle Puaut, qui a encadré et dirigé ma thèse, je tiens à témoigner toute ma gratitude pour ses remarques constructives et ses commentaires pertinents qui m'ont été d'une aide inestimable pendant ma thèse, et avant, pendant mon DEA. Je la remercie aussi très chaleureusement pour les discussions scientifiques passionnantes que nous avons partagées. Je lui suis également très reconnaissant pour les nombreuses heures que les relectures multiples de ce document lui ont volées : que ses précieux "gribouillis" (qui ne méritent d'ailleurs pas ce nom indigne) soient ici salués.

Je tiens à remercier vivement M. Michel Banâtre, responsable scientifique du projet Aces, de m'avoir fait confiance pour ce travail de thèse qu'il a co-dirigé avec Isabelle, et pour m'avoir permis de le mener à bien dans les meilleures conditions.

Je remercie Mme Françoise Simonot-Lion, Maître de conférences à l'Institut National Polytechnique de Lorraine, et M. Francis Cottet, Professeur d'Université à l'École Nationale Supérieure de Mécanique et d'Aérotechnique à Poitiers, d'avoir très aimablement accepté la charge de rapporteur. Je remercie M. Jean Camillerapp, Professeur à l'Institut National des Sciences Appliquées de Rennes, qui m'a fait l'honneur d'être le président du jury de ma thèse.

Je remercie également la DGA, et tout particulièrement mon correspondant DGA, M. Philippe Sarazin, d'avoir financé ma thèse par une bourse DGA/CNRS.

Je souhaite aussi saluer le personnel de l'Irisa, en particulier l'atelier et la bibliothèque, pour les moyens matériels qu'ils mettent à notre disposition, et pour leurs compétences et leur disponibilité.

Je tiens à remercier tous les membres des équipes Solidor/Aces, anciens ou actuels, pour la bonne humeur qu'ils contribuent tous à entretenir. Je tiens également à saluer ici Mme Emmanuelle Anceaume, qui m'a co-encadré avec Isabelle pendant mon DEA, et qui m'a initié avec Isabelle au monde de la recherche. Sur un plan plus personnel, j'adresse tout particulièrement un sincère merci à Gilbert Cabillic, Pascal Chevochot, Antoine Colin, Jean-Philippe Lesot, Isabelle Puaut et Stéphane Tudoret, la plupart anciens de la "Hades team", pour m'avoir témoigné leur chaleureuse sympathie à de multiples occasions.

Un grand merci aussi aux anciens de Télécom Bretagne pour leur précieuse amitié et leurs encouragements, en particulier David Mentré & Fabienne Pottier, Gaël Mahé, Mikaël Salaün, Karine Amis, Stéphane Chazelas, Clément Barry, et bien sûr mes amis de "la troupe du Studio". Un grand merci également à Thomas Petazzoni et Julien Munier pour m'avoir accordé leur amitié, et pour toutes les discussions animées que nous avons eues, et que nous continuons d'avoir, au sujet du projet Kos.

Enfin, je dois cette thèse à toute ma famille : un très affectueux merci à elle, qui a toujours su me faire confiance et me soutenir sans compter dans mes projets.

Table des matières

Introduction

Définition du *temps-réel* adoptée

La notion de "*temps-réel*" suggère bien évidemment celle de *temps*, dont l'acception est très large. Intuitivement, les systèmes informatiques "temps-réel" sont ceux pour lesquels le comportement temporel, non seulement en termes de séquence d'opérations (propriété logique), mais également en termes de quantification de l'écoulement de ces opérations (propriété physique), a une importance.

Dans le présent travail, nous reposons sur la définition du temps-réel suivante, largement adoptée dans le domaine [Sta88] : "*la correction du système ne dépend pas seulement des résultats logiques des traitements, mais dépend en plus de la date à laquelle ces résultats sont produits*". Cette définition implique que des *contraintes temporelles* sont à respecter, par exemple les échéances temporelles des traitements. Elle précise *i)* que ces contraintes sont relatives au temps en tant que *grandeur physique mesurable*, et *ii)* qu'elles font partie de la *spécification* du système à implanter, c'est-à-dire des propriétés que celui-ci doit respecter. En outre, elle signifie que la seule rapidité moyenne d'exécution du logiciel ne conditionne pas la validité du système. Elle suggère que dans tous les cas, même les *pires*, toutes les contraintes temporelles doivent être respectées, sans quoi le système est *défaillant*.

Dans ce contexte, la problématique qui nous intéresse est celle de la conception et de l'évaluation de systèmes informatiques qui sont soumis à des contraintes temporelles en plus des contraintes de correction fonctionnelles usuelles. Parmi les contraintes temporelles courantes, citons par exemple les échéances temporelles strictes des traitements, ou leur gigue de démarrage ; nous reviendrons sur ces notions dans la suite du document.

Classification sommaire des systèmes informatiques temps-réel

L'utilisation de l'outil informatique apparaît de plus en plus fréquemment dans les domaines où l'interaction avec l'environnement constitue une raison d'être essentielle du système. Un intérêt de recourir à l'informatique dans ce genre de système est dû au fait que les fonctionnalités offertes le sont à coût moindre (à fonctionnalités équivalentes) en termes de temps de développement, de temps et de coûts de fabrication, d'encom-

brement ou de poids, par rapport aux solutions électroniques ou mécaniques. En outre, l'ajout de nouvelles fonctionnalités, la constitution de mises à jour, ou la personnalisation du produit se révèlent souvent être des tâches moins lourdes.

Temps-réel critique

On trouve aujourd'hui des gros systèmes informatiques temps-réel dans des domaines aussi divers que l'aéronautique, l'aérospatiale, les transports ferroviaires, le contrôle de procédés industriels, la supervision de centrales nucléaires, la gestion de réseaux de télécommunication, la gestion de salles de marchés, ou la télémédecine. On trouve aussi de beaucoup plus petits systèmes (en termes de taille, de puissance du processeur, de capacité mémoire) *embarqués*, par exemple dans l'automobile (système de contrôle des freins, du moteur).

Pour ces exemples, une des caractéristiques est que le système informatique se voit confier une grande responsabilité en termes de vies humaines, de conséquences sur l'environnement, ou de conséquences économiques. On parle alors de systèmes *critiques*, qui sont soumis à des contraintes de fiabilité. Cette caractéristique amène les problématiques orthogonales de sûreté de fonctionnement et de tolérance aux fautes [CP99b, CP99a] que nous ne détaillerons pas dans ce travail.

Temps-réel strict et temps-réel souple

La majorité des systèmes temps-réel critiques est exclusivement constituée de traitements qui ont des contraintes temporelles *strictes* : on parle de systèmes *temps-réel strict*. C'est-à-dire qu'en condition nominale de fonctionnement du système, **tous** les traitements du système doivent impérativement respecter toutes leurs contraintes temporelles ; on parle alors de traitements *temps-réel strict* ou *dur* (*hard* en anglais). Ceci suppose deux choses : *i)* qu'on est capable de définir les conditions de fonctionnement nominales en termes d'hypothèses sur l'environnement avec lequel le système interagit ; *ii)* qu'on est capable de garantir avant exécution que tous les scénarios d'exécution possibles dans ces conditions respecteront leurs contraintes temporelles. Ceci suppose à son tour qu'on puisse extraire ou disposer de suffisamment d'informations sur le système pour déterminer tous les scénarios possibles.

Une autre classe de systèmes est moins exigeante quant au respect absolu de *toutes* les contraintes temporelles. Les systèmes de cette classe, dits *temps-réel souple* (*soft real-time* en anglais), peuvent souffrir un taux "acceptable" de *fautes temporelles* (non respect transitoire des contraintes) de la part d'une partie des traitements (eux-mêmes dits "*temps-réel souple*"), et sans que cela ait des conséquences catastrophiques. Cette classe comprend, entre autres, les systèmes où la qualité est appréciée par nos sens comme par exemple les systèmes et applications multimédia (téléphonie, vidéo, rendu visuel interactif par exemple). Pour ces systèmes, la mesure du respect des contraintes temporelles prend la forme d'une donnée probabiliste : la *qualité de service* relative à un service particulier (nombre d'images ou nombre d'échantillons sonores rendus par seconde, par exemple), ou relative au comportement du système dans son ensemble (nombre de traitements qui ont pu être rendus dans les temps, tous services confondus,

par exemple), ou les deux combinés. Une problématique de cette classe de systèmes est d'évaluer la qualité de service, avant ou pendant le fonctionnement, que le système offre ou va pouvoir offrir en cours de fonctionnement, en fonction des caractéristiques de l'environnement et du système.

Il est possible d'avoir des systèmes temps-réel strict non critiques (échantillonneur audio par exemple), ou des systèmes temps-réel critiques qui contiennent des traitements temps-réel souple (supervision de chaîne de montage – temps-réel critique – avec affichage de statistiques à but purement informatif – temps-réel souple – par exemple).

Présentation des travaux réalisés

Objectifs et démarche

Notre travail s'intéresse à l'*évaluation* de systèmes temps-réel ayant, ou non, une composante temps-réel strict. Dans le cadre de systèmes temps-réel strict, cette notion d'évaluation recouvre celle de *vérification* : la caractéristique fondamentale évaluée est le respect, ou le non-respect, de toutes les contraintes temporelles spécifiées, et est le plus souvent établie par analyse et preuves. Dans le cadre de systèmes temps-réel quelconques (temps-réel strict ou non), l'évaluation est le plus souvent d'ordre numérique, et résulte d'une analyse ou d'observations du comportement du système.

Une première partie de notre travail a consisté en l'étude d'approches existantes pour mener de telles évaluations, sous la forme d'une série de méthodes et d'outils (formels, logiciels, et parfois matériels) adaptés au domaine de l'informatique temps-réel. Plus précisément, nous nous sommes intéressés :

- aux méthodes qui permettent de *modéliser* un système temps-réel, et aux formalismes de modélisation associés. Nous nous sommes attachés aux approches qui ne s'éloignent pas trop des implantations effectives concrètes, afin de réduire les erreurs de représentativité du modèle dues à l'écart entre la modélisation et l'implantation. Nous avons limité cette présentation à la modélisation de systèmes centralisés, en ne décrivant pas les aspects liés à la problématique de modélisation de réseaux pour le distribué ;
- aux méthodes d'*évaluation* d'un système à partir du modèle, ou de l'implantation concrète, partagées entre deux voies : par analyse et preuves, et par exécution, qu'elle soit réelle ou simulée. L'élément central étudié dans cette optique est l'*ordonnancement*, c'est-à-dire l'arbitrage entre les besoins en ressources et les ressources disponibles, en tenant compte des contraintes temporelles spécifiées. Là encore, nous avons privilégié une présentation des méthodes d'évaluation de systèmes temps-réel centralisés.

Contributions

Ce travail a conduit à la réalisation d'un outil d'évaluation, Artisst, ainsi qu'à la proposition de règles de conception en ordonnancement, accompagnées d'évaluations expérimentales de leur impact à l'aide de l'outil.

Un outil de simulation : Artisst

Cette démarche nous a permis de proposer l'outil Artisst (pour *"*Artisst *is a* **R***eal-*__Ti__*me* **S***ystem* **S***imulation* **T***ool"*), pour la simulation visant à l'évaluation de systèmes temps-réel (système d'exploitation et application) à partir d'un modèle, ou à partir d'une implantation existante. Les modèles simulés sont essentiellement de type centralisé, mais nous montrerons cependant comment l'outil peut réaliser la simulation de systèmes temps-réel distribués et du réseau sous-jacent. Cet outil permet l'évaluation de systèmes temps-réel dans des environnements simulés aussi variés que possibles, et avec la prise en compte de coûts d'exécution de l'application et du système d'exploitation à un niveau de détails aussi fin que possible.

Sa capacité de personnalisation et d'extension, grâce à sa conception modulaire orientée-objet, lui permet en particulier d'être utilisé pour un grand nombre des solutions d'ordonnancement étudiées (par exemple pour l'évaluation du comportement temporel de différents ordonnanceurs), ou pour l'évaluation d'applications en présence de différents systèmes d'exploitation.

Travaux en ordonnancement temps-réel

Au cours de la réalisation et de la validation du simulateur, nous avons implanté de nombreux algorithmes d'ordonnancement de la littérature, et proposé des infrastructures génériques de simulation (familles d'ordonnancement), permettant la spécialisation de toute une série d'algorithmes d'ordonnancement à priorité pour différentes affectations (statiques et dynamiques) des priorités des tâches.

Nous avons également étudié l'influence de la granularité de l'horloge système sur quelques ordonnanceurs avec garantie en ligne de tâches apériodiques. Ceci nous a permis de donner quelques recommandations sur la prise en compte correcte de la granularité de l'horloge dans l'implantation d'un tel algorithme d'ordonnancement. Nous avons soumis une série d'ordonnanceurs respectant ces recommandations, à une évaluation par Artisst.

Organisation du document

Le document se compose de trois parties. La première partie décrit le formalisme dans lequel nous travaillons, présente les éléments de terminologie, puis introduit quelques méthodes d'évaluation de systèmes temps-réel dans ce formalisme. Deux voies de vérification et d'évaluation sont ensuite détaillées : l'analyse d'ordonnancement, puis la simulation.

La deuxième partie présente la plate-forme de simulation Artisst. Après avoir indiqué les motivations qui ont conduit à sa réalisation, nous en donnons un aperçu général avant de détailler sa structure interne. Nous décrivons ensuite les ordonnanceurs qui sont implantés, dont une partie se présente sous la forme de familles d'ordonnanceurs génériques paramétrables. Dans ce chapitre, nous donnons également quelques

recommandations pour la prise en compte de la granularité de l'horloge système dans les ordonnanceurs présentés.

La troisième partie propose une évaluation de différents ordonnanceurs avec garantie en-ligne de tâches temps-réel apériodiques. Ensuite nous proposons une évaluation de leur comportement en fonction de la granularité de l'horloge du système, sans et avec prise en compte des surcoûts d'exécution du système d'exploitation.

En fin de document, l'annexe est consacrée à une description plus technique de la plate-forme ARTISST, et à des exemples de réalisations l'utilisant.

première partie

Méthodes d'évaluation pour le temps-réel

Dans cette partie, nous présentons rapidement quelques formalismes de modélisation de systèmes temps-réel, avant de préciser celui sur lequel repose la suite du document (chapitre 1). Nous limitons cette présentation à la modélisation de systèmes centralisés : nous ne décrivons pas les aspects liés à la modélisation de réseaux.

Nous introduisons ensuite les approches courantes pour l'évaluation de systèmes temps-réel, à partir du modèle ou à partir de l'implantation effective (chapitre 2).

Enfin, les chapitres 3 et 4 détaillent deux de ces approches, en présentant quelques travaux dans ces domaines : les travaux en évaluation par analyse d'ordonnancement pour les systèmes temps-réel centralisés, et ceux en évaluation par simulation pour l'évaluation de systèmes temps-réel. Cette étude sert de base aux contributions proposées dans ce document, et présentées dans les parties suivantes.

Chapitre 1

Modèles et contraintes de conception pour le temps-réel

Le processus de fabrication d'un système informatique se décompose généralement en quatre étapes : la phase de *spécification* (quelles bonnes propriétés le système doit satisfaire), de *conception* (définition de la composition et de l'architecture du système), de *réalisation* (programmation ou câblage, selon la nature matérielle et/ou logicielle du système), et d'*intégration* du logiciel. Ce processus est jalonné de phases de *vérification* du système, qui s'occupent de confronter les spécifications au système tel qu'il est perçu à ce stade, et qui est représenté par un *modèle* abstrait (le plus souvent) ou concret (implantation d'une maquette opérationnelle ou simulée) capturant ses caractéristiques statiques, dynamiques, et temporelles. Les vérifications visent à prouver que le modèle respecte les propriétés spécifiées.

Dans notre travail, nous ne nous intéressons pas uniquement à la vérification de systèmes temps-réel, mais plus largement à leur *évaluation* : il s'agit d'étudier un système donné, représenté par un modèle, et d'en extraire un ensemble de mesures ou une mesure synthétique. Par exemple, pour un système temps-réel strict, l'évaluation peut consister en une analyse formelle d'un modèle abstrait du système, qui renvoie vrai ou faux suivant qu'on peut garantir que le système respectera toutes les contraintes temporelles (il s'agit dans ce cas d'une vérification). Ou, pour un système temps-réel quelconque, l'évaluation peut consister en l'observation du comportement d'un modèle concret du système pour en extraire des mesures (nombre de tâches ayant dépassé leurs contraintes temporelles par exemple).

Dans ce chapitre, nous nous intéressons à la modélisation du système et des contraintes temporelles, formant le fondement des démarches d'évaluation, en nous limitant à la modélisation de systèmes centralisés. Nous présentons différentes approches pour la modélisation de systèmes temps-réel, avant de détailler le modèle de système sur lequel repose toute la suite de notre travail. Nous préciserons les constituants de ce modèle, ainsi que les caractéristiques courantes qui les définissent, et que le concepteur de système temps-réel doit déterminer lors des phases de spécification et de conception.

1.1 Panorama des modèles de systèmes faisant intervenir le temps

Plusieurs formalismes de modélisation existent, plus ou moins contraignants vis-à-vis des hypothèses sur le système et du niveau d'abstraction. Nous en citons ici quelques uns qui font intervenir le temps [Kai01, Ost92, Kat98], sans prétention d'exhaustivité. Dans les formalismes qui suivent, sauf exception, le temps est une grandeur numérique (d'un espace discret ou dense) globale à tout le système.

Logiques et algèbres

Une première classe de formalismes pour la modélisation intègre la formalisation des spécifications. Ils ont pour objectif de démontrer formellement les propriétés spécifiées à partir du modèle de système, propriétés qui vont généralement au delà des propriétés temporelles :

- Les *logiques temporelles* (RTTL par exemple) étendent la logique propositionnelle en rajoutant des opérateurs (*henceforth* ou désormais, *eventually* ou finalement, ...) qui incluent la notion de temps discret, ou dense suivant les logiques. Elles permettent de montrer des propriétés temporelles qualitatives (équité, vivacité, sûreté par exemple), et quantitatives (vérification de telle propriété à tel instant, ou après telle attente), en exploitant les propriétés des *états* successifs du système.
- Les *algèbres de processus* (Timed CSP, Timed Lotos, CCSR par exemple). Il s'agit de décrire le comportement de processus à l'aide d'opérateurs de composition (parallélisme, séquence, occupation temporelle) et de règles syntaxiques, en fonction des *événements* qui provoquent des changements d'état dans le système, et de vérifier les propriétés fonctionnelles du système par équivalence et bissimulation, et en fonction des caractéristiques temporelles des processus modélisés.

Ces deux classes sont d'un niveau d'abstraction élevé, ce qui les rend parfois très éloignées d'une implantation concrète, et qui rend également la spécification de contraintes temporelles liées au temps-réel difficile. Elles souffrent d'autre part d'un problème d'efficacité, car elles conduisent, pour des modèles complexes, à une explosion combinatoire de la complexité de la vérification.

Méthodes structurelles

On peut distinguer une deuxième classe de formalismes adaptés aux preuves de propriétés formelles, qui vont aussi au-delà des preuves de propriétés temporelles. Ils sont en général distincts du formalisme d'expression des propriétés à vérifier, et s'intéressent à structurer le modèle à l'aide d'une sémantique bien définie, souvent associée à une représentation graphique.

Il existe un premier formalisme qui fait intervenir les ressources disponibles, contraintes par leur capacité disponible utilisable à chaque instant : ce sont les systèmes à *réseaux de files d'attente* (*queueing systems*). Chaque ressource est associée à une *capacité* et à une *file* de demandes correspondant aux demandes en cours compte-tenu des

capacités de la ressource, les files communiquant entre elles à la manière d'un réseau. Ce formalisme, et les travaux théoriques associés, permet d'évaluer la durée moyenne de service des demandes, ou le nombre moyen de demandes dans les différentes files. Il est naturellement bien adapté à la modélisation de réseaux, ou de serveur de calculs. Mais il permet difficilement d'exprimer des contraintes temporelles autres que la limitation des capacités des ressources à chaque instant.

De façon orthogonale, on peut distinguer les formalismes suivants qui permettent l'expression de l'écoulement du temps :

- Les modèles de *systèmes réactifs synchrones* (Esterel, Statecharts, Lustre, Signal, par exemple)[BGG+94, BG92, HCRP91]. Dans ces modèles, le système est constitué d'une suite infinie d'*actions* composées (*i.e.* des traitements), chaque action composée étant constituée de une ou plusieurs actions qui ont lieu *simultanément* sur l'échelle de temps manipulée, et qui peuvent venir en réaction à un événement de l'environnement. Le temps manipulé est explicitement un *temps logique*, et peut être relié au temps physique par l'intermédiaire d'horloges. À partir du modèle défini, en général sous la forme d'une composition hiérarchique de modules (graphes ou systèmes d'équations), il est possible de générer à la fois un automate, et l'implantation concrète (programme ou circuit). Des propriétés peuvent être exprimées dans le même formalisme (notion d'*observateur*), ce qui permet de vérifier la correction d'un modèle vis-à-vis de ces propriétés (états atteignables, invariants, vivacité, sûreté le plus souvent), en tenant compte des caractéristiques de l'environnement avec lequel il interagit. Dans d'autres cas, il est possible d'utiliser des outils d'aide à la preuve pour vérifier le respect de propriétés par l'automate du modèle. Cette approche s'apparente sensiblement à la conception de circuits électroniques logiques synchrones. Ces formalismes sont bien adaptés à la modélisation de contrôles de procédés (systèmes à boucle ouverte ou fermée), et sont utilisés dans l'industrie (contrôle correct d'un procédé linéaire ou non linéaire [KNT99], télécommunications, avionique).
- Les modèles de *systèmes états/transitions* tels que les automates temporisés ou les réseaux de Pétri temporels, qui associent des intervalles de validité temporelle aux tirages de transitions. Dans le premier cas, le formalisme est bien adapté à la modélisation de systèmes temps-réel complexes communicants, et des protocoles sous-jacents. Dans le deuxième cas, le formalisme est adapté à la modélisation de systèmes temps-réel pour vérifier des propriétés liées au partage de ressources ou de parallélisme. Les deux formalismes sont suffisamment proches pour qu'il soit possible de traduire les modèles d'un formalisme à l'autre.
- Les *langages asynchrones* permettent de spécifier, modéliser, et générer des systèmes temps-réel communicants sous une forme hiérarchique (SDL, très utilisé dans l'industrie des télécommunications), mais offrent peu de possibilités pour vérifier les propriétés temporelles des modèles autres que par simulation exhaustive (par exemple avec l'outil Telelogic ObjectGeode [53]).

Dans la grande majorité de ces formalismes, l'écoulement du temps manipulé, ainsi que la notion de traitement modélisée, supposent une capacité de traitement du système

suffisamment grande : dans les formalismes fondés sur des automates puisque plusieurs transitions peuvent potentiellement être tirées simultanément, ou dans les formalismes plus abstraits (algèbres de processus en particulier) puisque plusieurs traitements peuvent avoir lieu simultanément. En général, l'entrelacement des activités dans le système dû à la limitation en ressources disponibles n'est pas traité par ces formalismes.

Modèles orientés allocation de ressources

On distingue une troisième classe de formalisme de modélisation, fondamentalement asynchrone, visant exclusivement à la vérification de propriétés temporelles. Dans ce formalisme, le temps est une grandeur physique mesurable (espace discret ou dense) qui permet de caractériser les éléments du système, dont les traitements font partie (tâches). Le système est modélisé sous la forme de :

serveurs de ressources. Leur *capacité* peut correspondre à une densité temporelle (*ressources actives* : processeurs, réseau), ou à une capacité scalaire (*ressources passives* : verrous, sémaphores, rendez-vous par exemple).

demandeurs de ressources (les *tâches*, *travaux*, ou *traitements*), qui ont des *contraintes* de validité fonctionnelles et temporelles, en même temps que des *besoins* en ressources quantifiés.

L'ensemble de ces entités est arbitré par un composant central : le *support d'exécution* (en général un système d'exploitation). Ce formalisme est utilisé par la communauté du temps-réel qui s'intéresse aux problèmes d'*ordonnancement* ; par essence, il traite des problèmes d'entrelacement des activités dans le système en présence de ressources de capacités limitées, et se situe davantage au niveau du modèle comportemental du système, qu'au niveau formel de modélisation abstraite des formalismes précédents.

Pour notre part, nous nous intéressons à ce dernier formalisme, détaillé ci-dessous. Il présente l'avantage d'être proche des implantations concrètes, ce qui permet un dimensionnement du système d'après le modèle, et ce qui permet d'évaluer l'implantation avec les mêmes outils que pour l'évaluation du modèle.

1.2 Modèle de système considéré

Plus précisément, nous nous appuyons sur le modèle de système, constitué d'une seule unité de traitement, schématisé sur la figure 1.1.

Le *système* est soumis à des *événements* provenant de l'*environnement* du système, et générés par du matériel spécialisé (réseau, capteurs ou générateur d'impulsions d'horloge par exemple). Il effectue des *traitements* en tenant compte ou non de ces événements, qui peuvent décider d'*actions* à entreprendre sur l'environnement au moyen de témoins (écrans ou diodes par exemple), d'actionneurs (moteurs ou relais par exemple), ou de messages (réseau par exemple).

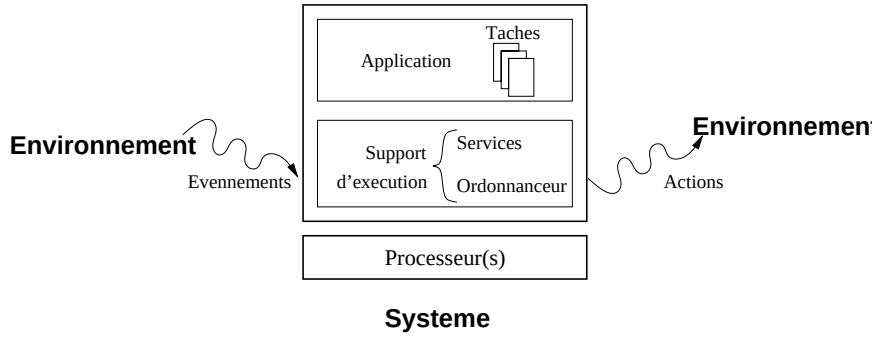


FIG. 1.1: Modèle de système considéré

1.2.1 Niveau logiciel

Les traitements sont constitués de l'*application* et du *support d'exécution* temps-réel. L'application effectue les traitements utiles du système pour lesquels elle a des besoins en *ressources* logiques ou matérielles (entre autres des ressources processeur), et est soumises à des *contraintes* liées aux spécifications, dont les contraintes temporelles font partie. Elle est structurée en *tâches*, c'est-à-dire en flots de contrôle concurrents (*i.e.* séquences d'opérations du processeur).

Le support d'exécution est chargé de répartir les demandes en ressources de l'application entre les ressources matérielles ou logiques disponibles, tout en veillant au respect de toutes les contraintes. Il peut prendre la forme d'un système d'exploitation contrôlant directement le matériel, ou d'un *intergiciel* (*middleware* en anglais) s'intercalant entre le système d'exploitation et l'application. La particularité d'un support d'exécution temps-réel est qu'il prend en compte les contraintes temporelles ainsi que le comportement temporel de l'application et du système d'exploitation. Tout particulièrement, c'est une composante du support d'exécution qui décide quelle tâche doit s'exécuter sur quel processeur de manière à ce que les contraintes temporelles soient respectées : l'*ordonnanceur*.

Dans les sections 1.3 et 1.4 qui suivent, nous revenons plus précisément sur la caractérisation temporelle des tâches et du support d'exécution.

1.2.2 Niveau matériel

Le système repose sur un ou plusieurs *processeurs*[1] regroupés en *nœuds* qui communiquent (par messages ou par mémoire partagée).

[1]Nous considérons qu'il s'agit de processeurs monoflot. Le domaine du temps-réel commence à voir apparaître des processeurs multiflot [JHA02], que nous assimilons ici à des multiprocesseurs.

1.3 Caractérisation de l'application

1.3.1 Description et définitions

Une tâche est en charge de fournir une partie des services de l'application. Elle correspond aux exécutions d'une séquence d'opérations donnée sur le processeur. Que les tâches partagent un unique espace d'adressage, ou que le cloisonnement mémoire soit possible (notion de processus Unix par exemple), est indifférent pour la suite.

Dans le modèle canonique du domaine, une *tâche* correspond à un traitement précis qui sera effectué par le processeur (composante statique, dont le code exécuté entre autres), et un *travail* (ou *job* en anglais : composante dynamique) correspond au flot d'exécution qui exécute effectivement le traitement défini dans la tâche. Le service fourni par la tâche peut être rendu plusieurs fois au cours de la vie du système, c'est-à-dire que la même tâche peut être *instanciée* plusieurs fois, sous la forme de plusieurs travaux. Et il peut y avoir plusieurs travaux d'une même tâche présents simultanément dans le système (actifs ou en attente, suivant les ressources disponibles).

Plusieurs travaux de plusieurs tâches (distinctes ou non) sont susceptibles de s'exécuter sur un même processeur, mais, par définition, ce dernier ne peut en exécuter qu'un seul à la fois. Un travail est donc associé à une structure de données qui renferme en particulier son *état* d'exécution. Le système d'exploitation s'occupe, entre autres, de passer l'état des travaux d'un état à un autre suivant le diagramme de transition des travaux (la figure 1.2 propose un exemple). Généralement, le diagramme d'état des travaux fait intervenir les états suivants :

ready : travail prêt à démarrer, *i.e.* pas encore démarré, ou dont l'exécution a été interrompue d'autorité (suite à un événement matériel par exemple) pour laisser la place à un autre travail.

running : travail en cours d'exécution sur le processeur ;

blocked : travail en attente de la disponibilité d'une ou plusieurs ressources ;

sleeping : travail en sommeil, en attente d'un événement de réveil ;

stopped : travail terminé.

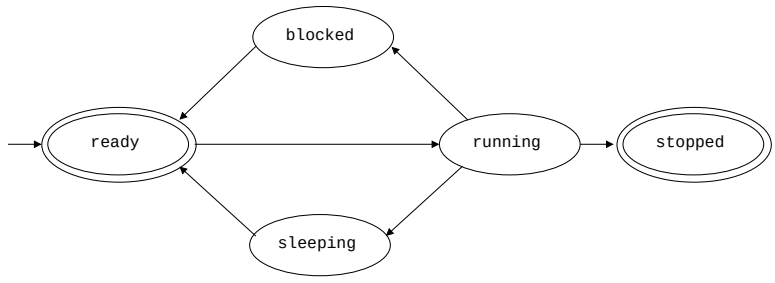


Fig. 1.2: Diagramme d'état des travaux

La vie d'un travail peut également être représentée suivant un *chronogramme* ou *diagramme de Gantt* tel que celui de la figure 1.3, dont la signalétique est reprise dans le reste du document, et qui permet d'introduire la terminologie suivante :

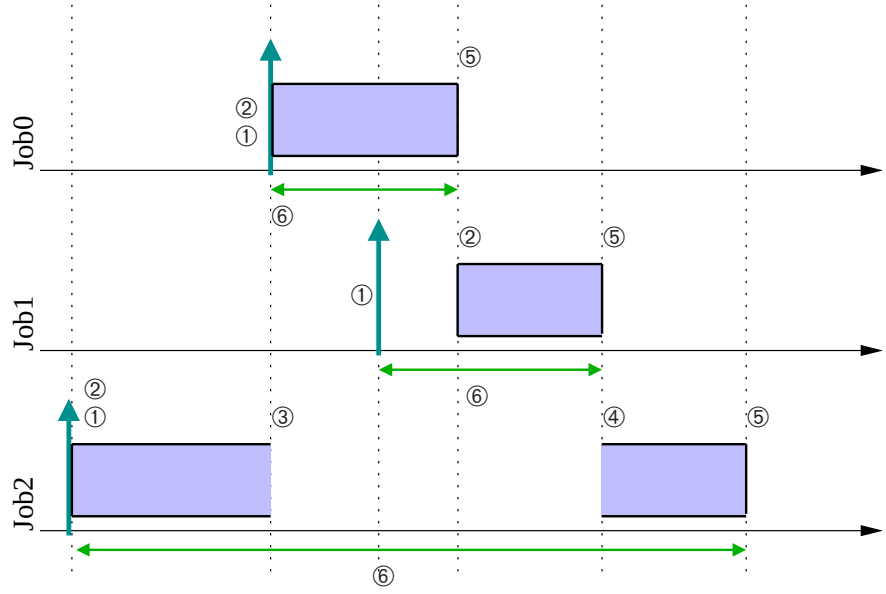


FIG. 1.3: Événements au cours de la vie d'un travail

① **: date de création ou d'activation** (*Arrival* ou *Request time* en anglais). Le travail est créé et prêt à être exécuté sur le processeur.

② **: date de démarrage** (*Start time* en anglais). Le travail commence à être exécuté sur le processeur pour la première fois.

③ **: dates de *préemption*.** Le travail est momentanément interrompu au profit d'un autre, dit *plus prioritaire* (plusieurs dates de préemption possibles durant la vie du travail).

④ **: dates de *reprise*.** Le processeur reprend l'exécution du travail là où il avait été précédemment préempté (plusieurs dates de reprise possibles durant la vie du travail).

⑤ **: date de terminaison** (*Finishing* ou *completion time* en anglais). Le travail termine de s'exécuter sur le processeur (fin volontaire du travail, ou suppression du travail par le support d'exécution ou par un autre travail).

⑥ **: temps de réponse** (*Response time* en anglais). L'écart entre la date de terminaison et la date d'activation d'un travail.

1.3.2 Caractéristiques et contraintes temporelles

Puisqu'il doit garantir que les contraintes temporelles des travaux sont respectées, le support d'exécution doit connaître les caractéristiques temporelles des travaux de chaque tâche, formant une partie du *modèle de tâche*. Dans le modèle de tâche figurent habituellement les caractéristiques temporelles suivantes :

Le temps d'exécution : il s'agit du temps d'exécution d'un travail de la tâche sur le processeur et considéré de manière isolée. En général, il s'agit d'un *temps*

d'exécution pire-cas (ou *WCET* pour *Worst Case Execution Time* en anglais), c'est-à-dire un majorant sur tous les temps d'exécution possibles de tous les travaux de la tâche, chacun pris en isolation du reste du système. Nous verrons en 2.2.1 que cette donnée constitue un paramètre d'entrée important aux méthodes et outils de vérification du respect des échéances. Quelques systèmes temps-réel souple considèrent un temps d'exécution moyen, minimal, ou connu de manière probabiliste.

La loi d'arrivée : Il s'agit de la répartition dans le temps des dates de création des travaux de la tâche. On distingue couramment trois lois :

périodique :

Les travaux de la tâche sont créés de façon rigoureusement périodique, et la période est précisée. Si la date d'activation du premier travail est donnée, la tâche est dite *concrète*. Si toutes les tâches du système sont périodiques, alors le motif d'activation des travaux se répète à l'identique par intervalles de durée multiple d'un intervalle, nommé l'*hyperpériode*. Si les périodes T_i des tâches du système sont telles que $\forall i, j,\ T_i > T_j \Rightarrow T_i$ multiple de T_j, alors le système est *harmonique*. Enfin, si toutes les tâches sont périodiques concrètes, et si les dates d'activation du premier travail sont identiques, les tâches sont dites *synchrones* (ou *à départ simultané*), et dans ce cas l'hyperpériode correspond au plus petit commun multiple des périodes des tâches du système (*i.e.* la plus grande période dans le cas particulier d'un système harmonique).

sporadique :

Les travaux de la tâche sont créés de sorte qu'une durée minimale sépare l'arrivée de deux travaux successifs : le *délai d'inter-arrivée*, qui est précisé. Une tâche périodique est un cas particulier de tâche sporadique.

apériodique :

Un travail de la tâche peut être créé à tout instant. Une tâche sporadique est un cas particulier de tâche apériodique.

D'autres lois d'arrivée moins usuelles sont possibles. La connaissance de cette donnée est un paramètre d'entrée important aux outils de vérifications de la garantie du respect des échéances (voir 2.2.1).

Le modèle de tâche indique également les contraintes temporelles des travaux. Les contraintes temporelles courantes sont (voir la figure 1.4) :

① **: date d'échéance :** il s'agit de la date à laquelle les travaux doivent être terminés. Cette date peut être *absolue* (*Absolute deadline* en anglais), ou définie relativement à la date de création des travaux (*Relative deadline* en anglais).

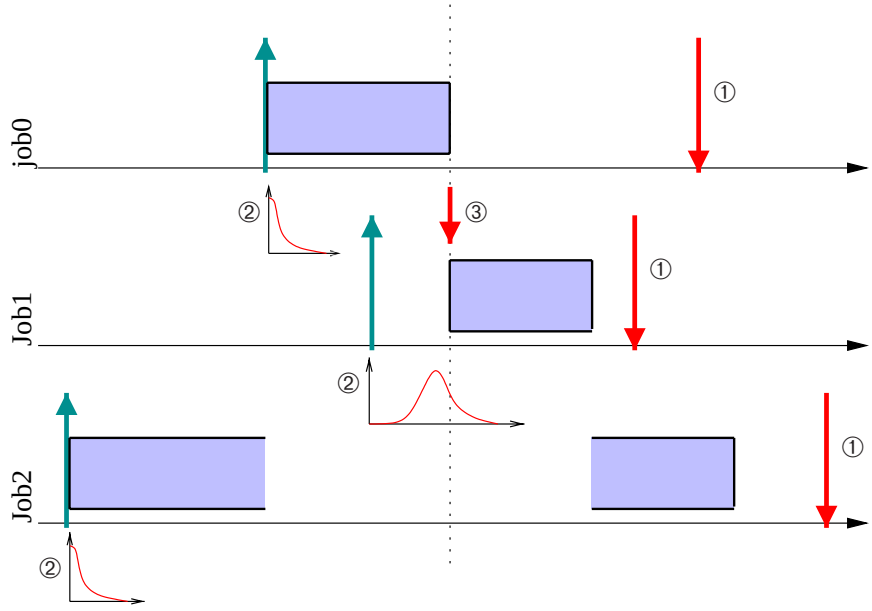


FIG. 1.4: Contraintes temporelles courantes

② **: gigue de démarrage :** il s'agit de la mesure de dispersion des délais entre la date de création et la date de démarrage des travaux d'une tâche. *Release jitter* en anglais.

③ **: précédence :** avant de pouvoir démarrer, un travail doit attendre le résultat d'un travail d'une autre tâche. On représente ce type de contrainte sous la forme d'un *graphe de précédence*, *i.e.* un graphe orienté dont les sommets sont les tâches et dont les arcs sont les relations de précédence (éventuellement augmentées des messages qui transitent).

Quelques mesures pour caractériser le comportement temporel sont dérivées des contraintes et des caractéristiques temporelles :

L'utilisation (ou *taux d'utilisation*) : c'est la proportion d'une ressource donnée occupée par les travaux d'une tâche. Le plus souvent, on s'intéresse à l'utilisation du processeur : si la tâche τ_i est périodique de période T_i et si chaque travail nécessite un temps d'exécution C_i sur le processeur, alors le taux d'utilisation du processeur par la tâche est : $u_i = \frac{C_i}{T_i}$. On définit également l'utilisation processeur pour un ensemble de tâches périodiques : $U = \sum_i u_i = \sum_i \frac{C_i}{T_i}$. On montre aisément que 1 est une borne supérieure d'utilisation : au delà de ce seuil, la ressource est dite *surchargée* puisque potentiellement la demande en ressource excède la capacité disponible.

Le retard : c'est l'écart entre la date de terminaison du travail et la date d'échéance. Négatif quand le travail respecte son échéance. *Lateness* en anglais ("*tardiness*" lorsqu'on ne s'intéresse qu'aux retards positifs).

La laxité : c'est l'écart maximal entre la date courante et la date au plus tard de la prochaine exécution (démarrage/redémarrage) du travail de sorte que l'échéance demeure respectée. *Laxity* ou *Slack time* en anglais. On peut également définir la laxité *statique* comme étant l'écart maximal entre la date d'activation et la date de démarrage au plus tard du travail de sorte que l'échéance demeure respectée.

Les contraintes temporelles doivent être impérativement respectées en temps-réel strict. En temps réel souple, des *dépassements* qui ne remettent pas en cause la structure de l'application sont tolérés. Dans ce cas, il est possible de définir une métrique qui mesure la *qualité de service* qui demeure assurée (par exemple le nombre de travaux qui respectent leur échéance relativement au nombre total de travaux créés pour la tâche). Cette qualité de service peut être évaluée globalement pour tout le système, ou relativement à chaque sous-système (ensemble de tâches de l'application), avant fonctionnement (par analyse) et/ou en cours de fonctionnement (exécution effective ou simulation). Dans certains systèmes, la qualité de service constitue une contrainte supplémentaire conditionnant la correction du comportement du système.

Dans la suite, sauf précision en cas d'ambiguïté, on confond les notions de *travail* et de *tâche*, puisqu'un travail est une instance d'une tâche précise.

1.3.3 Contraintes de ressources

Les tâches peuvent être vues comme des éléments du système qui consomment la ressource processeur. D'autres ressources existent au sein du système, qui peuvent être *matérielles* (processeur(s), mémoire, réseau, capteurs, actionneurs par exemple), ou *logiques* (sémaphores, files de messages par exemple). Le système d'exploitation a pour rôle de gérer toutes les ressources, c'est-à-dire d'arbitrer entre les demandes que font les tâches de l'application, et les ressources effectivement disponibles dans le système.

Dans le cas de ressources *actives* [SL94], c'est-à-dire celles qui permettent aux travaux de progresser dans leur exécution, telles que les processeurs et les réseaux, cet arbitrage consiste en un partage temporel (*multiplexage*) de l'accès aux ressources. Les décisions de partage sont faites suivant un *ordonnancement* (détaillé dans le chapitre 3 dans le cas de la ressource processeur).

Dans le cas des autres ressources (les ressources *passives*), l'objectif est de restreindre le nombre et les types d'accès (lecture/écriture) aux ressources afin de maintenir leur cohérence : c'est le rôle des mécanismes de *synchronisation* (verrous, sémaphores, conditions, moniteurs, files de messages par exemple), qui par extension permettent également aux tâches d'échanger des informations ou de se synchroniser sans qu'il y ait nécessairement de ressource à protéger (comme avec les *rendez-vous* par exemple).

Lorsque les tâches ne sont soumises ni à des contraintes de ressources, ni à des contraintes de précédence, elles sont dites *indépendantes*.

1.3.4 Autres contraintes

En dehors des contraintes de temps ou d'accès aux ressources, la spécification de systèmes temps-réel introduit parfois d'autres contraintes. Les plus courantes sont les

contraintes de :

Localisation ou localité : précisent si les tâches doivent s'exécuter sur un processeur donné (par exemple pour accéder à un capteur disponible uniquement sur une machine donnée, ou pour diminuer la quantité de messages réseau échangés). Valable pour les systèmes multiprocesseur seulement ;

Anti-localité : précisent si deux tâches doivent s'exécuter sur des processeurs différents (par exemple quand il s'agit d'introduire de la redondance spatiale dans les traitements afin que le système soit tolérant aux fautes physiques). Valable pour les systèmes multiprocesseur seulement ;

Énergie : il s'agit de minimiser l'énergie consommée par les traitements, par exemple en diminuant le plus possible la fréquence de fonctionnement du processeur. Cette contrainte est importante dans le domaine du temps-réel, souvent associé à des systèmes embarqués, pour lesquels les contraintes d'autonomie sont essentielles.

1.4 Caractérisation du support d'exécution

La phase d'implantation d'une application temps-réel repose à la fois sur les spécifications que nous venons de présenter, et sur les caractéristiques des supports langage et système pour le temps-réel sous-jacents. Ces dernières contraignent en partie la façon dont l'application doit être modélisée. Il est donc important de connaître les supports langages et système dès la phase de conception. Nous présentons ici quelques unes des caractéristiques à prendre en compte.

1.4.1 Perception du temps

Un système d'exploitation en général, et tout particulièrement un système d'exploitation temps-réel, doit percevoir l'écoulement du temps, notamment pour prendre ses décisions d'ordonnancement.

Couramment, un système informatique s'appuie sur une *horloge matérielle* spécialisée dans la **génération d'événements** à intervalles réguliers (les *ticks* ou les *tops* d'horloge), ce qui fournit au système d'exploitation une base de temps rudimentaire, dite *horloge système*, afin d'assurer au moins la vivacité du système. En contrepartie, la résolution de l'échelle de temps, utile aux décisions d'ordonnancement pour le temps-réel, est fonction de la fréquence de génération des événements d'horloge. Mais, puisque les coûts de traitement associés à la prise en compte de cette horloge ne sont pas négligeables, la fréquence n'est en pratique pas très élevée (de l'ordre de 1 à 10ms). Nous verrons à la partie III que cette résolution a quelques implications sur le comportement du système résultant.

Certains systèmes peuvent aussi *consulter* un compteur externe ou interne au processeur, qui possède une résolution beaucoup plus élevée (jusqu'à quelques fractions de nanoseconde). Pour les systèmes temps-réel, cette fonctionnalité permet de prendre des décisions d'ordonnancement de façon plus précise, et permet de prendre en compte des contraintes temporelles beaucoup plus courtes que la granularité de l'horloge système.

Dans ces deux cas, la perception du temps est locale à chaque nœud. Ceci peut poser problème lorsque le système temps-réel est distribué sur plusieurs nœuds : en général les horloges locales dévient et se *désynchronisent* l'une par rapport à l'autre, ce qui empêche de prendre des décisions relatives au temps qui sont globales au système. Dans ce cas, une perception du temps cohérente entre les nœuds nécessite de connaître ou d'établir les bornes sur les écarts entre les évolutions des horloges locales aux nœuds du système [AP97].

1.4.2 Supports langage et système

1.4.2.1 Supports langage

Le modèle de système temps-réel sur lequel nous reposons fait apparaître des notions que les langages de programmation savent manipuler directement ou non.

Ainsi, quelques langages savent manipuler des tâches directement (comme Ada ou occam2). D'autres langages (`C`, `C++` par exemple) imposent de recourir directement au système d'exploitation ou au support d'exécution pour les gérer. Certains langages proposent des mécanismes de synchronisation (par exemple : moniteurs en Modula-1, Concurrent Pascal, Mesa ; objets protégés en Ada ; files de messages en occam2 et Ada). Sinon, il est nécessaire de faire directement appel aux services du système d'exploitation ou d'une bibliothèque spécialisée, et/ou du matériel.

Pour faciliter la vérification des aspects temporels avant toute exécution (analyse hors-ligne), il serait désirable que le langage fournisse tous les éléments qui permettent d'évaluer le comportement temporel des tâches, ou, de façon plus réaliste, qu'il permette de l'exprimer : les temps d'exécution (par annotation du nombre d'itérations des boucles par exemple), les ressources utilisées. Malheureusement, il existe peu de langages dans l'industrie qui supportent cette fonctionnalité ; en général, il s'agit d'une série de restrictions d'un langage courant[2] (C, Ada) et de syntaxes d'annotation (par exemple : annotation du nombre d'itérations d'une boucle) qui sont non standards et spécialisées pour un secteur ou une entreprise particulière (par exemple SPARK Ada pour l'avionique anglaise [Bat98]). Ceci amène en pratique à séparer l'implantation du système, de sa caractérisation, ce qui peut être source d'erreur (le modèle de comportement ne correspond pas à la réalité).

1.4.2.2 Supports système

Quantité de supports d'exécution pour le temps-réel existent tant dans l'industrie (VxWorks, pSos, VRTX, OSE, OS9, LynxOS/LynuxOS, RTEMS, eCos par exemple) qu'au sein de la communauté scientifique (Mars [KFG+92], Maruti [SdSA95], Spring [SRN+98], Hades [CPC+00], RTMach [TNR90a], DIRECT [SG97a], Emeralds [Zub98], MaRTE [RH01], DICK [But97], Hartik/S.Ha.R.K. [GAGB01], variantes RTLinux [41][Bar97, WL99b, DFW02, Heu01] par exemple). Des normes pour les interfaces de programmation système existent également qui sont plus ou moins respectées par les systèmes d'exploitation (normes POSIX1003.1b et 1003.1c [fIT93] ou ITRON [56] par exemple).

[2] En général : pas de boucles infinies, récursivité contrôlée.

Nous présentons ici quelques unes des fonctionnalités qu'on retrouve couramment dans les supports d'exécution temps-réel :

Interface de création des tâches. Certains systèmes proposent des mécanismes pour associer la création de tâches à l'occurrence d'événements de l'environnement (création orientée événements, ou *event-driven*). D'autres ne permettent d'implanter que des tâches périodiques (exécution commandée par le temps, ou *time-driven*). Dans tous les cas, il est possible à une tâche d'en créer d'autres.

Gestion des délais. Le système peut proposer une fonctionnalité d'alarme différée relativement au temps processeur occupé par la tâche (indépendamment des préemptions), ou de sommeil pendant une durée donnée *i)* relativement à la date de début de la mise en sommeil (*délai relatif* qui accumule les erreurs dues à la granularité de l'horloge système et aux gigues de démarrage), ou *ii)* jusqu'à une date absolue (*délai absolu*). L'implantation des tâches périodiques par exemple dépend des méthodes de mise en sommeil disponibles.

Primitives de synchronisation. Les systèmes d'exploitation proposent en général des mécanismes de synchronisation basiques identiques, à savoir les sémaphores à compteur ou les verrous. Au delà, les systèmes peuvent adhérer à une norme, totalement ou partiellement, dans laquelle figurent une série de mécanismes de synchronisation supplémentaires (eventflag[3], boîte aux lettres, files de messages, verrous en lecture/écriture). Les systèmes peuvent également proposer leurs propres mécanismes de synchronisation, à la place ou en plus de ceux recommandés par les normes auxquelles ils peuvent décider de se conformer (par exemple pour prendre éviter les phénomènes d'inversion de priorité).

Politique d'ordonnancement. Pour définir la prochaine tâche à élire, le support d'exécution peut reposer sur un calendrier, ou *plan*, défini lors de la phase de conception (*ordonnancement cyclique hors-ligne*), ou sur un ordonnancement *en-ligne* reposant sur des priorités relatives entre les tâches, ou sur l'ordre d'activation (politique du premier arrivé – premier servi, ou FIFO, par exemple). Le chapitre 3 détaille les politiques d'ordonnancement temps-réel les plus courantes.

Profil d'appel à l'ordonnanceur. Le système d'exploitation peut prendre ses décisions d'ordonnancement lors de chaque événement système (blocage sur ressource, interruption matérielle), ou ne les prendre que lors d'une interruption d'horloge (*tick scheduling*). Dans le premier cas, les surcoûts système sont plus élevés, et dans le second cas les gigues d'activation sont plus importantes.

Comme nous le voyons en 2.2.1.2, dans le cas de l'implantation de systèmes temps-réel strict, il est nécessaire de disposer de suffisamment d'informations sur le comportement temporel du système d'exploitation sur lequel on repose. Cela suppose soit de disposer du code source du système d'exploitation et d'être capable de l'*analyser* [CP01a, CP01c] pour en extraire les caractéristiques temporelles ou pour en déduire le comportement de l'application dans son ensemble ; soit de disposer des caractéristiques temporelles *fournies* avec le système d'exploitation (le plus souvent obtenues par test

[3] une variante des *conditions* sous forme d'un champ de bits.

et mesure, ou *benchmark*) ; soit que l'utilisateur détermine les caractéristiques temporelles en évaluant lui-même le système d'exploitation ou l'application dans son ensemble. Parmi les caractéristiques temporelles couramment évaluées, on trouve [fIT93] les coûts liés aux appels système explicites :

Délai de prise de ressource lorsqu'une tâche acquiert un sémaphore, un verrou, un moniteur ou tout autre mécanisme d'exclusion non encore pris par une autre tâche.

Coût de changement de paramètre d'ordonnancement lorsqu'une tâche décide de changer de *priorité* (développé en 3) par exemple.

Coûts temporels d'autres services système tels que ceux de la création, suppression et terminaison de tâche, ou ceux liés au service d'ordonnancement.

D'autres coûts systèmes implicites sont également évalués :

Caractéristiques du service d'horloge système telles que sa résolution, ses paramètres de déviation ou d'irrégularité par rapport à une horloge de référence, les coûts temporels liés à l'appel des primitives de consultation et de modification, ainsi que ceux des primitives de programmation d'une alarme (*timeout*).

Délai de propagation d'un événement de réveil lorsqu'une tâche libère une ressource, signale une condition, ou envoie un événement (signal POSIX par exemple) à une autre tâche.

Coûts temporels des changements de contexte. C'est le temps nécessaire à la préemption d'un travail au profit d'un autre. Il s'agit d'une durée essentiellement dépendante de la configuration matérielle (type de processeur, cadence, taille des caches d'instructions, de données, et de traduction d'adresses par exemple).

Surcoûts temporels liés à la gestion des interruptions matérielles. Ce sont les temps nécessaires au passage de la tâche ou du traitant d'interruption en cours, vers le code de traitement de l'interruption levée ; et inversement. Il s'agit encore d'une donnée essentiellement dépendante de la configuration matérielle.

Autres coûts système. Les normes (comme POSIX1003.1b par exemple) proposent en plus de mesurer les coûts de services plus sophistiqués du système d'exploitation ou du support d'exécution, tels que ceux des services de gestion de fichiers ou d'allocation dynamique de mémoire par exemple.

Dans 2.2.1, nous verrons que la connaissance de majorants aux caractéristiques temporelles du système d'exploitation est un des paramètres d'entrée essentiels afin de garantir, avant exécution, qu'un système temps-réel strict respectera toutes ses contraintes temporelles en cours de fonctionnement. Or, certains services du système d'exploitation peuvent avoir des caractéristiques temporelles (telles que le temps d'exécution par exemple) pire-cas déraisonnables, comme par exemple l'allocation dynamique de mémoire [Pua02] ou le service de pagination par zone d'échange (*swap* en anglais), ou de gestion de fichiers sur disque. Ces services ont peu de chances d´être adaptés aux contraintes temps-réel courantes, ce qui explique pourquoi ce type de service est rarement utilisé dans le contexte du temps-réel strict.

Chapitre 2

Évaluation de systèmes temps-réel : vue d'ensemble

Dans ce chapitre, nous présentons d'abord quelques *métriques* (ou *mesures*) d'évaluation d'un modèle concret ou abstrait d'un système temps-réel. Puisque ces métriques permettent aussi de confronter le comportement du modèle aux spécifications, elles peuvent également être considérés comme des *critères de correction* lors des phases de vérification. Nous présentons ensuite les classes de méthodes qui visent à *vérifier* que la *modélisation* et l'*implantation* d'un système temps-réel sont correctes vis-à-vis des contraintes temporelles. Nous considérons ici tous les types de systèmes temps-réel, qu'ils soient critiques ou non, qu'ils soient constitués de tâches temps-réel strict et/ou souple, mais nous restreignons cette présentation au cas de systèmes centralisés, en privilégiant fortement la considération des aspects temporels.

2.1 Métriques d'évaluation et critères de correction usuels

Pour le temps-réel strict, le critère de correction fondamental est que toutes les tâches temps-réel strict respectent toutes leurs contraintes temporelles en toute circonstance nominale de fonctionnement. Au delà de cette contrainte fondamentale que nous développons au chapitre 3, différents mécanismes peuvent être évalués et comparés par des mesures quantitatives sur leur comportement, en s'inspirant des travaux en temps-réel souple.

En temps-réel souple, les contraintes de correction peuvent faire appel à des données numériques caractérisant le comportement du système. Suivant la complexité du système, il est possible de les obtenir par analyse (le plus souvent statistique) ou par évaluation (le plus souvent par simulation). Parmi les métriques les plus courantes, citons :

le taux de respect : c'est la proportion de travaux des tâches qui respectent leurs contraintes temporelles. *Hit ratio* en anglais. Lorsque les tâches doivent subir un test préventif lors de leur activation et avant leur exécution, afin de vérifier que leurs contraintes temporelles pourront être respectées sans remettre en cause le reste du système (*test d'acceptation*, voir 3.3.2), on parle de *taux de garantie*

(*guarantee ratio* en anglais).

la valeur dégagée : on associe une *fonction de valeur* à l'exécution de chaque tâche, et on mesure la somme (ou une autre fonction) des valeurs obtenues pendant le déroulement du système [BSS95]. *Hit value ratio* en anglais. Par exemple, on peut représenter une tâche temps-réel (dite *ferme*) par la fonction de valeur suivante : si la tâche se termine avant son échéance, elle dégage la valeur $X > 0$, et 0 sinon. Un cas particulier est celui où X vaut 1, auquel cas la valeur dégagée (*completion count* en anglais) est directement liée au taux de respect. D'autres fonctions de valeur font intervenir le temps d'exécution [BHS01], ou d'autres paramètres [AB99].

le débit : c'est le nombre de travaux d'une tâche qui respectent leurs contraintes temporelles par fenêtre de temps donnée [WS99b]. *Throughput* en anglais. En multimédia par exemple, il peut s'agir de la mesure du nombre d'images rendues par seconde (*fps* pour *frames per second*).

le seuil d'utilisation : c'est l'utilisation du processeur au dessus de laquelle des tâches commencent à dépasser leurs contraintes temporelles [VJP97, KAS93]. *Utilization breakdown* en anglais.

les caractéristiques temporelles (voir 1.3.2) telles que le retard, la gigue de démarrage, la laxité, le temps de réponse par exemple. Le cas particulier de la vérification des systèmes temps-réel strict consiste à garantir au moins que le retard maximal pour ces tâches est négatif ou nul.

les coûts systèmes (voir 1.4.2.2) tels que les coûts d'ordonnancement ou de service des appels système par exemple.

La majorité de ces métriques évoluent au cours du temps. L'évaluation fait intervenir les propriétés de leur distribution, telles que le maximum, le minimum, ou les caractéristiques statistiques (moyenne, écart-type par exemple). Le mode d'évaluation s'intéresse en général au régime permanent (stimuli de l'environnement réguliers), et plus rarement aux régimes transitoires [LSA+00] (délai de rétablissement, sur-utilisation temporaire). Lorsqu'il s'agit de proposer un nouveau mécanisme système pour le temps-réel, on le soumet à ces techniques d'évaluation, en intégrant dans le modèle ou dans l'implantation à évaluer soit des jeux de tests issus de systèmes réels (tels que ceux présentés dans [CFBW93, ABR+93, Tin92a, Aud91, Bur93, TC94, TBW92a, BWH93]), soit des jeux de tests générés aléatoirement (*charge synthétique*).

2.2 Évaluation à partir du modèle

Pour prévenir le développement coûteux d'un produit qui sera finalement jugé défaillant lors de l'évaluation finale de l'implantation (par confrontation aux spécifications), il est intéressant de pouvoir évaluer le système à partir d'un modèle abstrait, au plus tôt dans le cycle de développement (éventuellement dès la phase de spécifications). Cette démarche permet en plus de s'abstraire des détails d'implantation. Mais en contrepartie, la précaution à prendre pour que cette approche ait un intérêt, est que le modèle soit suffisamment fidèle à l'implantation pour toujours avoir un sens une

fois l'implantation réalisée [SL99]. De ce point de vue, formalisme du modèle et implantation ont intérêt à être en correspondance. Rappelons que dans toute la suite, nous nous intéressons à évaluer un système modélisé suivant le troisième formalisme détaillé en section 1.2, adopté par la communauté de l'ordonnancement sous contrainte de temps-réel, et qui présente l'avantage de ne pas trop éloigner le modèle abstrait de l'implantation concrète qui en dérive.

Nous présentons d'abord les méthodes d'analyse statique et stochastique, avant d'introduire les méthodes fondées sur la simulation à partir du modèle.

2.2.1 Méthodes analytiques déterministes

L'objectif que poursuit l'analyse déterministe, aussi nommée analyse statique hors-ligne, ou *analyse hors-ligne* dans la suite, est plus fort que l'évaluation du système : il s'agit en priorité de vérifier par analyse (le plus souvent d'ordre mathématique) du modèle, et avant toute exécution réelle du système temps-réel, que toutes les contraintes temporelles vont être toujours respectées, dans toutes les conditions nominales de fonctionnement possibles. Le plus souvent, l'analyse hors-ligne s'intéresse aux contraintes temporelles strictes, du type : aucune échéance temporelle dépassée, aucune contrainte de gigue de démarrage dépassée. C'est pourquoi ce type d'analyse est adapté en priorité aux systèmes qui ont une composante temps-réel strict. Lorsque les contraintes temporelles sont probabilistes, l'analyse stochastique (section 2.2.2) est mieux adaptée.

2.2.1.1 Terminologie et objectifs

Le cœur de l'analyse est l'étude de l'*ordonnancement* des tâches sur le ou les processeur(s). La problématique de l'*ordonnancement* revient à définir comment agencer les exécutions des tâches sur chaque processeur. Un ordonnancement est *faisable* (*feasible schedule* en anglais), si **toutes** les contraintes temporelles et de ressources sont respectées. Suivant les caractéristiques du modèle de tâches et du système, un ordonnancement faisable peut exister ou non. Lorsqu'un tel ordonnancement existe, le système temps-réel est dit *ordonnançable* (*schedulable* en anglais). Un système est dit *ordonnançable par un algorithme donné* si l'ordonnancement qui en résulte est faisable.

L'objectif que poursuit l'analyse statique hors-ligne est d'établir l'ordonnançabilité du modèle, et/ou la faisabilité d'un l'ordonnancement donné.

2.2.1.2 Paramètres d'entrée

L'ensemble des modèles des tâches est le paramètre le plus important lors de l'analyse. Deux informations sont essentielles : le comportement temporel des tâches (tel que le temps d'exécution sur le processeur par exemple), et les lois d'activation, sur lesquelles nous revenons ci-dessous.

En ce qui concerne le comportement temporel des tâches, il est nécessaire de prendre en compte tous ceux qui sont envisageables. La prise en compte exhaustive exacte de chacun de ces comportements temporels étant d'une grande complexité (surtout en l'absence d'une implantation concrète), voire impossible, on se limite en général à

déterminer un intervalle qui regroupe toutes les valeurs possibles de ces caractéristiques temporelles (par exemple, pour des tâches temps-réel strict, il est le plus souvent fait appel à la connaissance d'une borne supérieure sur le temps d'exécution).

En ce qui concerne les lois d'activation, il est nécessaire de prévoir les pires scénarios possibles, ce qui implique (1) que garantir l'exécution de tâches apériodiques au niveau de l'analyse hors-ligne est impossible puisque cela nécessiterait le recours à un processeur de puissance infinie : et (2) le temps d'inter-arrivée entre tâches sporadiques doit être un minorant sûr des espacements possibles entre les tâches sporadiques. Si les tâches sont activées suite à des événements extérieurs, alors les lois d'activation sont directement reliées au modèle de l'environnement.

2.2.1.3 Principe

Le principe de l'analyse hors-ligne est d'identifier la ou les pire(s) configuration(s) possible(s) (approche mathématique) de l'ensemble du système, ou par rapport à une tâche donnée, en termes de besoins en ressources, en particulier processeur. Une fois une telle configuration identifiée, il s'agit de vérifier qu'elle est compatible avec les contraintes temporelles et de ressources sur l'ensemble du système, ou sur la tâche considérée, et sans remise en cause du reste du système.

Il serait possible de conduire ce genre d'analyse en étudiant chaque modèle en particulier. Mais en pratique on se ramène à des modèles génériques abstraits pour lesquels des résultats existent, et on applique ces résultats au modèle particulier considéré. Le chapitre 3 donne un aperçu de modèles de systèmes qui ont déjà été traités.

2.2.1.4 Résultats

L'analyse hors-ligne établit l'ordonnançabilité du système ou non. Si le système est ordonnançable par le type d'ordonnancement proposé, alors l'analyse hors-ligne est également en charge de définir les paramètres nécessaires à l'exécution des tâches par le support d'exécution, tels que le plan si l'ordonnancement est de type cyclique, ou les priorités si l'ordonnancement est à priorité par exemple.

2.2.1.5 Intérêts et limitations

Si l'analyse conclut à l'ordonnançabilité du système, alors on est assuré que dans tous les cas de fonctionnement nominaux, toutes les contraintes temporelles sont respectées ; il en résulte une propriété de sûreté forte.

Mais l'analyse peut être extrêmement complexe, voire impossible à mener sur des modèles compliqués. Ainsi, dans les cas favorables, on est souvent amené à projeter le modèle sur un cas général plus simple, mais qui rajoute des contraintes plus fortes au système. Au final, l'analyse qui résulte de cette démarche est *pessimiste*, c'est-à-dire que le système peut être déclaré non ordonnançable à tort.

2.2.1.6 Outils

Il existe des outils qui partent d'un modèle de système donné, et qui effectuent l'analyse hors-ligne. En général, le modèle est simple, et prend en compte des tâches temps-réel strict périodiques ou sporadiques, avec contraintes de ressources et de précédence, avec possibilité de prise en compte de tâches apériodiques temps-réel souple. La plupart de ces outils sont articulés autour de l'approche RMA et RTA (resp. pour *Rate Monotonic Analysis* et *Response Time Analysis*, voir 3.2.3), et commencent à être utilisés dans l'industrie. Citons par exemple les outils System Engineering Workbench (CMU) [6], Tripacific Rapid RMA [42] (ex PERTS [LLD+96][24], Urbana Champaign University), TimeSys Timewiz [54], ou MAST[DHGP01] (Universidad de Cantabria).

2.2.2 Méthodes analytiques stochastiques

Lorsque les contraintes font intervenir des données probabilistes, il est nécessaire de les vérifier (également hors-ligne) à partir du modèle. Le formalisme de modélisation du système considéré ici (voir section 1.2) permet des études stochastiques tant que le système reste simple (tâches indépendantes) [Mig99, LSD89, Gar99, Hei93, AB98d, TDS+95]. Il n'est plus adapté dès que des contraintes plus fortes sont à prendre en compte (ressources, précédences). Dans ce cas, l'approche suivie est soit de se ramener à un autre formalisme (réseaux de pétri stochastiques par exemple), soit d'évaluer le comportement du modèle à l'aide de simulations.

2.2.3 Simulation

Une autre voie d'évaluation du modèle, éventuellement complémentaire aux précédentes, est la simulation du système modélisé.

2.2.3.1 Principe

Lors des phases de spécification et de modélisation, on ne dispose pas d'une implantation complète qu'il est possible d'évaluer. Dans ces conditions, la simulation demande d'anticiper sur l'implantation à venir, c'est-à-dire de générer artificiellement les informations qui manquent : dates d'activation des tâches, temps d'exécution... On parle de *charge de travail synthétique* (*synthetic workload* en anglais).

La simulation consiste alors en la prise en compte de ces informations pour l'exécution d'un système associé au modèle, dans un environnement simulé. En retour, il est possible d'extraire des mesures pendant et après la simulation qui peuvent être brutes (traces), ou traitées (analyse statistique par exemple).

2.2.3.2 Intérêts et limitations

En comparaison aux méthodes d'analyse, un avantage est que l'effort d'apprentissage est souvent moins élevé, puisqu'il ne s'agit pas de maîtriser une méthode formelle ou une démarche d'analyse, mais plus simplement de savoir utiliser un outil de simulation. Un autre intérêt de la simulation est qu'elle permet l'évaluation de modèles

très complexes voire impossibles à caractériser par analyse statique ou stochastique hors-ligne.

En contrepartie, la confiance qu'on peut attribuer à ce type d'évaluation n'est pas totale. À l'inverse de l'analyse statique, une simulation est *optimiste* : elle peut certes servir à montrer que le système n'est pas ordonnançable (en exhibant une *configuration* incorrecte). Mais, si aucune configuration incorrecte n'est trouvée, on ne pourra pas en déduire que le système est correct, *i.e.* ordonnançable, à moins que la simulation ait été *exhaustive*, c'est-à-dire que toutes les configurations possibles ait été *couvertes* (on parle de *couverture* ou d'*exhaustivité* de la simulation). Or, en pratique, sur des systèmes concrets, une couverture complète n'est pas réaliste tant le nombre d'états à explorer est important.

2.2.3.3 Outils

Le domaine de la simulation sur ordinateur est extrêmement vaste. En ce qui concerne la simulation pour le temps-réel, il existe essentiellement trois types de simulateurs : les langages ou bibliothèques génériques pour la simulation appliqués à la simulation de systèmes temps-réel, les simulateurs d'ordonnancement, et les simulateurs de systèmes complets. Le chapitre 4 s'attache à présenter ces différentes classes de simulateurs, et la plate-forme décrite en partie II se situe dans les 2 dernières catégories.

2.3 Évaluation à partir de l'implantation

Bien que le formalisme de modélisation considéré ait l'avantage d'être proche de l'implantation concrète, il peut exister un écart entre le modèle conçu et vérifié, et l'implantation concrète réalisée à partir de ce modèle. Ainsi, même dans le cas où le modèle abstrait a été évalué, il est important de confronter cette évaluation à l'évaluation de l'implantation qui en est dérivée.

Dans ce chapitre, nous présentons d'abord les méthodes d'évaluation à partir de l'implantation par analyse hors-ligne, avant de décrire celles par exécution réelle ou simulée.

2.3.1 Analyse hors-ligne de l'implantation

2.3.1.1 Principe et intérêt

De même que pour l'analyse statique hors-ligne à partir du modèle, il s'agit en priorité de *vérifier* que toutes les contraintes temporelles vont être toujours respectées par l'implantation.

L'intérêt du formalisme auquel on a choisi de se ramener (voir section 1.2) est que l'implantation découle assez directement du modèle abstrait, et inversement. Il s'ensuit qu'il est possible de remonter les paramètres concrets de l'implantation (*niveau bas* de l'évaluation : comportement temporel des tâches et du système sur le matériel réel par exemple) au niveau du modèle abstrait (*niveau haut*), afin d'évaluer l'implantation concrète par analyse hors-ligne du modèle ainsi actualisé.

Il en résulte que, si les informations qui sont remontées au niveau du modèle sont sûres, alors la phase de vérification de l'implantation sur le plan temporel est elle-même sûre.

2.3.1.2 Difficultés

En ce qui concerne le comportement temporel des tâches (tel que le temps d'exécution sur le processeur par exemple), évaluer les données à remonter dans le modèle est un domaine de recherche actif au sein de la communauté temps-réel. La solution couramment employée dans l'industrie repose sur la mesure des caractéristiques temporelles observées lors d'exécutions réelles. Mais cette approche pose des problèmes de sûreté, dans la mesure où on ne dispose en général pas de la garantie qu'aucune configuration possible, mais non testée, ne dépassera pas les pires observations (problème de couverture). Et ceci reste vrai même si un coefficient multiplicateur de sécurité (technique courante) est appliqué. Une deuxième technique, qui est en plein essor dans le milieu scientifique, consiste à analyser le code source de chaque tâche, et à calculer un majorant sur les comportements temporels, tels que le temps d'exécution des travaux de la tâche (le *temps d'exécution pire-cas*) [CP00, CPRS03, Col01].

Dans tous les cas, que ce soit en ce qui concerne la loi d'arrivée, ou dans l'établissement des caractéristiques temporelles, les modèles de tâches imposent un sur-dimensionnement de par le pessimisme nécessaire qu'ils introduisent. Dans ce cadre, de grands efforts de recherche en temps-réel strict consistent à réduire ce pessimisme, en affinant les modèles de lois d'activation (plus fin que les catégories canoniques : périodique/sporadiques/apériodiques), et/ou en raffinant le calcul des comportements temporels. Citons par exemple les travaux de détermination du temps d'exécution pire-cas par analyse statique du code source, qui tiennent compte de certaines optimisations matérielles (telles que le cache d'instructions, les pipelines, la prédiction de branchement) [Col01, CP01b]. Ces travaux sont cependant obligés de toujours considérer les pires configurations matérielles (pire conflit possible pour l'accès au cache par exemple), ce qui les rend pessimistes. D'autres travaux s'intéressent par conséquent à réduire ce pessimisme en rendant le comportement de l'architecture déterministe et facilement statiquement analysable (par exemple par des techniques de gel de cache, comme dans [PD02]).

Une autre série de paramètres d'entrée importants lors de l'analyse hors-ligne est celle liée au comportement temporel du système d'exploitation. En temps-réel strict, ce sont les paramètres pire-cas qui doivent être pris en compte, ce qui soulève les mêmes problèmes et induit les mêmes approches que pour définir le comportement temporel des tâches, ce qui en retour disqualifie certains services (voir 1.4.2.2).

2.3.1.3 Limitations

En temps-réel strict, la garantie de respect des contraintes temporelles repose fondamentalement sur l'hypothèse que les pires configurations et scénarios sont correctement identifiés, tant au niveau de l'application, que du système d'exploitation, ou encore de

l'environnement, et ceci en tenant compte des partages de ressources entre les traitements. Si tel n'était pas le cas, il pourrait y avoir *dépassement des hypothèses* en cours de fonctionnement, pouvant entraîner le non-respect de certaines contraintes temporelles. Nous verrons au paragraphe 3.4 certaines méthodes pour tolérer ce type de problème.

2.3.2 Tests et simulation

Une autre voie pour l'évaluation de l'implantation, et la confrontation de son comportement aux spécifications, est d'observer le comportement effectif du système, et d'en extraire les caractéristiques pertinentes.

2.3.2.1 Protocoles d'évaluation

Lorsqu'il s'agit d'évaluer le système complet, il s'agit de générer des *tests*, sous la forme de séquences d'événements de l'environnement, qui visent à amener le système à une configuration donnée, et à évaluer le comportement du système dans cette configuration. La pertinence de cette évaluation dépend de la pertinence des tests, et du degré de couverture de l'ensemble des tests. Avec le formalisme considéré, dans les cas les plus simples, il existe des tests qui permettent d'avoir une couverture complète des configurations possibles (*instant critique* pour les systèmes à ordonnancement *rate monotonic* par exemple). Par contre, le formalisme n'est plus adapté pour la génération de tests pour des systèmes beaucoup plus complexes. Pour ces cas pathologiques mais largement courants, il s'agit d'évaluer le comportement du système dans le plus grand nombre de configurations possibles, principalement en le soumettant à la plus grande variété de stimuli possibles. La plate-forme Dynbench par exemple permet de mener ce type de test [SWR+99b, WS99b, SWR+99a].

On peut aussi s'intéresser à *comparer* le comportement temporel de plusieurs systèmes d'exploitation (tout ou parties), ou de plusieurs bibliothèques de fonctions. Auquel cas il s'agit de partir sur la base de tests comparables, et, si possible, avec un degré de couverture correct. On utilise dans ce cas des applications synthétiques qui servent de charge référence pour les tests, ou *benchmark*s (bancs d'essai)[Wei89, VJP97, Heu01]. Pour le temps-réel, il n'existe à notre connaissance aucun standard dans le domaine, hormis les mesures de caractéristiques temporelles isolées présentées en 1.4.2.2. Les approches existantes consistent à réutiliser les applications de charge synthétique non temps-réel, qui ont pour objectif de mesurer la performance brute pour du calcul scientifique (comme par exemple le test Whetstone ou les tests C3I [MVP+96]) ou la performance brute du système d'exploitation (comme par exemple le test Dhrystone), et de les intégrer dans la procédure suivante : constitution d'un système avec des tâches contenant chacune un test de charge, et augmenter progressivement la charge du système (*i.e.* diminuer les périodes des tâches, augmenter le nombre d'opérations pour chaque travail...) jusqu'à dépassement d'une contrainte temporelle [Dru96, Poo96] (*harness test*, ou mesure du seuil d'utilisation). Les critères de comparaison étant alors la configuration au seuil du dépassement d'échéance.

Dans tous les cas, pour mener les tests, on doit choisir entre une exécution réelle

instrumentée d'une implantation existante, et l'évaluation sur une exécution simulée (*i.e.* sur une machine et dans un environnement fictifs).

2.3.2.2 Exécution réelle instrumentée

Il s'agit de disposer le système complet (matériel et logiciel) en présence d'un ou plusieurs tests, et d'extraire des informations sur son comportement en réponse aux tests, grâce à l'instrumentation du système. Cette méthode offre l'avantage de prendre en compte *de facto* toutes les caractéristiques de tous les éléments du système, en particulier les coûts temporels liés au système d'exploitation et au matériel.

La mise en place de l'instrumentation est le point délicat de cette méthode.

Il est possible d'instrumenter le système au niveau matériel (observateur de bus, sondes logiques par exemple) et de façon non intrusive. Mais cette technique se révèle délicate, limitée à une configuration matérielle donnée, souvent financièrement coûteuse, de plus en plus difficile à mettre en œuvre compte-tenu de l'intégration de composants toujours plus élevée, et peut nécessiter un puissant dispositif d'analyse des traces pour *interpréter* les observations, car les observations générées sont très nombreuses, et chacune porteuse d'une information élémentaire brute (trace des instructions exécutées, trace des interruptions matérielles, trace des changements de contexte par exemple).

Il est également possible d'ajouter les mécanismes d'instrumentation dans le logiciel (génération de traces de plus haut niveau, telles que l'activation de tâche, le dépassement d'échéance, le changement de date système), ce qui permet de disposer d'informations exploitables plus directement (phase d'interprétation des résultats plus légère). Cette approche reste délicate, car il s'agit de modifier le noyau ou l'application d'un système existant, ce qui peut s'avérer une tâche fastidieuse. De gros efforts sont cependant faits pour permettre le remplacement de *modules* entiers (en particulier l'ordonnanceur) dans le noyau (tels que DSLib [8] ou RTAI [41] pour Linux, ou Vassal [CJ98] pour Windows NT), ce qui rend l'instrumentation plus simple, puisqu'il suffit alors d'intercepter les interactions entre les modules.

Cette approche possède un autre inconvénient majeur : l'*effet sonde* (ou *probe effect* en anglais), autrement dit l'influence des dispositifs d'instrumentation sur le comportement du système, et donc sur les observations par instrumentation. Ces effets peuvent être directs (surcoûts d'exécution parfaitement localisés aux endroits où l'instrumentation se situe), auquel cas il est possible d'évaluer leur coût (principe de calibrage de la tare utilisé dans [MRW92]), ce qui rend possible la correction des mesures quand le comportement du système est déterministe. Mais ces effets peuvent être également indirects, soit au niveau du comportement microscopique (effet sur les pipelines, les caches d'instructions, de prédiction de branchement, de traduction d'adresse et de données), ou macroscopique (l'ajout des routines d'instrumentation allonge la durée des tâches, ce qui peut modifier leur ordonnancement), auquel cas la prise en compte de l'effet sonde est considérablement plus complexe (effets non linéaires), voire trop complexe en pratique. Une solution simple est de laisser le code d'instrumentation dans la version finale du produit, mais dans ce cas, le surdimensionnement inutile qui résulte peut conduire à des surcoûts financiers (besoins supplémentaires en mémoire, ou en fréquence de travail

du processeur par exemple), ou à une surconsommation d'énergie par exemple.

Enfin, il existe des méthodes hybrides, dans lesquelles le logiciel active certaines ressources matérielles (registres, ports d'entrée/sortie, plage d'adresses du bus) pour signifier l'occurrence d'un événement à traiter par le matériel, ou par un autre système. Avec cette méthode, on peut choisir de conserver l'instrumentation logicielle pour la version finale du produit avec un surcoût qui reste raisonnable (ce qui supprime l'effet de sonde). Mais lors de la phase de développement, les inconvénients liés à l'instrumentation matérielle restent valables.

2.3.2.3 Évaluation par simulation

Il s'agit d'exécuter le système dans un contexte matériel et d'environnement simulés. Le simulateur se charge de fournir les mécanismes d'instrumentation, voire d'analyse des observations. Puisque les dispositifs d'instrumentation font partie du simulateur, ils n'interfèrent pas avec l'exécution du système, ce qui signifie que l'effet sonde n'existe pas dans cette approche. Une autre conséquence est que la procédure de simulation et de comparaison de systèmes dans différentes configurations ou avec des variantes d'implantation, est plus légère : il suffit de se concentrer sur l'implantation du système, ou de réutiliser les implantations existantes, et de ne pas se soucier de l'intégration de l'instrumentation. La plate-forme présentée dans la partie II autorise les simulations de ce type pour des systèmes complets.

En contrepartie, en plus du problème des tests à définir pour l'évaluation, la représentativité des résultats de simulation dépend de sa fidélité. D'abord, une première contrainte est que le système simulé doit être pertinent vis-à-vis du produit final : l'idéal étant que simulation et produit final se partagent la même base de code. Une seconde contrainte forte est que les comportements des tâches et du système d'exploitation doivent rendre compte du comportement effectif le plus finement possible, c'est-à-dire avec le moins d'hypothèses simplificatrices possible, ce qui n'est pas forcément possible avec de nombreux simulateurs existants. En particulier, dans ce contexte, l'utilisation de lois statistiques pour définir le comportement temporel du système est insuffisante. Nous verrons au chapitre 4 qu'il est possible de rendre compte de ces comportements à plusieurs niveaux de précision : de la granularité instruction à celle de la tâche. Une difficulté est alors de faire un compromis entre précision et efficacité de la simulation. Une autre difficulté du même ordre est que les comportements temporels du système d'exploitation (en particulier les coûts liés à l'ordonnancement) et du matériel soient pris en compte dans la simulation, et à la même granularité que pour l'application, ce qui n'est pas possible avec de nombreux simulateurs existants.

Dans le chapitre suivant, nous nous intéressons aux résultats classiques obtenus par analyse hors-ligne de modèles simples. Le chapitre 4 entrera dans les détails des outils de simulation disponibles.

Chapitre 3

Ordonnancement et analyse d'ordonnancement

Nous nous concentrons maintenant sur une pièce fondamentale d'un système temps-réel : l'ordonnanceur. Pour chaque processeur, l'ordonnanceur est en charge de définir l'*ordonnancement*, qui est une séquence infinie d´éléments du type : `(date, identifiant_travail)`. Chaque élément correspond à l'*élection* de la tâche à exécuter, puis à un *changement de contexte* (si le travail n'est pas terminé), qui fait suite à une *décision d'ordonnancement* par l'ordonnanceur.

Dans ce chapitre, nous citons une série de résultats scientifiques qui ont été obtenus dans ce domaine central de l'ingénierie temps-réel que constitue l'analyse d'ordonnancement. Nous commençons par dresser un panorama des contraintes et propriétés d'ordonnancement, avant de donner les propriétés des problèmes algorithmiques associés en termes de complexité, et les grandes classes d'approches courantes (section 3.1). Nous développons ensuite plus avant les propriétés d'une classe particulière d'ordonnancement, les ordonnancements à priorité (section 3.2), pour des systèmes constitués exclusivement de tâches temps-réel strict, avec ou sans prise en compte des coûts système, avec ou sans contraintes de ressources, avec ou sans contraintes de précédence. Puis nous levons progressivement les restrictions de ce modèle : support de tâches apériodiques (section 3.3), gestion de la *surcharge* (section 3.4). Nous terminons (section 3.5) en donnant un aperçu de modèles de systèmes qui ne sont pas fondés sur un ordonnancement à priorité, puis en présentant brièvement quelques approches qui s'adaptent à un manque d'informations sur le comportement du système ou de l'environnement.

Ces présentations introduisent les ordonnanceurs qui figurent dans la plate-forme de simulation Artisst (décrite en partie II), dont certains sont soumis à évaluation (partie III).

3.1 Caractéristiques des problèmes et des méthodes d'ordonnancement

Après avoir introduit quelques éléments de terminologie (section 3.1.1), nous décrivons les caractéristiques de quelques problèmes d'ordonnancement (section 3.1.2), et présentons les grandes classes de solutions courantes à ces problèmes (section 3.1.3).

3.1.1 Classes des problèmes

On indique ici les caractéristiques liées aux systèmes cible de l'ordonnancement ou à l'ordonnancement lui-même, qui forment les paramètres du problème d'ordonnancement qu'on cherche à résoudre.

Monoprocesseur/multiprocesseur

L'ordonnancement est de type monoprocesseur si toutes les tâches ne peuvent s'exécuter que sur un seul et même processeur. Si plusieurs processeurs sont disponibles dans le système, l'ordonnancement est multiprocesseur. Dans toute la suite, nous nous limitons au problème de l'ordonnancement monoprocesseur.

Centralisé/distribué

Lorsqu'un système est multiprocesseur, l'ordonnancement est distribué si les décisions d'ordonnancement sont prises par un algorithme *localement en chaque nœud*.

Il est centralisé lorsque l'algorithme d'ordonnancement pour tout le système, multiprocesseur ou non, est déroulé *sur un nœud privilégié*.

Préemptif/non-préemptif

L'ordonnancement est préemptif lorsque les préemptions (voir 1.3.1) sont autorisées. On peut remarquer que dans le cas des ordonnanceurs monoprocesseur, la problématique de la synchronisation sur ressource n'existe pas en non-préemptif puisque il ne peut pas y avoir de concurrence sur l'accès aux ressources en l'absence de préemption.

Avec/Sans insertion de temps creux : oisif/non oisif

L'ordonnanceur peut avoir la propriété suivante : à partir du moment ou au moins une tâche est prête à être exécutée, alors l'ordonnanceur en élit forcément une parmi elles. Dans ce cas, l'ordonnanceur est *non oisif* ou *au plus tôt*, c'est-à-dire qu'il fonctionne *sans insertion de temps creux* (*non-idling* ou *work-conservative* en anglais).

Dans certains cas, un système peut n'être ordonnançable qu'en n'élisant aucune tâche pendant un certain temps, alors qu'il en existe au moins une prête à être exécutée. Dans ce cas, l'ordonnanceur est *oisif*, c'est-à-dire qu'il fonctionne par *insertion de temps creux* (*with inserted idle times* en anglais).

En-ligne/hors-ligne

Un ordonnancement hors-ligne (*off-line* en anglais) signifie que la séquence d'ordonnancement est prédéterminée : dates de début d'exécution des tâches, de préemption/reprise éventuelles. En pratique, l'ordonnancement prend la forme d'un *plan hors-ligne* (ou *statique*), exécuté de façon répétitive (on parle aussi d'ordonnancement *cyclique*, qui définit le *cycle majeur*) : à chaque *tick* d'horloge (on parle de cycle *mineur*) une tâche particulière du cycle majeur courant est exécutée jusqu'à terminaison ou jusqu'au *tick* d'horloge suivant. L'ordonnanceur s'appelle dans ce cas aussi un *séquenceur*.

Un ordonnancement en-ligne correspond au déroulement d'un algorithme qui tient compte des tâches prêtes effectivement présentes dans la *file d'ordonnancement* (*run-queue* en anglais) lors de chaque décision d'ordonnancement. Les ordonnanceurs en-ligne peuvent reposer sur la construction de plans d'ordonnancement en cours de fonctionnement, auquel cas ils sont dits à plan *dynamique* [KSSR96]. Mais plus couramment, ces ordonnanceurs sont fondés sur la notion de priorité (voir 3.1.3.2).

Il est par ailleurs possible de transformer un ordonnancement à plan statique en ordonnancement en-ligne (par exemple à priorité statique préemptif [DFP01] ou non-préemptif [Bat98], ou à priorité dynamique [CA97]).

Optimal/non optimal

Par définition, un algorithme d'ordonnancement *optimal* [GRS96] pour un problème d'ordonnancement donné parmi une classe donnée de solutions au problème (ordonnancement en-ligne, hors-ligne, distribué...) est tel que : si un système est *ordonnançable* (voir 2.2.1) par au moins un algorithme de la même classe, alors le système est ordonnançable par l'algorithme d'ordonnancement optimal considéré (il peut y avoir plusieurs algorithmes d'ordonnancement optimaux pour une classe donnée). En conséquence, si un système n'est pas ordonnançable par un ordonnanceur optimal d'une classe donnée, alors il ne l'est par aucun ordonnanceur de la même classe.

3.1.2 Complexité des problèmes

Le problème qui consiste à définir un ordonnancement peut être ramené à un problème d'optimisation. Par exemple, beaucoup de travaux s'intéressent à minimiser le retard maximal parmi toutes les tâches ("$L_{\max}$" dans la suite). Car une fois ce retard maximal minimisé, la correction temporelle du système est équivalente à vérifier que ce retard maximal est négatif ou nul.

Nous nous intéressons ici à la complexité [Coo71, Coo83, GJ79] de quelques problèmes pertinents pour l'ordonnancement [SSNB94][45], et nous nous limitons à présent à l'ordonnancement monoprocesseur.

Le tableau 3.1 indique la complexité de quelques problèmes d'ordonnancement respectivement non-préemptifs et préemptifs sur monoprocesseur, en fonction de la connaissance des dates d'activation (colonne *tâches concrètes*), l'insertion de temps creux (colonne *temps creux*), la prise en compte de contraintes de ressources (colonne *ressources*) ou de précédence (colonne *précédences*).

monoprocesseur non-préemptif						
tâches concrètes	**temps creux**	**précédences**	**ressources**	**problème**	**complexité**	
non	non	non	indifférent	minimiser $L_{\max}$	polynomiale	voir 3.1.3.2
non	oui	non		ordonnançabilité	NP-complet	[GJ79, HV95]
oui	non	non		minimiser $L_{\max}$	NP-complet	[GJ79, JSM91]
oui	non	*series-parallel*[a]		minimiser $L_{\max}$[b]	polynomiale	[SSNB94, Law78, GL95]
oui	non	oui		minimiser $L_{\max}$	NP-difficile	[SSNB94, Law78]
synchrones	non	oui		minimiser $L_{\max}$	polynomiale	[SSNB94, Law73]

monoprocesseur préemptif						
tâches concrètes	**temps creux**	**précédences**	**ressources**	**problème**	**complexité**	
non	non	non	verrous seuls	ordonnançabilité	NP-difficile	[Mok83]
non	non	non	oui	ordonnancement optimal	impossible	[Mok83], voir 3.2.5.2
non + temps d'exécution tous égaux	non	non	oui	minimiser $L_{\max}$	polynomiale	[Mok83]
indifférent	non	non	non	minimiser $L_{\max}$	polynomiale	voir 3.1.3.2
indifférent	non	oui	non	minimiser $L_{\max}$	polynomiale	voir 3.1.3.2
indifférent	non	oui	oui	ordonnancement	NP-difficile	[SSNB94]

TAB. 3.1: Complexité de problèmes d'ordonnancement monoprocesseur

[a] Ce type de graphe rassemble les arbres convergents et divergents, mais n'est pas du tout adapté aux cas courants où des échanges de messages peuvent se faire en travers de l'arbre [SSNB94].

[b] Ou toute fonction de chaînes de travaux vers $\mathbb{R}$ qui possède la propriété : $f(\alpha) \leq f(\beta) \Rightarrow f(a..\alpha\beta...b) \leq f(a..\beta\alpha..b)$.

On peut remarquer que le problème à résoudre est d'autant plus complexe que les préemptions sont interdites ou que les temps creux sont autorisés (insérer des temps creux dans l'ordonnancement revient à être capable de prévoir les activations à venir).

En pratique, pour résoudre les problèmes difficiles qui font intervenir des ressources et éventuellement des contraintes de précédences, il existe des ordonnancements statiques [JD90], des approches sub-optimales (telles que le *kernelized monitor* [Mok83] ou [Man98, MMM00b, RS94] par exemple), ou des protocoles pour accéder aux ressources en utilisant des algorithmes d'ordonnancement classiques (comme nous le verrons en 3.2.5.2).

3.1.3 Solutions aux problèmes d'ordonnancement

Dans la suite, nous nous intéressons d'abord au problème le plus simple : l'ordonnancement monoprocesseur sans ressource et sans précédence. Nous présentons ici les solutions simples les plus courantes, et verrons comment ces solutions peuvent être aménagées pour supporter des systèmes avec ressources et contraintes de précédence.

3.1.3.1 Ordonnancement hors-ligne

L'ordonnancement hors-ligne repose sur la définition d'un plan statique. Les traitements pour l'établissement du plan (comme [SA00, CCS02, Foh94] avec des algorithmes *ad hoc*, optimaux [JD90], ou sub-optimaux [EJ00] par exemple) sont effectués hors-ligne, et peuvent se permettre d'être très complexes. En particulier, ce type d'ordonnancement se prête bien aux cas de systèmes complexes tels que les systèmes temps-réel distribués [SRG89] (le problème principal restant alors la synchronisation des horloges locales pour un déroulement cohérent du plan).

Intérêts

Cette politique d'ordonnancement possède l'avantage de se contenter d'un support d'exécution très réduit (l'accès à une base de temps suffit). Par là même, elle introduit un surcoût d'exécution lié à l'ordonnancement très faible [KFG+92] : il suffit qu'une horloge système incrémente la position courante dans le plan d'ordonnancement. Également, l'environnement n'influence pas le déroulement du plan, ce qui rend le système robuste car immunisé contre une partie des dépassements d'hypothèse dus à la mauvaise modélisation de l'environnement. Enfin, la garantie que le système respectera toutes ses contraintes temporelles est une propriété facile à appréhender : il "*suffit*" que le plan soit vérifié hors-ligne, sous réserve que les temps d'exécution prévus soient corrects. Ceci fait de cette solution celle qui paraît la plus rigoureuse et la plus stricte ; nous verrons cependant que dans les cas de systèmes centralisés simples il existe des solutions en-ligne qui offrent tout autant de garanties.

Limitations

En contrepartie, il est nécessaire de se ramener à des plans périodiques, ou à des plans qui supposent connues toutes les dates d'arrivées (éventuellement non périodiques)

de toutes les tâches. Cela conduit à sur-contraindre le système, en particulier quand on doit traiter des tâches périodiques de période non multiple du cycle mineur, ou des événements de fréquence d'occurrence maximale élevée mais de faible fréquence moyenne. Au mieux, ces contraintes excessives entraînent un surdimensionnement du système, et au pire l'impossibilité de déterminer un ordonnancement faisable. De plus, les plans générés sont très sensibles aux petites modifications du code ou des fonctionnalités qui peuvent être faites (ajout d'une tâche supplémentaire par exemple) : ces modifications affectent le plan dans son ensemble (non localité), ce qui pose problème d'intégration lorsque plusieurs acteurs (entreprises, ingénieurs) coopèrent sur le même projet (contraire aux contraintes de confidentialité par exemple). Un autre problème qui se pose est que le plan doit être capable de définir l'ordonnancement sur une durée égale à l'*hyperpériode* des tâches, qui peut être très grande, ce qui peut impliquer une grande consommation mémoire pour stocker le plan, inadaptée aux petits systèmes embarqués.

Il existe cependant des ordonnancements par plan statique qui peuvent être complétés par un algorithme d'ordonnancement en-ligne (comme [Foh94, IF99], par une méthode similaire à la réquisition de temps creux, que nous verrons en 3.3.2.4), afin de prendre en compte l'activation de tâches sporadiques ou apériodiques par exemple.

Nous ne détaillons pas davantage ce type d'ordonnancement dans ce travail. Dans la suite, nous nous intéressons aux algorithmes d'ordonnancement en-ligne sur monoprocesseur.

3.1.3.2 Ordonnanceurs en-ligne à priorité

La majorité des ordonnanceurs temps-réel en-ligne reposent sur la notion de *priorité* : lors de chaque décision d'ordonnancement, la tâche de plus haute priorité est élue. Les priorités peuvent être attribuées hors-ligne, auquel cas l'algorithme d'ordonnancement est dit *à priorité statique* ou *fixe* ; ou en-ligne, auquel cas il est dit *à priorité dynamique*.

Intérêts

Par rapport à l'ordonnancement hors-ligne, l'ordonnancement à priorité est une solution plus souple lorsque des modifications sont introduites, moins gourmande en mémoire, et offre tout autant de garanties que l'ordonnancement hors-ligne. Comme nous le verrons par la suite, ces ordonnanceurs ont l'avantage de s'adapter aux évolutions de l'environnement (support des tâches sporadiques, admission de tâches apériodiques), ou du système (caractérisation temporelle incomplète de certaines parties du système par exemple). De plus, les algorithmes simples que nous décrivons ci-après peuvent être étendus pour supporter des modèles de tâches plus riches, des contraintes de précédence, ou des synchronisations de ressources, ce qui leur permet de rivaliser avec les ordonnancements hors-ligne de ce point de vue.

Limitations

D'un point de vue théorique, plus il y a de contraintes (ressources, précédences par exemple), moins le processeur peut être effectivement utilisé, et dans certains cas, aucun ordonnancement en-ligne faisable ne peut exister.

D'un point de vue pratique, ce type d'ordonnanceur est plus complexe que les ordonnanceurs à plan statique : les décisions d'ordonnancement consistent à calculer et à comparer des priorités relatives entre les tâches. Partant, ce type d'ordonnanceur est aussi plus exigeant en termes de services que doit fournir le support d'exécution : celui-ci doit, entre autres, pouvoir gérer des files de tâches à ordonnancer, et doit pouvoir déterminer quand une décision d'ordonnancement est nécessaire. Il en résulte par là même des appels à l'ordonnanceur plus fréquents qu'en ordonnancement hors-ligne, ce qui accroît encore le surcoût d'exécution engendré.

En pratique, les supports d'exécution les plus courants et les standards disponibles les plus utilisés, fournissent au moins les fonctionnalités nécessaires à l'ordonnancement à priorité fixe.

3.1.3.3 Choix d'ingénierie

Le choix entre ordonnancement à priorité statique et à priorité dynamique est d'autant plus difficile que les différentes extensions aux modèles simples vus dans la suite du chapitre existent en général pour les deux sous-familles.

Les ordonnancements à priorité statique s'intègrent très bien dans un système d'exploitation sans qu'il y ait besoin de lui faire gérer des contraintes de temps : il suffit que le système dispose d'un ordonnanceur à priorité, ce qui est pour le moins courant. En contrepartie, le seuil d'utilisation qu'il est possible d'atteindre (*i.e.* le rendement utile du processeur) est moins élevé qu'en priorité dynamique. De ce point de vue, il existe des versions mixtes (ordonnancements à priorité statique et dynamique combinés) qui effacent la frontière entre ordonnancement à priorité statique et dynamique [Zub98], et qui permettent de faire un compromis entre rentabilisation du processeur et complexité d'ordonnancement en-ligne.

D'autre part, en priorité dynamique, lorsqu'une échéance est dépassée, elle affecte potentiellement **toutes** les autres tâches, quel que soit leur niveau de *criticité*, ce qui peut amener des dépassements d'échéances en cascade non contrôlés (voir section 3.4.1). De ce point de vue, les ordonnancements à priorité statique ont l'avantage de mieux isoler ces fautes (les tâches de priorité supérieure à la tâche fautive ne sont pas touchées), mais moins encore que les ordonnancements hors-ligne par plan (une tâche programmée à telle date sera toujours exécutée, ou l'ensemble du système sera arrêté, que la tâche précédente ait terminé dans les temps ou non).

3.2 Ordonnanceurs à priorité

Dans cette partie, nous définissons les algorithmes d'ordonnancement à priorité que nous considérons dans la suite (section 3.2.1), avant de donner quelques résultats

d'optimalité les concernant (section 3.2.2). Nous indiquons ensuite les conditions d'ordonnançabilité qui leur sont associées, pour un modèle de système très restrictif (section 3.2.3). Cette présentation sera davantage détaillée et calculatoire que dans toutes les sections qui suivront, afin de donner un aperçu de la démarche employée dans le domaine. Nous levons ensuite certaines restrictions du modèle (sections 3.2.4), cette fois-ci sans rentrer dans les détails calculatoires. Puis nous donnons quelques extensions pour supporter des contraintes d'ordonnancement supplémentaires (section 3.2.5), telles que la prise en compte des accès concurrents à des ressources protégées, toujours en préférant donner un aperçu des techniques et de leur principe, plutôt qu'en les détaillant.

3.2.1 Politiques d'ordonnancement à priorité courantes

Nous nous plaçons ici dans le cadre simple de l'ordonnancement monoprocesseur non oisif à priorité, lorsque toutes les tâches sont indépendantes.

Les ordonnancements à priorité les plus courants sont les suivants :

RM (pour *Rate Monotonic*) : ordonnancement à priorité statique pour tâches périodiques/sporadiques avec échéance relative égale à la période/délai d'inter-arrivée. Une tâche est d'autant plus prioritaire que sa période est petite.

DM (pour *Deadline Monotonic*) : ordonnancement à priorité statique pour tâches sporadiques. Une tâche est d'autant plus prioritaire que son échéance relative est petite. DM équivaut à RM quand l'échéance relative est égale à la période.

EDF (pour *Earliest Deadline First*) : ordonnancement à priorité dynamique. Une tâche est d'autant plus prioritaire que sa date d'échéance absolue est proche de la date courante. Le cas particulier où les tâches sont synchrones est parfois appelé **EDD** (pour *Earliest Due Date*), ou algorithme de Jackson.

LLF (pour *Least Laxity First*) : ordonnancement à priorité dynamique. Les tâches sont d'autant plus prioritaires que leur laxité (la tâche étant considérée en isolation) est faible à la date courante.

En dépit de leur simplicité, ces ordonnanceurs possèdent des propriétés relativement fortes. C'est ainsi qu'ils servent de base à quantités de variantes, comme nous le voyons par la suite.

3.2.2 Résultats d'optimalité

Les ordonnancements simples précédents ont les propriétés intéressantes suivantes :

1. EDF et LLF sont optimaux parmi les ordonnancements préemptifs non oisifs [LL73]. EDF est également optimal relativement à la minimisation du retard maximal des tâches [But97].
2. EDF est optimal parmi les ordonnancements non-préemptifs non oisifs [GMR95, GRS96].
3. RM est optimal parmi les ordonnancements préemptifs non oisifs à priorité statique, pour tâches périodiques non-concrètes, ou périodiques synchrones, ou sporadiques, lorsque l'échéance relative est égale à la période (ou au délai d'inter-arrivée dans le cas sporadique) [LL73].

4. DM est optimal parmi les ordonnancements préemptifs non oisifs à priorité statique, pour tâches périodiques non-concrètes ou synchrones, ou sporadiques, lorsque l'échéance relative est inférieure à la période [ABRW91, But97, LW82].
5. Dans les cas où les tâches périodiques concrètes ne sont jamais synchrones, ni RM, ni DM ne sont optimaux parmi les ordonnancements préemptifs non oisifs à priorité statique. Dans [Aud91], on donne l'affectation optimale des priorités pour l'ordonnancement préemptif non oisif à priorité statique pour le cas général (*i.e.* qui reste valable quand les échéances relatives et les périodes ne sont pas reliées).
6. DM est optimal parmi les ordonnancements non-préempifs non oisifs à priorité statique, pour tâches périodiques non-concrètes, ou périodiques synchrones, ou sporadiques, lorsque l'échéance relative est inférieure à la période [Bat98].

3.2.3 Quelques conditions de faisabilité

Dans la suite, les notations suivantes seront utilisées : dans le système Γ de N tâches, chaque tâche $\tau_i, i \in [1, N]$ possède un temps d'occupation[1] sur le processeur C_i et une échéance relative D_i. Si τ_i est périodique, T_i représente la période. Si τ_i est sporadique, T_i représente le délai d'inter-arrivée.

Le taux d'utilisation de ce système de tâches s'écrit alors : $U = \sum_{i=1}^{N} \frac{C_i}{T_i}$, et on rappelle qu'une condition nécessaire pour que le système soit faisable est $U \leq 1$ (voir 1.3.2).

3.2.3.1 EDF préemptif

Pour un ensemble de tâches périodiques tel que $\forall i, D_i = T_i$, la condition nécessaire et suffisante de faisabilité [LL73] est :

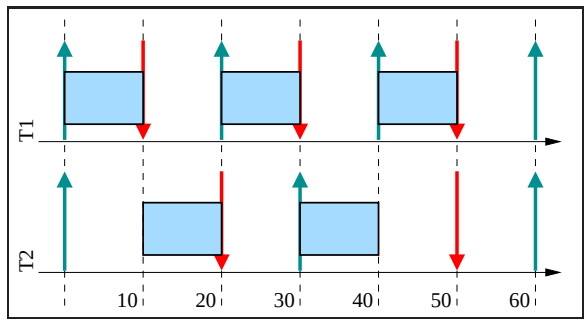
$$U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1 \tag{3.1}$$

Cette condition reste nécessaire et suffisante dans le cas où $\forall i, D_i \geq T_i$ [BMR90].

Pour un ensemble de tâches périodiques tel que $\forall i, D_i \leq T_i$, la condition précédente n'est bien sûr plus suffisante puisque D_i peut être arbitrairement petit. Dans ce cas, une condition suffisante de faisabilité analogue existe, qui revient à considérer le système comme étant constitué de tâches sporadiques de délai inter-arrivée D_i, et qui s'écrit $\sum_{i=1}^{N} \frac{C_i}{D_i} \leq 1$. Mais cette condition pessimiste n'est pas nécessaire comme le prouve l'exemple de la figure 3.1.

Dans [BMR90, GRS96] (par calcul du *taux de demande* du processeur, ou *processor demand* en anglais) et [Spu96] (par calcul du temps de réponse), deux conditions nécessaires et suffisantes pour la faisabilité d'un ensemble de tâches périodiques avec D_i et T_i arbitraires sont données. Ces formules ont l'inconvénient d'être complexes, et

[1] Pour le moment, nous considérons qu'il s'agit du temps d'exécution effectif, toujours constant. En 3.2.4, nous nuancerons cette définition.



	τ_1	τ_2
T_i	20	30
C_i	10	10
D_i	10	20

$$\Rightarrow \sum_{i=1}^{N} \frac{C_i}{D_i} = 1.5 > 1$$

$(U = 83.33 < 1)$

FIG. 3.1: Système de 2 tâches périodiques synchrones ordonnançable par EDF

c'est pourquoi en pratique, on utilise souvent la condition suffisante (*i.e.* pessimiste) plus simple suivante :

$$\sum_{i=1}^{N} \frac{C_i}{\min(D_i, T_i)} \leq 1$$

3.2.3.2 EDF non préemptif

Considérons d'abord le cas non oisif. La condition nécessaire et suffisante de faisabilité pour un ensemble de tâches concrètes apériodiques correspond à dérouler l'ordonnancement. Dans le cas de tâches apériodiques non-concrètes, une condition nécessaire et suffisante en $O(N^2)$ est donnée dans [GMR95].

Dans le cas de tâches sporadiques ou périodiques non-concrètes, une condition nécessaire et suffisante pseudo-polynomiale est donnée dans [JSM91] et repose sur la demande en ressource processeur.

Dans le cas de tâches périodiques concrètes, la même condition devient suffisante seulement, et le problème qui consiste à déterminer l'ordonnançabilité du système est NP-complet [GMR95, JSM91] (des conditions de faisabilité dans le cas synchrone et non synchrone sont données dans [GMR95]).

Puisque l'ordonnancement non oisif peut être vu comme un cas particulier d'ordonnancement oisif, toutes ces conditions de faisabilité restent valables dans le cas oisif, mais ne sont plus que suffisantes seulement. Dans le cas oisif (NP-complet en général), [GMR95] montre qu'à partir d'un ordonnancement oisif faisable, on peut dériver un ordonnancement par EDF oisif ; il en est déduit un algorithme de type *branch and bound* pour la détermination d'un ordonnancement EDF oisif (complexité exponentielle).

3.2.3.3 Ordonnancement préemptif à priorité statique

Dans ce paragraphe et le suivant, les tâches τ_i sont classées par ordre de priorités π_i décroissantes : $i < j \Rightarrow \pi_i \geq \pi_j$ ($\pi_i = 0$ étant la plus faible priorité possible) ; dans

le cas d'une affectation des priorités suivant RM ou DM, $\pi_i = \frac{1}{\min(D_i,T_i)}$ par exemple. Dans toute la suite, on ne considère que les ordonnancements à priorité fixe non oisifs.

Une première condition de faisabilité nécessaire simple est $U \leq 1$ (puisque EDF est optimal parmi tous les ordonnancements non oisifs).

*Condition suffisante fondée sur l'utilisation, et notion d'*instant critique

Dans [LL73], la condition suffisante suivante sur la faisabilité d'un ensemble de tâches périodiques synchrones est montrée, qui suppose l'affectation des priorités suivant RM ou DM :

$$\sum_{i=1}^{N} \frac{C_i}{\min(D_i, T_i)} \leq N.(2^{\frac{1}{N}} - 1) \tag{3.2}$$

Cette condition s'écrit $U \leq 1$ dans le cas particulier de tâches harmoniques (et devient alors nécessaire et suffisante).

Dans [LL73], il est également montré la propriété fondamentale selon laquelle la pire configuration de tâches périodiques non-concrètes est obtenue pour le premier travail des tâches lorsque toutes les tâches sont activées au même instant (*instant critique*, correspondant à un ensemble de tâches synchrones). Il en découle que la condition 3.2 reste une condition de faisabilité suffisante pour des ensembles de tâches périodiques non-concrètes ou sporadiques.

Toute une série de résultats sont énoncés dans la littérature du domaine, qui étendent cette analyse fondée sur l'utilisation, dite RMA (pour *Rate Monotonic Analysis*), pour supporter par exemple des contraintes de dépendances entre les tâches (nous y revenons dans les sections qui suivent).

Condition nécessaire et suffisante fondée sur la demande en processeur

Une première condition pseudo-polynomiale nécessaire et suffisante pour l'ordonnançabilité de tâches périodiques non-concrètes ou sporadiques est donnée dans [LSD89]. Cette condition, appelée TDA (pour *Time Demand Analysis*), fait intervenir une formulation de la demande en ressource processeur pour chaque tâche, et est valable pour le cas simple $\forall i, D_i \leq T_i$ avec affectation des priorités quelconque.

Condition nécessaire et suffisante fondée sur le calcul du temps de réponse

Tâches périodiques non concrètes, ou sporadiques. Une condition nécessaire et suffisante équivalente à la précédente, mais plus couramment utilisée, pour la faisabilité de tâches périodiques non concrètes ou sporadiques est donnée dans [JP86, Tin92a, Bur94]. Elle repose sur le calcul du temps de réponse pire cas rt_i des tâches, s'intitule RTA (pour *Response Time Analysis*), et s'écrit $\forall i, rt_i \leq D_i$. Le calcul de rt_i ne suppose pas d'affectation des priorités particulières, et peut se calculer sans hypothèse sur la relation entre les T_i et les D_i.

– Dans le cas le plus simple où $\forall i, D_i \leq T_i$ [JP86], rt_i se calcule en prenant en compte les interférences venant des tâches de priorités supérieures : c'est un calcul itératif qui vise à élargir un intervalle de temps jusqu'à ce que celui-ci contienne à la fois le temps d'exécution de τ_i, et les interférences par les tâches de priorité supérieure (*i.e.* pour une fenêtre de taille w, cette interférence s'écrit $\sum_{j<i} C_j.\lceil \frac{w}{T_j} \rceil$). Lorsque $U \leq 1$ et $\forall i, D_i \leq T_i$, rt_i est ainsi le point fixe de la suite[2] :

$$\begin{aligned} rt_i^{(0)} &= C_i \\ \forall k \geq 1, rt_i^{(k)} &= C_i + \sum_{j<i} C_j . \left\lceil \frac{rt_i^{(k-1)}}{T_j} \right\rceil \end{aligned} \tag{3.3}$$

– Dans le cas général [Tin92a, Bur94] (tâches périodiques non concrètes où sporadiques, D_i et T_i non reliés), il faut en plus prendre en compte les interférences de τ_i avec elle-même quand $D_i > T_i$ (*i.e.* dans ce cas, plusieurs travaux de τ_i peuvent être prêts à s'exécuter en même temps).

Tâches périodiques concrètes. Lorsque les tâches périodiques sont concrètes, ces conditions d'ordonnançabilité deviennent suffisantes seulement. En effet, lorsque les dates d'activation des premiers travaux des tâches font que les tâches ne sont jamais activées simultanément, le temps de réponse pire cas qui se manifeste effectivement peut être strictement plus petit que celui calculé dans les formules ci-dessus, qui considèrent le pire temps de réponse toutes configurations confondues. Dans [Aud91], en plus de définir l'affectation optimale des priorités, on présente à la fois un test qui permet de savoir si l'ensemble des tâches concrètes peut ou non être synchrone (auquel cas les conditions précédentes sont nécessaires et suffisantes), et le test de faisabilité de l'ensemble de tâches qui tient compte des dates de démarrage des premiers travaux.

3.2.3.4 Ordonnancement non préemptif à priorité statique

Dans le cas général (D_i et T_i non reliés) de tâches périodiques non concrètes ou sporadiques, [GRS96] donne une condition nécessaire et suffisante d'ordonnançabilité d'un ensemble de tâches, analogue à RTA. Il y est également montré que l'affectation des priorités suivant RM ou DM n'est plus optimale dans le cas non-préemptif (sauf si $\forall i, D_i \leq T_i$ et $\forall i \neq j, D_i < D_j \Rightarrow C_i \leq C_j$), mais que l'affectation optimale pour le cas préemptif donnée dans [Aud91] reste optimale dans le cas non préemptif non oisif.

3.2.3.5 Autres ordonnancements à priorité statique

Les travaux [WS02, WS99a, GP96] proposent un modèle de tâches à priorité statique intermédiaire entre le préemptif et le non préemptif qui permet d'allier l'intérêt des deux méthodes. Il s'agit d'associer à chaque tâche deux priorités : la priorité de la tâche tant qu'elle n'a pas commencé son exécution (le *seuil de préemption* ou *preemption threshold*

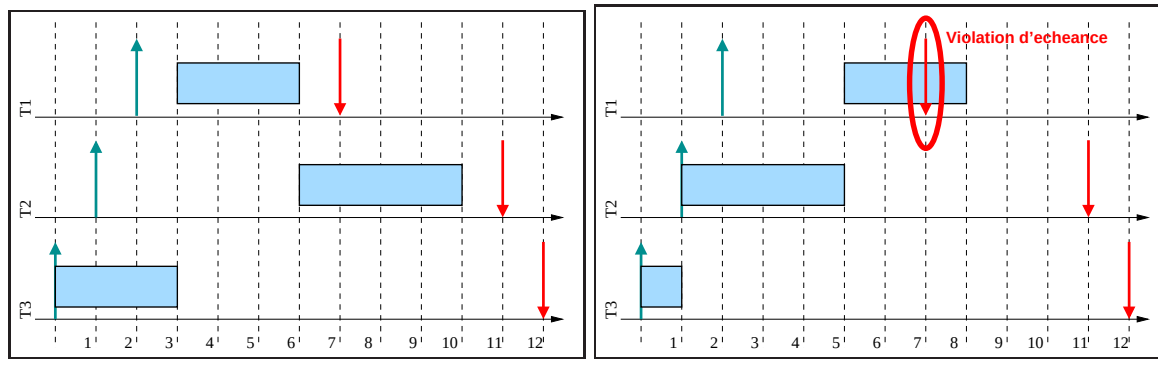
[2] Cette suite admet un point fixe dès que $\sum_{j \leq i} \frac{C_j}{T_j} \leq 1$, or ceci est vrai puisque $\sum_{j \leq i} \frac{C_j}{T_j} \leq U \leq 1$.

en anglais), et la priorité π_i quand elle a commencé son exécution. Les cas préemptif et non préemptif sont des cas particuliers avec un seuil de préemption respectivement de π_i et 0. Ces travaux proposent à la fois un test de faisabilité, et un algorithme d'affectation optimal polynomial des seuils de préemption.

3.2.4 Limitations

Les conditions de faisabilité qui ont été formulées ci-dessus supposent un modèle de système monoprocesseur idéal assez restrictif. En particulier :

- Toutes les tâches sont périodiques ou sporadiques, et supposées connues avant l'exécution du système.
- Toutes les tâches ont un temps d'exécution exactement connu avant exécution, toujours égal à leur temps pire-cas.
  - Dans le cas préemptif (avec tâches indépendantes), cette limitation peut être supprimée : les tâches peuvent terminer avant leur temps d'exécution pire-cas.
  - Dans le cas non-préemptif non oisif, la terminaison plus tôt d'une tâche peut conduire au cas pathologique (on parle d'*anomalie*) illustré figure 3.2, même si le système est ordonnançable lorsque les temps d'exécution pire cas sont considérés.



(a) Ordonnancement faisable (temps d'exécution pire cas pour toutes les tâches)

(b) Dépassement d'échéance si la tâche τ_3 est plus courte que le temps pire-cas

FIG. 3.2: Anomalie d'ordonnancement par terminaison plus tôt en non préemptif (ordonnancement de type EDF ou DM)

- Aucun mécanisme de synchronisation n'est considéré.
- Les tâches sont supposées sans relation de précédence.
- Les surcoûts inhérents au matériel (*i.e.* temps de changement de contexte par exemple) et/ou liés à l'exécution des services du système d'exploitation (dont ceux de l'ordonnanceur) sont négligés.

– Les activations des tâches périodiques sont supposées être faites à la granularité de l'horloge système.
– Comportement binaire (totalement faisable/non faisable) : aucune mesure de qualité d'aucune sorte n'est formulée lorsque le système n'est pas faisable.
– Les ordonnancements considérés sont non-oisifs.

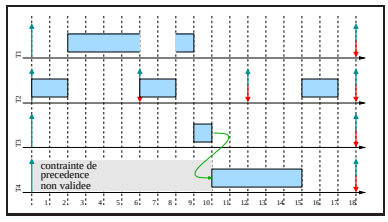
Dans les paragraphes qui suivent, nous présentons quelques extensions qui ont été apportées aux ordonnancements à priorité que nous venons de décrire, et qui visent à supprimer tout ou partie de ces limitations.

3.2.5 Ajout de contraintes

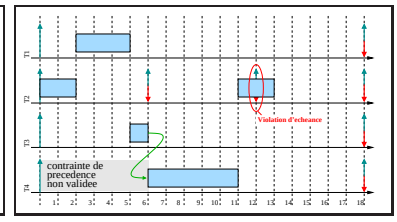
3.2.5.1 Contraintes de précédence

En ce qui concerne l'ordonnancement préemptif à priorité dynamique, [Law73] propose un algorithme qui minimise le retard maximal tout en prenant les contraintes de précédences en compte (*Latest Deadline First*). Le principe est de laisser en attente une tâche prête à être exécutée tant que sa/ses précédences n'a/n'ont pas été validée(s). Pour l'ordonnancement suivant EDF, la prise en compte des contraintes de précédence peut se faire par modification des échéances relatives [HMT90, SS93, Jef92]. L'ordonnancement résultant est alors optimal vis-à-vis des ordonnancements préemptifs non oisifs qui prennent en compte les précédences [HMT90].

En ordonnancement à priorité fixe, une difficulté s'ajoute : suivant l'affectation des priorités, la terminaison en avance d'une tâche peut rendre non ordonnançable un système qui était ordonnançable lorsque les temps d'exécution étaient tous considérés toujours égaux au temps pire-cas. On parle d'*anomalie d'ordonnancement*. La figure 3.3 propose une illustration.



(a) Temps d'exécution égal au WCET



(b) τ_1 termine en avance

	τ_1	τ_2	τ_3	τ_4
C_i	5	2	1	5
$D_i = T_i$	18	6	18	18
π_i	1	2	0	3

FIG. 3.3: Contrainte de précédence $\tau_3 \prec \tau_4$ et dates de terminaison

Des travaux [Tin94, Tin92b, Tin93, ATB93] se sont attachés à étendre les conditions de faisabilité RTA de manière à intégrer des contraintes de précédence (sous la forme d'un délai minimal incompressible entre l'activation d'une tâche et celles de ses successeurs : notion de *transaction*) avec prise en compte des terminaisons en avance sous la forme d'une gigue d'activation en aval des précédences. D'autres études s'intéressent à des chaînes de tâches (*i.e.* 0 ou 1 tâche précède 0 ou 1 tâche) et vérifient le respect des

contraintes temporelles parmi tous les entrelacements des chaînes possibles [SNF98] et en tenant compte des variations des temps d'exécution [RRGC02, SGL97].

3.2.5.2 Accès protégés à des ressources partagées

Comme dans les systèmes informatiques non temps-réel, le principe est d'isoler l'accès à une ou plusieurs ressources sous la forme de *sections critiques*, c'est-à-dire de portions de code à l'intérieur des tâches. En temps-réel, ceci s'accompagne de problèmes dus au *temps de blocage* inévitable qu'une tâche doit subir dès qu'elle demande une ressource possédée par une tâche de priorité inférieure.

En temps-réel viennent se rajouter deux problèmes supplémentaires :

- Une anomalie d'ordonnancement similaire à celle présentée en 3.2.4 : la terminaison plus tôt d'une tâche peut rendre infaisable un ordonnancement qui l'était lorsque les temps pire-cas étaient considérés.
- Les phénomènes d'*inversion de priorité*. Le jeu des conflits liés aux ressources peut entraîner qu'une tâche τ_1 soit mise en attente d'une ressource possédée par une tâche de priorité inférieure τ_3. Mais cette tâche de faible priorité peut être préemptée par une tâche τ_2 de priorité intermédiaire entre celles de τ_1 et de τ_3, et qui n'est pas en conflit ni avec τ_1, ni avec τ_3 pour la ressource. Il en découle que τ_1 a en fait été indirectement préemptée par une tâche τ_2 de priorité inférieure. Si ce phénomène d'*inversion de priorité* n'a pas été pris en compte au moment de la conception, des dépassements d'échéance sont possibles, comme cela a été le cas pour le célèbre dysfonctionnement de la mission Mars Pathfinder [50].

Pour résoudre ces problèmes, de nombreux *protocoles d'accès aux ressources* sont proposés et intégrés aux systèmes d'exploitation temps-réel. En particulier afin de maîtriser les phénomènes d'inversion de priorité, ils visent à borner le temps de blocage associé à sa prise en compte. Parmi les protocoles d'accès aux ressources en priorité statique, citons : le protocole à héritage de priorité (PIP, pour *Priority Inheritance Protocol*) [SRL90, Kai81], le *protocole à seuil de priorité* (PCP, pour *Priority Ceiling Protocol*) [SRL90], le *protocole à priorité de pile* (SRP, pour *Stack Resource Policy*) [Bak91]. Ces travaux peuvent également être intégrés dans un système avec ordonnancement à priorité dynamique [Spu96, SS93, Bak91].

Une fois les temps de blocage déterminés, les conditions de faisabilité vues précédemment dans ce chapitre ont été étendues pour les prendre en compte, tant pour EDF [SS93, Bak91, SS93, Spu96], qu'en ordonnancement à priorité fixe préemptif [SRL90, Bur94, ABR+93, Tin92a]. Ces conditions sont toutes suffisantes seulement (les tâches peuvent ne subir aucun blocage en réalité), et sont plus ou moins pessimistes.

3.2.5.3 Prise en compte des coûts système et matériels

Prendre en compte les interruptions matérielles peut se faire en les modélisant sous la forme de tâches de haute priorité. Prendre en compte les changements de contexte (préemptions) peut se faire en rajoutant le coût du changement de contexte (dont l'estimation non pessimiste est un domaine de recherche actif) soit au temps pire-cas de

chaque tâche, soit au temps pire-cas des tâches qui peuvent être préemptées (ce qui est possible avec les algorithmes à priorité fixe, mais pas avec ceux à priorité dynamique).

En ce qui concerne l'ordonnancement à priorité dynamique (préemptif et non-préemptif), les travaux de [JS93, Jef92] et [SP97, Spu96] intègrent le surcoût des interruptions matérielles indépendantes des tâches du système, dans le calcul du taux de demande du processeur (voir 3.2.3.1).

En ce qui concerne l'ordonnancement à priorité fixe, les travaux de [JS93] et [SNF98] indiquent comment intégrer le surcoût des interruptions matérielles dans TDA et RTA respectivement. Le document [KAS93] détaille comment intégrer différents surcoûts fixes pire-cas (tels que le coût des préemptions, de l'ordonnanceur) dans TDA en *tick scheduling* (aisément transposable à RTA). Les travaux [Tin93, BWH93] précisent comment intégrer les surcoûts liés à l'interruption d'horloge pour du *tick scheduling*, en tenant compte de la durée d'exécution variable de l'ordonnanceur, qui est fonction du nombre de tâches à activer à chaque interruption.

3.2.5.4 Gigue d'activation

En pratique, entre la date où une tâche périodique doit être théoriquement activée, et la date réelle à laquelle la tâche est mise dans la file des tâches prêtes à être exécutées, il peut y avoir un délai variable dû par exemple à la granularité de l'horloge système (en particulier pour un mode de fonctionnement de type *tick scheduling*, dans lequel les tâches ne peuvent démarrer que lors des *ticks* d'horloge), aux surcoûts induits par les décisions d'ordonnancement, ou au temps de génération d'un message nécessaire avant le commencement de la tâche (pour la validation d'une contrainte de précédence par exemple). Les conditions de faisabilité données ci-dessus ne sont alors plus suffisantes.

Des extensions à ces conditions existent pour prendre en compte une telle gigue. Le principe est de ramener la gigue à un temps de blocage (voir 3.2.5.2). Ainsi, pour l'ordonnancement préemptif à priorité fixe, RTA est étendu pour supporter la gigue [ABR+93, Tin92a, Bur94]. Le même type d'extension existe pour les ordonnancements à priorité dynamique [Spu96]. D'autres travaux s'intéressent aussi à contrôler les gigues d'activation et à les borner dans des ordonnancements à priorité fixe ou dynamique avec contraintes de précédence [LD01, DC01].

3.2.5.5 Extensions

Modèles de tâches plus riches

Certains travaux s'intéressent à des modèles de tâches plus complexes avec les ordonnanceurs à priorité statique et dynamique simples. Comme par exemple :

- des tâches "périodiques de façon sporadique" : une tâche est périodique avec une courte période pendant un intervalle donné, puis cet intervalle se répète de façon sporadique. Des travaux existent pour prendre en compte ce modèle de tâche en priorité statique par RTA [Tin93, ABR+93, Tin92a] et dynamique [Spu96].
- des tâches caractérisées par une fréquence d'activation (modèle de *Rate Based Execution* en anglais). Le profil d'activation d'une tâche correspond à la définition

d'une densité d'activation de la tâche (une borne supérieure sur le nombre d'activations de la tâche dans une fenêtre glissante de temps donnée). Dans [JG99], une condition nécessaire et suffisante d'ordonnançabilité est donnée pour un ordonnancement suivant EDF, et il est montré qu'aucun ordonnancement à priorité fixe ne peut ordonnancer de telles tâches.

- des tâches avec "échéance avant la fin de la tâche", c'est à dire que la tâche est en deux morceaux : les traitements utiles qui doivent se terminer avant une échéance, et des opérations de maintenance exécutées immédiatement après (prise en compte des coûts de préemption par exemple, ou de libération des ressources système). Les travaux [Bur93, Tin93] proposent une extension de RTA pour intégrer ce modèle de tâches en ordonnancement à priorité statique.
- des tâches qui peuvent avoir des temps d'exécution qui suivent un motif donné (*i.e.* ce sont des vecteurs plutôt que des scalaires) : le modèle *multiframe*. Les travaux [MC96b, MC96a] et [BCM99] proposent d'étendre respectivement la condition de faisabilité 3.2 et RTA pour supporter ce modèle dans l'ordonnancement RM. Ce modèle multiframe est lui-même généralisé dans [BCG+99] afin de supporter des tâches dont les échéances relatives et les périodes sont elles-mêmes variables suivant un motif, et une condition nécessaire et suffisante de faisabilité par RTA est proposée.
- des tâches qui peuvent être constituées de sous-tâches avec des échéances relatives internes et des précédences : c'est le modèle de tâches "récurrent" de [Bar98]. Une condition suffisante de non faisabilité en préemptif y est donnée.

Extensions du modèle de système : fonctionnement par phases

Pour certains systèmes, les tâches n'ont pas toutes besoin d'être prises en compte à tout moment. C'est le cas lorsque le système fonctionne par phases : un avion par exemple aura besoin de certaines fonctionnalités pendant le décollage, d'autres pendant la croisière, et d'autres encore à l'atterrissage. Des travaux se sont penchés sur l'intégration de tels modèles de système pour l'ordonnancement en plan statique [Foh94, KNH+97] (recours à un plan intermédiaire de transition immédiat ou différé) et à priorité statique [TBW92b] (extension de RTA) par exemple.

Le fonctionnement par phases pose le problème de la cohérence d'état global du système lors des transitions, dont la résolution est une approche propre à chaque application.

3.3 Ordonnancement avec acceptation en-ligne

Un système temps-réel peut être en charge du contrôle d'un procédé par exemple, et en même temps avoir à fournir une palette de services sans contrainte temporelle (affichage d'une interface graphique, exécution des services d'un système d'exploitation généraliste par exemple [Bar97, GAGB01]), ou avec contrainte temporelle mais facultatifs (affichage de statistiques sur le procédé par exemple).

Nous étudions d'abord le cas où les tâches à prendre en compte dynamiquement sont périodiques (section 3.3.1), puis apériodiques avec ou sans contraintes de temps-réel (section 3.3.2). Nous donnons ensuite des améliorations aux mécanismes pour profiter de la terminaison plus tôt des tâches temps-réel strict (section 3.3.3).

3.3.1 Tâches périodiques et sporadiques

Si le comportement et les contraintes temporelles de la tâche à prendre en compte (échéances, temps d'exécution, ressources nécessaires, contraintes de précédence) sont connus, alors il est possible de garantir le respect de toutes ses contraintes temporelles en cours de fonctionnement, au même titre que les tâches garanties hors-ligne. L'établissement de cette garantie procède de la vérification des conditions d'ordonnançabilité données en 3.2.3, ou d'une variante (comme [BS93] pour EDF, ou [McE94, RJMO98] pour l'ordonnancement à priorité statique par exemple), en présence de la nouvelle tâche.

Si l'établissement d'une telle garantie est trop coûteuse en ressources processeur, ou si le comportement temporel de la tâche n'est pas suffisamment connu, on se ramène aux dispositifs de prise en compte de tâches apériodiques présentés ci-dessous, en considérant que la tâche est apériodique.

3.3.2 Tâches apériodiques

On se place toujours dans le cadre des ordonnancements à priorité, et on considère que le système est désormais constitué d'une sous-partie temps-réel strict garantie hors ligne, comprenant uniquement des tâches sporadiques et/ou périodiques ; et d'une sous-partie constituée de tâches apériodiques (*sous-partie dynamique* dans la suite) dont les travaux sont à prendre en compte en ligne individuellement.

Le plus souvent, les deux sous-systèmes sont indépendants, mais quelques travaux [CLB99, LB00a, DTB93] s'intéressent en plus à la prise en compte du partage de ressources entre les tâches temps-réel strict garanties hors-ligne, et celles dynamiquement activées.

3.3.2.1 Principe

Tâches sans contraintes temporelles

Lorsque les tâches apériodiques n'ont pas de contraintes temporelles, l'objectif est de continuer de garantir la faisabilité des tâches temps-réel garanties hors-ligne, tout en obtenant un temps de réponse le plus petit possible pour les tâches apériodiques.

Tâches avec contraintes temporelles

Lorsque les tâches apériodiques qui se présentent dynamiquement au cours de la vie du système ont des contraintes temporelles (traitement exceptionnel avec échéance stricte par exemple), il s'agit en plus de garantir le respect des contraintes temporelles des travaux associés.

Puisqu'il est impossible de garantir l'ordonnançabilité de l'intégralité du système hors-ligne (cela reviendrait à pouvoir prévoir l'avenir, ou à disposer d'une capacité de traitement infinie), en pratique la technique est de soumettre individuellement chaque travail à un *test d'acceptation* avant de l'ordonnancer, suivant une des techniques ci-dessous. Le test d'acceptation, dépendant de l'ordonnancement, a pour rôle de vérifier que si le travail est pris en compte, alors *i)* ses propres contraintes temporelles sont respectées en présence des autres tâches déjà présentes, et *ii)* il est possible d'ordonnancer le travail sans remettre en cause les contraintes temporelles des autres tâches garanties. Si ce *test d'acceptation* échoue, le travail est *refusé*.

Le test d'acceptation repose sur le calcul du temps de réponse du travail en fonction de toutes les tâches déjà présentes dans le système, et consiste en la vérification que ce temps de réponse est compatible avec les contraintes temporelles. Dans la section 8.1 de la partie II, nous développons une infrastructure générique pour une grande palette d'ordonnanceurs à priorité (fixe et dynamique), qui permet l'acceptation en-ligne de tâches apériodiques. Elle repose sur un algorithme de calcul de temps de réponse paramétrique, adaptable à un grand nombre d'affectations de priorités.

Dans la suite, nous indiquons les mécanismes d'ordonnancement sans détailler les tests d'acceptation associés.

3.3.2.2 Ordonnancement en *tâche de fond*

La méthode la plus simple pour prendre en compte les tâches apériodiques est de les ordonnancer lorsqu'aucune tâche périodique/sporadique temps-réel strict n'est prête. Avec cette méthode (*ordonnancement en tâche de fond* ou *background scheduling* en anglais), les conditions de faisabilité et l'algorithme d'ordonnancement ne sont pas modifiés, mais le temps de réponse des tâches apériodiques qui découle est long.

3.3.2.3 Approches par tâches *serveurs*

Afin d'améliorer le temps de réponse des tâches apériodiques dans le contexte d'un ordonnancement à priorité, l'idée est de les faire prendre en charge par une tâche dédiée du système, sans toutefois remettre en cause l'ordonnancement des tâches périodiques et sporadiques temps-réel strict : on parle de tâches *serveurs*.

Même si nous nous intéressons ici aux ordonnancements à priorité, une technique similaire existe pour les ordonnancements à plans [Foh94, IF99].

Priorité statique

En ordonnancement à priorité statique, les premières approches [But97] considèrent une tâche serveur périodique de *capacité* fixée re-remplie à chaque début de période du serveur, et qui décroît en fonction du temps (méthode à *serveur périodique* ou *polling server* en anglais), ou en fonction de l'utilisation par les tâches apériodiques [SLS95] (DS, pour *deferrable server*). Dans le premier cas, l'introduction du serveur ne modifie pas les conditions de faisabilité (le serveur est une tâche périodique classique, ordonnancé même s'il n'a aucune tâche apériodique à exécuter). En contrepartie, les

tâches apériodiques ne sont pas prises en compte quand elles arrivent lorsque le serveur a été complètement ordonnancé dans la période courante. Dans le deuxième cas, les tâches apériodiques y gagnent en réactivité puisqu'elles sont prises en compte à tout moment dans la période courante : on parle de *serveur avec préservation de la bande passante* (*bandwidth preserving* en anglais). Mais l'introduction du serveur perturbe l'ordonnancement des autres tâches temps-réel (le serveur n'est pas ordonnancé pendant la période courante jusqu'à ce qu'il ait une tâche apériodique à exécuter), ce qui modifie les conditions de faisabilité (l'utilisation maximale garantie, analogue à la condition 3.2, est plus faible).

Une variante de DS, plus simple et dont les conditions de faisabilité sont analogues au cas du *polling server*, est la méthode avec *serveur sporadique* [Spr90] où le serveur est une tâche sporadique dans laquelle la seule contrainte est que les re-remplissages de la capacité du serveur sont séparés par un délai d'inter-arrivée du serveur. En contrepartie, le temps de réponse des tâches apériodiques est plus compliqué à déterminer. D'autres variantes plus complexes existent, comme la méthode à *échange de priorité* [But97] (*priority exchange* en anglais), qui visent à accumuler de la capacité d'exécution tant que le serveur n'a pas de tâche apériodique à exécuter. La condition de faisabilité est moins défavorable que DS (l'utilisation maximale garantie est plus élevée), mais les tâches apériodiques ont un temps de réponse plus élevé et les conditions de faisabilité sont plus complexes que DS.

Priorité dynamique

En ordonnancement à priorité dynamique, une première technique est de déclarer une tâche serveur, de capacité et de période (ou délai d'inter-arrivée) données. EDF peut être utilisé pour ordonnancer les serveurs. On peut utiliser les méthodes à échange de priorité ou de serveur sporadique indiquées ci-dessus, pour définir le comportement du serveur [But97]. Dans les deux cas, la condition d'ordonnancement revient à considérer le serveur comme une tâche temps-réel quelconque, la réactivité des tâches apériodiques étant d'autant plus faible que la période du serveur est grande.

Il existe d'autres méthodes qui évitent d'avoir à attendre le début de la période suivante avant de pouvoir exécuter la tâche apériodique, de façon à réduire leur temps de réponse. Le principe est de réserver une portion de la capacité du processeur à une tâche serveur (sa capacité), et de modifier en-ligne l'échéance relative du serveur suivant la présence ou l'absence de tâches apériodiques à exécuter, tout en garantissant que les tâches temps-réel garanties hors ligne ne sont pas perturbées. C'est le fonctionnement des *serveurs à bande passante donnée* : TBS [SB94, CLB99, LB00a, SBS95] pour *Total Bandwidth Server* en anglais (pour des requêtes apériodique dont on connaît le temps d'exécution pire-cas), ou CBS [AB98a, ALB99, JG01, LB01] pour *Constant Bandwidth Server* en anglais (pour des requêtes apériodiques quelconques) par exemple. La condition de faisabilité revient toujours à considérer le serveur comme une tâche temps-réel quelconque du système.

3.3.2.4 Approches par réquisition de temps creux

Les meilleures méthodes vis à vis de la minimisation du temps de réponse des tâches apériodiques sont celles qui permettent à ces tâches d'être exécutées en priorité par rapport aux tâches garanties hors-ligne. Leur principe est intermédiaire entre l'ordonnancement en tâche de fond, et les approches par serveurs : il s'agit de retarder les tâches garanties hors-ligne au profit des tâches apériodiques, en veillant cependant à continuer de respecter toutes les contraintes temporelles des tâches garanties hors-ligne : on parle de *réquisition de temps-creux* (*slack steeling*).

Priorité statique

Il existe une méthode très consommatrice en mémoire, qui minimise le temps de réponse de la première tâche apériodique en tête de la file des apériodiques sur le serveur [TLS95] : l'*ordonnancement par réquisition de la laxité* [LT94] (*slack stealing* en anglais). Il s'agit de maintenir en mémoire un calendrier (la *fonction de laxité*), établi hors-ligne, et qui couvre une hyperpériode du système. À chaque instant, le calendrier indique la capacité d'exécution que le serveur des tâches apériodiques est autorisé à prendre au détriment des tâches temps-réel garanties hors ligne, tout en garantissant que les échéances de ces dernières restent respectées. Les travaux [DTB93, Dav93] présentent une version exacte mais complexe de cette technique, qui calcule la fonction de laxité en-ligne (pas de calendrier), ainsi que des versions approchées moins complexes.

Enfin, il existe une méthode aux performances (réactivité des tâches apériodiques) comparables, dite *à double priorité* (ou *dual priority*) [Dav94, GNM99], mais moins coûteuse en mémoire et en temps de calcul tant hors-ligne que en-ligne. Elle consiste à retarder au maximum les tâches temps-réel garanties hors ligne d'un retard calculé hors-ligne par analyse de temps de réponse (le *retard de promotion* ou *promotion time* en anglais) s'il y a des tâches apériodiques présentes. Pour cela, pendant la durée de ce retard de promotion, les tâches temps-réel garanties hors ligne ont une priorité plus faible que les tâches apériodiques qui peuvent être présentes, ce qui assure une bonne réactivité des tâches apériodiques. Afin de continuer de garantir les tâches temps-réel garanties hors ligne, une fois que le retard de promotion d'une tâche temps-réel garantie hors ligne est écoulé, celle-ci retrouve sa priorité nominale, supérieure à la priorité de toutes les tâches apériodiques éventuellement présentes. Lorsque le retard de promotion est nul, cette méthode est équivalente à l'ordonnancement des tâches apériodique en tâche de fond. Ces travaux fournissent un algorithme pour calculer les retards de promotion maximum possibles pour des ensembles de tâches sporadiques (non optimaux dans le cas de tâches toutes périodiques [Nic98]).

Priorité dynamique

Il existe une technique similaire à l'ordonnancement par réquisition de la laxité pour EDF [TLSH94].

3.3.3 Récupération de ressources pour favoriser l'ordonnancement de tâches activées dynamiquement

Lorsque l'ordonnancement de tâches temps-réel activées dynamiquement dépend d'un test d'acceptation, il est important de pouvoir compter sur le maximum de ressources disponibles, en particulier en ce qui concerne les ressources processeur. Ceci suppose d'être capable d'évaluer précisément les ressources disponibles et celles nécessaires : si on se limite à ne prendre en compte que les pires comportements des tâches en cours (temps d'exécution pire cas, fréquence d'activation maximale pour les tâches sporadiques), alors des tâches temps-réel dynamiquement activées risquent d'être refusées à tort. La *récupération de ressources* est un mécanisme complémentaire à l'acceptation dynamique en-ligne vu précédemment, qui permet de profiter des ressources réservées pour d'autres tâches, mais qui sont finalement non utilisées (par exemple : terminaison plus tôt que le temps d'exécution pire-cas, abandon de la tâche de sauvegarde quand la tâche primaire d'un couple primaire/sauvegarde termine correctement), afin de limiter le pessimisme introduit par la réservation de ressource processeur qui avait été faite sur la base d'un comportement pire-cas, pour pouvoir accepter davantage de tâches en-ligne.

Il existe des méthodes qui réactualisent les ressources disponibles à la fin de chaque tâche, pour récupérer celles réservées mais non utilisées [SBS95, LB00a, Dav93]. Il existe d'autres méthodes plus précises, qui mesurent l'accélération par rapport au temps d'exécution pire cas, au cours de l'exécution de la tâche [HS90, MZ93, DTB93] : il s'agit de découper la tâche en sections (*milestones* en anglais), dont on connaît le temps pire cas, et encadrées par l'émission de signaux de repérage envoyés au système d'exploitation par exemple.

Ces techniques sont en général accompagnées d'un mécanisme de *repêchage* des tâches refusées (ou *second chance* en anglais) : lorsque la récupération des ressources libère suffisamment de ressources, il peut être possible d'ordonnancer une tâche dynamiquement activée qui avait été précédemment refusée et mise dans la file de repêchage (*reject queue* en anglais).

Le test d'acceptation paramétrique décrit en 8.1 de la partie II propose également des mécanismes de récupération de ressources en fin de tâche, avec repêchage.

3.4 Ordonnancement avec politique de rejet

3.4.1 Gestion de la surcharge et notion de rejet

Lorsque les ressources nécessaires pour exécuter les tâches présentes dans le système sont plus importantes que les ressources disponibles (en particulier la ressource processeur), le système entre en phase de *surcharge*. À la différence de l'acceptation en-ligne de travaux apériodiques vue précédemment dont le rôle est de *prévenir* la surcharge, la *gestion de surcharge* intervient *a posteriori*, une fois la surcharge établie.

Le phénomène de surcharge peut être lié à un fonctionnement anormal du système, résultat d'un comportement non prévu de l'environnement (activation de tâches spo-

radiques sans respect du délai minimal d'inter-arrivée par exemple), ou du système (temps d'exécution réels supérieurs aux temps pire-cas spécifiés, surcoûts du système d'exploitation ou du matériel sous-estimés). Le phénomène peut aussi être le résultat d'une décision de réalisation, comme la volonté de considérer des temps d'exécution moyens plutôt que des temps pire-cas dans l'ordonnancement (afin de limiter le surdimensionnement du système, quand par exemple le comportement au pire-cas n'est pas *raisonnable*, comme pour un mécanisme d'allocation mémoire par exemple).

Si aucune précaution n'est prise, la conséquence d'une surcharge est que certaines tâches ne respectent pas leur échéance, et le système peut s'écrouler : les tâches qui manquent leur échéance retardent les autres qui à leur tour manquent leur échéance, et ainsi de suite. C'est l'effet *domino* [SSDB94] particulièrement sensible avec un ordonnancement de type EDF, puisque celui-ci fonde ses décisions d'ordonnancement sur la proximité de l'échéance de la tâche, donnant ainsi plus grande priorité à la tâche qui est en train de manquer son échéance.

En présence de surcharge, la solution pour contrôler le comportement du système est de borner ou de diminuer les besoins en ressources. Cela suppose :

- Qu'on est capable de détecter en-ligne le commencement de la surcharge. Ce problème est NP-complet [BHR93] dans le cas général. Dans certains cas cependant, il est possible de profiter de bonnes propriétés des algorithmes d'ordonnancement qui permettent de détecter le début de la saturation du processeur [KS93]. Sinon, on fait appel à des approximations qui consistent à mesurer le taux d'occupation du processeur, ou *charge de travail*, afin de *prévoir* la surcharge (peut-être à tort) [MS95, BBL01]. La formalisation puis la mesure de la charge sont elles-mêmes des problèmes en tant que tels, puisqu'il s'agit de tenir compte à la fois des ressources processeur demandées *globalement*, et des contraintes temporelles qui sont *locales* à chaque tâche, et qui participent de la surcharge (pas de surcharge en l'absence de contrainte de temps).
- Qu'on accepte le fait que le système puisse terminer d'autorité une tâche si nécessaire.

En aval, une fois la surcharge détectée ou anticipée, il y a plusieurs méthodes pour diminuer le besoin en ressource processeur : soit certaines tâches sont volontairement abandonnées, *i.e.* terminées d'autorité (décisions microscopiques) ; soit la structure globale du système est modifiée, *i.e.* le système est *reconfiguré* (décisions macroscopiques). Nous nous intéressons maintenant à ces deux voies d'étude (sections 3.4.2 et 3.4.3 respectivement) dans le cas d'ordonnanceurs pour le temps-réel strict. En section 3.4.4, nous indiquons des modèles de systèmes qui tolèrent des dépassements d'hypothèses dans certaines limites.

3.4.2 Ordonnancement sans reconfiguration

En général, on découpe le système en deux sous-parties : la sous-partie critique définie et garantie hors-ligne, et la sous-partie non critique. La sous-partie critique n'est jamais remise en cause, seule la sous-partie non critique présente les problèmes de surcharge. Nous nous concentrons ici sur cette sous-partie non critique.

On distingue habituellement plusieurs classes d'ordonnanceurs de cette catégorie : les ordonnanceurs *au mieux* (*best effort* en anglais) sans test d'acceptation dans lesquels les tâches sont ordonnancées jusqu'à leur terminaison ou jusqu'à dépassement d'échéance ; les ordonnanceurs avec test d'acceptation et sans remise en cause des tâches acceptées (dits aussi *à garantie*) ; les ordonnanceurs avec exécution conditionnelle (test à l'activation ou avant démarrage) et *politique de rejet*, *i.e.* abandon (dits aussi *robustes*).

3.4.2.1 Notion de valeur

Que ce soit pour évaluer la qualité d'un ordonnanceur en présence de surcharge, pour donner des garanties numériques sur le comportement du système (tous ordonnanceurs), ou pour arbitrer entre les tâches à accepter (ordonnanceurs à garantie) ou abandonner (ordonnanceurs robustes), il est nécessaire d'introduire les notions d'*importance* des tâches représentées numériquement par des *fonctions de valeur*, et de *valeur globale* dégagée par le système, qui définit la *qualité du service* effectivement fournie par le système.

Les fonctions de valeur associées aux tâches peuvent être par exemple constantes, ou fonction du temps d'exécution, ou relatives à la date de démarrage, ou fonction de la décision qui doit être prise (acceptation, refus ou abandon ; comme dans [MB97, Del96]). La valeur globale dégagée par le système peut à son tour être la somme des valeurs dégagées par chacune des tâches, ou une fonction plus complexe [BPB+00].

Nous ne considérons pas ici le problème décisionnel lié la définition des fonctions de valeur, qui peut lui-même être un problème complexe [BPB+00].

3.4.2.2 Limites théoriques

Les travaux de Sanjoy Baruah montrent que pour des fonctions de valeur simples, il existe une limite sur la valeur globale maximale garantie qu'on peut atteindre avec un algorithme en-ligne, par rapport à ce qu'il est possible d'obtenir avec un algorithme d'ordonnancement idéal qui disposerait de toutes les informations sur les activations de tâches à venir, dit *ordonnanceur clairvoyant* (comme par exemple [AB98b]). Le quotient entre les deux valeurs est le *facteur de compétitivité*.

Dans le cas simple où la fonction de valeur pour une tâche est : 1 quand la tâche termine dans les temps, et 0 si elle dépasse son échéance où si elle est refusée ou abandonnée, les travaux [BHS01] montrent que dans le cas général, par rapport à un algorithme clairvoyant, les algorithmes d'ordonnancement en-ligne peuvent dégager une valeur (appelée dans ce cas le *taux de réussite* ou *completion count* ou *hit ratio*) arbitrairement plus petite (*i.e.* le facteur de compétitivité vaut 0).

3.4.2.3 Ordonnanceurs à garantie et ordonnanceurs robustes

En général, les *ordonnanceurs à garantie* reposent directement sur le test d'acceptation (tel que ceux évoqués en 3.3) : si celui-ci réussit, la tâche est définitivement intégrée dans le système. C'est le fonctionnement de GED (pour *Guaranteed Earliest Deadline*)

[BS93], une variante de EDF avec test d'acceptation. On pourrait cependant imaginer des ordonnanceurs qui refusent une tâche alors que le test d'acceptation réussit, par anticipation de l'activation prochaine d'une tâche (sporadique par exemple) plus importante.

Les *ordonnanceurs robustes* sont des ordonnanceurs à priorité classiques, avec test d'acceptation, et dans lesquels une *politique de rejet* est activée en cas de refus d'une tâche : quand le test d'acceptation échoue, l'ordonnanceur tente d'abandonner une ou plusieurs tâches de moindre importance. C'est le fonctionnement de RED (pour *Robust Earliest Deadline*) [BS93], une variante robuste de GED. Le test d'acceptation paramétrique avec récupération de ressources décrit en 8.1 de la partie II généralise RED à un ensemble plus large d'ordonnanceurs à priorité (statique et dynamique).

D'autres ordonnanceurs robustes ne reposent pas sur un test d'acceptation, mais sur un test de démarrage, comme D^{over} [KS93] qui est issu de EDF. D^{over} repose sur une propriété fondamentale de EDF qui lui permet d'identifier la saturation transitoire du processeur (*i.e.* un risque de surcharge) au démarrage d'une tâche, auquel cas le démarrage est conditionné par un test qui optimise la valeur dégagée à hauteur de la limite théorique donnée dans [BKM+91].

3.4.2.4 Ordonnanceurs *au mieux* et/ou à valeur *moyenne* garantie

Dans les paragraphes précédents, il était question de garanties strictes sur la valeur qu'il est possible d'obtenir de façon sûre, sachant qu'on dispose des informations pour identifier les configurations pire-cas. On s'intéresse ici aux systèmes où on ne dispose que d'informations stochastiques qui ne permettent pas d'identifier de telles configurations, et on cherche à évaluer et/ou garantir la valeur *moyenne* qu'on obtiendra à l'exécution.

Avec ce type d'ordonnanceurs, dits *au mieux*, une tâche est ordonnancée jusqu'à son terme, ou jusqu'à dépassement de son échéance, de manière à borner le besoin en ressources processeur. Cette stratégie peut être associée à un ordonnancement à priorité classique (RM, EDF), dont les comportements stochastiques peuvent être évalués par exemple par simulation, comme dans les travaux de [AB99, BSS95].

D'autres travaux, comme ceux de [Gar99], étudient formellement le comportement stochastique de RM, DM et EDF. Les travaux de [AB98d, AB98c] proposent une extension stochastique de RMA : le concepteur fournit à la conception les caractéristiques temporelles stochastiques des tâches (distribution des temps d'exécution et des dates d'activation), ainsi que les contraintes probabilistes de respect des échéances (taux de respect moyens par tâche), et ces travaux permettent de vérifier que ces contraintes moyennes seront respectées ou non.

Ce type de travail permet de garantir qu'un système de tâches donné dégagera une valeur moyenne donnée. Ceci permet d'avoir une idée globale sur la qualité du service *moyen* fourni, mais sans disposer de garantie stricte sur la qualité *minimale* du service fourni.

Ces travaux forment en soi une branche très vaste des travaux d'ordonnancement, à savoir le temps-réel souple et le multimedia. Nous ne les détaillons pas ici, et nous nous intéressons en priorité aux méthodes d'analyse et d'ordonnancement de systèmes

qui ont une composante temps-réel strict.

3.4.3 Ordonnancement avec reconfiguration

Les ordonnanceurs de ce type sont capables de passer d'une configuration du système à une autre (*changement de mode*). Une configuration peut être un ensemble de tâches défini hors-ligne, ou défini en-ligne : le système est constitué d'une série de *services* ; un service étant une fonction particulière du système ou un groupement de tâches établi par un algorithme spécialisé [IMR96]. Chaque service peut être rendu avec une plus ou moins grande qualité, ou différentes contraintes en ressources, en précédences, en temps, par des ensembles de tâches différents ayant chacun leur coût d'exécution.

Ce type d'ordonnanceur [BSS94, GSSR97, Del96, DK98, DM00] repose sur deux mécanismes :

- un mécanisme de suivi de l'état courant du système, avec détermination de la configuration la mieux adaptée à la charge du processeur courante, ou à ses évolutions. Ce mécanisme se doit d'être conçu de telle sorte que des oscillations entre deux configurations voisines soient évitées, phénomène qui est dû en particulier aux mécanismes de changement de configuration eux-mêmes, qui ont un coût d'exécution non nul qui influence donc les décisions de changement de configuration.
- un mécanisme de changement de configuration. Ce mécanisme doit maintenir le système dans un état cohérent à la fois en ce qui concerne les ressources partagées par les services entre deux configurations, et en ce qui touche à l'application elle-même, ce qui peut être résolu par exemple en pré-définissant des points de changement de configuration [SdSA95, KNH+97].

Les deux problèmes ci-dessus sont des problèmes très complexes, d'autant plus complexes si le système est multiprocesseur. Leur résolution est en générale limitée à un système donné.

3.4.4 Extensions au modèle de tâche

Il existe des modèles qui intègrent directement le fait que des tâches puissent être totalement ou en partie abandonnées, et ceci avec garantie en-ligne, ou hors-ligne dès la phase de validation par analyse de faisabilité.

Ainsi, il existe des modèles pour lesquels la spécification doit préciser quels travaux d'une tâche peuvent être supprimés (modèle à perte sporadique ou *skip-over* [KS95, CB97], modèle avec pertes contraintes par fenêtre ou *window-constrained*, modèle avec fréquence de pertes contrainte ou *weakly-hard* [Nic98, BBL01] par exemple).

Il existe des modèles où seule une partie de la tâche est garantie hors-ligne, l'exécution du reste dépendant de la charge du système (modèle de de *calcul imprécis* ou *imprecise computation* [LLS+91, SL95, BH98], modèle à transformation de tâche ou *transform-task* [TDS+95, GL99, Gar99]).

Dans cette dernière classe de modèles, citons aussi le modèle à exception (*taskpair scheduling* ou TPS) [Str95, Net97], qui est la démarche inverse du modèle à calcul

imprécis : une tâche (dite *tâche principale*) est associée à une tâche fantôme (dite *exception*). L'exception n'est exécutée que si la tâche principale s'approche *trop* de son échéance : la tâche principale est abandonnée au profit de l'exception quand la date de démarrage au plus tard de l'exception est atteinte. Ainsi, l'exception est toujours garantie (par test d'acceptation ou hors-ligne), par contre la tâche principale ne l'est pas. Ce modèle permet d'isoler les fautes temporelles et de garantir un fonctionnement minimal. Ce modèle est également compatible avec une connaissance non sûre des temps d'exécution de la tâche principale (par exemple statistique [GS96, NGM01, NG97b], ou par instrumentation et extrapolation [SG97a, NGS97, NG97a, NGM98, SG97b]).

3.5 Extensions du modèle de système

3.5.1 Systèmes ouverts

Nous avons présenté les méthodes d'ordonnancement classiques en temps-réel. Dans la plupart de ces travaux, la phase d'analyse d'ordonnancement recourt le plus souvent à des démonstrations fondées sur l'étude du système dans son ensemble. Des travaux récents s'attachent à alléger cette contrainte, c'est-à-dire qu'ils s'intéressent aux *systèmes ouverts*. Dans ces systèmes, l'objectif est qu'une série d'applications temps-réel, chacune chargée de gérer ses propres tâches suivant sa propre politique d'ordonnancement, puissent coopérer. Les travaux dans ce domaine ont en commun d'aller dans le sens d'une virtualisation des ressources partagées, et une coopération sous forme *boîte noire* des traitements temps-réel effectués par le système.

Citons les travaux dans le domaine des systèmes à ordonnanceurs multiples [Mig99, Nav99, RS01] ou hiérarchiques [Sta93, WL99a, RH01, CJ98, RSH00, LMA88, MF02, LB00b, GAGB01] dans lesquels se pose le problème de prouver la correction temporelle de la composition des ordonnanceurs qui cohabitent [RS01]. Citons aussi les modèles avec ordonnanceurs fluides [SAWJ+96, JG01, ALB99], et l'ordonnancement de ressources virtuelles [MF02], dont l'objectif est de dégager les paramètres pour caractériser les besoins en ressources des différents sous-systèmes afin de prouver la correction de leur composition.

3.5.2 Relâchement d'hypothèses sur l'environnement et le système

Jusqu'à présent, le système était conçu à des fins de prévention des fautes temporelles : il était admis que toute tâche en cours d'exécution dans le système avait un comportement temporel et des besoins en ressources connus dès la phase de conception, ou au moment de l'acceptation. Il existe deux types d'approches dont le rôle est de s'adapter à l'absence, avant exécution, d'informations nécessaires aux décisions d'ordonnancement, ce qui facilite leur conception, au détriment des garanties offertes. Ces mécanismes trouvent largement leur place en temps-réel souple, et nous ne les détaillons pas ici, bien que certains parmi eux sont intégrés dans les ordonnanceurs fournis avec la plate-forme décrite en partie II :

Approches par prévision inspirée des systèmes en boucle fermée pour fonder les décisions d'ordonnancement à venir sur le passé du système, en générant les informations manquantes (*feedback scheduling*). Citons par exemple les extensions de EDF reposant sur un contrôleur PID (pour *proportional integral derivative* en anglais) FC-EDF et FC-EDF^2 [SLS98, LSA+00], et TPS (présenté en 3.4.4) [GS96, SG97a, NG97a, NGM01].

Systèmes réflexifs : approche symétrique à la précédente. Le système dispose de toutes les informations élémentaires qui lui permettent de reconstruire, moyennant un effort modeste, l'information manquante nécessaire à l'ordonnancement. Une autre caractéristique des systèmes réflexifs est qu'ils sont capables de modifier leur propre structure s'ils jugent que l'état courant l'exige, les rendant *de facto* compatibles avec l'ordonnancement en présence de surcharge avec reconfiguration. Citons par exemple le système Spring [SRN+98], dont l'ordonnancement est de type plan dynamique, qui utilise cette approche afin de générer ses plans.

Chapitre 4

Simulation pour l'évaluation de systèmes temps-réel

Ainsi qu'il a été vu en section 2.2.1, les méthodes d'analyse d'ordonnancement présentées dans le chapitre 3 précédent permettent de vérifier hors ligne et de façon sûre la correction temporelle du comportement du système modélisé.

Mais ces méthodes supposent la connaissance de toutes les configurations possibles du système, ce qui peut s'avérer impossible en pratique. En effet, lorsque l'interaction entre l'environnement et le système est insuffisamment caractérisée sur le plan temporel, ou lorsque le système lui-même est insuffisamment caractérisé, ces méthodes ne peuvent conclure à aucun résultat sûr. De même, ces méthodes ne sont pas adaptées dans le cas où les caractéristiques temporelles sont connues, mais ne peuvent pas être prises en compte car elles ne sont pas compatibles avec les contraintes temporelles ; c'est le cas lorsque le système comprend des mécanismes systèmes d'assez haut niveau qui ont des besoins pire-cas en temps processeur considérables, tels que la gestion dynamique de mémoire [Pua02], de systèmes de fichiers, ou de bases de données par exemple. De plus, avec ces méthodes par analyse, l'obtention de résultats quantitatifs autres que le verdict binaire des conditions d'ordonnançabilité est en général complexe.

Dans ce chapitre, nous nous intéressons aux méthodes de simulation. Contrairement aux méthodes analytiques, elles n'offrent pas une garantie complète sur la sûreté des évaluations, mais elles sont compatibles avec l'évaluation quantitative de systèmes complexes, avec des mécanismes systèmes de plus haut niveau, et en présence d'un environnement dynamiquement évolutif.

Nous présentons ci-dessous les grandes techniques de simulation existantes sans prétention d'exhaustivité, et en ne nous limitant pas au domaine du temps-réel : la simulation à temps continu, puis la simulation à événements discrets et ses deux sous-familles (orientée événements, orientée processus). Nous nous intéressons ensuite plus spécifiquement à la simulation appliquée à l'évaluation de systèmes temps-réel.

4.1 Classes de simulateurs

4.1.1 Simulation à temps continu

Historiquement, la simulation a été une des premières utilisations de l'ordinateur, qui remonte à la seconde guerre mondiale [Nic]. Les phénomènes étudiés étaient principalement du domaine de la physique classique ; on parle par conséquent de *simulation à temps continu*. L'ordinateur avait pour rôle de calculer numériquement des opérations mathématiques en grand nombre et plus rapidement que l'homme.

Dans ce type de simulation, le système à simuler est modélisé sous la forme d'un système d'équations, en général des équations différentielles, qui font intervenir un paramètre temps en tant que variable de $\mathbb{R}$. Le simulateur est en charge de résoudre numériquement le système, mais par approximations, d'une part à cause des limitations liées à la représentation des réels en informatique (virgule fixe ou flottante le plus souvent), et d'autre part à cause du fait que la résolution de tels systèmes fait souvent intervenir une discrétisation du temps (cas de la méthode des éléments finis). Plus la discrétisation est fine, plus la simulation est précise, mais plus elle est coûteuse en termes de temps de calcul.

Ces simulateurs (par exemple avec ACSL [1] ou son ancêtre le langage CSSL, ou les bibliothèques matlab/simulink ou scilab/scicos) sont adaptés à la simulation de phénomènes physiques macroscopiques, microscopiques, en électronique analogique, en physique classique, voire quantique. Ces outils proposent bien souvent une interface graphique où les éléments à simuler sont interconnectés à la manière d'un circuit électronique, et encapsulés sous forme de "boîtes" (composition *hiérarchique*), afin de rendre la modélisation du système dynamique à simuler plus intuitive, et de faciliter la réutilisation de composants. Ce principe, illustré dans la figure 4.1 suivante (une capture d'écran de Simulink), est par ailleurs repris dans un bon nombre des simulateurs à événements discrets que nous voyons dans la suite.

4.1.2 Simulateurs à événements discrets

Pour ce type de simulation, on s'intéresse aux systèmes dont l'*état* change "par paliers" irréguliers dans le temps simulé. Par exemple, pour la simulation d'un réseau téléphonique, si on s'intéresse à son dimensionnement, on s'intéresse seulement au taux d'occupation des liaisons, lui-même déterminé par les demandes et les fins de communications. Ainsi, on peut se limiter à simuler l'état du système seulement aux instants précis où les *événements* qui agissent sur l'état du système se produisent. On parle de *simulation à événements discrets*, ou *DES* pour *Discrete Event Simulation*.

Pour l'évaluation de systèmes temps-réel par simulation, on s'intéresse spécifiquement à la simulation à événements discrets, qui est bien adaptée au formalisme choisi pour la modélisation du système. C'est pourquoi nous la détaillons ci-dessous, sans nous restreindre pour le moment à la simulation pour l'évaluation de systèmes temps-réel.

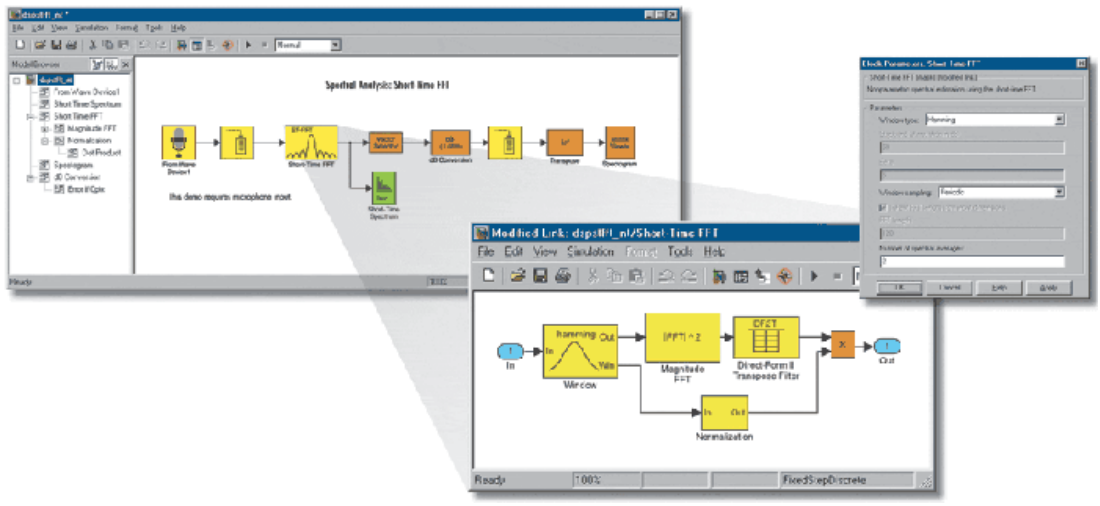


FIG. 4.1: Modélisation graphique avec les outils de simulation

4.1.2.1 Principe

La simulation à événements discrets repose sur la gestion d'un *calendrier d'événements*, c'est-à-dire la liste des événements à venir classés dans l'ordre chronologique. Les *événements* sont des données simples rassemblant une date d'apparition dans l'échelle de temps simulé, un type d'événement, et des informations associées. Ils sont produits au cours de l'évolution du système simulé, ou par l'environnement simulé. Ces définitions font apparaître deux notions fondamentales : la notion d'*échelle de temps simulé* qui définit la relation d'ordre dans le calendrier, et la notion *d'entité* sur laquelle agissent les événements.

Le simulateur fonctionne très simplement en extrayant à chaque *pas de simulation* l'événement à venir le plus proche, et en s'occupant de classer les événements produits suivant l'ordre chronologique de leur date d'apparition.

Illustration

Par exemple, si on désire simuler un réseau téléphonique, les événements considérés sont du type `décrochage du combiné` (produits par exemple par un générateur aléatoire), `début de conversation` (produit par le système simulé, résultant de la gestion des ressources simulées), `fin de conversation` (programmé lors du traitement de l'événement `début de conversation` à l'aide d'un générateur aléatoire).

4.1.2.2 Implantation du système simulé

On peut distinguer les outils de simulation à événements discrets suivant qu'ils sont *orientés événements* ou *orientés processus*.

Le cas des simulateurs orientés événements correspond le plus directement à la

définition canonique des simulateurs à événements discrets. Il s'agit de se concentrer sur la génération d'événements, et chaque événement agit sur l'état *global* du système. Cette approche est par exemple bien adaptée à la simulation d'automates à états finis.

Les simulateurs orientés processus fonctionnent à un niveau d'abstraction plus élevé : ils se focalisent sur la simulation du comportement des différentes entités du système, et de leurs interactions. C'est-à-dire que le parallélisme d'exécution entre les acteurs apparaît explicitement. En ce sens, l'état du système est fractionné en plusieurs états (les *processus*) et les événements dans le système matérialisent les interactions entre processus, mais sont moins explicites qu'en simulation orientée événements, et agissent sur ces états partiels.

Un même modèle peut être transposé dans l'une ou l'autre des approches, mais, suivant le formalisme choisi pour décrire le modèle (automates d'états finis, réseaux de files d'attente, réseaux de Pétri par exemple), l'une ou l'autre approche sera plus naturelle ou plus adaptée.

Illustration

Pour reprendre l'exemple téléphonique précédent, l'approche orientée processus permettrait de décrire un client de la manière suivante (`Poisson()` et `Erlang()` sont des générateurs de séquences aléatoires) :

```
void processus_client ()
{
  while (1)
    {
      wait(Poisson(6,3)); /* le client fait autre chose */
      ligne->acquire();   /* le client decroche et attend
                             de pouvoir parler */
      wait(Erlang(6,3));  /* le client est en communication */
      ligne->release();   /* le client raccroche */
    }
}
```

`wait(t)` est la primitive qui met le processus en sommeil afin de traiter les autres processus, et programme un événement de réveil du processus à la date `date_courante + t`. `acquire()` et `release()` sont des *méthodes* d'une entité (*objet*) de type ligne téléphonique, s'occupant de la gestion de cette ressource. Ici, le principe est celui des sémaphores valués, pour autoriser l'accès simultané à la ligne à au plus `nb_demandes_max` clients (la *capacité* de la ligne).

L'approche orientée événements se traduirait par exemple par :

```
void evenement_client_decroche ()
{
  ligne->nb_demandes++;

 /* le client peut-il utiliser
    la ligne immediatement ? */
  if (ligne->nb_demandes <= nb_demandes_max)
    {
      schedule(client_raccroche, Erlang(6,3));
      /* Le client est en communication */
    }

  /* Sinon le client attend */
}
```

```
void evenement_client_raccroche ()
{
  ligne->nb_demandes--;

  /* Y'a-t'il des clients en attente ? */
  if (ligne->nb_demandes >= nb_demandes_max)
    {
      /* Oui : on en prend un en
         charge sur la ligne */
      schedule(client_raccroche, Erlang(6,3));
      /* Le client est en communication */
    }

  /* prepare la prochaine demande */
  schedule(client_decroche, Poisson(6,3));
}
```

`schedule(fct, t)` est la primitive qui programme l'exécution de la fonction `fct` à la date `date_courante + t`. Le paramètre `fct` joue ici à la fois le rôle du type de l'événement, et des informations associées.

4.1.2.3 Génération des événements

Quand on veut extraire des mesures quantitatives sur le comportement du système, une grande difficulté est (i) de définir la loi statistique des dates des événements produits, mais également (ii) de définir la production par informatique des séquences aléatoires.

La production des séquences pose à son tour 3 problèmes. Il s'agit de disposer de générateurs aléatoires, dits "pseudo-aléatoires" quand on ne dispose pas de matériel spécialisé (générateur de bruit blanc par exemple), qui soient à la fois (i) fidèles à la distribution souhaitée[1], (ii) tels que les échantillons soient statistiquement indépendants, et (iii) tels que si plusieurs séquences sont générées, elles ne soient pas *trop* corrélées entre elles. Cette problématique constitue elle-même un domaine de recherche actuel, que ce soit au niveau des générateurs pseudo-aléatoires, ou de la recherche de générateurs aléatoires physiques à haut débit dans les architectures informatiques complexes [SS02].

4.1.2.4 Exécution de la simulation et analyse des observations

Une fois le système implanté, deux autres points essentiels sont à considérer : l'analyse des observations, et la méthodologie de simulation.

En ce qui concerne l'analyse des observations, la difficulté est d'avoir la possibilité d'insérer les points de mesure le plus librement possible, et de pouvoir composer facilement les résultats obtenus. Si on reprend l'exemple téléphonique, on peut s'intéresser par exemple au nombre moyen de lignes occupées, ou à la corrélation entre les temps d'occupation et le délai entre décrochage du combiné et début de conversation.

[1] Il suffit de disposer d'une distribution uniforme, et d'appliquer la transformation inverse de la fonction de répartition souhaitée.

En ce qui concerne la méthodologie de simulation, elle a un rôle prédominant sur la qualité des résultats fournis (en particulier l'intervalle de confiance). Elle consiste à définir le nombre de simulations nécessaires, ainsi que la durée moyenne (en termes de nombre d'échantillons obtenus) de chaque simulation, qui doit tenir compte de l'établissement du régime permanent de la simulation (*warm-up period* en anglais).

4.1.2.5 Exemples de solutions génériques

Les logiciels de simulation à événements discrets disponibles se comptent par centaines [28]. Nous n'avons pas la prétention de les citer tous ici, mais préférons donner leurs grandes caractéristiques communes.

Il existe une grande quantité de langages pour la simulation de systèmes à événements discrets (Simula, Simscript [35], Modline ex-Qnap2 [17], GPSS/H [10], Parsec [23] par exemple). Le plus célèbre exemple est celui du langage Simula 67, qui a introduit à la fois la notion de coroutine (ancêtre des *threads*), et les bases de la programmation orientée objet, comme le faisait naturellement ressortir l'exemple de simulation orientée processus en illustration de 4.1.2.2. Certains langages, comme Simula, sont *généralistes*, c'est-à-dire qu'ils supportent une grande variété de domaines de simulation. D'autres langages se spécialisent dans un domaine donné, avec un formalisme donné (QNap2 pour les réseaux de files d'attente[2], Esterel, Statecharts, Signal et Lustre pour les systèmes temps-réel réactifs synchrones par exemple). À titre d'illustration, la figure 4.2 présente 2 extraits de codes pour la simulation d'un même modèle (une file d'attente à un guichet) : une simulation avec Simula 67 (orientée processus) et une simulation avec QNap2 (formalisme de réseaux de files d'attente).

Il existe également une grande quantité de bibliothèques de fonctions pour la simulation à événements discrets avec des langages généralistes, tels que le `C` (Mesquite CSim [15], NEST [BSY88], Simpack [33]), le `C++` (MetaSim [16], ADEVS [2], Sim++ [30], CNCL [7], AKSL [3], C++-Sim [4], PSim [25]), ou Java (Javasim [12]). La plupart se classent dans la catégorie *simulation orientée événements*, et offrent les mécanismes de génération de séquences aléatoires, de gestion du calendrier d'événements (en accordant une grande importance à la méthode de tri, allant du classement naïf à l'arbre binaire équilibré), et d'exploitation statistique des mesures effectuées. Certaines bibliothèques s'orientent vers la simulation distribuée pour davantage de performances.

Enfin, il existe une grande variété d'outils tout intégrés pour la simulation (ERS [9], Ptolemy [26] et Ptolemy II [27], Moose [18][CGF97], OPnet [21], Hyperformix Workbench [11], Lockheed Martin CSim [13], OMNet++ [20], Simulink/Matlab, Scilab+Scicos par exemple), permettant de lier élégamment les étapes de modélisation et d'évaluation à partir du modèle. Beaucoup de ces outils sont graphiques, et consistent en la composition hiérarchique de boîtes interconnectées pour l'échange d'événements, les boîtes étant elles-mêmes composées d'un réseau de boîtes interconnectées (voir la figure 4.3 pour une illustration). Ces outils sont en général spécialisés pour un forma-

[2] QNap2 fournit également une série de *solveurs* qui permettent une résolution analytique de certaines catégories de systèmes, mais la simulation reste la seule méthode d'évaluation pour les modèles les plus complexes.

```
Simulation begin
    ref(head) Q;
    Integer   U;
    Boolean   Guichet_Occupe;

    process class client;
    begin
        activate new client delay negexp(1/10, U);

        if Guichet_occupe then begin
           into(Q);
           passivate;
           out;
        end;

        Guichet_occupe := TRUE;
        hold( uniform( 0,18, U ) );
        Guichet_occupe := FALSE;

        if not Q.empty then activate Q.first;
    end;

    Q :- new Head;
    U := clocktime;

    activate new client delay randint(0,5,U) ;

    hold(200);
end
```

(c) Simula67

```
/DECLARE/ QUEUE Client,Guichet;

/STATION/ NAME=Client; INIT=1; TYPE=INFINITE;
TRANSIT=Guichet; SERVICE= EXP (0.1);

/STATION/ NAME=Guichet; TRANSIT=Client;
SERVICE= UNIFORM (0.0,18.0);

/CONTROL/ TMAX=200.0;

/EXEC/ BEGIN
 SIMUL;
END;

/END/
```

(d) QNap2

FIG. 4.2: Simulation d'un même système avec Simula 67 et QNap2

lisme donné (simulation de réseaux avec Hyperformix Workbench, OPnet et l'utilisation classique de OMNET++ par exemple), bien qu'il en existe qui permettent à chacune des boîtes d'être décrite dans des formalismes différents, allant du modèle d'automate temporisé à états finis, aux files d'attente (ERS par exemple, qui permet de composer différents outils pour analyser des modèles dans différents formalismes).

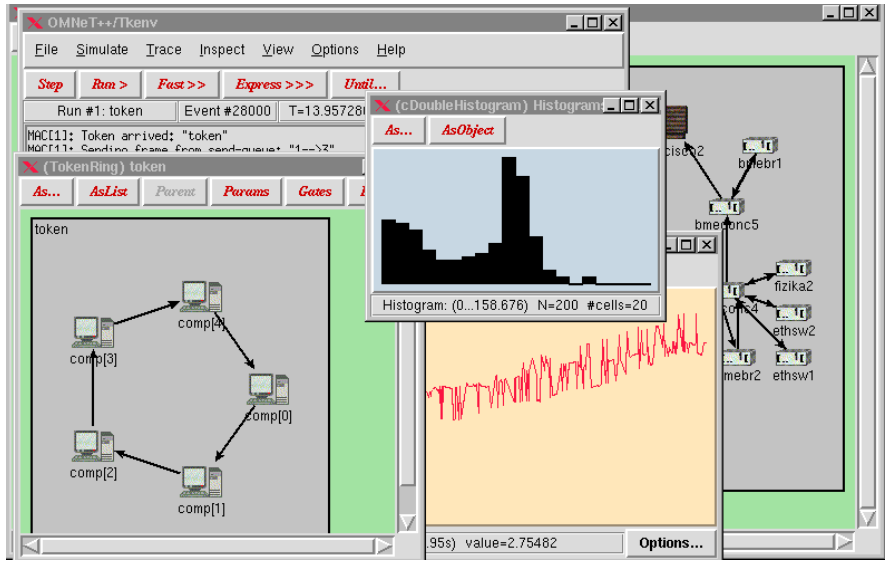


FIG. 4.3: Exemple de réseau modélisé avec OMNET++

4.1.3 Simulateurs hybrides

Les deux catégories de simulateurs à temps continu / à événements discrets ne forment pas deux compartiments hermétiques du domaine de la simulation. Les simulateurs hybrides se chargent de faire cohabiter les deux : on parle de simulation multi-modèle [FNSB94] (Moose) ou à domaines multiples (Ptolemy). Ils rendent par exemple possible la simulation d'un système de contrôle (simulation à événements discrets) qui interagit avec un environnement physique (simulation à temps continu : mécanique, thermodynamique, physique, ...). Citons par exemple Matlab, Scilab, Moose, Ptolemy.

4.2 Application à l'évaluation de systèmes temps-réel

Dans le chapitre 2, nous avons indiqué que la simulation pour l'ordonnancement a d'autant plus d'intérêt qu'elle permet de reconstituer fidèlement le comportement du système. Nous voyons ci-dessous les possibilités offertes et les éventuelles limitations de ce point de vue avec les outils existants, génériques et dédiés. Les outils que nous considérons sont (i) les simulateurs par utilisation de langages, d'outils ou de bibliothèques génériques de simulation à événements discrets ; (ii) les simulateurs dédiés à

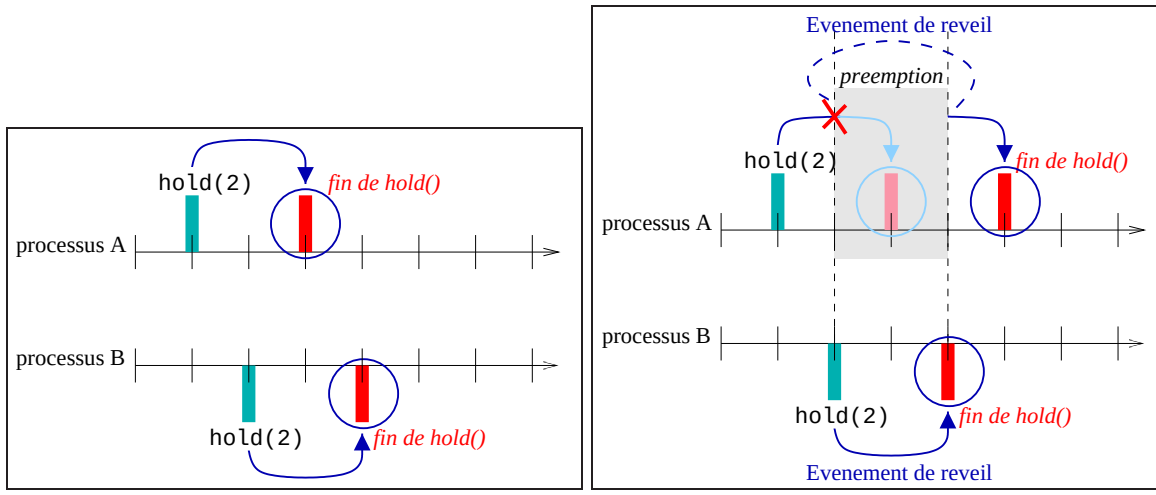
l'évaluation de systèmes temps-réel à partir du modèle ; (iii) les simulateurs de systèmes temps-réel complets pour l'évaluation de l'implantation.

4.2.1 Langages, bibliothèques et outils génériques

Dans son fonctionnement le plus simple, la simulation à événements discrets implique la gestion des dates des événements dans une seule et même échelle globale, avec le plus souvent la propriété simple suivante : la programmation d'un événement dans le calendrier ne remet jamais en cause les événements déjà présents dans le calendrier (ni leur date, ni leur sens). Par exemple, quand on simule un processus (en ordonnancement temps-réel, on parlerait de *tâche*) qui utilise une ressource pendant un intervalle de temps de durée t, cela se traduit par la programmation d'un événement de réveil à la date `date_prise_de_ressource + t`, et aucun événement qui serait programmé entre `date_prise_de_ressource` et `date_prise_de_ressource + t` ne viendrait perturber ce schéma. Autrement dit, cela revient à établir qu'il n'y a pas de notion de préemption, ou qu'une fois que la ressource est acquise, elle le reste indépendamment d'autres processus qui viendraient en faire la demande.

Une composante centrale en ordonnanceur temps-réel étant l'ordonnanceur, la problématique de la simulation de tels systèmes passe au moins par la prise en compte des préemptions du processeur (tout du moins pour les ordonnancements préemptifs, largement répandus), ou, plus généralement, par la prise en compte du partage des ressources actives préemptibles (voir 1.3.3). Ainsi, il s'agit de pouvoir simuler que le processus utilise la ressource processeur pendant t secondes pour effectuer ses traitements, et ceci même si la ressource processeur vient à lui être réquisitionnée au profit d'un autre processus. En reprenant le fonctionnement simple des simulateurs à événements discrets ci-dessus, ceci conduit aux manipulations de la figure 4.2.1 du calendrier des événements : à chaque fois que le traitement d'un processus est préempté, l'événement de réveil qui marque la fin de la simulation du traitement (programmé par l'appel à la fonction `hold(delay)` sur la figure) est repoussé du délai introduit par la préemption.

Parmi les langages, bibliothèques, ou outils de simulation à événements discrets génériques, simuler l'occupation du processeur en tenant compte des préemptions est parfois disponible (par exemple dans QNap2). Mais le formalisme de modélisation est généralement celui de réseaux de files d'attente, éloigné de celui considéré ici, ce qui nécessite d'une part de remodéliser le système étudié suivant ce formalisme en le décomposant en un grand nombre d'éléments (portion de code des tâches par exemple), ce qui s'accompagne de problèmes d'efficacité de la simulation, tant le nombre d'éléments simulés devient élevé. D'autre part, les résultats de l'évaluation s'intéressent à caractériser le comportement du modèle en régime permanent en termes de nombre de *clients* moyen ou maximal dans chaque queue, le temps moyen de présence des clients dans les queues, ... Or, en ce qui concerne l'ordonnancement, l'évaluation du système passe aussi par l'étude de la séquence d'exécution et de préemption des tâches, ce qui nécessite de modifier les outils et/ou les éléments simulés puisque les mesures que le simulateur effectue sont alors insuffisantes, ou peu pertinentes de par le fait que la notion de régime permanent pour de tels systèmes est difficile à définir.



(a) Sans contrôle de partage

(b) Ressource active préemptible

FIG. 4.4: Gestion du calendrier des événements pour la simulation d'une ressource partagée que chacun des processus souhaite utiliser pendant 2 unités de temps

Lorsque l'outil de simulation ne propose pas cette notion de ressource préemptible, une première solution est de manipuler à la main (à son insu) son calendrier des événements (suppression, ajournement), comme le fait par exemple le simulateur de Basement [HS] au dessus de la bibliothèque C++-Sim. Une technique alternative (utilisée dans OPNet par exemple) consiste à modéliser les processus sous la forme d'automates, pour lesquels on associe à chaque état la simulation d'une séquence d'occupation *non préemptible* du processeur pendant une durée donnée ; les préemptions ont lieu lors d'une transition entre états dans un automate : elles correspondent à sélectionner un état de l'automate d'un processus (le même ou un autre). Ce principe revient à définir les points de préemption possibles dans le processus simulé, et évite d'avoir à repousser les dates de réveil des séquences d'occupation du processeur simulées à cause des préemptions, puisque précisément les préemptions sont interdites pendant la simulation de l'occupation du processeur. Cette approche est utilisée par exemple dans [EC99] et [Cas00] décrits en 4.2.2.2.

Afin d'éviter la tâche complexe de la traduction d'un modèle d'un formalisme dans un autre et de se heurter aux problèmes d'efficacité qui en résultent, ou afin d'éviter de refaire la gestion du calendrier d'événements au dessus d'un mécanisme de gestion de calendrier existant, le domaine de la simulation de systèmes temps-réel s'est doté d'outils dédiés, que nous présentons dans la suite.

4.2.2 Simulateurs pour l'évaluation à partir de modèles de systèmes temps-réel

Lorsqu'il s'agit d'évaluer le modèle d'un système temps-réel par simulation, deux grands types de simulations sont possibles, suivant la granularité, et donc la précision avec laquelle les simulations sont effectuées. Nous distinguons deux grandes classes de granularité :

- Les tâches correspondent à des périodes d'occupation du processeur dont la durée est définie de bout en bout sous la forme d'une loi statistique simple (section 4.2.2.1).
- Les tâches sont définies plus précisément, pour faire apparaître le profil de comportement d'un algorithme donné (qui peut être constitué de tests, de boucles, ...). Dans ces conditions, une loi statistique de bout en bout qui modélise cette caractéristique serait extrêmement complexe (section 4.2.2.2).

Pour chacune des deux catégories, nous donnons d'abord une description générale du principe, puis une série d'exemples et de références, avant de conclure sur les limitations.

4.2.2.1 Simulation avec *charge synthétique*

Définitions et principe

Le principe est de générer aléatoirement des applications synthétiques, c'est-à-dire des séries de tâches "*creuses*" qui n'effectuent aucun traitement, et qui sont uniquement caractérisées par leur date aléatoire d'activation (entre autres liée aux stimuli de l'environnement), leur temps d'exécution aléatoire, leurs besoins aléatoires en ressources, et d'autres contraintes (de précédence par exemple).

Toutes les données aléatoires doivent cependant correspondre à une loi statistique donnée connue, et qui respecte les hypothèses sur le comportement de l'environnement et de l'application. Par exemple, pour la simulation de tâches sporadiques, les dates d'activation des travaux des tâches générées doivent respecter l'hypothèse sur la durée d'inter-arrivée minimale spécifiée dans le modèle de la tâche. De même, les temps d'exécution des travaux générés doivent être inférieurs au WCET spécifié dans le modèle de la tâche.

Les simulateurs de cette catégorie prennent la forme d'un simulateur à événements discrets orienté événements, reposant ou non sur une bibliothèque ou un outil de simulation à événements discrets générique existant. Ils se focalisent sur la simulation d'ordonnancement sur monoprocesseur, en général en négligeant l'impact des coûts système et matériels. Certains sont limités à la simulation d'ordonnancement avec une échelle de temps discrète (*i.e.* à valeurs dans $\mathbb{N}$), c'est-à-dire avec des temps et des durées multiples entiers d'un quantum de temps fixé (la granularité de l'horloge système), ce qui correspond à une granularité élevée. Des simulateurs plus complexes supportent la simulation de multiprocesseur, voire de système distribué avec échange et ordonnancement de messages.

Les plus simples génèrent des traces d'exécution brutes (création/suppression de tâches, préemptions) sous forme textuelle ou graphique (par des chronogrammes ou dia-

grammes de Gantt). Les plus évolués permettent de filtrer les événements à considérer, et de construire des métriques synthétiques complexes.

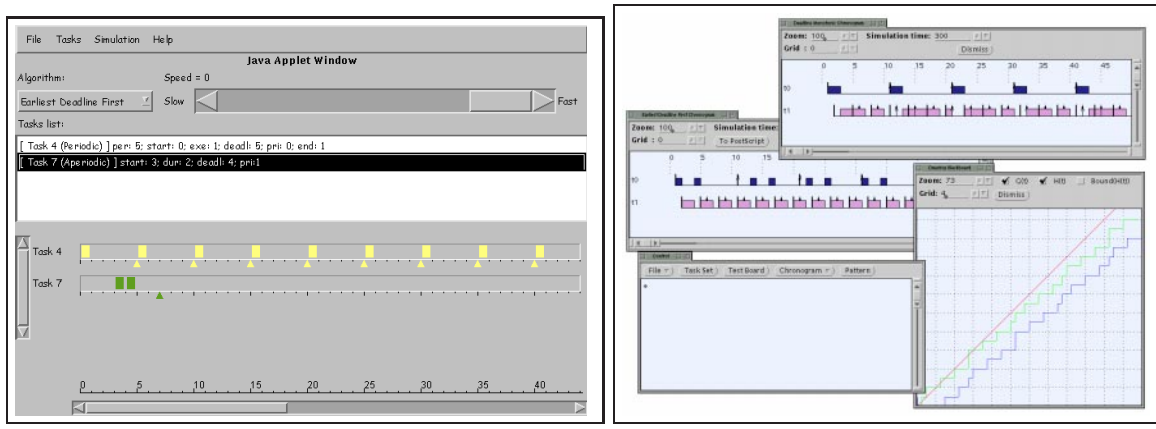
Ces simulateurs se prêtent bien à l'évaluation d'algorithmes d'ordonnancement, pour lesquels les modèles les plus simples négligent les coûts d'exécution en dehors de ceux des tâches. Ils sont très utilisés par les universitaires lorsqu'une nouvelle politique d'ordonnancement est proposée. Ils sont utilisés en particulier pour évaluer le comportement d'ordonnanceurs avec sous-partie dynamique (par exemple pour évaluer les taux de garantie ou les temps de réponse des tâches apériodiques), ou pour évaluer le comportement des ordonnanceurs avec gestion de la surcharge.

Exemples

Pratiquement tous les laboratoires de recherche en ordonnancement temps-réel ont développé leur propre outil de simulation avec tâches creuses, adapté au modèle de tâches sur lequel ils travaillent.

Les plus simples sont des simulateurs à événements discrets non personnalisables, qui simulent les systèmes à ordonnancement à priorité statique, tels que Scheduler 1-2-3 [TNR90b] utilisé pour RTMach, Schedsim [37], ou SRMS Workbench [39] par exemple. D'autres sont dédiés à d'autres ordonnanceurs, comme Dharma pour l'ordonnancement myope en distribué [MMM00a].

Il existe également des simulateurs qui supportent un modèle de tâche plus développé afin de pouvoir simuler une palette donnée de plusieurs ordonnanceurs (en général EDF, LLF, ainsi que RM et DM), tels que IEDFC d'Ismael Ripoll [51], Cheddar de l'université de Brest [52], le simulateur pour Java d'E. Lefevre [49] ou de l'UTSA [43], ou LUCAS [14] pour EDF sur Matlab/Simulink. Avec ces simulateurs, une interface graphique permet le plus souvent de modéliser le système et d'observer son comportement sous forme de chronogramme, comme l'illustrent les captures d'écran de la figure 4.5.



(a) Simulateur Java E. Lefevre (b) IEDFC

FIG. 4.5: Captures d'écran de simulateurs d'ordonnancement synthétiques

D'autres simulateurs offrent une structure plus modulaire (orientée objet) leur permettant d'étendre et le modèle de tâches, et l'ordonnanceur, comme RAPIDS [29] de l'université du Massachusetts, RTSim et Ghost de l'université de Pise (avec prise en compte de surcoûts système simulés par loi statistique), et GAST [JV96, JV97, Jon98] de l'Université de Chalmers (qui s'intéresse à l'ordonnancement et à l'affectation des processeurs pour le distribué pour des tâches communicantes organisées en graphes).

Limitations

La simulation par charge synthétique présente l'inconvénient d'éloigner le modèle utilisé dans la simulation, de l'implantation ultérieure. Ceci augmente le risque de non pertinence du comportement temporel simulé par rapport au comportement temporel effectif de l'implantation ultérieure.

Une première source de non pertinence possible provient du fait que si on connaît l'algorithme déroulé par une tâche, trouver une loi probabiliste représentative de son comportement temporel peut être très difficile.

Une autre limitation commune aux ordonnanceurs de ce type est que la simulation des coûts système est soit inexistante, soit (plus rarement) prend la forme d'un surcoût temporel aléatoire : aucun de ces simulateurs ne permet de rendre compte plus finement du comportement temporel de ces coûts.

Enfin, outre le fait que les tâches n'effectuent aucun traitement, les simulateurs d'ordonnancement de cette classe ont en général l'inconvénient de contraindre les modèles de tâches supportés (le plus souvent un simple modèle à priorité), et de se limiter à la simulation d'une palette fixée non extensible d'ordonnanceurs, qui ne sont la plupart du temps associés à aucun protocole d'accès aux ressources partagées. Ces deux caractéristiques mises bout à bout, il en découle qu'il est difficile de simuler d'autres ordonnanceurs que ceux proposés par défaut sans une modification en profondeur du simulateur.

4.2.2.2 Simulation avec tâches "pleines"

Principe

Ces simulateurs se spécialisent également dans la simulation d'ordonnancement. Ils ajoutent les mécanismes pour simuler plus précisément le comportement temporel des tâches, voire du support d'exécution, en l'intégrant avec l'algorithme qui est exécuté par chaque tâche (resp. par le support système). En ce sens, ces simulateurs peuvent davantage trouver leur place plus en aval dans le cycle de développement et de validation de systèmes temps-réel.

L'outil prend la forme d'un simulateur à événements discrets orienté processus, dans lequel chaque tâche dans le système simulé est un processus de la simulation. Chaque tâche peut effectuer de vrais traitements, ou au moins dérouler un algorithme, et est associée à la simulation d'occupation de la ressource processeur pendant une durée contrôlée. Souvent, les traitements effectués dans les tâches sont exprimés dans un langage interprété dédié.

Exemples

L'outil de simulation le plus simple dans cette catégorie est intermédiaire entre les approches à tâches creuses et à tâches pleines. Il s'agit d'exprimer chaque tâche sous la forme d'un traitement effectué en temps simulé nul, et d'une occupation simulée du processeur de bout en bout pendant un temps contrôlé défini par un algorithme personnalisable.

Les outils les plus simples parmi eux n'effectuent aucun traitement dans les tâches, mais utilisent un modèle de comportement temporel non probabiliste, défini sous la forme d'un automate : Retis C++ Simulator de l'université de Pise, le simulateur synthétique de PERTS [LRD+93] repris dans l'outil commercial RAPID Sim [42], ou Sew [6] de Carnegie Mellon.

Plus complexes et précises, les approches du projet Trio [CSSLC00] et de la boîte à outils matlab/simulink de [EC99] reposent sur ce principe, et consistent à représenter le système, éventuellement distribué, sous la forme d'un graphe de tâches communicantes, au dessus d'outils de simulation à événements discrets plus génériques (OPNet et Hyperformix workbench pour [CSSLC00], simulink pour [EC99]). Avec cette approche, les traitements sont personnalisables (C pour [CSSLC00], matlab/C pour [EC99]), l'ordonnanceur également (par défaut : priorité fixe pour [EC99], priorité quelconque préemptif et non préemptif pour [CSSLC00]) ; et dans [EC99], il peut être étendu pour supporter d'autres modèles de tâches que ceux par défaut. Mais les coûts d'ordonnancement et du système d'exploitation ne sont pas pris en compte. L'outil commercial timewiz [54] est une alternative à ces outils, qui propose un environnement graphique de modélisation accompagné d'un outil de validation par analyse statique, mais se limite à un ordonnanceur (à priorité fixe).

Le simulateur de l'Université Libre de Belgique [VGH96, GH98, GM99] est plus intégré, et permet de modéliser conjointement le traitement des tâches et leur temps d'occupation du processeur, puisqu'il repose sur un langage simple dédié pour exprimer les traitements, sans boucle possible, qui est équipé d'une primitive particulière (l'opérateur "crochet") pour simuler l'occupation du temps processeur. Le langage vise en priorité la simulation de la gestion des ressources (évaluation de protocoles de gestion de ressources). De la même manière, l'outil repose sur un autre langage spécialisé permettant de définir son propre ordonnanceur, ce qui confère une grande souplesse pour l'expérimentation de nouveaux ordonnanceurs. Mais les coûts de l'ordonnanceur ne sont pris en compte que sous la forme d'un surcoût aléatoire. Le même type d'approche est adopté dans Stress [ABRW94], qui est équipé d'un langage de description des traitements des tâches un peu plus riche (permet les boucles), mais le système simulé repose sur le principe du *tick scheduling*. L'exemple ci-dessous (repris de [ABRW94]) donne un exemple de simulation (1 tâche + ordonnanceur EDF) :

```
periodic task1
  period 10
  deadline 5
  variable var
  variable tri
  if tri > 100 then
    tri := 0
  var := 10
  loop var > 0 max 20
    { tri := tri + var * var
      [2, 4] -- occupation du CPU
      var := var - 1 }
endper

scheduler periodic early
  period 1
  deadline 0
  offset 0
```

```
priority 0                                          if task != mytask
hidden                                              then effpri of task := deadline of task
variable task                                   endper
for task in tasklist of myproc max 999999
```

Le simulateur pour l'ordonnancement de [JRR97] (prévu pour l'évaluation de l'ordonnanceur Rialto) propose une approche comparable au niveau de la simulation du traitement des tâches, mais à la fois le modèle des tâches et l'interface du système d'exploitation simulé (un large sous-ensemble de l'API kernel win32) ne sont pas extensibles ni personnalisables.

L'outil DRTSS [24] intégré dans l'environnement PERTS (ancêtre de l'outil commercial RAPID Sim [42]) est à l'origine un outil de simulation par tâches creuses, mais il a été étendu pour supporter la notion de tâche pleine, ou *livetask* [SL96], sans que cette évolution ait été reprise dans RAPID Sim. DRTSS correspond à une bibliothèque de fonctions `C++` : les tâches sont codées en `C++` et une primitive dédiée permet de simuler l'occupation du processeur. L'ordonnanceur se présente sous la forme d'une tâche spéciale avec laquelle les autres tâches du système interagissent par transactions, ce qui permet *de facto* de prendre en compte les coûts d'ordonnancement. Le simulateur impose cependant la modélisation du système sous la forme d'une composition hiérarchique de tâches communicantes dont le modèle est imposé, ne propose pas la simulation d'un système d'exploitation, et impose au modèle de système simulé la notion de tâche dédiée pour l'ordonnancement ainsi que la notion de transaction. L'outil s'intéresse en priorité à la vérification de la correction du comportement temporel du système vis-à-vis des contraintes. Il vient également avec un dispositif évolué de filtrage des traces de simulation et d'établissement de métriques composites, reposant sur un langage dédié.

Limitations

Ces simulateurs présentent l'avantage de pouvoir rapprocher le modèle simulé de son implantation ultérieure. Ils limitent ainsi le risque de non pertinence des lois d'occupation du processeur simulé des tâches, par rapport au comportement temporel effectif de l'implantation ultérieure.

Avec les simulateurs les plus simples cependant, modèle de comportement temporel et algorithmes déroulés par les tâches sont exprimés sous deux formes distinctes (la plupart du temps : deux langages). Ceci crée de nouveau un risque de non pertinence possible, puisqu'il faut maintenir le modèle de comportement temporel en cohérence avec l'algorithme déroulé. Il en résulte de plus une duplication de l'effort de développement par absence de partage de code entre les deux.

Enfin, les outils de simulation de cette catégorie présentent en général les mêmes limitations que les simulateurs fondés sur des tâches creuses en ce qui concerne la prise en compte des coûts du système d'exploitation et de l'ordonnanceur, la souplesse vis-à-vis des modèles de tâches supportés, de l'extension de la palette d'ordonnanceurs supportés, ou de la résolution des simulations qui fonctionnent en général en *tick scheduling*, avec des temps et des durées multiples entiers d'un quantum de temps fixé (en général la granularité de l'horloge système).

4.2.3 Simulateurs de systèmes complets

Description

Ces outils reprennent l'implantation de l'application dans son ensemble, et requièrent peu de modifications pour simuler le comportement du système dans son intégralité. Un autre intérêt de ce type de simulateur est que le système d'exploitation, en tant qu'interface de programmation, et en tant que consommateur de temps processeur, est pris en compte.

Il s'agit de variantes des simulateurs de modèles à tâches pleines, mais avec une granularité temporelle de simulation plus fine ou plus contrôlable : il existe des simulateurs dont la granularité de simulation est l'instruction machine sur la plate-forme cible, ou la ligne de code source, ou une granularité intermédiaire définie par l'utilisateur. D'autre part, pratiquement tous les systèmes d'exploitation temps-réel commerciaux sont accompagnés d'un simulateur, qui permet de simuler le comportement de l'application sans avoir à faire l'effort (en temps ou en matériel) de la déployer sur machine cible.

Ce type de simulation a un intérêt lorsque à la fois le système d'exploitation, l'application, et l'environnement sont simulés avec des finesses et des précisions comparables. Par exemple, si l'application simulée l'est à la granularité de l'instruction machine, ne pas prendre en compte les coûts de préemption annule l'intérêt de la simulation au niveau instruction. Pour les simulateurs qui ne fixent pas de granularité de simulation particulière (tel que celui présenté en partie II), une difficulté est ainsi de définir une granularité de simulation qui soit cohérente pour l'ensemble du système et de l'environnement simulé. Une autre difficulté reliée est d'établir un compromis entre précision et rapidité d'exécution de la simulation.

Exemples

Les simulateurs les plus simples négligent la simulation précise du comportement temporel, en confondant temps-réel simulé et temps de la simulation, comme le simulateur OSE Softkernel [22] pour le noyau temps-réel OSE, ou Precise MQXSim [19] pour le noyau MQX par exemple. D'autres distinguent les deux échelles de temps, mais le comportement temporel n'en est pas plus finement restitué pour autant : par exemple, le simulateur TAPS [40] pour le noyau temps-réel AMX possède une primitive spéciale pour signifier la consommation de temps processeur dans l'échelle de temps simulé, mais celle-ci est non préemptive, ce qui n'est plus conforme au comportement effectif de l'implantation réelle.

Ces simulateurs, et d'autres (comme le simulateur WindRiver VxSim [44] pour VxWorks), sont limités au niveau des interfaces de système d'exploitation qu'ils peuvent prendre en compte, ou des ordonnanceurs et modèles de tâches qui sont disponibles. Ceci limite l'évaluation de nouveaux mécanismes système par exemple.

À l'inverse, il est possible d'utiliser les simulateurs de machine cible, pour processeur nu (comme les simulateurs de processeurs commerciaux, ou les simulateurs SoftPC [36] pour Intel 80286, SPIM [38] pour MIPS, ou SimpleScalar [34]), ou pour une architecture matérielle simulée complète (horloge système, gestionnaire d'interruptions, DMA,

... Comme bochs [48] pour Intel x86/AMD x86-64, Sid [47] pour ARM/Sanyo/x86 *via* bochs, Nachos [55] pour MIPS, ou SimOS [32] pour MIPS et Alpha, SimICS [31] pour de très nombreuses architectures par exemple). Cette approche permet de s'affranchir des problèmes liés à la limitation à un système d'exploitation particulier, mais possède l'inconvénient d'être très lente à l'exécution, de plusieurs ordres de grandeurs par rapport à une exécution sur machine réelle. Et elle se heurte aux mêmes problèmes d'expressivité des observations (*i.e.* des traces de très bas niveau) que l'évaluation par instrumentation matérielle (voir 2.3.2.2).

Enfin, il existe un outil original, Carbonkernel [5][DG01], qui fait un compromis entre la simulation par tâches pleines, et les simulateurs de systèmes complets. Les tâches simulées peuvent partager le code source avec l'implantation, et le compilateur s'occupe de générer, pour chaque ligne de code source, un appel à une primitive de simulation d'occupation non préemptible du temps processeur. En ce sens, CarbonKernel se rapproche des simulateurs par tâches pleines. Mais l'outil permet de définir la notion de système d'exploitation (incluant l'ordonnanceur) sous la forme d'une interface de programmation (une *personnalité*) avec laquelle l'application interagit (par défaut : `eCos` et `RTLinux`), et dont les coûts d'exécution sont pris en compte de la même manière que pour les tâches. En ce sens, CarbonKernel se rapproche des simulateurs de systèmes complets. Malheureusement, le simulateur n'offre aucun contrôle plus précis sur le temps d'occupation du processeur associé à chaque ligne de code source, ce qui peut rendre le comportement temporel de la simulation non représentatif de celui de l'implantation effective.

Limitations

Ces outils de simulation souffrent de deux limitations majeures liées à leur origine. D'abord, ils sont le plus souvent adaptés au système d'exploitation auquel ils sont associés, et en plus ils sont restreints à ce seul système d'exploitation. Que ce soit pour modifier le modèle de tâches, la politique d'ordonnancement ou de gestion de ressources, ou l'interface de programmation du système d'exploitation, le travail nécessaire est conséquent (modifications sur du code système et modifications des outils), voire la plupart du temps impossible (outils commerciaux fermés).

Également, de par l'objectif de ces outils, qui est en priorité de valider l'aspect fonctionnel du système par simulation, l'aspect temporel est relégué au second plan, ou en tout cas la simulation ne vise pas à être pertinente vis-à-vis du comportement temporel de l'implantation effective : il s'agit le plus souvent de détecter les erreurs de réalisation classiques (algorithmes incorrects, fuite ou erreur de manipulation de la mémoire, effets de bord non pris en compte, mauvaise gestion des cas d'erreur par exemple), ou liées à la synchronisation de ressources (interblocages, inversions de priorité). À tel point qu'il existe des simulateurs qui fonctionnent en confondant l'échelle de temps simulée, avec l'échelle de temps de la machine de simulation (*i.e.* si la tâche met 10ms à s'exécuter sur la machine de simulation, cela correspond à 10ms de temps simulé).

deuxième partie

Plate-forme de simulation
ARTISST

Dans cette partie, nous présentons la plate-forme d'évaluation par simulation qui a été développée : ARTISST, pour "ARTISST is a **R**eal-**Ti**me **S**ystem **S**imulation **T**ool". Nous commençons par indiquer les motivations qui nous ont conduits à sa réalisation (chapitre 5), puis nous en donnons un bref aperçu (chapitre 6). Nous présentons ensuite sa structure interne en commençant par le cas de la simulation simple de systèmes centralisés, puis de systèmes distribués (chapitre 7). Dans le chapitre 8, nous présentons les ordonnanceurs fournis par défaut avec ARTISST, et indiquons quelques travaux qui ont été effectués dans ce domaine (en particulier concernant la prise en compte de la granularité de l'*horloge système*). Une synthèse des caractéristiques d'ARTISST est rassemblée au début du dernier chapitre, avant de résumer la démarche d'un utilisateur qui veut réaliser une simulation, qui est détaillée dans le tutoriel [Dec02] (en anglais) fourni avec le code source de la plate-forme ARTISST.

Chapitre 5

Motivations

5.1 Limitations des méthodes d'évaluation existantes

Comme nous l'avons vu dans la première partie, l'évaluation d'un *modèle* de système temps-réel par analyse hors-ligne s'avère d'autant plus complexe, voire impossible, que le modèle de système possède une composante dynamique, ou que le comportement temporel du système et/ou de l'environnement n'est pas entièrement connu a priori (risque de surcharge par exemple). Et elle s'avère d'autant plus pessimiste que des ressources système sont partagées, que les tâches sont soumises à des contraintes de précédence, que les lois d'arrivée sont pessimistes, ou que les temps d'exécution pire-cas sont sur-évalués. De plus, lorsqu'on s'intéresse à l'évaluation de l'*implantation* par analyse, se pose le problème de *remonter* les informations sur le comportement temporel effectif de l'implantation vers le modèle soumis à analyse. Ceci s'accompagne de risques de non sûreté lorsque le formalisme du modèle est éloigné de l'implantation effective.

Pour ces raisons, l'évaluation du modèle de système par exécution (simulée pour le modèle, réelle instrumentée ou simulée pour l'implantation) est une alternative qui se justifie, bien qu'elle pose des problèmes de sûreté par absence de couverture complète. À cause des problèmes d'évaluation par exécution réelle instrumentée que nous avons présentés dans la première partie, en particulier les phénomènes d'intrusion et de lourdeur de mise en place de protocoles d'évaluation, nous nous sommes orientés vers l'évaluation par simulation, à partir du modèle ou portant sur l'implantation du système complet.

Dans ce chapitre, nous commençons par énoncer les propriétés recherchées pour l'évaluation de systèmes temps-réel par simulation à partir de modèle abstrait, ou à partir d'une implantation concrète. Nous rappelons ensuite les limitations de l'existant vis à vis de ces vœux.

5.2 Vœux pour un simulateur

Nous souhaitons disposer d'une infrastructure permettant d'évaluer aussi bien un modèle de système abstrait (*i.e.* simulation par tâches creuses pour l'évaluation du

comportement général d'algorithmes d'ordonnancement par exemple), qu'une version la plus proche possible de l'implantation réelle (*i.e.* simulation par tâches pleines pour l'évaluation du comportement d'un système particulier).

5.2.1 Évaluation à partir de modèles

Modèle de tâches facilement personnalisables

Pour faciliter la simulation de modèles aussi nombreux que possible, en particulier pour permettre l'évaluation d'ordonnanceurs aussi variés que possible, le simulateur ne doit pas être associé à un modèle de tâches unique : il doit être possible de définir les modèles de tâches facilement. Il conviendrait que le simulateur ne pose aucune hypothèse sur un "portrait type" (ou *patron*) de modèle de tâche attendu.

Ordonnanceur personnalisable

Le simulateur ne doit pas supposer que l'ordonnanceur possède des caractéristiques particulières. Par exemple, les ordonnanceurs ne fonctionnent pas tous au dessus de files de tâches classées par priorité. Il doit être possible à l'utilisateur de définir son ordonnanceur de pair avec le modèle de tâches, facilement et de façon localisée dans le code (ne pas à avoir à intervenir sur un nombre importants de composants du système).

Protocoles de gestion de ressources personnalisables

Le simulateur doit permettre d'implanter des protocoles d'accès aux ressources, sans pour cela contraindre leur interface de programmation, ni leur interaction avec l'ordonnanceur.

5.2.2 Évaluation d'une implantation

On l'a vu en première partie, l'évaluation de l'implantation est d'autant plus pertinente que l'implantation simulée est fidèle à l'implantation réelle. Que ce soit au niveau de la fidélité *structurelle et fonctionnelle*, qu'au niveau de la fidélité du *comportement temporel*.

Simulation des traitements avec une base de code source commune

Au niveau de la fidélité structurelle et fonctionnelle, l'objectif principal qu'on se fixe est qu'il doit être possible de restituer les traitements de l'application.

Afin d'éliminer les risques de divergence de comportement logique entre implantation et simulation, on se donne comme contrainte forte de pouvoir reprendre le code source d'une implantation existante, et de l'utiliser dans la simulation avec le moins de modifications possible.

Simulation fonctionnelle et temporelle intégrée

Au niveau de la fidélité temporelle, la contrainte qu'on se fixe est de ne pas dissocier la simulation du comportement temporel, de la simulation du comportement fonctionnel. Le but étant de diminuer les risques de non pertinence de la modélisation du comportement temporel du modèle, par rapport à l'exécution réelle.

En tenant compte de la contrainte de partage de code source, ceci signifie qu'on souhaite pouvoir intégrer la simulation de l'occupation du processeur dans le code source même du système simulé, sans pour autant être obligé de faire fonctionner la simulation sur la machine cible réelle (il s'agirait de la faire fonctionner sur station de travail). Ceci permet de plus d'éviter l'effort de re-modélisation du comportement temporel dans un autre formalisme.

Une autre contrainte qu'on se donne à ce sujet, est qu'on ne définit pas une granularité fixée pour la simulation du comportement temporel : toute valeur numérique pour la simulation de l'occupation du processeur doit être tolérée, afin d'autoriser la simulation du système avec un niveau de précision temporelle de la simulation arbitrairement élevé. La fidélité du comportement temporel simulé sera d'autant meilleure que l'utilisateur du simulateur choisira de simuler le comportement du système à plus bas niveau (simulation au niveau instruction, ou simulation de la micro architecture du processeur), mais ce sera au détriment des temps de simulation.

Simulation du système d'exploitation personnalisable

Dans les systèmes temps-réel concrets, l'application repose sur les services d'un système d'exploitation qui s'occupe de la gestion des ressources (dont la ressource processeur *via* l'ordonnanceur).

Pour que l'approche par simulation soit plus souple à mettre en œuvre que l'exécution réelle instrumentée, il est souhaitable que le système d'exploitation, en tant qu'interface de programmation et que services fournis, soit personnalisable. Ceci généralise le souhait ayant trait à la personnalisation de l'ordonnanceur et des protocoles de gestion de ressources pour l'évaluation de modèles.

Prise en compte des coûts d'exécution du système d'exploitation

La fidélité de simulation de comportement temporel du système ne repose pas uniquement sur la restitution du comportement temporel de l'application : il est aussi souhaitable que le simulateur prenne en compte les coûts d'exécution du système d'exploitation, ceux-ci incluant l'ordonnanceur. Il serait également intéressant que le simulateur sache distinguer les temps d'exécution liés aux traitements de l'application, de ceux liés au système d'exploitation simulé.

Pour les mêmes raisons que précédemment, il est souhaitable d'intégrer la simulation du comportement temporel dans le code source du système d'exploitation et de l'ordonnanceur simulés.

Décorrélation des échelles de temps-réel et de temps-système

Les systèmes temps-réel n'ont en général pas accès à l'échelle de temps-réel, par nature continue et globale à tous les nœuds. Mais ils disposent le plus souvent d'une base de temps locale, mise à jour *régulièrement* par l'intermédiaire de matériel spécialisé (une *horloge*), qui leur permet de disposer d'une échelle de temps discontinue évoluant par paliers : l'*échelle de temps système* locale.

Afin de reproduire les effets de la granularité de l'*horloge système* dans le comportement du système simulé, notamment au travers des décisions d'ordonnancement lorsque celles-ci font intervenir le temps, il est souhaitable de reproduire cette décorrélation entre les deux échelles de temps, caractéristique des systèmes temps-réel concrets. Nous verrons dans la partie III ces effets sur une palette d'ordonnanceurs temps-réel.

Une autre facette de cette caractéristique, est que, pour les systèmes distribués, une synchronisation exacte des visions locales du temps, *via* l'horloge système, est rarement garantie. Il est donc aussi souhaitable de reproduire cette décorrélation entre l'échelle de temps-réel commune, et l'échelle de temps système locale à chaque nœud, afin de pouvoir simuler les désynchronisations et les anomalies des horloges.

5.3 Limitations des outils de simulation existants

Dans la première partie, nous avons présenté une palette d'outils utilisables pour la simulation de modèles ou d'implantations de systèmes temps-réel. Cependant, aucun des outils qui ont été indiqués ne permet de satisfaire simultanément tous les souhaits émis ci-dessus.

Si on s'intéresse aux outils de simulation à événements discrets généralistes (langages, outils, ou bibliothèques de fonctions), certains proposent la simulation de ressources actives préemptibles, telles que le processeur, mais leur formalisme de modélisation (en général des réseaux de files d'attente) nécessite une traduction lourde et fastidieuse depuis le modèle dans le formalisme considéré ici (modèles orientés allocation de ressources), d'autant plus qu'on souhaite permettre la simulation de tâches pleines pour lesquelles on souhaite partir du code source. D'autres outils de ce type ne permettent pas par défaut la simulation de telles ressources préemptibles : lorsqu'un événement est programmé, comme par exemple un événement marquant la fin de la simulation d'occupation du processeur, celui-ci doit être repoussé lors de chaque préemption (voir 4.2.1 de la partie précédente). Implanter ce fonctionnement au-dessus de ces outils revient à court-circuiter leur mécanisme de gestion du calendrier des événements, ce qui enlève une grosse partie de leur intérêt, tout en alourdissant et en ralentissant la phase de simulation. C'est la raison principale pour laquelle nous nous sommes tournés vers les outils plus spécifiques à la communauté du temps-réel, car ils supportent en général cette notion de ressource active préemptible.

D'autre part, nous souhaitons pouvoir restituer les traitements effectués par une implantation réelle. Ce souhait écarte par définition les simulateurs par tâches creuses, et correspond par contre à la catégorie des simulateurs par tâches pleines. Cependant, les outils de cette catégorie évoqués en 4.2.2.2 de la première partie, ne permettent

pourtant pas de répondre simultanément à tous les souhaits qu'on a exprimés plus haut. Soit les outils ne sont pas extensibles au niveau du modèle de tâches ou de l'ordonnanceur, soit l'intégration de la simulation d'occupation du processeur dans les traitements n'est pas possible, soit les durées simulées sont forcément multiples d'un quantum de temps donné, soit les coûts d'exécution du système d'exploitation (y compris ceux de l'ordonnanceur) ne sont pas pris en compte, ou alors ils ne peuvent être pris en compte que sous la forme d'une variable probabiliste. De plus, la grande majorité de ces outils ne permet pas le partage de code entre implantation et modèle simulé, ce qui accroît le risque de non conformité entre le modèle simulé et une implantation concrète, réduisant ainsi le degré de confiance qu'on peut accorder aux résultats de l'évaluation.

Enfin, les outils de simulation de systèmes complets évoqués en 4.2.3 de la première partie ne répondent pas non plus à tous les souhaits émis plus haut : en particulier le système d'exploitation qu'ils simulent (interface de programmation et ordonnanceur en particulier) est rarement, voire pas du tout personnalisable, et/ou les considérations de fidélité du comportement temporel simulé sont négligées.

Toutes ces raisons font que nous nous sommes attachés à réaliser un simulateur qui permette de répondre simultanément à tous les souhaits qui ont été émis. Nous présentons cette plate-forme de simulation, ARTISST, dans les chapitres suivants.

Chapitre 6

Aperçu d'ARTISST

La plate-forme de simulation ARTISST, pour "ARTISST is a **R**eal-**Ti**me **S**ystem **S**imulation **T**ool", est disponible sous licence libre `LGPL` à l'adresse `http://www.irisa.fr/aces/software/artisst`, et est accompagnée du tutoriel [Dec02] en anglais. Un dépôt à l'Agence pour la Protection des Programmes est effectué.

Elle se présente sous la forme d'une bibliothèque de fonctions `C/C++` pour Unix (Solaris et Linux en particulier) : elle est écrite en `C`, et il existe des objets `C++` qui encapsulent ses fonctionnalités. Il est ainsi possible de l'utiliser pour simuler un système écrit en `C`, en `C++`, ou en tout autre langage pour lequel une passerelle vers la bibliothèque ARTISST serait développée. Elle a été conçue en abordant une méthodologie objet, ce qui la rend extensible et facilement personnalisable pour de nombreux aspects détaillés dans la suite du document.

Dans ce chapitre, nous donnons un aperçu des caractéristiques principales de la plate-forme. Le chapitre suivant entrera dans les détails du fonctionnement et des fonctionnalités fournies.

6.1 Infrastructure de simulation

La plate-forme est une infrastructure de simulation ouverte plutôt qu'un outil fermé, qui repose sur la notion de modules remplaçables. Après avoir présenté l'aspect modulaire d'ARTISST, nous indiquons les différents modules disponibles par défaut, et qui peuvent servir de base pour la réalisation d'autres modules personnalisés.

6.1.1 Modularité

Le cœur d'ARTISST est un simulateur à événements discrets reposant sur la notion de *propagation* des *événements de simulation* entre *modules* ARTISST. Au niveau le plus haut, les modules produisent et consomment les *événements* de la simulation. Au niveau bas, les modules sont reliés entre eux par le *circuit de simulation*, qui achemine les événements de la simulation entre les modules. La plate-forme ARTISST définit les notions d'événements et de modules, et fournit les mécanismes pour la construction du circuit de simulation et la gestion des événements dans le circuit.

Un *événement* répond à la définition habituelle dans le domaine de la simulation à événements discrets ; il permet d'indiquer par exemple le changement de valeur d'un détecteur, ou le commencement d'une tâche, ou une préemption, ou l'envoi d'un message sur un réseau simulé, ou la commande d'un actionneur. Dans le cas d'Artisst, une palette d'événements pertinents pour l'exécution de systèmes temps-réel est définie, et l'utilisateur peut en ajouter d'autres s'il le souhaite.

Un *module* est un composant logiciel chargé de produire et/ou de consommer des événements ; il peut s'agir par exemple d'un générateur de séquences aléatoires en charge de modéliser les stimuli de l'environnement, ou d'un générateur d'événements à partir d'un fichier de traces, ou d'un outil d'analyse des événements générés, ou d'un outil de simulation d'un système temps-réel, ou d'un outil de simulation d'un réseau. Artisst fournit un ensemble de modules prédéfinis et extensibles, et l'utilisateur peut en définir de nouveaux.

Nous revenons sur la définition précise de ces notions dans le chapitre suivant.

Cette modularité autorise le *branchement* de nouveaux modules sans modification de code autre que la mise à jour du circuit de simulation. Par exemple, on peut rajouter un module de génération de stimuli de l'environnement simulé sans modifier le module responsable de la simulation d'un système temps-réel : il suffit de le *brancher* en entrée du module de simulation du système temps-réel. Cette approche permet également de remplacer un module par un autre de façon transparente pour le reste de la simulation. Par exemple, il est possible de remplacer un module d'analyse statistique simple (calcul de moyenne par exemple), par un module d'affichage des traces de simulation sous forme graphique (chronogramme).

La figure 6.1 présente un exemple de circuit de simulation formé de 6 modules. Il permet de simuler un système temps-réel centralisé soumis à deux flux de stimuli aléatoires provenant de l'environnement simulé, et d'observer son comportement temporel sous la forme de traces et d'un chronogramme. Nous détaillons ci-dessous chacun des constituants.

6.1.2 Modules fournis

Par défaut, Artisst fournit les modules suivants :

- Module de génération de séquences aléatoires d'événements. 20 lois de distribution sont disponibles, et il est possible de fournir sa propre distribution sous la forme d'une fonction de répartition.
- Module de génération d'événements à partir d'un fichier de traces de format prédéfini.
- Module de simulation de système temps-réel, qui constitue le cœur de l'infrastructure, et qui est décrit dans la suite.
- Module de simulation de réseau, et module de simulation de carte réseau.
- Module de diffusion des événements (joue le rôle équivalent à une prise multiple en électricité).
- Module d'enregistrement des traces de simulation dans un fichier texte.

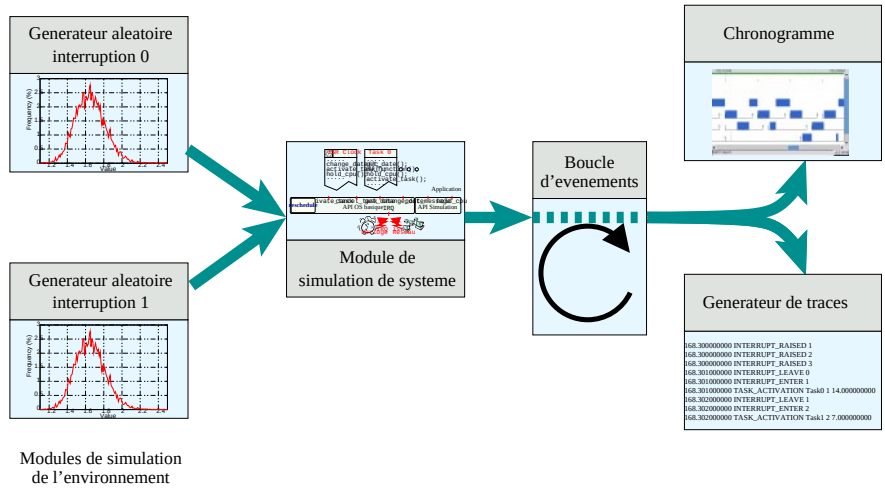


FIG. 6.1: Exemple de circuit de simulation

– Module de représentation graphique des traces (chronogramme).

Ces modules ne sont pas tous du même niveau de service : les 2 premiers s'occupent de la simulation de l'environnement, les 2 suivants servent à la simulation des systèmes observés proprement dite, les 2 derniers servent à l'exploitation des observations.

6.1.3 Extensibilité

Les modules ARTISST sont des classes qui héritent de la même classe abstraite. Les modules fournis par défaut ne dérogent pas à cette règle. Implanter un nouveau module revient soit à hériter d'un module existant, soit à définir son propre module par héritage de la classe abstraite.

6.2 Module de simulation de système

Le module fondamental dans ARTISST est le module en charge de simuler le système temps-réel à évaluer. Il peut y avoir plusieurs tels modules dans un même circuit de simulation, par exemple pour simuler un système distribué autour d'un réseau.

6.2.1 Présentation

Le rôle du module de simulation de système temps-réel est de servir d'interface entre l'infrastructure de simulation à événements discrets (le circuit de simulation), et le système temps-réel simulé. En entrée du module se trouvent les modules responsables de la simulation de l'environnement (simulation de capteurs ou de détecteurs) : les événements qu'ils génèrent sont assimilés à des interruptions matérielles. Et en sortie, le module génère (i) les événements correspondants à l'instrumentation du système

simulé et (ii) les événements à destination de l'environnement simulé (commande d'actionneurs, envoi de messages réseau par exemple). La figure 6.2 illustre l'interaction de ce module avec le circuit de simulation.

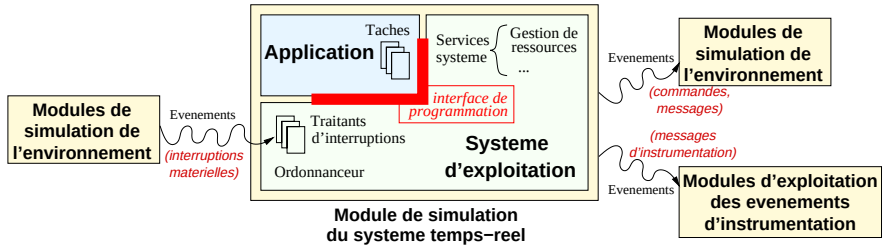


FIG. 6.2: Modèle de système simulé et interaction avec le circuit de simulation

Le module de simulation de système joue le rôle d'un processeur virtuel : il permet l'exécution du système temps-réel en réponse aux interruptions matérielles provenant de l'environnement simulé. Il propose en plus l'infrastructure d'instrumentation sous la forme de la génération d'événements de la simulation. Dans ce module, le modèle de système est défini conformément au modèle du début de la première partie : le système est constitué d'un système d'exploitation et d'une application, elle-même constituée de tâches.

6.2.2 Tâches de l'application

Un travail d'une tâche de l'application peut être activé par un traitant d'interruption ou par une autre tâche. Le point d'entrée de chaque travail est une fonction. L'exécution d'un travail correspond au déroulement de cette fonction, qui peut faire appel aux fonctions de la bibliothèque ARTISST par exemple (les appels système au système d'exploitation simulé en font partie), ou de toute autre bibliothèque de la machine hôte de la simulation. De cette façon, il est possible de reprendre le code d'une application existante.

Chaque travail est associé à un *modèle de tâche* qui sert à l'ordonnanceur pour prendre ses décisions d'ordonnancement. Il prend la forme d'une structure qu'il est possible d'étendre par héritage. De la sorte, le cœur du simulateur ne contraint pas la palette de modèles de tâches qu'il est possible de simuler. Pour plus de commodité, un modèle de tâches par défaut propose les attributs les plus courants dans la littérature en ordonnancement temps-réel (loi d'activation, WCET, entre autres).

6.2.3 Système d'exploitation simulé et ordonnanceur

Le système d'exploitation correspond à une classe abstraite. Les méthodes de cette classe sont des fonctions de rappel automatiquement appelées par le module de simulation (par exemple lors de l'arrivée d'une interruption), ou par les tâches (appel système, fin de tâche), y compris la fonction de rappel d'ordonnancement. Ces fonctions de rappel implantent :

- la gestion des interruptions matérielles (`isr()`)
- l'ordonnanceur (`reschedule()`)
- les services système fondamentaux (création, suppression, fin de tâche)
- les services système définis par l'utilisateur

Implanter un système d'exploitation revient à instancier cette classe abstraite. Avec cette approche, il est aisé d'ajouter de nouveaux services système, tels que des protocoles de gestion de ressources par exemple. Il est également possible de simuler l'interface de programmation d'un autre système d'exploitation existant, en implantant la couche de compatibilité nécessaire.

Puisque l'ordonnanceur constitue une simple méthode de l'objet système d'exploitation, il est possible de définir un nouvel ordonnanceur, ou d'en réutiliser un parmi la palette de ceux disponibles par défaut (détaillés au chapitre 8). Pour ses décisions, l'ordonnanceur utilise la liste des travaux des tâches de l'application présents dans le système simulé et maintenue à jour par le module de simulation, avec le modèle de tâche associé, et repose sur une échelle de temps discrète gérée localement (en général associée à un générateur d'interruption, l'*horloge système*) : l'échelle de temps système (nous reviendrons sur cette notion en 7.2.4.3).

6.2.4 Gestion des interruptions matérielles

Les interruptions matérielles sont prises en charge par le système d'exploitation simulé *via* la fonction de rappel de traitement des interruptions, ou *traitant d'interruption* (`isr()`, pour *Interrupt Service Routine* en anglais). Elles peuvent être masquées temporairement ou ignorées définitivement, et sont interruptibles par les interruptions de plus haute priorité (*i.e.* l'interruption de *niveau* i est autorisée à préempter l'exécution des traitants des interruptions de niveaux $j > i$).

6.2.5 Coûts d'exécution pris en compte

Que ce soit au niveau des services du système d'exploitation simulé (ceci incluant les traitants d'interruption, l'ordonnanceur, et les services système), ou des tâches de l'application, il est possible de simuler l'occupation du processeur pendant un intervalle de temps donné, et en tenant compte des préemptions (voir la partie précédente, section 4.2.1) : il suffit d'appeler la primitive ARTISST `hold_cpu(durée)` dans le code de ces entités. En paramètre, toute expression valide du langage `C` ou `C++` est possible.

6.2.6 Instrumentation du système simulé

Le module de simulation, en plus de s'occuper d'appeler automatiquement les fonctions de rappel du système d'exploitation simulé, et de gérer les événements conformément aux appels à la primitive `hold_cpu()`, se charge de générer automatiquement les événements de simulation qui correspondent à l'instrumentation du système simulé.

Les événements d'instrumentation générés reflètent la prise en charge des interruptions (date d'occurrence, interruption acceptée, refusée, préemptée), la gestion des

tâches (création, suppression, fin, préemption), les appels à l'ordonnanceur (date d'appel, date de fin) et ses décisions, et les appels système (date d'appel, date de fin, identification de l'appel, paramètres).

6.3 Modules de simulation de réseau

La simulation d'un système distribué avec Artisst consiste en l'échange de messages réseau simulés qui prennent la forme d'événements de simulation, et repose sur 3 types de modules (voir figure 6.3) : les modules de simulation des systèmes temps-réel, un module unique d'acheminement des messages réseau (`NET` pour *Network*), et un module d'interconnexion au réseau (coupleur) par système temps-réel simulé (`NIC` pour *Network Interface Controller*).

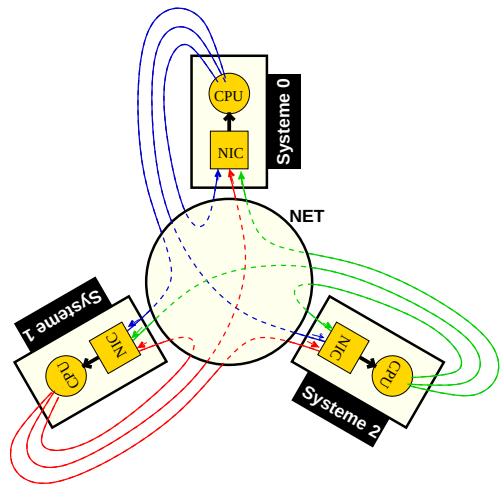


Fig. 6.3: Configuration réseau dans une simulation Artisst

Le module d'acheminement `NET` des messages s'occupe de *diffuser* les messages réseau de/à tous les modules d'interconnexion `NIC` qui sont *branchés* sur le réseau. Il n'est pas personnalisable et correspond à une diffusion instantanée totalement connectée sans perte : toute la simulation du réseau est reléguée aux modules d'interconnexion. Ces modules `NIC` sont personnalisables ; ce sont eux qui sont responsables de la simulation du/des réseau(x) proprement dit(s), à savoir : les retards (variables), les pertes et les altérations de messages sur les trajets `système simulé` → `réseau d'acheminement` et `réseau d'acheminement` → `système simulé`. La réception de messages réseau par les modules de simulation de système temps-réel peut se faire suivant deux modes complémentaires : par signalisation (*i.e.* une interruption simulée est levée dès qu'un message réseau est disponible) et par attente active (*polling*).

Enfin, de par la décorrélation entre l'échelle de temps système locale à chaque nœud, et le temps-réel global à l'ensemble de simulation, il est possible de simuler la désynchronisation des horloges locales des nœuds.

Chapitre 7

Description détaillée d'Artisst

Dans ce chapitre, nous décrivons plus en détails la structure et le fonctionnement de la plate-forme Artisst.

7.1 Infrastructure de simulation

Dans les outils de simulation à événements discrets habituels, le principe est de gérer la programmation des événements dans un calendrier d'événements global.

Dans la plate-forme Artisst, le principe central est celui de la *propagation* de *flux d'événements* de simulation, entre un ou plusieurs *module(s) producteur(s)* et un ou plusieurs *module(s) consommateur(s)*. Les connexions entre modules sont établies sous la forme d'un *circuit de simulation* par le concepteur. L'*infrastructure de simulation* :

- définit les types de données et les interfaces fondamentales utiles pour le déroulement de la simulation : événements de la simulation (7.1.1), modules (7.1.2) ;
- implante les fonctions fondamentales pour créer le circuit de simulation ;
- implante les fonctions fondamentales pour assurer la propagation des événements de la simulation le long du circuit de simulation (7.1.3), en définissant la notion de temps-réel simulé, et en assurant que cette échelle de temps est définie globalement à l'ensemble du circuit de simulation (7.1.4).

Cette infrastructure est la seule partie d'Artisst qui n'est pas extensible ou personnalisable. La souplesse de personnalisation globale d'Artisst est obtenue grâce à la modularité constitutive de cette structure, et à la souplesse de personnalisation des modules et du circuit de simulation.

7.1.1 Les *messages* de la simulation

Les événements de simulation dans Artisst sont des structures de données qui rassemblent une estampille, un type d'événement, et les informations associées au type d'événement. Puisque les événements de simulation sont accompagnés d'informations, ils sont appelés *messages*. Sous forme de traces, ils ont l'allure suivante :

Date	Type d'événement	Information complémentaire (dépend du type d'événement)
<date>	INTERRUPT_RAISED	<IRQ_LEVEL>
<date>	SUSPEND_TASK_EXECUTION	<TASK_ID>

7.1.1.1 Estampilles

L'*estampille* des messages permet de définir l'*échelle de temps-réel simulé* : la valeur de cette estampille correspond à une date dans cette échelle. C'est à chaque module de définir les estampilles sur les messages qu'il produit. Nous verrons en section 7.1.4 ci-après comment l'infrastructure de simulation garantit que l'échelle de temps-réel simulé ainsi définie correspond à un ordre total global (*i.e.* sur l'ensemble du circuit de simulation).

En pratique, l'estampille est une donnée numérique représentée en virgule fixe sur 64 bits afin d'éviter les erreurs d'arrondis ; sa capacité correspond à l'intervalle $[-9.2 * 10^9,\ 9.2 * 10^9]$[1], et sa résolution à 10^{-9}. Dans la plate-forme Artisst, la partie entière est appelée "seconde", et la partie fractionnaire "nano-seconde", bien que cette dénomination soit arbitraire ; dans ce cas, l'amplitude de l'échelle est de ±292 ans.

7.1.1.2 Types de messages et informations associées

Par défaut, les *types* des messages de simulation se divisent en 6 grandes catégories (la section 1.2 de l'annexe A les présente tous). Ils sont pratiquement tous dévoués au module de simulation de système temps-réel :

Gestion des interruptions. Ces messages indiquent l'occurrence d'une interruption matérielle simulée, et sa prise en charge (qui peut être retardée par le jeu des priorités d'interruptions et des désactivations temporaires) ou sa non prise en charge par le système simulé.
L'information associée à ces messages est le *niveau d'interruption* concerné.

Gestion des tâches. Ces messages indiquent l'activation ou la suppression d'une tâche, ainsi que son début et fin d'exécution, ses préemptions par d'autres tâches, ou ses interruptions, ainsi que ses reprises après préemption ou interruption.
L'information associée à ces messages est l'identifiant de la tâche concernée.

Passage en mode noyau. Lors des appels système, les dates de passage en mode noyau (*i.e.* appel d'un service système par l'application), et retour en mode utilisateur (*i.e.* retour à l'application après la fin de l'appel à un service système), sont restituées.
L'information associée à ces messages est le service noyau appelé.

Exécution de l'ordonnanceur. Ces messages encadrent l'exécution de l'ordonnanceur (début, fin).
L'information associée à la fin de l'exécution de l'ordonnanceur indique l'identifiant de la tâche élue.

[1] Plus précisément : $[-2^{63} + 2 * 10^{15}, 2^{63} - 1]$ nano-secondes. La marge $2 * 10^{15}$ sert à détecter les opérations sur des dates non définies, qui sont positionnées par défaut à la valeur $-2^{63} + 10^{15}$.

Échange de messages réseau. L'information associée à un message de ce type est le contenu du message réseau.

Messages personnalisables. Il est possible de définir ses propres messages en supplément aux messages indiqués précédemment afin d'étendre la palette des messages possibles.
L'information associée à ce type de message est un sous-type de message défini par l'utilisateur, ainsi que des données associées également définies par l'utilisateur.

En pratique, l'information associée aux messages est définie sous la forme d'une union (une donnée par type de message). Quand l'information associée référence une donnée externe à la structure de message, cette référence est gérée par un mécanisme de comptage de références afin de prendre en compte correctement les diffusions des messages en multiples exemplaires, conjointement avec les libérations de mémoire, afin d'éviter les problèmes de fuites de mémoire.

7.1.2 Modules

Les modules Artisst correspondent à une interface (structure de données) qui rassemble les prototypes de méthodes, qui sont les fonctions de rappel nécessaires à l'infrastructure de simulation. Ces fonctions de rappel servent :

- À la construction du circuit de simulation, pour avertir les modules qu'on les connecte à tel ou tel autre module. Ces méthodes sont présentes dans un but informatif.
- À la propagation des messages le long du circuit de simulation, détaillée en 7.1.3. Ce sont les méthodes les plus importantes ; on compte parmi elles les méthodes `push()`, `pull()` et `front()` qui sont évoquées dans la suite.

Implanter un nouveau module revient à implanter toutes les méthodes spécifiées dans l'interface, ou une partie seulement, suivant le *type* de module vu ci-après. Il n'est pas possible d'étendre l'interface des modules, car cette interface n'est utile qu'à l'infrastructure de simulation, qui est le cœur fondamental non extensible d'Artisst. Par contre, à chaque module il est possible d'associer une donnée propre, ce qui permet de les étendre, éventuellement en leur ajoutant une interface propre et séparée de l'interface "module".

Par exemple, un module de génération de séquence pseudo-aléatoire possède son interface d'appel propre (pour indiquer la loi de probabilité à restituer), et sa donnée propre renferme les caractéristiques de la variable pseudo-aléatoire utilisée dans la séquence (essentiellement la loi de probabilité modélisée). De même, le module de simulation de système temps-réel possède son interface d'appel propre, servant au système simulé pour définir les notions de système d'exploitation simulé, de tâches, ainsi que pour simuler l'occupation du temps processeur ; il possède ses données propres pour maintenir l'état du système simulé.

7.1.3 Circuit de simulation

Dans Artisst, le circuit de simulation n'est pas une donnée globale : il est défini de proche en proche, et correspond à une suite d'interconnexions d'un module sur un autre, qui lui-même est interconnecté à un autre module, et ainsi de suite (la section 1.4 de l'annexe A détaille les fonctions utilisées pour réaliser ces interconnexions). Nous donnons d'abord les propriétés des interconnexions entre modules, puis décrivons comment la simulation se déroule. La section qui suivra s'intéressera à présenter comment l'échelle de temps-réel simulé peut être définie globalement grâce aux propriétés locales des interconnexions et à des règles de construction du circuit.

7.1.3.1 Interconnexion de modules

Une *interconnexion* $A \rightarrow B$ est un lien orienté entre 2 modules A et B sur lequel transitent les messages de la simulation dans le sens de l'orientation de la connexion (*i.e.* de A vers B), et sans perte ni retard[2] ni altération : on dit que A est la *source* de l'interconnexion et qu'il *précède* B, que B est la *destination* de l'interconnexion et qu'il est un *successeur* de A, et que A *génère* les messages alors que B les *reçoit*.

On distingue d'autre part deux types d'interconnexions suivant l'*initiateur* de la propagation du message sur l'interconnexion :

- Si l'initiateur est le module source de la connexion (A), alors on parle de *soumission* des messages par le module source au module destination. La propagation d'un message de A vers B correspond à ce que A appelle la méthode `push()` de B, avec le message transmis en paramètre.
- Si l'initiateur est le module destination de la connexion (B), on parle de *consommation* des messages du module source par le module destination. La propagation d'un message de A vers B correspond à ce que B appelle la méthode `pull()` de A afin de récupérer le message propagé.

Pendant toute la durée de la simulation, pour une connexion donnée entre 2 modules, la direction de la connexion ainsi que l'identité de l'initiateur de la propagation des messages sont connues et ne peuvent pas changer. Un même module peut être initiateur à la fois en mode consommation et en mode soumission (mais sur deux connexions différentes) ; inversement, un module peut n'être qu'une entité passive à laquelle on soumet des messages, ou de laquelle on consomme des messages. La restriction étant que, pour une interconnexion donnée, il y ait exactement un initiateur de la propagation.

Plusieurs modules peuvent être connectés à un module donné, en mode soumission et/ou en mode consommation.

Lorsqu'un module donné soumet des messages à plusieurs modules destination, l'infrastructure de simulation Artisst s'occupe de la *soumission* des messages vers tous les modules destination.

Lorsqu'un module donné consomme ses messages depuis plusieurs modules source, l'infrastructure de simulation s'occupe de faire consommer un seul message à la fois depuis l'un de ces modules : Artisst garantit que le message récupéré est *le plus*

[2] L'estampille sur les messages n'est pas modifiée.

proche (chronologiquement) de la date courante (nous détaillerons cette notion en 7.1.4) parmi tous les messages récupérables depuis les modules source connectés. En pratique, l'opération de consommation, que nous simplifions à "`pull`" dans la suite, se fait en deux étapes : une *consultation* des modules source pour sélectionner le module qui fournit le message le plus proche (la méthode `front()` des modules sert à cet effet), puis la *consommation* proprement dite sur le module sélectionné par appel à la méthode `pull()` (avec la contrainte que le message consommé doit être identique au message consulté). La section 1.3 de l'annexe A rentre plus en détails dans la définition pratique de la structure et des fonctions de rappel d'un module.

La figure 7.1 donne une illustration des configurations de circuit types, suivant que la propagation est exclusivement de type consommation (flèches courbes : 7.1(a)), exclusivement de type soumission (flèches droites : 7.1(b)), ou avec certains modules qui sont initiateurs à la fois en mode soumission et en mode consommation (7.1(c), les modules 1, 5 et 6 sont initiateurs de soumission et consommation). Nous verrons en 7.3 que ce dernier circuit est celui d'une simulation de système distribué (le module 1 est le réseau, 5 et 6 sont les 2 modules de simulation de système temps-réel, 2 et 3 sont leur interface réseau respective).

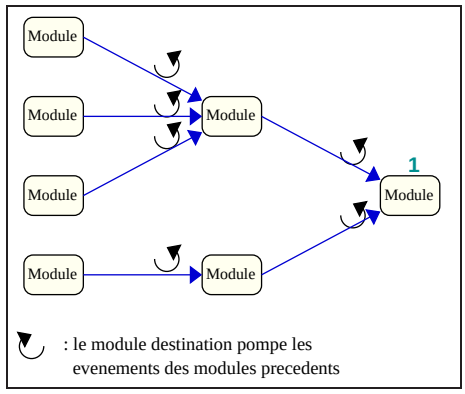
7.1.3.2 Déroulement de la simulation

Pour simuler le modèle, la première étape est de construire le circuit de simulation (sur lequel il sera possible de rajouter ou de supprimer des modules en cours de simulation), et de désigner un module particulier sur le circuit. La simulation consiste alors en une *boucle d'événements* qui, à chaque itération ou *pas de simulation*, consomme le *prochain* message de simulation (au sens chronologique, relativement à l'échelle de temps-réel simulé ; cette notion est définie en 7.1.4) de ce module. Le fait de consommer les messages de ce module entraîne le déclenchement du mécanisme de propagation des messages décrit plus haut. Les modules concernés par cette consommation sont dits "en amont" de la boucle d'événements. À chaque pas de simulation, la boucle d'événements soumet le message qu'elle a extrait aux modules dits "en aval".

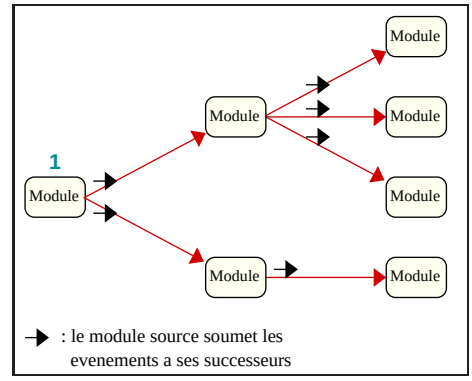
Pour illustrer, dans l'extrait de code suivant, le module qui est désigné pour servir la boucle d'événements s'appelle `module`, et chaque itération de la boucle `while` en extrait le *prochain* message, le stocke dans la variable `message`, et le soumet aux modules connectés en mode soumission :

```
/* Main message loop */
while (st_inout_pull_message(module, & message))
  {
    st_inout_push_message_on_all_outputs(module, & message);
  }
```

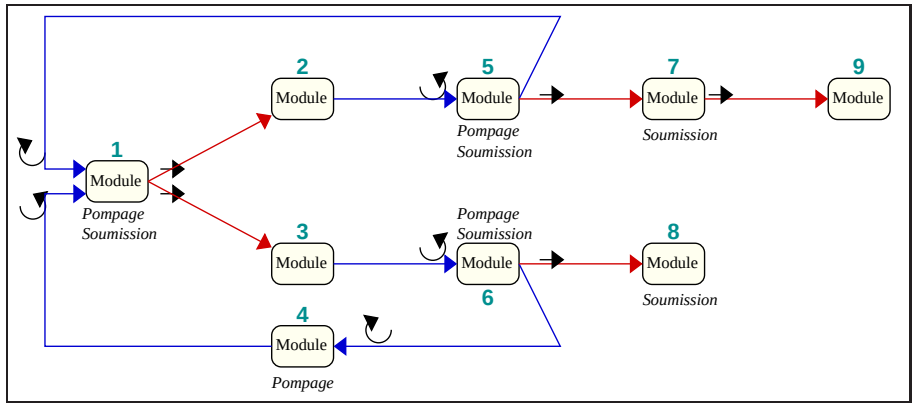
Dans la figure 7.1 précédente, la boucle d'événements s'appuie sur le module 1 de 7.1(a), 7.1(b), et 7.1(c).



(a) Consommation seule



(b) Soumission seule



(c) Plusieurs modes de propagation

Fig. 7.1: Schémas de propagation des messages

7.1.3.3 Propagation des messages et échelle de temps logique

La *propagation des messages* le long du circuit de simulation correspond à la chaîne d'appels aux méthodes `pull()` et `push()` des modules connectés les uns aux autres : un module invoque la méthode `pull()` ou `push()` d'un autre module, qui elle-même invoque `pull()` et/ou `push()` sur d'autres modules, etc... Pour générer les messages qu'elles produisent, la méthode `pull()` d'un module peut (choix inclusif) :

- appeler la méthode `pull()` des modules qui le précèdent. Si le message récupéré correspond au message généré par le module (*i.e.* le message que la méthode `pull()` retourne), il peut être altéré, retardé mais jamais avancé (*i.e.* son estampille ne peut pas être décrémentée) ;
- générer ses propres messages ;
- appeler la méthode `push()` des modules qui lui succèdent, avec un message identique au message généré, ou différent.

Les méthodes `push()` et `front()`, quant à elles, ne peuvent appeler que les méthodes `push()` et `front()` sur les modules qui succèdent/précèdent respectivement.

Ce principe de propagation est par essence totalement séquentiel (c'est une suite d'imbrications d'appels de fonctions), ce qui définit une *échelle de temps logique* globale (*i.e.* à l'échelle de tout le circuit de simulation) implicite, sous la forme d'une relation d'ordre total entre les messages générés. Toute la difficulté de la simulation est de faire en sorte que les estampilles sur les messages générés soient en ordre cohérent avec cet ordre logique, afin de définir la notion d'échelle de temps-réel simulé globale à l'ensemble de la simulation (voir en 7.1.4 ci-dessous).

7.1.4 Échelle de temps-réel simulé globale

Le rôle des estampilles des messages est de définir le temps-réel simulé. Mais dans Artisst, chaque module définit ou redéfinit lui-même les estampilles sur les messages qu'il génère ou qu'il relaye (on parle de *création* d'estampille dans la suite), c'est-à-dire que chaque module impose sa propre vision du temps-réel simulé. Par conséquent, pour définir une échelle de temps-réel simulé globale (*i.e.* sur l'ensemble du circuit de simulation), la propriété centrale que doit vérifier une simulation est que les visions locales du temps-réel soient cohérentes. Pour cela, il suffit d'assurer que l'ordre défini par les estampilles des messages générés soit cohérent avec l'ordre logique (voir 7.1.3.3) de génération des messages par les modules le long du circuit de simulation : un message m_1 d'estampille t_1 (relativement à l'échelle de temps-réel simulé) n'est *jamais* généré *après* (relativement à l'échelle de temps logique) un message m_2 d'estampille $t_2 \geq t_1$, même si m_1 et m_2 ne transitent pas sur la même interconnexion. Dans la suite, on appelle cette propriété "la *propriété d'accord*" entre les deux ordres (temps-réel simulé, et ordre logique).

Dans le cas général, aucune bonne propriété du principe de propagation ne peut garantir que le circuit de simulation ne provoquera pas la violation de la propriété d'accord. C'est pourquoi Artisst repose sur une série de trois règles qui contraignent la nature des modules, leurs propriétés, et la topologie du circuit de simulation. Le

principe est que la boucle d'événements canalise (directement ou indirectement) **toutes** les estampilles transitant (créées ou relayées) dans le système, afin d'assurer qu'elles sont (non strictement) croissantes, de sorte que la propriété précédente d'accord avec l'ordre logique soit vérifiée. Ce principe fait intervenir les estampilles, et non les messages associés qui peuvent être quelconques, et régit en particulier la *création* de nouvelles estampilles. Les règles sont les suivantes :

1. **Progrès local des estampilles.** Si un module génère ou reçoit un message m_1 estampillé avec la date t_1 (temps-réel simulé), l'infrastructure de simulation d'Artisst *vérifie* que tout message m_2 qu'il génère ou qu'il reçoit par la suite (ordre logique) est tel que son estampille $t_2 \geq t_1$. Quand la vérification échoue, c'est toute la simulation qui est abandonnée (cas d'erreur fatale).
2. **Toutes les connexions de type *consommation* sont en amont de la boucle d'événements.** Pour toute connexion en mode consommation, il existe une chaîne de connexions en mode consommation partant de la connexion considérée, et menant au module servant à la boucle d'événements (en respectant l'orientation des connexions, qui ne change pas au cours de la simulation, et qui est définie au moment de la construction du circuit). L'infrastructure de simulation d'Artisst n'effectue aucune vérification de cet aspect.
3. **Seuls les modules source utilisés en mode consommation peuvent créer ou modifier des estampilles.** Ces mêmes modules sont aussi autorisés à propager, en mode soumission, tout ou partie des messages qui sont consommés, et seulement ceux-là, à d'autre modules destination. L'infrastructure de simulation d'Artisst n'effectue aucune vérification de ces aspects.

La règle 1 garantit que, *localement à chaque module*, la vision locale du temps-réel simulé est cohérente avec l'ordre logique. Il en découle en particulier que si un module retransmet un message qu'il a reçu, alors il ne peut modifier l'estampille du message qu'en l'incrémentant.

La règle 2 découle de la définition de la propagation de flux d'événements : une *consommation* ne peut pas avoir lieu spontanément ; il faut qu'un autre module qui est connecté au module source initie la consommation, et ainsi de suite jusqu'à l'origine de la consommation, qui n'est autre que le module associé à la boucle d'événements.

La règle 3, combinée à la précédente, impose que les seuls modules qui ont le droit de créer de nouvelles estampilles, ou de les modifier (*i.e.* en les incrémentant, en vertu de la règle 1), sont ceux qui se trouvent en amont de la boucle d'événements : ainsi la boucle d'événements centralise les visions locales du temps-réel simulé, par l'intermédiaire des estampilles qui transitent. En corollaire, ces règles imposent que (*i*) aucun module *strictement en aval* (*i.e.* en aval, et sans connexion qui ramène à la boucle d'événements) de la boucle d'événements ne peut définir ses propres estampilles, et (*ii*) aucun module utilisé en tant que module source en mode soumission ne peut modifier les estampilles des messages qu'il propage si elles ne sont pas également consommées par la boucle d'événements.

Par exemple, le circuit de simulation de la figure 7.2 correspond à la simulation de systèmes temps-réel centralisés, et respecte ces trois règles à partir du moment

où les modules `chronogramme`, `enregistreur de traces` et `analyse statistique` ne créent aucune estampille nouvelle (ce qui est le cas en pratique puisque ces modules ne produisent aucun message). Nous verrons en 7.3 le circuit type pour la simulation de systèmes distribués.

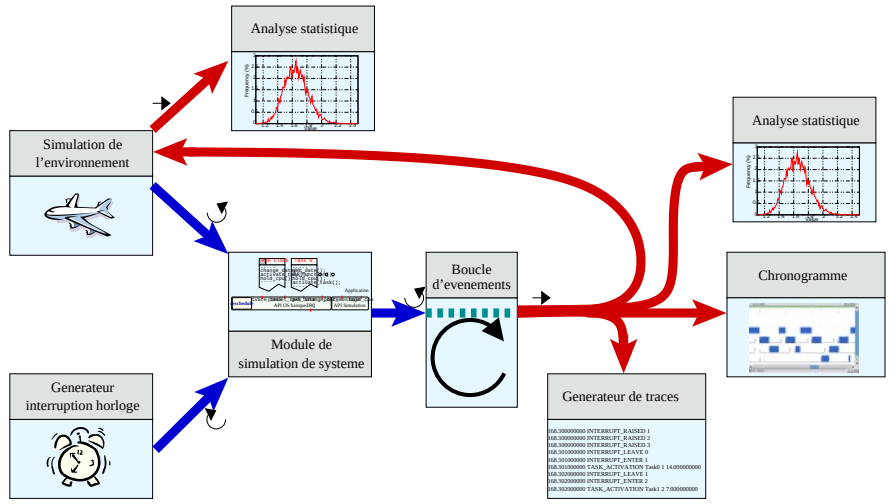


FIG. 7.2: Exemple de circuit de simulation

7.2 Module de simulation de système

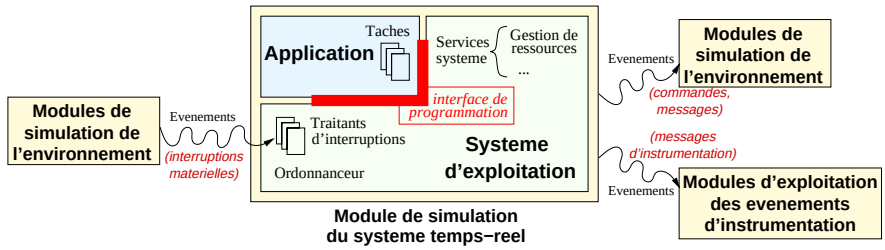
Une simulation avec ARTISST a pour vocation d'évaluer le comportement d'un système temps-réel. Dans ce but, le module central que propose la plate-forme ARTISST est le module de simulation de système temps-réel, appelé `rtsys`, que nous présentons ici.

Dans un premier temps, nous donnons quelques détails sur ce module, sa structure, son interaction avec l'infrastructure de simulation et l'environnement simulé en faisant abstraction des détails techniques sous-jacents (la section 1.5 de l'annexe A fait une synthèse de ces aspects). Nous présentons ensuite les caractéristiques de l'application et du système d'exploitation simulés.

Tout au long de ces présentations, nous indiquons en quoi et comment les différentes entités peuvent être personnalisées et étendues. Nous nous intéressons ici en priorité à définir l'implantation de ces différentes entités, plutôt qu'à indiquer comment les utiliser : ceci fait en revanche l'objet du tutoriel [Dec02].

7.2.1 Caractéristiques du module

Le module de simulation de système temps-réel joue à la fois le rôle d'un module ARTISST dans le circuit de simulation, et le rôle de support d'exécution pour le système temps-réel simulé. La figure 6.2 précédente et reprise ci-dessous illustre ce double rôle :



Le module se présente sous la forme d'une interface de module ARTISST munie de la fonction de rappel `pull()`, afin d'être utilisé en amont de la boucle d'événements, en mode consommation. La méthode `pull()` a pour rôle de faire avancer la simulation du système temps-réel au gré de la simulation d'occupation du temps processeur (primitive `hold_cpu()` décrite plus loin) et des stimuli de l'extérieur (interruptions matérielles). Par ailleurs, cette méthode `pull()` soumet chaque message généré à tous les modules qui sont connectés en mode soumission. Ceci autorise à connecter directement au simulateur de système temps-réel des modules d'analyse des messages produits, sans passer par le module servant de boucle d'événements.

Le module possède une deuxième interface : l'interface de programmation proposée au système simulé, qui lui confère son rôle de support d'exécution. Ce sont les fonctions de cette interface que l'application simulée va utiliser pour simuler l'occupation du temps processeur simulé, ou pour faire appel aux services du système d'exploitation simulé. Il est possible d'ajouter de nouveaux services système afin d'enrichir le système d'exploitation simulé, ou afin de reproduire l'interface de programmation d'un système d'exploitation existant. De la même manière, il est possible de personnaliser l'ordonnanceur sous-jacent au système d'exploitation simulé.

Dans ce qui suit, nous nous concentrons sur cet aspect "support d'exécution", l'aspect "module" au sens de l'infrastructure de simulation étant conforme à la description de la section 7.1 précédente.

7.2.2 Présentation de la structure interne

Le module est entièrement écrit en `C`, et une série de classes équivalentes en `C++` est fournie pour représenter les interfaces sous forme d'objets classiques. Il est découpé en deux couches, comme l'illustre la figure 7.3.

La couche basse fournit les services fondamentaux d'un processeur, en ce sens qu'elle définit et gère les *contextes d'exécution*. Dans ARTISST, un *contexte d'exécution* est un flot de contrôle qui occupe du temps (simulé) sur le processeur simulé, qui peut être interrompu (par interruption matérielle simulée), et qu'il est possible de changer en cours de fonctionnement (*changement de contexte*). Dans ARTISST, un contexte d'exécution propre est associé à chaque travail de chaque tâche, mais également au traitement de chaque interruption[3]. C'est cette couche basse qui définit le mécanisme de simulation d'occupation du temps processeur, et qui joue le rôle d'interface entre les contextes d'exécution et l'infrastructure de simulation.

[3] Ce qui n'est en général pas le cas sur les processeurs réels.

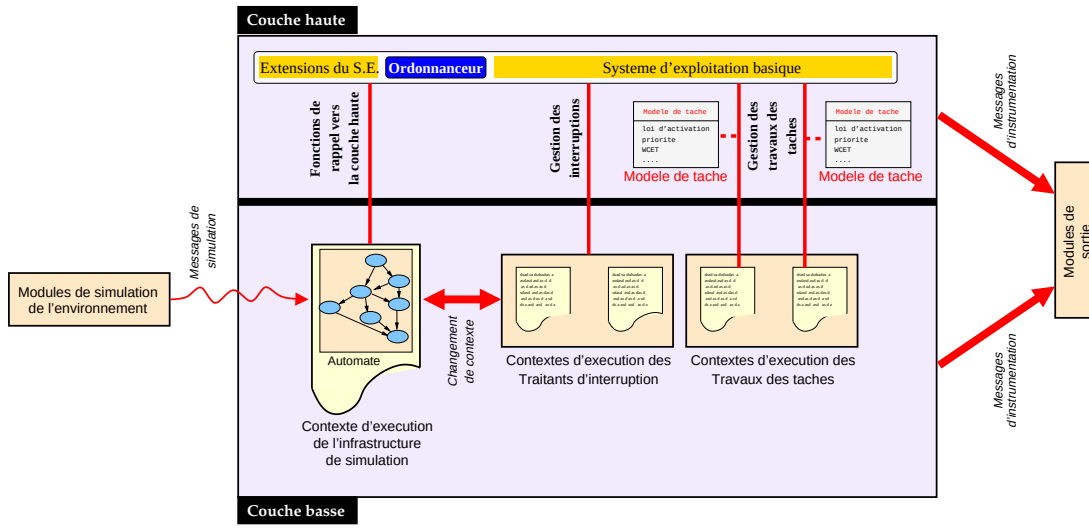


FIG. 7.3: Structure interne du module de simulation de système

La couche haute définit le système d'exploitation simulé, et joue le rôle d'interface entre l'application et les contextes d'exécution.

Chacune des deux couches génère les messages d'instrumentation à son niveau : messages de création/suppression/fin/changement de contexte pour la couche basse, messages de début/fin d'appel système/d'ordonnancement pour le couche haute.

Dans les sections qui suivent, nous présentons plus en détail chacune des deux couches du module.

7.2.3 Couche basse

La couche basse du support d'exécution associé au module de simulation de système correspond à :

- Un automate d'états finis associé à la fonction de rappel `pull()` du module (voir 1.5.4). Cet automate s'occupe de définir le *prochain* message à générer en fonction des stimuli provenant de l'environnement et de l'instrumentation des comportements des *contextes d'exécution* (*i.e.* tâches et traitants d'interruptions).
- Les mécanismes pour créer, exécuter, changer de contexte d'exécution.

Les éléments de la couche basse ne sont personnalisables qu'à l'aide de quelques directives (indiquées dans la suite) lors de la compilation de la bibliothèque ARTISST, qui permettent de modifier certaines de ses caractéristiques.

Lors de l'instanciation de la couche basse du module, une série de fonctions de rappel vers la couche haute doit être fournie. Ces fonctions de rappel correspondent en particulier à la méthode `isr()` de traitement des interruptions, `end_isr()` de signalisation de fin d'interruption, et `end_task()` de signalisation de fin de travail d'une tâche. Nous y faisons référence dans la suite.

Nous commençons par décrire les mécanismes élémentaires, et remontons jusqu'à leur interaction dans l'automate d'états finis.

7.2.3.1 Contextes d'exécution

Que ce soit pour les tâches ou pour les traitants d'interruptions, le principe est le même : toutes ces entités possèdent leur propre *contexte d'exécution*. Ces contextes d'exécution sont proches des *threads* : il s'agit de flots d'instructions indépendants qui sont exécutés sur la machine hôte de la simulation, et qui se partagent le même espace d'adressage (celui du processus du simulateur). À la différence des bibliothèques de *threads*, qui reposent sur un ordonnancement implicite visant à la parallélisation des exécutions tout en offrant des mécanismes de synchronisation, les contextes d'exécution utilisés dans Artisst s'apparentent à des coroutines ou à des continuations, et sont ordonnancés explicitement par le cœur du simulateur. Plusieurs implantations sont disponibles pour plus de portabilité, qui sont automatiquement sélectionnées lors de la compilation. Ces implantations reposent sur des mécanismes de plus bas niveau pour gérer les changements de contexte du processeur hôte de la simulation, tels que ceux des bibliothèques de *threads* (*threads POSIX*, ou les *GNU Portable threads* [46]), ou tels que proposés par des interfaces de programmation spécifiques (*Single Unix Specification v2*, ou le cœur de gestion des contextes de `GNU Pth`).

Les contextes d'exécution sont créés en indiquant l'adresse de la fonction qui constitue le traitement à effectuer. Il existe également un contexte d'exécution spécial qui ne correspond à aucune tâche de l'application, ni à aucun traitant d'interruption : c'est le *contexte d'exécution originel*, celui par lequel s'exécute l'infrastructure de simulation (propagation des messages, invocation de la méthode `pull()` du module de simulation de système temps-réel).

Pour chaque pas de simulation mené par l'invocation de la méthode `pull()` du module, la simulation consiste en un ou plusieurs aller-retour(s) entre le contexte d'exécution originel et le contexte d'exécution en cours dans la simulation (tâche ou traitant d'interruption). Ces aller-retours sont définis par l'automate d'états finis décrit plus loin.

7.2.3.2 Travaux des tâches

Au niveau de la couche basse du module, un travail d'une tâche de l'application correspond à un contexte d'exécution, auquel sont adjoints des mécanismes de détection de démarrage et de fin de tâche afin de générer les messages d'instrumentation, et afin de signaler l'événement "fin de travail de la tâche" à la couche haute (par l'intermédiaire de la fonction de rappel `end_task()`). Au niveau de la couche basse, seule la notion de *travail de tâche* est reconnue, ce qui entraîne que plusieurs travaux d'une même tâche peuvent être présents dans le système.

Pendant son exécution, un travail d'une tâche peut effectuer n'importe quelle opération : il s'agit de code compilé pour la machine hôte de la simulation. En particulier il peut appeler les fonctions de l'interface de programmation du système d'exploitation

simulé définie par la couche haute, ou les fonctions de la couche basse qui permettent de changer “manuellement” de contexte ou de générer “manuellement” des messages d'instrumentation (primitive `post_message()`).

En dehors d'une fonction spéciale (`hold_cpu()` décrite ci-après), toutes les opérations que le travail fait sont *instantanées* vis-à-vis du temps-réel simulé.

Pour simuler l'occupation du processeur pendant un intervalle de temps donné, ainsi que l'*interruptibilité*, le module définit la primitive `hold_cpu(delay)`. Cette primitive est en charge d'effectuer un changement de contexte vers le contexte d'exécution originel (celui de l'infrastructure de simulation). Le paramètre `delay` qui est fourni, est utilisé dans l'automate d'états finis décrit ci-dessous pour simuler l'occupation du processeur par le travail pendant l'intervalle de temps `delay` donné, relativement à l'échelle de temps-réel simulé. Ce paramètre figure le temps-réel simulé occupé par le travail sur le processeur, en tenant compte des interruptions et des préemptions : il correspond au temps que mettrait cet appel à s'exécuter (dans l'échelle de temps-réel simulé) si le travail était pris en isolation. Il peut être le résultat de n'importe quelle expression `C` valide (par exemple, une constante, ou un appel à une fonction de génération de séquence aléatoire).

7.2.3.3 Interruptions

Dans Artisst, les interruptions matérielles sont des messages de simulation. Ces messages sont les seuls stimuli de l'environnement simulé qui sont pris en compte par le module ; c'est la couche basse qui s'en charge. Chaque interruption possède un *niveau d'interruption* ; par défaut, Artisst supporte 32 niveaux, mais cette valeur est configurable (pas de limitation).

La prise en charge des interruptions est régie par deux mécanismes : le filtrage global et local, puis les règles de préemption. La figure 7.4 présente une série de scénarios pour illustrer ces mécanismes : l'interruption 0 est masquée globalement, et l'interruption 2 est prise en compte en différé ou immédiatement suivant le filtrage local du contexte d'exécution courant, et suivant les règles de préemption. Nous détaillons ci-dessous tous ces mécanismes.

Filtrage des interruptions

Le traitement d'une interruption est effectué dans un contexte d'exécution propre. Mais avant toute affectation de contexte d'exécution pour le traitant d'interruption, la prise en charge d'une interruption est soumise à deux phases de filtrage :

1. **Filtrage global** (ou *masquage des interruptions*) : l'interruption peut être définitivement ignorée. Ce filtrage fait intervenir le niveau l de l'interruption levée. Son verdict dépend de la valeur du $l^{\text{ème}}$ bit d'un champ de bits global à tout le module. Si l'interruption n'est pas ignorée à ce niveau, elle subit le filtrage local suivant. Le champ de bits est géré par les tâches ou les traitants d'interruption eux-mêmes. Lors de l'initialisation du module, toutes les interruptions sont masquées.

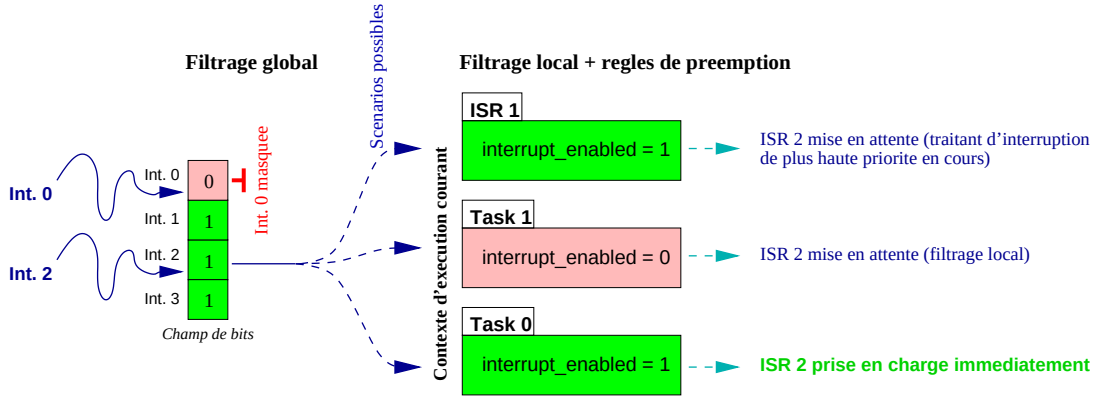


FIG. 7.4: Les mécanismes de prise en charge des interruptions

2. **Filtrage local** (ou *activation/désactivation locale des interruptions*) : chaque contexte d'exécution (tâche ou traitant d'interruption) définit son statut vis-à-vis des interruptions. Il peut soit tolérer les interruptions (tous niveaux confondus), soit en être temporairement protégé (tous niveaux confondus). Le contexte d'exécution peut changer ce statut à tout moment.
   Pendant que les interruptions sont localement désactivées (*i.e.* il y a *passivation*), une interruption qui est reçue par le module est *mise en attente* (au maximum une interruption en attente par niveau d'interruption). Par la suite, une interruption mise en attente peut être (*i*) soit annulée dans le cas où le système modifie le filtrage global pour masquer le niveau d'interruption considéré, (*ii*) soit réactivée lorsque les interruptions sont réactivées : (*ii.a*) soit parce que le contexte d'exécution courant effectue lui-même cette réactivation, (*ii.b*) soit parce qu'il y a changement de contexte vers un contexte d'exécution qui ne désactive pas localement les interruptions.

Règles de préemption par les traitants d'interruptions

Une fois les deux étapes de filtrage passées, un contexte d'exécution est affecté pour effectuer le traitement. Le traitement effectué est toujours le même : il s'agit de l'appel de la fonction de rappel `isr(level)` définie par la couche haute. Cette fonction prend en paramètre le niveau `level` de l'interruption prise en charge. Avant que le traitement de l'interruption ne débute, le traitant d'interruption obéit à deux règles de préemption :

Préemption des tâches. Le traitant d'interruption (de tout niveau) interrompt immédiatement le contexte d'exécution en cours si c'est le contexte d'exécution d'un travail de tâche : il y a *interruption* du travail de la tâche.

Préemptions inter-interruptions. Lorsque le contexte d'exécution en cours est celui d'un traitant d'interruption, il y a deux politiques de préemption. La politique adoptée est définie lors de la compilation du module.

Avec la politique par défaut, le niveau d'interruption correspond à un niveau de préemption : un traitant d'interruption de niveau i peut préempter immédiatement un traitant d'interruption de niveau $j > i$ (*i.e.* le niveau 0 correspond à la plus grande priorité) ; dans ce cas, il y a *imbrication des traitants d'interruption*. Dans le cas contraire, le traitant d'interruption i est mis en attente jusqu'à ce que tous les traitants d'interruption en cours de niveaux $j \leq i$ aient terminé leur traitement.
Avec la deuxième politique (dite `LIFO`), toute interruption peut préempter tout traitant d'interruption en cours. Dans ce cas, il y a imbrication systématique des traitants d'interruption.

Traitements dans les traitants d'interruption

Une fois que le traitant d'interruption a démarré, ses caractéristiques sont identiques à celles des tâches, si ce n'est que la fonction de rappel pour la signalisation à la couche haute de la fin de traitant d'interruption est `end_isr()`. Le fait en particulier que les traitants d'interruption puissent faire appel aux services du système d'exploitation (appels système) est relativement peu usuel en programmation système.

7.2.3.4 Automate

Le cœur de la couche basse du module de simulation de système temps-réel est l'automate qui régit les allers depuis l'infrastructure de simulation (contexte d'exécution originel) vers les contextes d'exécution. Son rôle est de prendre en compte les stimuli de l'environnement (interruptions), de gérer les dates de réveil des contextes d'exécution qui sont en cours de simulation d'occupation du temps processeur (*i.e.* qui ont appelé la primitive `hold_cpu()`), et de générer les messages d'instrumentation de la couche basse. Les retours depuis les contextes d'exécution vers l'automate sont provoqués par de simples appels à la primitive `hold_cpu()`.

Le fonctionnement basique de l'automate est présenté dans l'algorithme 7.1. La figure A.1 en 1.5 de l'annexe A donne le diagramme d'états complet : les étiquettes accolées aux transitions correspondent aux messages d'instrumentation qui sont générés, et non aux événements qui tirent ces transitions.

7.2.4 Couche haute

La couche haute du support d'exécution associé au module de simulation de système :

- définit la notion de modèle de tâche, et spécifie l'association travail de tâche – modèle de tâche ;
- définit l'interface du système d'exploitation proposé aux travaux des tâches et aux traitants d'interruption. Par défaut, le *système d'exploitation basique* ainsi défini est une classe abstraite que l'utilisateur est en charge de compléter (en particulier : il doit choisir un ordonnanceur parmi ceux disponibles, ou en implanter un autre) ;
- instrumente les invocations et les décisions du système d'exploitation.

• t_{in} = date du prochain événement provenant d'un module d'entrée; {*i.e.* invocation de la méthode `front()` des modules qui précèdent}

• t_{hold_cpu} = `date_courante` + `contexte_d_execution_courant.remaining_holding_time`; {date de terminaison du `hold_cpu()` du `contexte_d_execution_courant`}

{**Sélection du prochain contexte d'exécution**}

• `contexte_d_execution_suivant` := `contexte_d_execution_courant`; {Contexte du prochain pas de simulation par défaut}

SI ($t_{hold\_cpu} < t_{in}$) **ALORS**

{*Doit traiter la fin du* `hold_cpu()` *en cours d'abord*}

• Reprendre l'exécution du `contexte_d_execution_courant` jusqu'au prochain `hold_cpu()`;

{*exécution en cours ; retour ici*}

SI Fin du travail de tâche courant **ALORS**

• Signaler la fin du travail à la couche haute, qui définit le travail suivant (fonction de rappel `end_task()`);

• `contexte_d_execution_suivant` := travail ainsi défini;

SINON SI Fin de traitant d'interruption **ALORS**

• Signaler la fin de l'interruption à la couche haute, qui définit le travail suivant (fonction de rappel `end_isr()`);

SI Pas d'interruption en attente **ALORS**

• `contexte_d_execution_suivant` := travail ainsi défini;

SINON

{*Au moins une interruption en attente*}

• `contexte_d_execution_suivant` := interruption en attente; {Suivant priorités ou ordre LIFO : dépend de la configuration à la compilation}

FIN SI

FIN SI

• `date_courante` = t_{hold_cpu};

SINON

{*Un message d'un module d'entrée arrive en cours du* `hold_cpu()` *courant*}

m = prochain message du module d'entrée

SI m est de type `INTERRUPT_RAISED` **ALORS**

• Déterminer si l'interruption est ignorée, mise en attente, ou traitée immédiatement;

SI L'interruption doit être traitée immédiatement **ALORS**

• Diminuer `contexte_d_execution_courant.remaining_holding_time` de t_{in} - `date_courante`; {Enregistre le temps en `hold_cpu()` qu'il reste à simuler}

• `contexte_d_execution_suivant` := interruption à traiter;

FIN SI

FIN SI

• `date_courante` = t_{in};

FIN SI

{**Changement d'état jusqu'à la prochaine itération**}

• retourner le message si le contexte d'exécution courant en a généré un;

• `contexte_d_execution_courant` := `contexte_d_execution_suivant`; {Mise à jour du contexte d'exécution jusqu'au prochain pas de simulation}

ALG 7.1: Un pas de simulation du système temps-réel simulé (*i.e.* méthode `pull()`)

L'ensemble de ces composants forme la base extensible pour la réalisation de systèmes simulés.

7.2.4.1 Modèle du système

Au niveau de la couche haute, la vision globale qu'on a du système correspond au modèle objet donné en figure 7.5. Nous détaillons dans les sections qui suivent les caractéristiques des différents constituants de ce modèle.

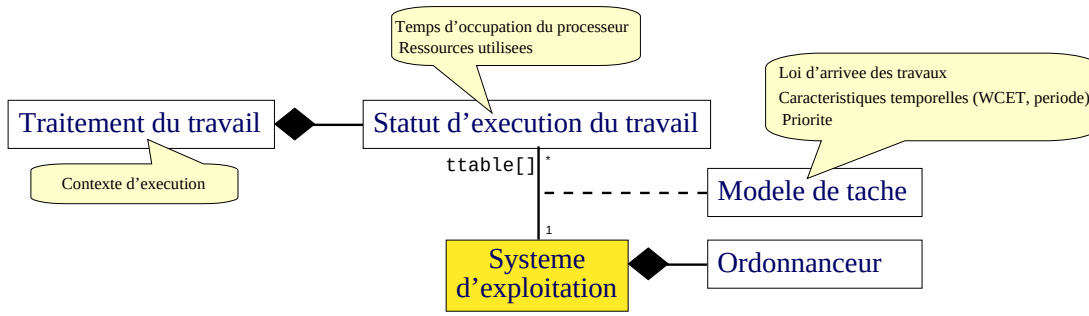


FIG. 7.5: Modèle objet du systeme simulé

7.2.4.2 Tâches de l'application

Conformément à la définition des tâches donnée dans la section 1.3 de la première partie, le module de simulation de système temps-réel représente une *tâche* comme étant un ensemble de *travaux* qui sont exécutés. Dans le module, la notion de tâche est scindée en trois :

- Le *modèle de tâche*.
- Le *statut d'exécution* de chaque travail en cours.
- Le contexte d'exécution de chaque travail (couche basse).

Le système d'exploitation simulé maintient la liste des travaux en cours dans le système et propose les mécanismes de création et de suppression des travaux des tâches, comme nous le verrons dans la section suivante.

Modèle de tâche

Le modèle de tâche est une structure de données qui renferme les caractéristiques de la tâche ; elle doit être impérativement définie avant l'activation de travaux de cette tâche. Cette structure est partagée par tous les travaux de la tâche, et est utilisée par le système d'exploitation (tout particulièrement par l'ordonnanceur) pour l'arbitrage des besoins et des demandes en ressources. Dans tous les cas, la simulation d'occupation effective des ressources, telles que le processeur, est liée aux traitements effectués par les travaux de la tâche, qui peuvent dépasser les hypothèses émises dans le modèle ;

ceci permet par exemple d'évaluer le comportement du système en cas de dépassement d'hypothèses.

Le module exporte une structure de données par défaut qui renferme :

- Le temps d'exécution pire cas, moyen, et au mieux.
- L'échéance relative à la date d'activation.
- La loi d'arrivée (périodique, sporadique, apériodique) avec les attributs associés (respectivement période et déphasage, délai d'inter-arrivée minimum).
- La priorité.

Ces informations correspondent aux attributs les plus couramment utilisés par les ordonnanceurs les plus simples proposés dans la littérature du domaine ; l'objectif étant d'épargner à l'utilisateur le travail qui consiste à souvent redéfinir les mêmes modèles de tâches.

Le système d'exploitation basique que nous décrivons dans la section suivante ne repose en aucune manière sur le modèle de tâche par défaut. En particulier, lors de la définition d'un modèle de tâche périodique, les travaux de la tâche ne sont pas automatiquement périodiquement activés par le système d'exploitation basique : c'est à l'utilisateur de gérer cette réactivation périodique, par exemple en la confiant au traitant d'une interruption dédiée périodiquement générée par un module externe.

Suivant l'ordonnanceur utilisé ou défini, l'utilisateur peut ne renseigner qu'une partie des informations de cette structure (le reste étant alors *indéfini*). Inversement, si pour un ordonnanceur, un protocole d'accès aux ressources, ou d'autres services système donnés, ce modèle de tâches s'avérait incomplet ou incorrect, l'utilisateur peut étendre le modèle par défaut (héritage de structure en `C`, héritage en `C++`), ou le remplacer. La personnalisation du modèle de tâches ainsi réalisée a pour but de satisfaire aux besoins de l'ordonnanceur et des autres services système que l'utilisateur doit choisir parmi ceux existants, ou implanter par ses soins.

Statut d'exécution

Le statut d'exécution d'un travail renferme les informations sur l'état du travail, et son comportement temporel effectif. C'est une structure de données qui est créée lorsque le travail est activé, et que le système d'exploitation simulé doit tenir à jour.

Le module vient avec un statut d'exécution par défaut qui renferme les informations suivantes sur l'exécution de chaque travail :

- L'état du travail de la tâche (prêt, en cours d'exécution, préempté, interrompu, bloqué).
- Les caractéristiques temporelles du travail, et les statistiques d'utilisation : date d'arrivée, date de début d'exécution, temps passé dans les services du système d'exploitation, temps passé plus précisément dans l'ordonnanceur, temps passé dans les traitants d'interruption. Indique également le nombre de préemptions et le nombre d'interruptions.

  Les temps et les durées sont donnés relativement aux échelles de temps-réel et de temps système (cette notion est définie dans la section suivante).

– La priorité courante, initialisée à celle définie dans le modèle de tâche. En cours d'exécution du travail, elle peut être égale à ou différente de celle indiquée dans le modèle de tâche (suite à l'action d'un protocole de gestion de ressources par exemple).

Les deux premières informations sont mises à jour automatiquement par le système d'exploitation basique. La troisième n'est pas modifiée par le système d'exploitation basique, mais peut l'être par les personnalisations et les extensions que l'utilisateur implante.

Cette structure de données est mise à disposition de l'utilisateur de ARTISST, tout particulièrement pour effectuer les décisions d'ordonnancement. Elle peut être étendue (héritage de structure en `C`, héritage en `C++`) par l'utilisateur afin par exemple d'indiquer les ressources qui sont utilisées par le travail. Ces personnalisations vont de pair avec le choix de l'ordonnanceur ou des services du système d'exploitation.

Contexte d'exécution

Il s'agit des traitements réels qui sont effectués par le travail. Cette notion est définie dans la couche basse, et a été présentée en 7.2.3.1.

7.2.4.3 Système d'exploitation

Le système d'exploitation simulé a une structure qui se présente comme illustré sur la figure 7.6. Son rôle est de :

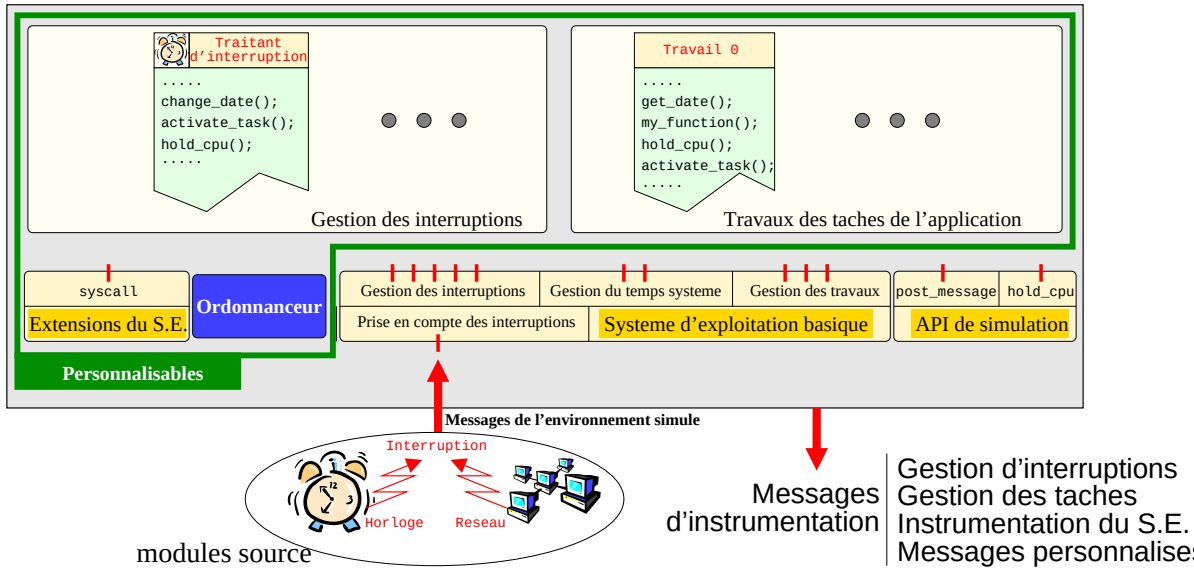


FIG. 7.6: Structure du système d'exploitation simulé

– Définir une interface de programmation unique visible depuis les tâches et les interruptions.

- Déclarer une série de fonctions de rappel que l'utilisateur doit définir, et qui sont invoquées automatiquement par les fonctions de l'interface de programmation, ou par le système d'exploitation lui-même. L'ordonnanceur fait partie de ces fonctions de rappel.
- Mettre à jour les statuts d'exécution des travaux des tâches.
- Générer les messages d'instrumentation du système d'exploitation, et fournir les fonctions de rappel nécessaires à la couche basse.

La plupart des composants du système d'exploitation simulé sont personnalisables et extensibles (la figure les met en valeur), comme nous le verrons dans la section 7.2.4.3 ci-dessous.

Fonctionnalités et structure du système d'exploitation simulé

C'est le système d'exploitation basique qui définit l'interface de programmation visible depuis les tâches et les traitants d'interruption. Elle permet d'accéder aux *services système* pour :

- Gérer les travaux des tâches,
- Gérer les interruptions,
- Gérer le temps système simulé,
- Invoquer les extensions personnalisées du système d'exploitation basique.

En `C`, cette interface prend la forme d'une série de fonctions. En `C++`, elle correspond à une partie des méthodes de la classe *système d'exploitation basique* "`RTSys`". Ce sont elles qui peuvent être personnalisées : remplacées ou étendues. Par défaut, les fonctions de rappel sont *instantanées* en ce qui concerne le temps-réel simulé. En les personnalisant, il est possible de rajouter des appels à la primitive `hold_cpu()` afin de simuler l'occupation du temps processeur, et même d'invoquer un autre service système (ou le même).

Le système d'exploitation basique s'occupe d'encadrer automatiquement l'appel de toutes les fonctions de l'interface, ou des fonctions de rappel ci-dessus, par la génération de messages d'instrumentation. Il s'occupe également de mettre à jour les statistiques d'utilisation du statut d'exécution du contexte d'exécution courant (travail de tâche, traitant d'interruption). Ainsi, les surcoûts système (invocations de services système, exécution de l'ordonnanceur, et interruptions par un/des traitants d'interruption) qui résultent des appels à la primitive `hold_cpu()` dans ces fonctions de rappel, sont automatiquement pris en compte dans la simulation.

Le tableau 7.1 résume les fonctionnalités proposées par l'interface de programmation du système d'exploitation simulé, et leurs relations avec les fonctions de rappel impliquées directement ou indirectement (entre parenthèses).

La figure 7.7 donne un aperçu du comportement du système complet, et en particulier de celui du système d'exploitation simulé. Elle présente les messages d'instrumentation qui sont générés (en oblique en haut de la figure), la localisation de l'appel à la fonction de rappel d'ordonnancement, ainsi que les imbrications des traitants d'interruption (règle de préemption par priorité).

Fonction de l'interface de programmation *(visible par les tâches/interruptions)*	**Fonction de rappel** *(personnalisable)*
Gestion des travaux des tâches	
`st_rtsys_activate_task()`	`notify_task_activation()` *(+ `reschedule()`)*
`st_yield()`	`reschedule()`
`st_rtsys_cancel_task()` et *Fin de tâche*	`notify_task_termination()` *(+ `reschedule()`)*
Gestion des interruptions	
`st_rtsys_get_irq_mask()`	
`st_rtsys_set_irq_mask()`	*(+ `isr()` éventuellement, via couche basse)*
`st_rtsys_get_local_interrupts_state()`	
`st_rtsys_local_disable_interrupts()`	
`st_rtsys_local_restore_interrupts()`	*(+ `isr()` éventuellement, via couche basse)*
Traitement d'une interruption	`isr()` *(+ `reschedule()` pour l'ISR la plus externe)*
Gestion de l'échelle de temps système	
`st_rtsys_get_date()`	
`st_rtsys_change_date()`	*(+ `reschedule()`)*
Extensions du système d'exploitation basique	
`st_syscall()`	`syscall()` *(+ `reschedule()` si demandé)*
Ordonnancement	
Décision d'ordonnancement nécessaire	`reschedule()`

TAB. 7.1: Interface du système d'exploitation, fonctions de rappel, et leurs relations

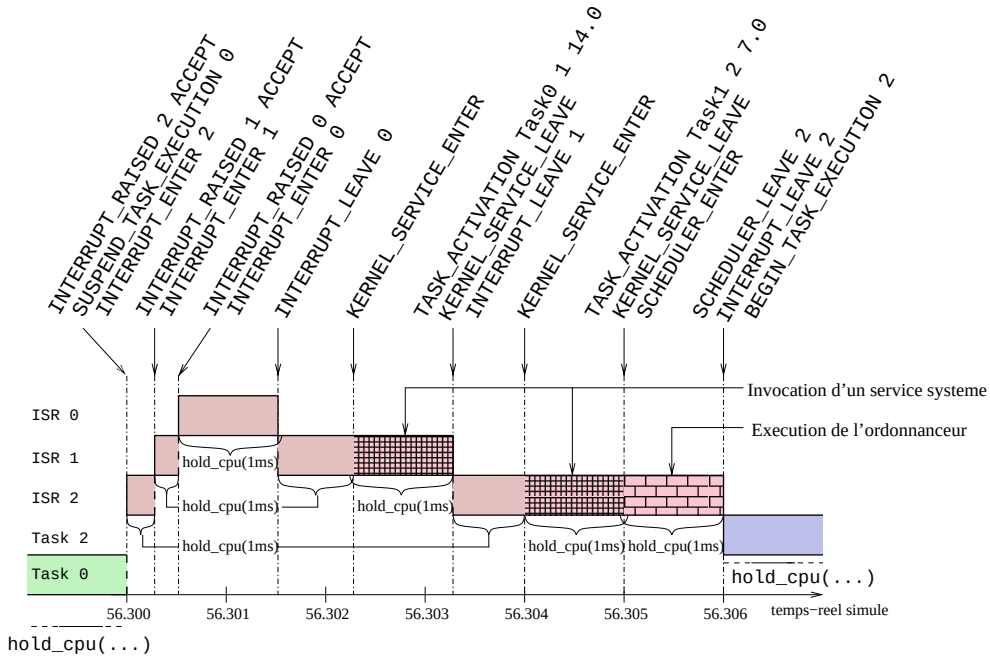


FIG. 7.7: Scénario d'exécution typique

Dans ce qui suit, nous détaillons chacune des rubriques présentes dans le tableau 7.1.

Gestion des travaux des tâches

La création d'un travail correspond à appeler la fonction `st_rtsys_activate_task()` qui prend en paramètre un modèle de tâche associé, et un statut d'exécution. C'est à l'appelant de cette fonction d'allouer et d'initialiser le modèle de tâche, et d'allouer le statut d'exécution. Le système d'exploitation basique s'occupe d'initialiser le statut d'exécution : entre autres l'état du travail est mis à `READY` (*prêt*) pour que le travail soit éligible par l'ordonnanceur.

Une fois le travail créé, le système d'exploitation s'occupe de mettre à jour les informations contenues dans le statut d'exécution en fonction des invocations de service système et des interruptions/préemptions. En particulier l'état du travail est régi par l'automate de la figure 7.8.

L'interface de programmation du système d'exploitation simulé propose aussi des fonctions pour supprimer (`st_rtsys_cancel_task()`) des travaux activés (y compris le travail courant), ou pour demander à l'ordonnanceur d'élire une tâche à exécuter (`st_yield()`).

Enfin, la fin d'un travail conduit à appeler automatiquement la fonction de rappel `notify_task_termination()` du système d'exploitation simulé.

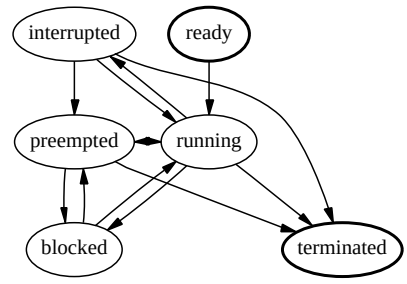


FIG. 7.8: Diagramme de transition des états des travaux de tâches

Gestion des interruptions

La levée et le traitement des interruptions simulées sont automatiquement gérés par la couche basse du module (voir 7.2.3.3). L'interface de programmation fournie par la couche haute propose les fonctions qui permettent de modifier :

- Le masque de filtrage global des interruption (fonctions `st_rtsys_get_irq_mask()` et `st_rtsys_set_irq_mask()`),
- Le statut du contexte d'exécution courant (travail, traitant d'interruption) vis-à-vis du filtrage local des interruptions (fonctions `st_rtsys_local_disable_interrupts()`, `st_rtsys_local_restore_interrupts()` et `st_rtsys_get_local_interrupts_state()`).

La prise en charge des interruptions est effectuée par la fonction de rappel `isr(level)` du système d'exploitation simulé, qui a les caractéristiques définies dans la présentation de la couche basse (7.2.3.3).

Échelle de temps système

Dans les systèmes réels, la notion de temps-réel est abstraite, et leur est en général non directement accessible. Ainsi, la plupart des systèmes ne disposent que d'une approximation discontinue (par paliers) de l'échelle de temps-réel, grâce à un matériel externe qui génère *régulièrement* (*i.e.* le plus régulièrement possible vis-à-vis de l'évolution du temps-réel) des impulsions signifiant l'écoulement du temps : l'*horloge système*.

Dans le module de simulation de système temps-réel d'ARTISST, le principe est le même : le système simulé n'a pas connaissance du temps-réel défini par l'infrastructure de simulation (voir 7.1.4). Il repose sur un *stimulus* externe pour gérer (en général : incrémenter) la *date système* courante, *i.e.* pour définir l'*échelle de temps système*.

Alors que l'échelle de temps-réel est globale à toute la simulation (même s'il y a plusieurs modules de simulation de système temps-réel), l'échelle de temps système est locale à chaque module de simulation de système temps-réel. De cette manière, il est possible de simuler des systèmes distribués sujets à des désynchronisations des horloges locales, ou d'évaluer l'impact de la granularité de l'horloge système sur le comportement de différents ordonnanceurs (voir partie III). La figure 7.9 illustre la différence entre les

échelles de temps-réel et de temps système pour un système distribué formé de deux modules de simulation de système.

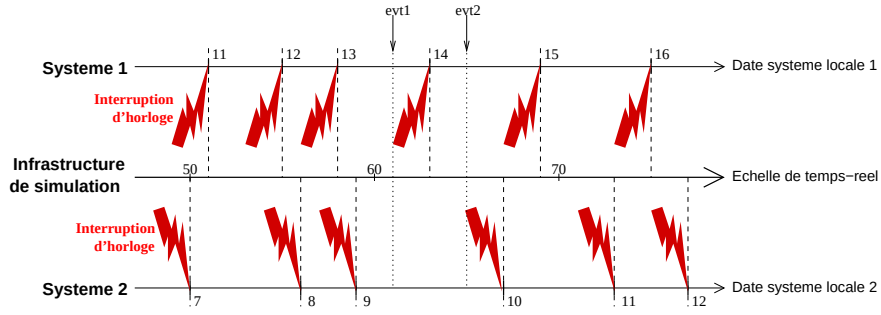


FIG. 7.9: Échelles de temps-réel et de temps système

Cette échelle de temps système définit la perception qu'a le système de l'évolution du temps. Il en découle que, si le système simulé prendre une décision ou effectuer des calculs liés au temps perçu, il doit avoir conscience que la date qu'il perçoit correspond à la date de la dernière interruption d'horloge, et non à la date courante effective dans l'échelle de temps-réel simulé. Nous verrons en partie III que cette granularité dans la mesure du temps a un impact sur le comportement du système simulé, et nous l'évaluerons pour différents ordonnanceurs.

Ainsi, c'est cette échelle de temps système qui doit être utilisée dans le système d'exploitation simulé et dans l'application dès qu'il s'agit de mesurer une durée ou connaître une date : sur la figure 7.9, il est par exemple impossible pour le système 2 de distinguer les 2 événements `evt1` et `evt2`, alors que le système 1 peut (par chance) les distinguer. En particulier, toutes les décisions d'ordonnancement doivent (mais rien ne l'impose techniquement) se fonder sur cette échelle, et non sur l'échelle de temps-réel qui en théorie est inaccessible au système simulé. Ceci ne signifie pas pour autant que les décisions d'ordonnancement peuvent être prises uniquement lors des interruptions d'horloge système (*tick scheduling*) : elles demeurent indépendantes des paliers de variation de l'échelle du temps système.

Cependant, toutes les simulations d'occupation du temps processeur (*i.e.* expression `delay` passée à la primitive `hold_cpu(delay)`) sont relatives à l'échelle de temps-réel.

En pratique, la date système a exactement la même structure (en termes de langage de programmation) que les estampilles des messages de simulation (voir 7.1.1.1). Et le stimulus pour la gestion de la date système est en général délivré par un module source, appelé *horloge système*, sous la forme d'une interruption ; le traitant d'interruption associé est alors en charge d'incrémenter (le plus souvent) la date système, bien que ceci puisse également être effectué par un autre élément du système (comme un mécanisme de synchronisation d'horloge en distribué par exemple).

L'interface de programmation de la couche haute propose les fonctions pour gérer cette échelle de temps système : une fonction pour consulter la valeur de cette date (`st_rtsys_get_date()`), et une fonction pour la modifier (`st_rtsys_change_date()`).

À chaque fois que la date système courante est modifiée, les statistiques d'utilisation relatives au temps système du statut d'exécution du contexte d'exécution courant (travail de tâche, traitant d'interruption) sont mises à jour.

Extensions et personnalisation

Une fonction de l'interface de programmation est réservée pour l'appel de services système implantés par l'utilisateur : `st_syscall()`. Cette fonction est associée à la fonction de rappel `syscall()` que l'utilisateur peut personnaliser (`C`) ou surcharger (`C++`). C'est dans cette fonction de rappel que des services tels que des mécanismes de synchronisation sur ressources doivent être implantés.

Ces fonctions prennent en argument un identifiant d'appel système que l'utilisateur définit, et des paramètres également définis par l'utilisateur. Par exemple, pour l'implantation d'un mécanisme de synchronisation de ressource de type sémaphore, un identifiant peut servir à signifier la prise du sémaphore, et un autre son relâchement, et les paramètres peuvent représenter l'identifiant du sémaphore.

Le corps de la fonction de rappel doit indiquer si une décision d'ordonnancement doit être prise à l'issue de son invocation ou non. La fonction de rappel par défaut ne fait rien, et indique qu'aucune décision d'ordonnancement n'est nécessaire.

Fonction de rappel d'ordonnancement

L'ordonnanceur est une simple fonction de rappel (`reschedule()`) appelée automatiquement par le module lors :

- de l'invocation des fonctions de gestion des travaux des tâches,
- de la fin du travail d'une tâche,
- de la fin du traitant d'interruption le plus externe (car il y a imbrication possible des traitants d'interruption),
- de la fin de l'invocation d'un service système personnalisé, si l'utilisateur en a décidé ainsi.

Cette fonction est en charge de définir le travail de tâche à élire avant de l'exécuter sur le processeur simulé, et connaissant la liste des travaux disponibles dans le système (variable `ttable` disponible dans le système d'exploitation basique). L'ordonnanceur peut dépendre d'un modèle de tâche et/ou d'un statut d'exécution particulier pour les travaux possibles, autres que ceux proposés par défaut. Comme il a été indiqué précédemment, l'ordonnanceur doit prendre ses décisions liées au temps relativement à l'échelle de temps système.

Par défaut, ARTISST vient avec plusieurs ordonnanceurs implantés ; nous les décrivons au chapitre 8.

7.2.5 Configuration et initialisation

Au démarrage de la simulation, le support d'exécution est configuré par défaut de telle manière que :

– toutes les interruptions sont ignorées,
– tous les services du système d'exploitation simulé sont totalement interruptibles, et l'ordonnanceur est supposé réentrant. Il est possible de modifier ces caractéristiques à la compilation du module.

À la création du module, l'utilisateur doit indiquer le travail de tâche initial (*travail d'initialisation*) à exécuter au lancement de la simulation. Ce travail est en charge d'activer tout ou partie des interruptions si l'utilisateur le désire, et d'activer d'autres travaux de tâches si nécessaire.

7.2.6 Un exemple

L'annexe A indique comment mettre en place une simulation avec ARTISST. Cependant, dans cette section nous allons donner un exemple de la séquence d'événements générée à partir d'un exemple très simple dans lequel deux tâches sont en présence. Nous considérons que :

– L'ordonnanceur est de type *tourniquet* (*round-robin* en anglais), et chaque invocation occupe le processeur pendant 1ms dans l'échelle de temps-réel simulé. Nous ne détaillons pas son code ici.
– Les deux tâches τ_1 et τ_2 effectuent une boucle infinie :

```
while(1)
{
  st_hold_cpu(t_2ms);
  st_syscall(1, 0, NULL); // appel du service systeme 0
                          // avec comme parametre NULL (inutilisé)
}
```

`t_2ms` est une variable représentant une durée de 2 ms sur l'échelle de temps-réel simulé.

– Nous considérons que la fonction de rappel d'appel système `syscall()` du module de simulation de système temps-réel (*i.e.* celle qui prend en charge les appels à `st_syscall()` ci-dessus) simule l'occupation du processeur pendant 1ms dans l'échelle de temps-réel simulé. Voici le code correspondant :

```
st_hold_cpu(t_1ms);
```

`t_1ms` est une variable représentant une durée de 1 ms sur l'échelle de temps-réel simulé.
Nous considérons que l'appel de ce service se termine par l'invocation de l'ordonnanceur.
– La seule interruption matérielle du système correspond à l'interruption d'horloge, émise toutes les 5ms dans l'échelle de temps-réel simulé, et prise en charge par un traitant d'interruption qui nécessite un traitement de 1ms dans l'échelle de temps-réel simulé. Voici le code correspondant :

```
st_hold_cpu(t_1ms);
```

Voici un exemple de séquence d'événements valides générée (nous n'indiquons pas leur nom réel, interne à ARTISST, l'annexe A s'en charge) :

Date	Événement de simulation
0 :	**Message de début de la tâche** τ_1
0-2 :	`hold_cpu()` par τ_1 en cours
2 :	Fin du `hold_cpu()` de τ_1, invocation de l'appel système
	Message de début d'appel système
2-3 :	`hold_cpu()` par l'appel système
3 :	Fin du `hold_cpu()` de l'appel système initié par τ_1
	$\Rightarrow$ Invocation de l'ordonnanceur
	Message de d'invocation de l'ordonnanceur
3-4 :	`hold_cpu()` par l'ordonnanceur, élection de la τ_2
4 :	**Message de fin d'invocation de l'ordonnanceur**
	Message de fin d'appel système
	Message de préemption de la tâche τ_1
	Message de début de la tâche τ_2
4-... :	`hold_cpu()` par τ_2 en cours (interrompu)
5 :	*Interruption d'horloge*
	Message d'interruption d'horloge (prise en charge immédiate)
	Message de notification d'interruption de τ_2
	Message de début de traitant de l'interruption d'horloge
5-6 :	`hold_cpu()` du traitant d'interruption d'horloge
6 :	Fin de la fonction de rappel de traitement d'interruption
	$\Rightarrow$ Invocation de l'ordonnanceur
	Message d'invocation de l'ordonnanceur
6-7 :	`hold_cpu()` par l'ordonnanceur, élection de la τ_1
7 :	**Message de fin d'invocation de l'ordonnanceur**
	Message de fin de traitant de l'interruption d'horloge
	Message de notification de reprise de la tâche τ_1
	Reprise de la tâche τ_1 (deuxième itération du `while`), début du `hold_cpu()`
7-9 :	`hold_cpu()` par τ_1 en cours
...	

Lorsque la tâche τ_2 sera de nouveau ré-élue, le `hold_cpu()` qui était en cours à la date 5 sera repris, de sorte qu'une occupation du processeur pour les 1ms restantes soit simulée.

7.2.7 Points délicats

Comme il a été signalé dans la première partie, la simulation d'un système en vue de son évaluation a d'autant plus d'intérêt qu'elle est pertinente, c'est-à-dire représentative du comportement de l'implantation effective.

C'est pour cela qu'ARTISST vise à fournir le nécessaire pour implanter la simulation

d'applications existantes au-dessus d'un système d'exploitation simulé. C'est dans cet objectif qu'elle permet que le système d'exploitation simulé puisse reprendre l'interface de programmation d'un système d'exploitation existant. C'est pour cela aussi qu'elle autorise et encourage l'utilisateur à ajouter les appels à `hold_cpu()` pour prendre en compte, simplement et le plus fidèlement possible, les surcoûts système.

Cependant, la principale difficulté dans l'ajout des appels à `hold_cpu(delay)` réside dans l'expression du paramètre `delay` passé, elle-même liée à la granularité des appels à `hold_cpu()`. Ce paramètre indique le temps d'occupation du processeur à simuler relativement à l'échelle temps-réel ; il sera d'autant plus simple à déterminer que l'appel à `hold_cpu()` se réduit à couvrir un bloc d'exécution élémentaire (séquence d'instructions). À une granularité plus grossière, la présence de boucles et de conditionnelles rend en effet l'expression de ce paramètre plus compliquée si on veut qu'elle soit fidèle au comportement effectif, et nécessite l'étude approfondie du code source avant simulation (analyse statique, annotations de boucles). Mais en contrepartie, plus la granularité des appels à `hold_cpu()` est fine, plus la simulation est lente.

D'autre part, même si l'expression du paramètre passé à `hold_cpu()` reste simple, on peut juger utile de lui adjoindre des fluctuations aléatoires, d'autant plus justifiées que la granularité est élevée : elles peuvent rendre compte de l'influence du matériel par exemple (cache, pipeline par exemple). Dans ce cas se pose en plus le problème de la définition de la distribution de la loi de probabilité à simuler (type, paramètres) pour qu'elle soit fidèle au comportement effectif, ce qui forme un domaine de recherche actif [BCP02].

7.3 Simulation de systèmes distribués

7.3.1 Définitions et principe

Avec ARTISST, la simulation de systèmes constitués de plusieurs processeurs correspond à la simulation de systèmes distribués au-dessus d'un réseau simulé. La simulation continue cependant de s'effectuer au sein du même processus sur la machine hôte de la simulation, par passage de messages de simulation.

ARTISST ne permet pas de simuler des systèmes multiprocesseurs à mémoire partagée gérés par un seul système d'exploitation simulé commun : le module de simulation de système temps-réel décrit en 7.2 ne permet pas de partager un même système d'exploitation simulé (*i.e.* couche haute) entre plusieurs processeurs simulés (*i.e.* couches basses). L'ensemble *processeur simulé + système d'exploitation + application* est appelé "*nœud*" dans toute la suite.

Le circuit de simulation typique d'une simulation de système distribué est indiqué dans la figure 7.10 (système à deux *nœuds*), et obéit aux règles définies en 7.1.4. En particulier, la règle 3 "*Seuls les modules source utilisés en mode consommation peuvent créer ou modifier des estampilles*" n'est pas remise en cause en ce qui concerne le module de simulation de système temps-réel : car celui-ci produit effectivement de nouveaux messages avec de nouvelles estampilles, et en plus les propage à des modules destination (en l'occurrence les modules de simulation d'environnement et de chronogramme) en

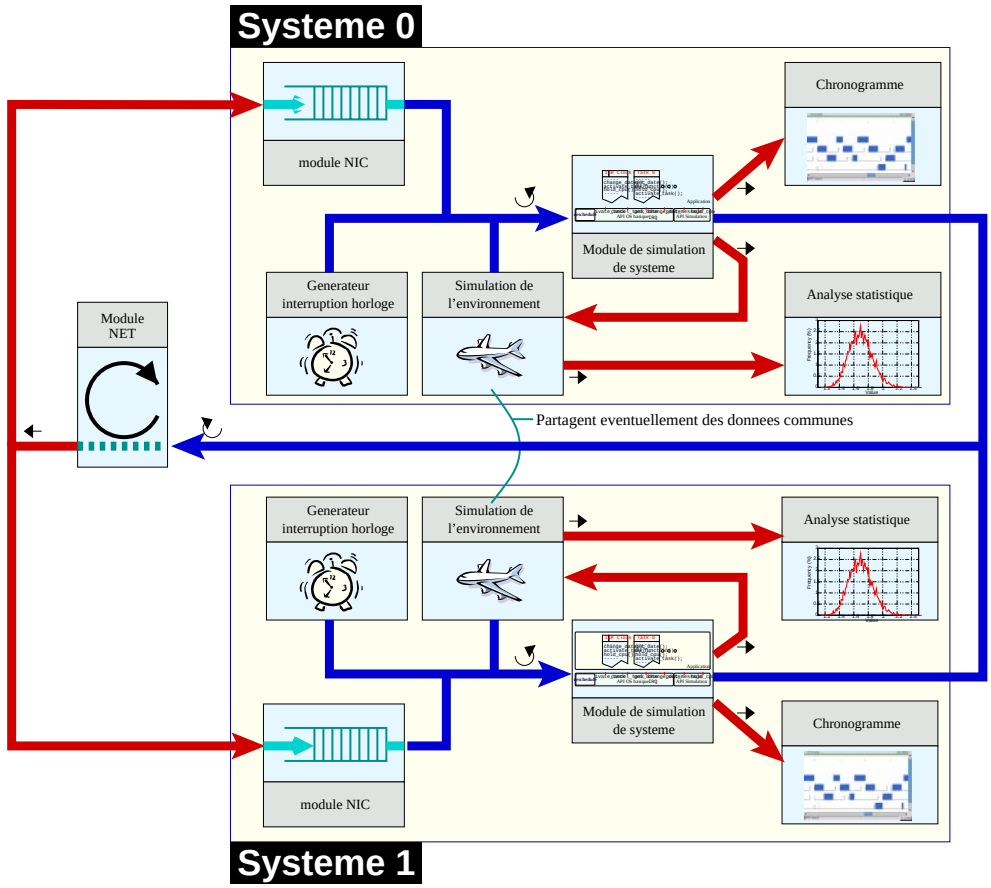


FIG. 7.10: Circuit de simulation de système distribué typique

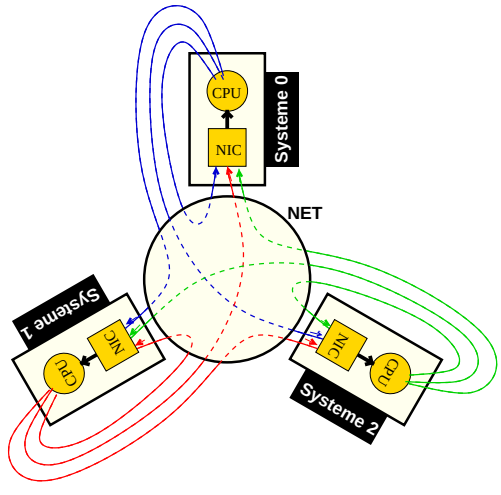
mode soumission, mais chaque message soumis correspond exactement à un message consommé par la boucle d'événements. La même remarque s'applique aux modules de simulation d'environnement qui propagent leurs messages produits vers les modules d'analyse statistique. À noter que les deux modules de simulation d'environnement peuvent partager des données, afin de simuler un environnement commun dans lequel serait plongé le système distribué.

La simulation de systèmes distribués repose sur deux types de modules, présentés dans la suite :

- Un seul module d'acheminement des messages (`Net`, pour *Network*) pour tout le circuit, qui sert à la boucle d'événements,
- Des modules d'interface réseau (`Nic`, pour *Network Interface Controller*) : un module associé à chaque nœud simulé.

Le réseau simulé est capable de simuler des pertes, des délais, et des altérations de messages réseau, ainsi que de gérer les interconnexions des nœuds (point à point, diffusion, plusieurs sous-réseaux, ...).

La figure 6.3 reprise ci-dessous donne la topologie globale du réseau simulé et le rôle respectif des modules `Net` et `Nic`, abstraction faite de l'infrastructure de simulation, et lorsque toutes les possibilités d'interconnexion entre nœuds sont exploitées. Les flèches représentent les interconnexions logiques réalisées par le module `Net` (*i.e.* une diffusion totale).



7.3.2 Les messages réseau

Les messages réseau correspondent à un type de message de simulation précis (`NET_SEND`). ARTISST fournit une primitive pour l'envoi de messages réseau, qui correspond ni plus ni moins au remplissage de la structure *message de simulation* pour figurer un message réseau, et à sa génération sur l'infrastructure de simulation (appel de la primitive `post_message()`). L'information associée à ces messages est :

- Date temps-réel de l'émission du message,
- Identifiant d'émetteur (utile aux modules `Nic`),

– Identifiant de destination (utile aux modules `Nic`),
– Le contenu du message réseau ainsi que sa longueur.

L'identifiant de l'émetteur et la date d'émission sont automatiquement définis pour chaque message envoyé. L'identifiant d'émetteur est fixé lors de la création des modules de simulation de système temps-réel : le premier système simulé créé reçoit l'identifiant `0`, le deuxième `1`, etc... (pas de limitation).

Aucune de ces informations n'est utilisée par le module `Net`. Et les modules `Nic` peuvent choisir de les exploiter ou non. Ainsi, il est possible de simuler le routage de messages de type point à point (en considérant que l'identifiant de destination est en fait l'identifiant d'un système temps-réel simulé), multicast (en considérant que l'identifiant de destination est une adresse multicast), à diffusion (en ne tenant pas compte de l'identifiant de destination), ou autre. De même, le contenu du message peut correspondre à l'encapsulation de protocoles de plus haut niveau, ceci restant tout à fait opaque à la partie qui nous concerne ici ; à ce titre, la simulation de réseau dans ARTISST correspond à la simulation de la couche physique d'un réseau d'interconnexions.

7.3.3 Module d'acheminement de messages "NET"

Le module `Net` est un module très simple dont le rôle est de consommer les messages de simulation des modules source qui lui sont connectés (des modules de simulation de système), et de renvoyer tous les messages réseau à l'identique (les autres messages de simulation sont ignorés) vers tous les modules destination qui lui sont connectés en mode soumission (des modules `Nic`, y compris le module `Nic` associé à l'émetteur du message).

Contrairement à ce que le nom laisse paraître, ce type de module ne simule pas les pertes, les délais d'acheminement, les altérations des messages, ni leur routage, et n'est pas du tout personnalisable ou extensible. Son rôle est exclusivement de contrôler les visions du temps-réel locales à chacun des nœuds simulés, au travers des estampilles des messages générés, afin d'assurer leur cohérence. Pour ce faire, ce module exploite les fonctionnalités de l'infrastructure de simulation, puisqu'il doit être capable de consommer le *prochain* message provenant de plusieurs modules source en mode consommation. Ce module sert en général de boucle d'événements (comme sur la figure 7.10).

7.3.4 Module de simulation de réseau "NIC"

Les modules coupleurs `Nic` (pour *Network Interface Controller*) servent de passerelle entre le module `Net` décrit précédemment, qui leur soumet (mode soumission) les messages réseau émis par tous les nœuds simulés ; et les nœuds simulés, qui peuvent consulter leur module `Nic` afin de récupérer les messages disponibles. Ces modules s'apparentent par conséquent à des files de messages réseau.

Ce sont ces modules qui sont responsables de la simulation du réseau proprement dite : ce sont eux qui simulent la *transmission* des messages et de ses caractéristiques (délais de propagation, pertes, altération, routage), par la manipulation directe de leur file des messages (7.3.4.1). Et ce sont à eux que s'adressent les nœuds associés pour

récupérer les messages réseau (7.3.4.2). Enfin, ces modules sont totalement personnalisables pour simuler toutes les caractéristiques de réseau possibles (7.3.4.3).

7.3.4.1 Simulation de transmission de message et signalisation

Pour la simulation de la transmission des messages réseau, les modules `Nic` agissent sur leur file de messages locale. Ils peuvent ainsi :

- Supprimer des messages de la file ⇒ simulation de pertes de messages (critères en général probabilistes), ou de routage (critère lié aux identifiants d'émetteur et de destination),
- Modifier les estampilles des messages, qui (dans la file uniquement) correspondent à la date temps-réel à partir de laquelle les messages peuvent être reçus par le nœud associé ⇒ simulation de délais d'acheminement,
- Modifier le contenu des messages réseau ⇒ altération des messages.

Ces actions sont effectuées par deux fonctions de rappel que l'utilisateur définit à la construction du module, ainsi que nous le voyons ci-dessous, en 7.3.4.3.

Vis-à-vis du nœud associé, on distingue deux types de messages présents dans la file :

- Ceux susceptibles d'être récupérés par le nœud associé : ce sont les messages réseau dont l'estampille a été positionnée à une date temps-réel antérieure à la date temps-réel courante. On dit dans ce cas que le message réseau a été *reçu* : ils sont en attente d'être *délivrés* (*i.e.* récupérés par le nœud associé)
- Ceux qui sont "en transit" : ce sont les messages réseau qui ont été soumis au module `Nic`, mais pour lesquels celui-ci a modifié l'estampille de telle sorte qu'elle fasse référence à une date temps-réel à venir. Les messages de ce type sont conservés dans la file en attendant d'être déclarés "reçus", *i.e.* en attendant que la date temps-réel soit postérieure à l'estampille.

Lorsqu'un message dans la file passe de la catégorie "en transit" à la catégorie "reçu" (l'inverse n'est pas possible), une interruption simulée est automatiquement et systématiquement générée par le module `Nic` à destination du nœud simulé associé. Le niveau de cette interruption est défini lors de la construction du module `Nic`. C'est par ailleurs le seul message de simulation qui est propagé entre `Nic` et nœud associé par l'infrastructure de simulation : les messages réseau sur le lien `Nic` → nœud (et uniquement celui-là) sont propagés par appel de fonction, ainsi que nous le voyons en 7.3.4.2 (d'où la sémantique particulière de l'estampille des messages réseau stockés dans la file de messages du `Nic`).

Le nœud associé au `Nic` peut choisir d'ignorer cette interruption (mode de réception par attente active par exemple), ou de la prendre en compte (mode de réception par signalisation).

7.3.4.2 Livraison du message

Une fois un message reçu, le nœud associé peut le récupérer. Pour cela, le module `Nic` possède une interface de programmation spécifique très simple, formée des deux

fonctions `st_nic_is_empty()` et `st_nic_next_message()`. Leur rôle est respectivement de savoir si la file de messages du `Nic` contient des messages de la catégorie "reçus" ; et de récupérer le message de cette catégorie en tête de cette liste (ordre chronologique des estampilles) : on parle de *livraison* du message. Zéro ou plusieurs messages de la catégorie "reçu" peuvent éventuellement être dans la liste, surtout en mode de réception par attente active. C'est pourquoi la livraison des messages au nœud simulé consiste en une boucle du type :

```
while (!st_nic_is_empty(my_nic_module)) {
  struct st_message_t msg;

  st_nic_next_message(my_nic_module, & msg);
  /* Le message msg vient d'etre recu,
     il peut etre traite */
  // ...
}
```

7.3.4.3 Fonctionnement interne et personnalisation

En interne, la file de messages est une simple file des messages réseau classés suivant l'ordre chronologique des estampilles. Pour l'instant, l'implantation est de type liste chaînée (circulaire double).

L'action de chaque module `Nic` sur cette liste se fait à deux niveaux :

1. Lors de chaque soumission d'un message réseau par le module `Net` (*i.e.* à chaque invocation de la méthode `push()` du module par l'infrastructure de simulation),
2. Lors de chaque livraison de message (appel à la fonction `st_nic_next_message()` de l'interface de programmation du module).

Pour cela, deux fonctions de rappel sont appelées, qui doivent être fournies par l'utilisateur au moment de la construction de chaque module `Nic`.

Dans ces fonctions de rappel, l'utilisateur a le droit :

- de supprimer des messages de la file ;
- d'incrémenter les estampilles des messages (*i.e.* simuler un délai de propagation), en les réordonnant si besoin. Mais il ne peut pas décrémenter ces estampilles ;
- de modifier les contenus des messages réseau.

Ce fonctionnement permet de modéliser des réseaux et des protocoles aussi variés que possible (à détection de collision comme Ethernet par exemple, à priorité comme le bus CAN par exemple, ...).

7.4 Autres modules

La bibliothèque Artisst est accompagnée d'autres modules de simulation très simples. On distingue les modules qui servent à générer des messages de simulation servant de stimuli à un ou plusieurs modules de simulation de système temps-réel

(dits *modules d'entrée*), de ceux qui servent à analyser les messages d'instrumentation générés par les modules de simulation de système temps-réel (dits *modules de sortie*).

Il existe en plus un module de type spécial, dit *gateway*, dont le rôle est de servir de relai : il sert dans une boucle d'événements, et est en charge de consommer les messages qui sont générés par les modules qui se trouvent en amont, et de les rediffuser tous, en mode soumission, aux modules qui lui sont connectés en aval. Il joue en quelque sorte le rôle de "multiprise", ou de brasseur.

7.4.1 Modules d'entrée

Les modules d'entrée Artisst sont des modules qui ne sont accessibles qu'en mode consommation.

7.4.1.1 Générateurs aléatoires

Un module de génération aléatoire de messages est un module qui génère un message donné toujours identique, avec une estampille qui est régie par une loi statistique donnée : entre deux messages m_i et m_{i+1} successifs générés par le module, la loi statistique fixe la distribution de $estampille(m_{i+1}) - estampille(m_i)$. 20 distributions standard sont disponibles :

- Constante
- Binaire (*i.e.* lancé de pièce)
- Uniforme
- Gauss/normal
- Poisson
- Erlang
- Exponentielle
- Hyper-exponentielle
- Laplace
- γ
- χ^2
- β
- Student
- Log normal
- Log γ
- Weibull
- Géométrique
- Binomiale
- Neg. binomiale
- Triangulaire

On peut également fournir une distribution personnalisée en définissant soit la fonction de génération aléatoire, soit la fonction de répartition, soit la densité de probabilité de la distribution souhaitée.

Pour simuler une loi d'activation périodique par exemple, on utilisera la distribution constante. Pour une loi d'activation sporadique, on utilisera une fonction aléatoire personnalisée, qui pourra utiliser les distributions prédéfinies, et qui se chargera de la borner afin de garantir le délai d'inter-arrivée spécifié.

7.4.1.2 Lecteur de traces

Le module de lecture de traces transforme un fichier texte, dans lequel figure un message par ligne, en le flux de messages de simulation équivalent. Un exemple typique de traces interprétées est le suivant :

```
4.150500000 INTERRUPT_RAISED 0 ACCEPT
4.150500000 SUSPEND_TASK_EXECUTION 0
4.150500000 INTERRUPT_ENTER 0
4.151500000 SCHEDULER_ENTER
4.152500000 SCHEDULER_LEAVE 0
4.152500000 INTERRUPT_LEAVE 0
4.152500000 RESUME_TASK_EXECUTION 0
4.152500000 INTERRUPT_WAKEUP_POSTPONED INTERNAL
4.160500000 INTERRUPT_RAISED 0 ACCEPT
```

Le premier champ fixe l'estampille du message, les champs suivants renseignent le type et l'information associée au message.

7.4.2 Modules de sortie

Les modules de sortie sont accessibles en mode consommation et en mode soumission. En mode consommation, leur fonction de rappel `pull()` reproduit exactement les messages consommés à partir de son/ses module(s) source.

7.4.2.1 Enregistreur de traces

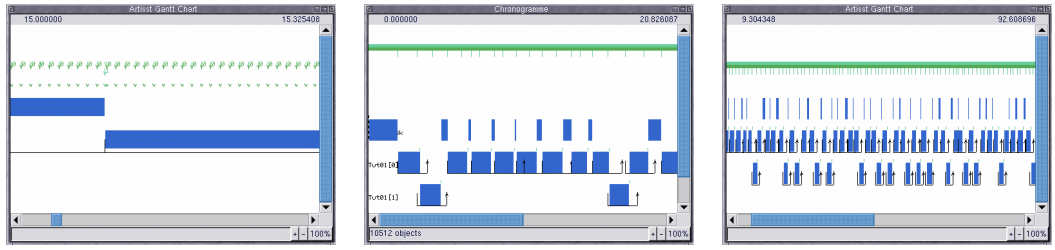
Le module d'enregistrement de traces stocke les messages de simulation qu'il reçoit dans un fichier. Le format des enregistrements est le même que celui utilisé par le lecteur de traces.

7.4.2.2 Chronogramme

Le module chronogramme (ou diagramme de Gantt) permet de représenter sous forme graphique les messages d'instrumentation issus d'un module de simulation de système temps-réel. Le principe est de représenter les événements liés à la gestion des interruptions sur une flèche du temps-réel, puis les événements liés aux différents travaux des tâches sur une flèche du temps par travail de tâche. Si plusieurs travaux de la même tâche existent, le nom de la flèche du temps associé est suffixé par `[N° du travail]` : pour un modèle de tâche de nom `Task1`, si deux travaux s'exécutent simultanément, le premier s'appelle `Task1[0]`, et le deuxième `Task1[1]`.

Le module repose sur une représentation interne sous la forme d'un arbre d'objets graphiques, sans limitation de taille (en dehors des limites liées à la taille mémoire), en particulier sans limitation sur la longueur des flèches de temps, ni sur la résolution (*i.e.* facteur de zoom)[4]. Le but est d'abstraire la partie manipulation graphique, de la partie représentation, et, par la structure en arbre, d'accélérer le rendu partiel de la scène (*clipping*). Il est en théorie possible de convertir cette représentation interne en n'importe quel format graphique (image, affichage à l'écran, postscript...). Le module propose deux formats de représentation : affichage à l'écran, et Xfig 3.2.

[4]Paradoxalement, nombre de simulateurs d'ordonnancement affichent leur chronogramme directement dans une fenêtre, ce qui limite leur largeur, et, partant, le temps de simulation autorisé, en particulier en cas de zoom (de l'ordre de 16384 pixels, voire moins).



(a) Zoom avant

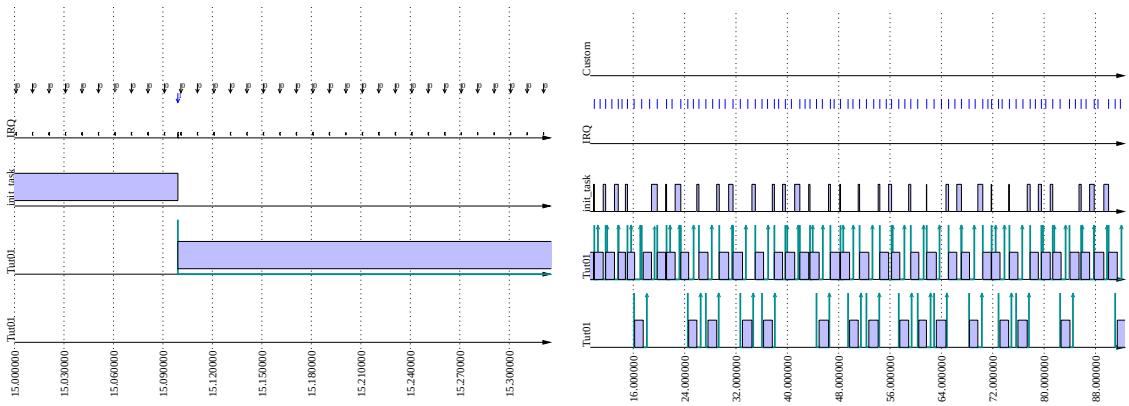
(b) Facteur de zoom par défaut

(c) Zoom arrière

Fig. 7.11: Fenêtre graphique de représentation du chronogramme

Le format de représentation par affichage à l'écran correspond à une fenêtre graphique reposant sur la bibliothèque `gtk`, sur laquelle l'utilisateur peut zoomer, se déplacer, et à partir de laquelle il peut décider de générer une représentation XFig de la zone qu'il est en train de visionner. En outre, la fenêtre indique la date temps-réel de l'endroit où se trouve le curseur de la souris, ainsi que l'objet sur lequel il pointe accompagné de ses paramètres temporels (exécution d'une tâche : date de début/fin, traitement d'interruption : date de début/fin, ...). La figure 7.11 est une capture d'écran donnant une idée du résultat. L'implantation de cette méthode de représentation suppose que la bibliothèque `libglade` est disponible : elle permet de construire la fenêtre à partir d'une description en `XML` (emplacement des boutons et des autres éléments graphiques). Il est ainsi très aisé de personnaliser l'aspect de la fenêtre même après que le module a été compilé.

La deuxième méthode de représentation correspond à la génération d'un fichier XFig (protocole version 3.2), qui autorise par la suite la transformation en une quantité d'autres formats d'images, vectorielles ou par points. Cette méthode de représentation peut être invoquée par la méthode de représentation sur fenêtre graphique (par appui sur une touche), ou directement dans le code de l'infrastructure de simulation : l'interface de programmation du module est prévue à cet effet. La figure 7.12 propose une version postscript du chronogramme, issue de la traduction depuis la version XFig.



(a) Zoom avant

(b) Zoom arrière

Fig. 7.12: Représentation vectorielle du chronogramme

Chapitre 8

Ordonnanceurs fournis avec ARTISST

L'un des objectifs d'ARTISST est de pouvoir évaluer le comportement d'une même application sous différents ordonnanceurs. Il est possible d'implanter ses propres ordonnanceurs, ou de personnaliser des ordonnanceurs existants. Par défaut, ARTISST propose plusieurs ordonnanceurs pour tâches indépendantes :

Ordonnanceurs à priorité simples :

- Un ordonnanceur préemptif à priorité générique[1] pour toutes les affectations de priorités statiques ou dynamiques : `JDP` (pour *Job-level Dynamic Priority* en anglais) ;
- Un ordonnanceur à priorité fixe mixte préemptif/non-préemptif, dit *à seuil de préemption* (voir 3.2.3.5 de la partie I), avec garantie hors-ligne ;
- Un ordonnanceur à priorité fixe préemptif pour un modèle de tâches différent, fondé sur la fréquence d'activation [JG99] (*RBE* pour *Rate-Based Execution*, voir 3.2.5.5 de la partie I), avec garantie hors-ligne.

Ordonnanceurs avec acceptation en-ligne : tous ces ordonnanceurs prennent en compte la granularité de l'horloge système (la dernière section 8.5 indique de quelle manière).

Avec sous-système garanti hors-ligne :

- Un ordonnanceur EDF préemptif avec serveurs pour tâches apériodiques du type *Constant Bandwidth Server* (voir 3.3.2.3 de la partie I), avec ordonnancement à priorité *conservative* quelconque (notion définie en 8.1) des tâches apériodiques entre elles, et garantie en-ligne de ces tâches ;
- Un ordonnanceur préemptif à réquisition de temps-creux à *double priorité* (*Dual Priority*, voir 3.3.2.4 de la partie I), avec ordonnancement à priorité *conservative* quelconque (notion définie en 8.1) des tâches apériodiques entre elles, et garantie en-ligne de ces tâches.

[1] Spécialisation par utilisation des *templates*, ou patrons, du langage `C++`.

Sans sous-système garanti hors-ligne :

- Une famille (*i.e.* arborescence d'objets) d'ordonnanceurs préemptifs à priorité *conservative* quelconque (notion définie en 8.1) pour tâches exclusivement apériodiques, avec garantie et/ou avec repêchage des travaux refusés : la famille `JFP` (pour *Job-level Fixed Priority* en anglais). Cette famille d'ordonnanceurs est *paramétrique*, c'est à dire que chaque ordonnanceur de cette famille peut être spécialisé pour différentes notions de priorité conservative, qui incluent entre autres toutes les priorités fixes, EDF (correspond alors à `GED` décrit en 3.4.2.3 de la partie I), FIFO et une approximation de LLF.

Ordonnanceurs avec politique de rejet : tous ces ordonnanceurs prennent en compte la granularité de l'horloge système.

- Une famille d'ordonnanceurs préemptifs à priorité conservative avec politique de rejet sans reconfiguration, et test d'acceptation en-ligne pour l'ordonnancement de tâches exclusivement apériodiques : ces ordonnanceurs constituent une partie de la famille `JFP`. Cette famille inclut les variantes robustes de DM, EDF (correspond alors à `RED` décrit en 3.4.2.3 de la partie I), FIFO et une approximation de LLF ;
- Un ordonnanceur avec un modèle de tâche étendu, où une partie de chaque tâche est susceptible d'être rejetée : TPS (pour *Taskpair Scheduling*, voir 3.4.4 de la partie I).

Ordonnanceur pour systèmes ouverts :

- L'ordonnanceur EEVDF [SAWJ+96] (pour *Earliest Elligible Virtual Deadline First*) pour le modèle fluide, et garantie en-ligne des requêtes apériodiques temps-réel (voir 3.5 de la partie I).

Ordonnanceurs *au mieux* avec approche par prévision :

- Des ordonnanceurs *au mieux* avec test d'acceptation simple contrôlé par les dépassements d'échéance passés : FC-EDF et FC-EDF^2 (voir 3.5.2 de la partie I).

Tous les ordonnanceurs fournis supportent les tâches qui n'ont pas d'échéance temporelle : quand aucune tâche possédant une contrainte temps-réel n'est prête, l'ordonnanceur élit la tâche non temps-réel de plus haute priorité, suivant une affectation de type priorité statique, et en mode préemptif (*i.e.* ordonnancement des tâches non temps-réel en tâche de fond).

Dans les quatre sections qui suivent, nous décrivons quatre de ces ordonnanceurs, puisqu'ils sont soumis à évaluation dans la partie III, et parce qu'ils présentent des caractéristiques originales : l'ordonnanceur à bande passante constante (`CBS-JFP`), l'ordonnanceur à double priorité (`DP`), la famille d'ordonnanceurs `JFP`, et l'ordonnanceur TPS. Nous présentons la famille `JFP` en premier car ses caractéristiques et son fonctionnement sont repris en partie dans `DP` et `CBS-JFP`.

Dans ces sections, nous considérons que le système a moyen de déterminer la date temps-réel actuelle exacte (temps continu), ainsi que les temps exacts que les travaux

ont passé sur le processeur. Nous levons cette hypothèse dans la dernière section du chapitre, en exposant les problèmes liés à la granularité de l'horloge système pour les ordonnanceurs avec acceptation en-ligne de tâches apériodiques temps-réel, et en indiquant comment la prendre en compte dans ces ordonnanceurs.

Les autres ordonnanceurs disponibles et non présentés en détail ci-dessous correspondent à l'implantation directe des travaux décrits dans les articles qui les proposent. Les sources d'Artisst contiennent une documentation complète pour tous ces ordonnanceurs.

8.1 Famille JFP (*Job-level Fixed Priority*)

On peut remarquer [Liu00, Mig99] que l'ordonnancement à priorité statique (voir 3.1.3.2 de la partie I) correspond à la définition d'une priorité à l'échelle de la tâche (*i.e.* tous travaux confondus), et que l'ordonnancement à priorité dynamique revient la plupart du temps à définir la priorité à l'échelle de chaque travail : ceci est valable par exemple pour EDF (priorité = échéance absolue du travail), FIFO (priorité = date d'activation), mais n'est pas valable pour LLF (la priorité dépend du temps que la tâche a passé sur le processeur). On peut par conséquent distinguer une propriété commune aux ordonnanceurs à priorité statique et à la plupart des ordonnanceurs à priorité dynamique :

> **(affectation conservative)** Une fois la priorité d'un travail définie, celle-ci reste constante jusqu'à sa terminaison.

On parle dans la suite d'affectation *conservative* des priorités, et la famille d'ordonnanceurs reposant sur cette propriété est appelée *à priorité fixe à l'échelle du travail*, ou JFP pour *Job-level Fixed Priority*. Dans le cas de l'ordonnancement à priorité statique, cette priorité est celle de la tâche à laquelle le travail appartient.

Nous avons ainsi défini dans Artisst toute une lignée d'ordonnanceurs paramétriques reposant sur cette propriété, adaptés aux systèmes temps-réel prenant en compte des travaux de tâches apériodiques sans ou avec garantie, et sans sous-partie garantie hors-ligne. Cette famille d'ordonnanceurs est inspirée de la lignée EDF/GED/RED de [BS93], et permet éventuellement d'obtenir une garantie en-ligne, éventuellement accompagnée de politiques de rejet variées, pour des affectations de priorités aussi différentes que DM, RM, EDF, FIFO ou une approximation de LLF (décrit ci-dessous et noté *FLLF* par la suite).

Elle est écrite en C++, et définit son propre modèle de tâches. Elle s'appuie sur le mécanisme de paramétrisation du langage C++ (*template*) : une hiérarchie objet correspondant à la lignée EDF/GEDF/RED est fournie, et est paramétrable par une ou plusieurs *fonction(s) de comparaison de priorité*. Le modèle objet associé à cette hiérarchie est représenté dans la figure 8.1. Dans ce modèle, le modèle des tâches et le statut d'exécution des tâches (JFPTaskModel et JFPTask respectivement) héritent des modèles et statuts définis par défaut dans Artisst (st_Task_Model et st_Task_Status respectivement). La lignée d'ordonnanceurs JFP possède les caractéristiques d'un système quelconque (héritage de RTSys), et le spécialise à partir de l'ordonnanceur JFPSys :

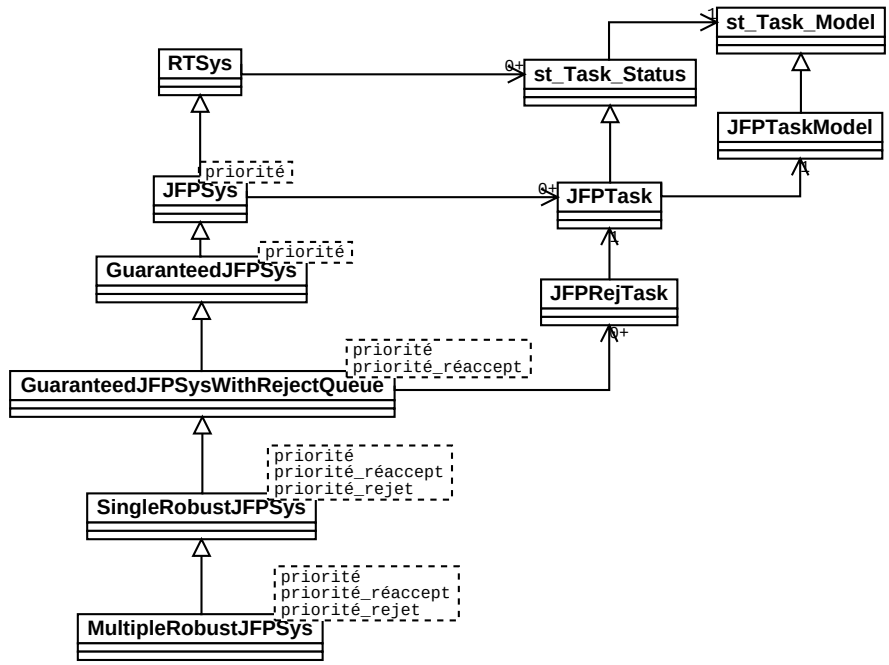


FIG. 8.1: Modèle objet pour la famille d'ordonnanceurs JFP

chacun de ces ordonnanceurs est paramétré, le ou les paramètre(s) étant indiqué(s) sur le schéma dans les boîtes en trait pointillé, et correspondant à une fonction de comparaison de priorité (définie en 8.1.2 ci-dessous). Pour utiliser l'un ou l'autre des ordonnanceurs de cette famille, l'utilisateur d'ARTISST doit dériver le système temps-réel simulé de l'une de ces classes (par héritage), en le paramétrant par la ou les fonction(s) de comparaison de priorité. Les 1, 2 ou 3 fonctions de comparaison de priorité passées en paramètres permettent d'établir 3 niveaux d'affectation des priorités entre les tâches :

- La première (`priorité` sur le schéma) établit l'affectation des priorités pour le test de garantie et les décisions d'ordonnancement (utilisée dans tous les ordonnanceurs de la famille) ;
- La seconde (`priorité_réaccept` sur le schéma) permet d'ordonner la file des tâches refusées, pour les soumettre à une étape de réacceptation/repêchage ultérieure (utilisée dans 3 des 5 ordonnanceurs de la famille) ;
- La troisième (`priorité_rejet` sur le schéma) permet d'ordonner la liste des tâches prêtes, afin de prendre les décisions de rejet (utilisée dans 2 des 5 ordonnanceurs de la famille).

Dans la suite, nous commençons par définir le modèle de tâches (section 8.1.1), puis nous détaillons la notion de fonction de comparaison de priorité conservative (section 8.1.2), et enfin nous détaillons la hiérarchie objet d'ordonnancement JFP (section 8.1.3).

8.1.1 Modèle de tâches

La lignée d'ordonnanceurs décrite ici est inspirée de celle décrite dans [BS93], qui suppose l'affectation des priorités suivant EDF. Nous avons généralisé ce travail à toute affectation conservative des priorités.

Dans leur travail et le nôtre, le modèle de tâches (défini *via* les classes `JFPTaskModel` et `JFPTask` dans le cas d'Artisst) est le suivant :

- Tous les travaux qui sont activés correspondent à des tâches apériodiques indépendantes totalement préemptibles.
- Certains de ces travaux ont une échéance stricte, auquel cas ils sont dites temps-réel, et on suppose leur temps d'exécution pire-cas connu. Suivant l'ordonnanceur choisi dans la famille `JFP`, il peut y avoir garantie en-ligne, ou pas. Les travaux sont ordonnancés préemptivement suivant la fonction de comparaison de priorité qui paramètre l'ordonnanceur.

8.1.2 Fonctions de comparaison de priorité conservatives

Dans la famille d'ordonnancement `JFP`, les tâches sont classées suivant leur priorité. Ce classement est utilisé pour les décisions d'ordonnancement, de garantie de tâches apériodiques, ou de rejet. Afin de pouvoir utiliser les structures de données ordonnées les plus efficaces de la bibliothèque STL[2] (arbres binaires de recherche en particulier), ce classement ne repose pas sur la notion de priorité en tant que telle, mais sur la définition des *fonctions de comparaison de priorité*. Ce sont de simples fonctions booléennes chargées de comparer deux travaux `t1` et `t2` passés en paramètres, en fonction du critère de comparaison choisi (échéance absolue pour EDF, priorité statique pour RM/DM, date de création pour FIFO, ...), et de renvoyer `true` si `t1` est plus prioritaire que `t2`.

Pour la famille JFP, il est **nécessaire** que ces fonctions de comparaison soient *conservatives*. Par définition :

> Une fonction de comparaison de priorité `prio` *conservative* est telle que : si `prio(t1, t2)` vaut `true` (resp. `false`) à un instant donné, alors tant que `t1` et `t2` restent valides, `prio(t1, t2)` continue de valoir `true` (resp. `false`).

Les fonctions de comparaison de priorité conservatives suivantes sont définies par défaut, mais il est bien sûr possible d'en définir d'autres, à condition qu'elles soient conservatives (par exemple, il n'existe pas de fonction de comparaison de priorité conservative pour l'affectation de priorités suivant LLF) :

FP *i.e.* priorité fixe (rassemble DM, RM, et toutes les affectations à priorité fixe) : la priorité est statique définie hors-ligne dans le modèle de tâche ;

EDF : la priorité est liée à l'échéance absolue du travail de la tâche ;

FIFO : la priorité est liée à la date d'activation du travail de la tâche ;

[2] *STL* : *Standard Template Library*, bibliothèque de classes paramétriques accompagnant de nombreux compilateurs `C++`.

FLLF : cas particulier de `FP`, pour l'approximation de LLF. Une tâche est d'autant plus prioritaire que sa laxité initiale, la tâche étant considérée en isolation (*i.e.* différence entre échéance relative et temps d'exécution pire-cas), est petite ;

Value : variante de FP qui effectue les comparaisons de priorité sur un autre champ du modèle de tâche que SP : sur le champ `value`. Cette fonction est utilisée dans le cas des ordonnanceurs fondés sur la politique FP, lorsqu'une politique de rejet est définie, qui repose sur une affectation de type priorité fixe différente de celle choisie pour l'ordonnancement.

Puisque les ordonnanceurs de la famille `JFP` peuvent être paramétrés par 1, 2 ou 3 fonction(s) de comparaison conservative(s) suivant leur type (voir figure 8.1, et les sections qui suivent), à partir de ces 5 fonctions de comparaison de priorités, il est théoriquement possible de définir $5 + 5 + 5^2 + 5^3 + 5^3 = 285$ ordonnanceurs. Nous proposons 25 spécialisations intéressantes parmi elles.

8.1.3 Lignée d'ordonnanceurs implantée

Dans ARTISST, nous fournissons l'arborescence d'ordonnanceurs à 5 niveaux suivante, tous les ordonnanceurs étant paramétrables par une ou plusieurs fonction(s) de comparaison de priorité conservative (voir figure 8.1).

8.1.3.1 **Ordonnanceur basique sans garantie** (JFPSys)

Il s'agit d'un ordonnanceur simple prenant en compte tous les travaux de tâches qui sont activés, mais sans aucune garantie. Il est paramétré par une fonction de comparaison de priorité, et fonctionne de la manière suivante :

– Si au moins un travail avec échéance temps-réel est prêt, celui de plus grande priorité (selon la fonction de comparaison de priorité qui paramètre l'ordonnanceur) est élu.

La complexité algorithmique d'ordonnancement en-ligne est en $O(\log(\text{nombre de tâches}))$, et est due à l'utilisation des structures de données de la STL.

Si tous les travaux qui sont activés appartiennent à des tâches qui ont été garanties par analyse hors-ligne (voir la partie I), cet ordonnanceur s'apparente à DM, RM, ou EDF simple (suivant la fonction de comparaison de priorité choisie) puisqu'il n'y a pas besoin de garantie en-ligne supplémentaire des travaux.

8.1.3.2 **Ordonnanceur à garantie simple** (GuaranteedJFPSys)

Il s'agit d'un ordonnanceur paramétré par une fonction de comparaison de priorité, bâti au dessus du précédent, et qui rajoute une étape de garantie en-ligne pour tous les travaux des tâches avec échéance temps-réel : chaque travail de chaque tâche (qu'elle soit périodique, sporadique, ou apériodique) est individuellement soumis à l'algorithme de garantie. Nous commençons par définir la notion de temps résiduel à la base de l'algorithme de garantie, avant de détailler ce dernier.

Notion de temps résiduel

Par définition :

> Le *temps résiduel* d'un travail donné du système, est la mesure du temps minimal entre la date de terminaison au pire-cas de ce travail et sa date d'échéance, et en tenant compte des travaux de priorité supérieure.

Cette définition suppose que l'affectation des priorités est conservative (voir ci-dessus), afin que la valeur du temps résiduel ne dépende pas de la date à laquelle elle est calculée, tant que *i)* le travail reste présent dans le système, *ii)* aucun travail de priorité supérieure ne termine plus tôt que son temps d'exécution pire-cas, et *iii)* aucun travail de priorité supérieure n'est activé d'ici à l'échéance du travail considéré. Lorsque l'une de ces conditions n'est pas respectée, le temps résiduel des travaux concernés doit être recalculé.

La figure 8.2 propose une illustration des temps résiduels pour un ensemble de trois travaux, calculés à la date 4, et qui restent inchangés jusqu'à la date 8 pour J1, 11 pour J2, et 14 pour J3, à condition que ni J1, ni J2, ni J3 ne terminent plus tôt que leur temps d'exécution pire cas (dans le cas contraire, une ré-évaluation intermédiaire des temps résiduels s'impose).

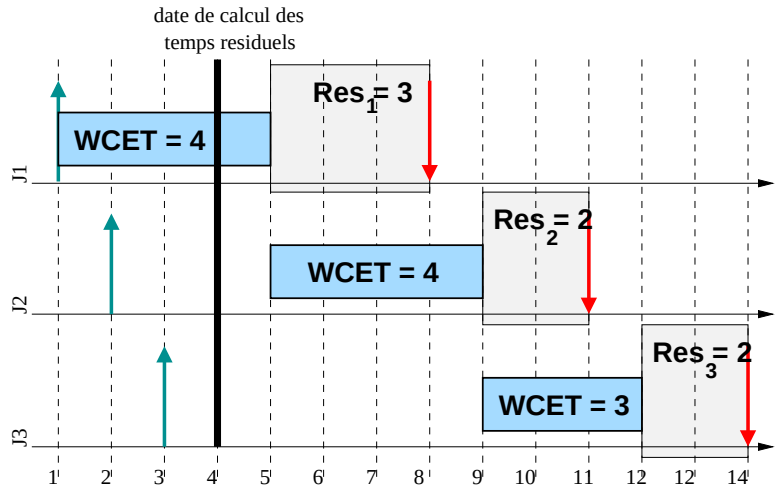


FIG. 8.2: Temps résiduels (zones grisées) pour 3 travaux apériodiques, affectation des priorités suivant EDF.

Dans [BS93], le calcul du temps résiduel Res_i (avec $i \geq 1$) à un instant t, pour un travail temps-réel τ_i, sachant que les travaux sont classés suivant les priorités décroissantes, s'écrit :

$$d_0 = t \quad Res_0 = 0 \quad \text{(Ne correspond à aucun travail réel)}$$
$$i \in \{1 \cdots N\} \quad Res_i = Res_{i-1} + (d_i - d_{i-1}) - c_i$$

Où d_i est l'échéance absolue du travail. Et c_i est le temps d'exécution pire-cas *restant à exécuter* mis à jour par exemple à chaque interruption d'horloge système à partir du temps d'exécution pire-cas C_i et du temps d'exécution effectif mesuré $exec_i$: $c_i =$

$C_i - exec_i$; nous verrons dans le chapitre 8.5 les problèmes que pose la granularité de l'horloge système dans l'établissement de c_i. Dans l'expression ci-dessus, le calcul de Res_i n'est pas très intuitif, et peut sembler spécifique à l'affectation des priorités suivant EDF (car pour d'autres affectations, $d_i - d_{i-1}$ peut être négatif, ce qui est *a priori* difficile à appréhender).

Dans ARTISST, nous sommes partis de la définition littérale des temps résiduels suivante :

$$i \in \{1 \cdots N\} \quad Res_i = d_i - t - c_i - c_{i-1} - \ldots - c_1$$

Il est facile de montrer que cette expression est équivalente à la précédente. Cependant, elle présente l'avantage d'être plus intuitive, d'être plus explicitement indépendante de l'affectation des priorités du moment que celle-ci reste conservative, et enfin de conduire à une complexité algorithmique d'implantation du test d'acceptation sensiblement moindre.

Algorithme de garantie

L'algorithme prend comme entrée un travail à accepter ou à refuser, et la *file des travaux temps-réel* déjà acceptés. On suppose que les travaux de cette file sont classés suivant la fonction de comparaison des priorités passée en paramètre de l'ordonnanceur : le premier élément de la file, d'indice 1, est le travail de plus haute priorité. L'algorithme procède en 3 pas :

1. On cherche à quel rang dans la file des travaux temps-réel déjà présents le nouveau travail viendrait s'insérer, suivant la fonction de comparaison des priorités choisie. Soit i ce rang.
2. On vérifie que le temps résiduel Res_i calculé par l'expression donnée ci-dessus est positif, auquel cas les travaux de priorité supérieure laissent suffisamment de temps libre pour que le travail s'exécute. Sinon, le travail est *refusé*.
3. On vérifie que les temps résiduels Res_j pour $j > i$ sont tous positifs, auquel cas l'introduction du nouveau travail est compatible avec les échéances des autres travaux temps-réel. Dans ce cas, le nouveau travail est *accepté*, c'est-à-dire qu'il est inséré au rang i dans la file des travaux temps-réel. Si non (au moins un travail de priorité inférieur n'a pas suffisamment de marge pour être exécuté dans les délais), le nouveau travail est *refusé*.

Du fait que la fonction de comparaison des priorités est conservative, on est assuré que la file des travaux temps-réel restera classée après que l'algorithme ait été appliqué (que le travail ait été accepté ou refusé).

Puisque les temps résiduels Res_i ne sont calculés qu'au moment où un travail est activé, avec l'expression de Res_i choisie, l'algorithme tient mécaniquement compte de la terminaison en avance des travaux pour pouvoir accepter d'autres travaux. En effet, les travaux qui terminent (que ce soit en avance ou pas) sont aussitôt enlevés de la file des travaux temps-réel, et ne figurent donc plus dans le calcul de Res_i pour les activations de travaux ultérieures.

Si la fonction de comparaison des priorités est EDF, cet algorithme correspond exactement à GED, pour (*Guaranteed Earliest Deadline First*, voir 3.4.2.3 de la partie I).

8.1.3.3 **Ordonnanceur à réacceptation des travaux refusés** (GuaranteedJFPSysWithRejectQueue)

Il s'agit d'un ordonnanceur paramétré par deux fonctions de comparaison de priorité, qui rajoute une étape lors du refus et lors de la terminaison plus tôt de travaux temps-réel pour la récupération des ressources inutilisées. La première fonction de comparaison de priorité permet de paramétrer l'ordonnanceur à garantie sous-jacent vu précédemment. La deuxième fonction de comparaison de priorité sert à l'étape de refus/réacceptation (ou *repêchage*) suivante, pour classer les travaux à réexaminer lors du *repêchage* :

1. Quand un travail est refusé par l'algorithme à garantie précédent, il est mis dans une *file de travaux rejetés*. Cette file de travaux rejetés est classée selon la deuxième fonction de comparaison de priorité fournie en paramétrage de l'ordonnanceur.
2. Quand un travail temps-réel termine plus tôt que son temps d'exécution pire-cas, une routine de réacceptation des travaux rejetés est appelée.
3. La routine de réacceptation des travaux rejetés refait passer le test de garantie décrit précédemment (et qui tient compte de la terminaison plus tôt ainsi que nous l'avons vu plus haut) au *premier* travail sur la file des travaux rejetés. Si le travail est accepté, elle essaye de réaccepter le deuxième travail sur la file des travaux rejetés, etc... Sinon, elle ne fait rien.
4. Périodiquement, les travaux dans la file des travaux rejetés qui n'ont aucune chance de pouvoir être réacceptés (*i.e.* $t > d_i - c_i$) sont définitivement refusés.

Cet ordonnanceur, comme le précédent, permet de récupérer les ressources laissées disponibles par la terminaison plus tôt de travaux temps-réel. Mais, à la différence du précédent, les ressources récupérées servent en priorité à donner une *deuxième chance* à des travaux qui avaient été précédemment refusés. Seulement quand les ressources récupérées ne sont pas suffisantes pour permettre ceci, elles sont récupérées sous la même forme que dans l'ordonnanceur précédent : afin de permettre d'accepter davantage de travaux temps-réel ultérieurs.

8.1.3.4 **Ordonnanceur robuste à politique de rejet simple** (SingleRobustJFPSys)

Il s'agit d'un ordonnanceur paramétré par 3 fonctions de comparaison de priorité, qui est fondé sur l'ordonnanceur à réacceptation de travaux refusés, et qui rajoute une étape lors du refus d'un travail temps-réel. Les deux premières fonctions de comparaison de priorité servent à paramétrer l'ordonnanceur à réacceptation de travaux refusés sous-jacent comme précédemment. La troisième fonction de comparaison de priorité, dite fonction de *valeur*, sert à la politique de rejet, dont le fonctionnement est le suivant :

1. Quand un travail τ_r est refusé, on regarde s'il pourrait être accepté si on supprimait un travail temps-réel τ_s présent dans le système qui a la plus faible *valeur* (*i.e.* suivant la troisième fonction de comparaison de priorité), à condition que celle-ci soit inférieure à celle du travail refusé.
2. Si oui, le travail τ_s ainsi supprimé est mis dans la file des travaux rejetés, et le travail τ_r est accepté. Si non, le travail τ_r subit exactement le même traitement que dans l'ordonnanceur à réacceptation des travaux refusés décrit ci-dessus (*i.e.* il est mis dans la file des travaux rejetés).

Si la première fonction de comparaison de priorité est EDF, et si les deux autres sont la fonction de comparaison de valeur (*value*), cet ordonnanceur correspond à la version à rejet simple de RED (voir 3.4.2.3).

8.1.3.5 **Ordonnanceur robuste à politique de rejet multiple** (MultipleRobustJFPSys)

Il s'agit d'un ordonnanceur comparable au précédent. La différence est qu'au lieu d'essayer de réaccepter une tâche en tentant d'en supprimer une seule de plus faible valeur, on essaye en en supprimant autant que possible.

Si la première fonction de comparaison de priorité est EDF, et les deux autres sont la fonction de comparaison de valeur (*value*), cet ordonnanceur correspond à la version à rejet multiple de RED (voir 3.4.2.3).

8.2 Ordonnanceur à bande passante constante (CBS-JFP)

Nous avons repris les travaux [AB98a] sur le serveur à bande passante constante (*CBS* pour *Constant Bandwidth Server* en anglais, voir 3.3.2.3 de la partie I), pour y ajouter la garantie en-ligne de travaux de tâches apériodiques, en utilisant l'algorithme de garantie de la famille JFP. Il s'agit brièvement d'un ordonnanceur hiérarchique dont le premier niveau est l'ordonnancement des tâches et des serveurs garantis hors-ligne suivant EDF, et dont le deuxième niveau est la garantie et l'ordonnancement des travaux apériodiques dans chacun des serveurs (de type JFP). C'est pourquoi nous appelons cet ordonnanceur : `CBS-JFP`.

Contrairement aux ordonnanceurs de la famille JFP, ce type d'ordonnanceur permet de faire cohabiter tâches garanties hors-ligne et garantie en-ligne de travaux de tâches apériodiques.

8.2.1 Modèle de système

Les tâches serveurs du système répondent à la définition des serveurs donnée en 3.3.2.3 de la partie I. Plus précisément, en ce qui concerne l'implantation dans ARTISST, le modèle est le suivant :

– Des tâches avec des contraintes temps-réel sont garanties hors-ligne, en présence des tâches *serveurs*.

– L'ordonnanceur du système est EDF, chargé d'ordonnancer les tâches et les serveurs garantis hors-ligne.
– Les travaux apériodiques qui sont soumis sont affectés en-ligne à un serveur donné du système. Les serveurs pour tâches apériodiques prennent la forme de travaux dont l'échéance est ajustée suivant les travaux de tâches apériodiques qu'ils servent, et suivant les paramètres du serveur. Lorsque l'ordonnanceur EDF du système élit un serveur, celui-ci est chargé de sélectionner un travail de tâche apériodique qu'il sert, pour exécution immédiate sur le processeur, suivant un algorithme d'ordonnancement de type `JFP` à priorité conservative avec garantie en-ligne. Nous détaillons le fonctionnement d'un serveur dans la section suivante.

8.2.2 Serveurs de tâches apériodiques

Conformément à [AB98a], un serveur correspond à la définition de deux paramètres fournis hors-ligne : une *période* P_S, et une *capacité d'exécution* (ou *budget*) C_S, qui définissent la fraction du processeur $B_S = \frac{C_S}{P_S}$ disponible (la *bande passante*) pour exécuter les travaux des tâches apériodiques assignés au serveur. À partir de ces deux paramètres, et en fonction de la présence ou de l'absence de travaux apériodiques en attente, on définit en-ligne l'échéance du serveur, compatible avec la garantie hors-ligne des autres tâches et serveurs, ce qui permet l'ordonnancement du système suivant EDF.

Dans Artisst, un serveur est un objet `C++` paramétrique qui possède une file de travaux de tâches apériodiques en attente sur le serveur. Cette file comprend deux types de travaux apériodiques : des travaux avec une échéance temps-réel, et des travaux non temps-réel. Le paramètre du serveur est une fonction de comparaison de priorité conservative (voir 8.1.2) utilisée pour classer les travaux apériodiques temps-réel dans la file ; les travaux apériodiques non temps-réel sont classés suivant une affectation du type priorité fixe. Si plusieurs serveurs existent, ils peuvent avoir chacun en paramètre une fonction de comparaison de priorité différente.

Lors de l'activation d'un travail de tâche apériodique temps-réel, il est soumis, vis-à-vis du serveur, à un test d'acceptation similaire à celui de JFP, reposant sur le calcul des temps résiduels (voir 8.1.3), en prenant soin de ne tenir compte que de la fraction du processeur effectivement disponible pour le serveur. Lorsque le serveur est élu par l'ordonnanceur EDF du système, il indique le travail temps-réel de plus haute priorité dans sa file s'il en existe, ou sinon le travail non temps-réel de plus haute priorité s'il en existe. Si la file est vide, le serveur se met en sommeil en attendant la prochaine activation de travail apériodique qui lui sera affectée.

8.3 Ordonnanceur à double priorité (DP)

Cet ordonnanceur, appelé *DP* dans Artisst, est une implantation directe des travaux de [Dav94] (voir 3.3.2.4 de la partie I). Elle permet la garantie en-ligne de travaux de tâches apériodiques en présence de tâches temps-réel garanties hors-ligne ordonnancées en priorité fixe. Les travaux apériodiques sont soumis entre eux à une politique

d'ordonnancement à priorité à l'échelle des travaux conservative, et sont garantis en-ligne s'ils ont une échéance.

Dans ARTISST, l'ordonnanceur se nomme `DP`, et est paramétrique, le paramètre étant la fonction de comparaison de priorité conservative (voir 8.1.2) des travaux des tâches apériodiques temps-réel. L'algorithme de garantie en-ligne des travaux des tâches apériodiques est similaire à celui de JFP (voir 8.1.3), et tient compte de la présence des tâches temps-réel garanties hors-ligne.

8.4 Ordonnanceur TPS

L'ordonnanceur TPS (pour *TaskPair Scheduling*) prend en compte les travaux apériodiques, avec garantie en-ligne, et sans sous-partie garantie hors-ligne : il permet de garantir en-ligne une partie (l'*exception*) d'un traitement dont on ne connaît pas en-ligne le comportement temporel intégral, le reste du traitement (la *tâche principale*) étant pris en charge par une politique d'ordonnancement *au mieux*. L'implantation dans ARTISST est directement issue de [Str95, Net97] (voir 3.4.4 de la partie I).

Dans ARTISST, l'ordonnanceur TPS a un intérêt tout particulier, puisqu'il montre que le modèle de tâches peut être bien différent du modèle de tâches défini par défaut. Ainsi, l'ordonnanceur TPS doit gérer des *taskpairs*, c'est-à-dire des couples *tâche principale+exception*, plutôt que des tâches simples : ceci entraîne l'extension du modèle de tâches et du statut d'exécution afin de définir les notions de tâche principale, d'exception, et afin de leur ajouter leurs relations, représentées par l'objet *taskpair*.

À titre d'exemple, le modèle objet qui est développé pour cet ordonnanceur est donné en figure 8.3 (une flèche fermée indique une relation d'héritage, une flèche ouverte indique une association, un texte précédé d'un dièse indique un attribut privé, et un + indique une méthode publique), qui illustre les capacités d'extension du modèle de tâches. Dans ce modèle, l'ordonnanceur (`TPSched`) possède les caractéristique d'un système quelconque (héritage de `RTSys`), et le spécialise puisqu'il est en charge de gérer non pas directement des travaux, mais des instances des *taskpairs* (`TaskPair_Status`). Ces instances sont formées d'une tâche principale et d'une exception, qui sont en fait des spécialisations (par héritage) d'un objet générique `TaskPair_Task`, lui-même étant une spécialisation par héritage des travaux de tâches ARTISST habituels.

8.5 Prise en compte de la granularité de l'horloge système dans les algorithmes d'ordonnancement avec acceptation en-ligne de travaux apériodiques temps-réel

Dans les sections qui précèdent, nous avons présenté les ordonnanceurs disponibles dans ARTISST en supposant que la date temps-réel et les temps d'exécution effectifs mesurés étaient connus à tout moment. Dans ce chapitre, nous présentons la problématique associée au fait qu'un système informatique n'a en général accès à la notion de temps qu'au travers d'une approximation discontinue du temps-réel : l'échelle de

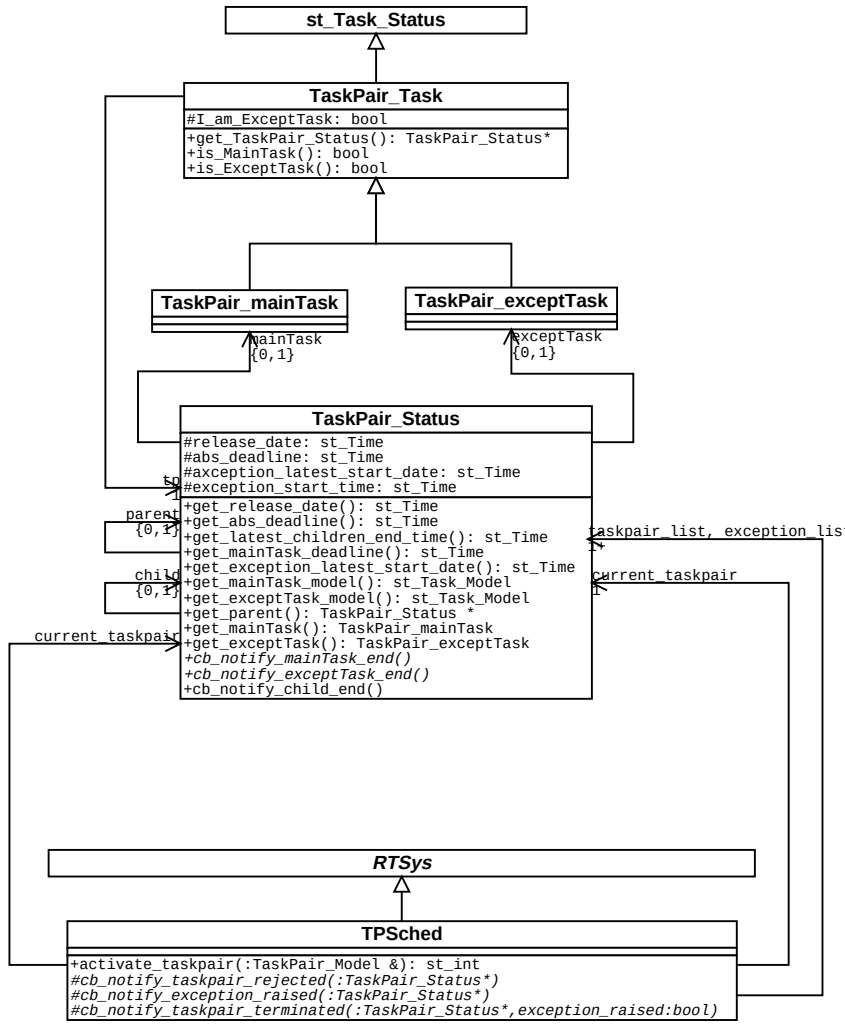


FIG. 8.3: Modèle objet pour l'ordonnancement avec TPS

temps système. Nous dégageons ensuite trois grandeurs affectées par cette approximation, qui sont utilisées dans les algorithmes d'ordonnancement avec acceptation en-ligne de travaux apériodiques temps-réel présentés précédemment. Puis nous indiquons comment la granularité de l'horloge système peut être prise en compte dans l'évaluation de ces grandeurs.

Tous les ordonnanceurs présentés précédemment utilisent les résultats qui suivent, afin de prendre en compte la granularité de l'horloge système.

8.5.1 Problématique

L'horloge système est une grandeur discrète, qui évolue par paliers de hauteur la *granularité* de l'horloge, à chaque interruption d'horloge. Par conséquent, lorsqu'un événement système doit être traité (acceptation de tâche par exemple), le système n'a qu'une vision approximative de la date de l'événement, des durées entre événements, et du temps processeur effectivement consommé par les tâches. Pour fonctionner correctement, tous éléments du système simulé qui effectuent des calculs ou des décisions fondés sur le temps doivent donc tenir compte de cette caractéristique.

En particulier, les algorithmes d'acceptation proposés doivent être revus, de sorte que le système continue de se comporter de façon pessimiste vis-à-vis des grandeurs liées au temps, sans quoi des contraintes temporelles risqueraient d'être dépassées.

Dans la suite, toutes les grandeurs manipulées sont relatives à l'échelle de temps système. Et il s'agit de prendre en compte le fait que l'horloge système qui définit ce temps évolue de façon discontinue, par palier de hauteur la granularité de l'horloge système, supposée constante par rapport au temps-réel.

8.5.2 Grandeurs affectées

Le principe des tests d'acceptation présentés dans les sections précédentes est toujours de calculer le temps résiduel de chaque travail, en tenant compte des tâches déjà garanties, et de manière à vérifier le respect des contraintes temporelles. Le calcul de ce temps résiduel fait intervenir (voir 8.1.3.2) trois types de grandeurs affectées par la marge d'erreur liée à la granularité de l'horloge système :

- Les dates : que ce soit pour déterminer la date courante, ou les échéances absolues ;
- La mesure du temps d'exécution sur le processeur effectivement consommé par un travail : pour déterminer le temps pire-cas d'exécution restant ;
- La mesure du temps d'exécution réservé lors de l'acceptation d'un travail, mais finalement inutilisé suite à la terminaison plus tôt du travail. Cette grandeur est utilisée dans le cas de l'ordonnancement `JFP` à réacceptation des travaux refusés (voir 8.1.3.3).

8.5.3 Dates

La prise en compte de la granularité d'horloge système nécessite :

– Que l'établissement de l'échéance absolue du travail soit fait de façon pessimiste, c'est-à-dire sur la base de la date système de début du *tick* courant (donnée par `get_date()` dans ARTISST) : `échéance_absolue = get_date() + échéance_relative`
– Que le calcul du temps de réponse considère que la date de démarrage du travail au plus tôt correspond à la fin du *tick* d'horloge courant : `démarrage_au_plus_tôt = get_date() + granularité_horloge`. Ceci est dû au fait qu'on ne peut pas savoir si la date courante se situe plus près de la fin du *tick* courant, que du début.

8.5.4 Évaluation du temps d'exécution pire-cas restant à exécuter

En cours de fonctionnement, les systèmes simulés tiennent à jour les temps d'exécution pire-cas restant à exécuter par les différents travaux, afin de calculer au plus juste les temps résiduels. Ceci s'effectue simplement en décrémentant le temps d'exécution pire-cas restant, de la valeur de la granularité de l'horloge système, lors de certains *ticks* d'horloge. Il faut que cette mise à jour s'effectue de façon sûre, et donc que les *ticks* quand la mise à jour s'effectue soient bien choisis.

Si on se contente de décrémenter le temps d'exécution restant de la tâche *en cours* au moment de **chaque** interruption d'horloge, on prend le risque de surestimer la progression de la tâche en cours. Car on ne cherche pas à savoir si elle a occupé le processeur réellement pendant toute la durée du *tick*, ou si elle a été élue juste avant que l'impulsion d'horloge n'apparaisse. En comptabilisant le temps d'exécution restant de la sorte, on prend donc le risque d'accepter des travaux de tâche apériodiques à tort, par surestimation des ressources disponibles.

La figure 8.4 illustre ce phénomène, pour une granularité d'horloge unitaire. Dans le système figure une tâche sporadique τ_1 garantie hors-ligne, de temps d'exécution pire-cas 3, et d'échéance relative 4, et une tâche apériodique temps-réel τ_0 plus prioritaire garantie en-ligne, de temps d'exécution pire-cas 1, et d'échéance relative 2. Dans cet exemple, τ_0 se termine plus tôt que son temps d'exécution pire-cas, ce qui fait qu'au moment du *tick* d'horloge, c'est τ_1 qui est en cours d'exécution. Si à chaque *tick* on décrémente d'une unité le temps d'exécution pire-cas de la tâche *en cours*, on accepte à tort le deuxième travail de τ_0 (date 3) puisque les calculs de temps de réponse de τ_1 sont effectués sur la base du temps d'exécution restant de τ_1 égal à 1, alors qu'en réalité τ_1 n'a pas bénéficié du processeur pendant 2 unités de temps pleines. Il en résulte un dépassement d'échéance de τ_1 à la date système 5, alors que τ_1 est censé avoir consommé plus de temps processeur que requis (dès la date 4).

Les systèmes évalués doivent par conséquent surestimer les temps d'exécution pire-cas restants. Une méthode sûre est de considérer qu'il est toujours égal au temps d'exécution pire-cas initial (WCET). Dans ARTISST, afin d'être moins pessimistes, nous profitons de la progression des tâches, mais en la sous-évaluant afin de garantir la sûreté de l'approche : les systèmes ne décrémentent le temps d'exécution pire-cas des tâches restant lors des *tick* d'horloge, que si la tâche n'a jamais été préemptée pendant la durée du *tick* d'horloge qui vient de s'écouler.

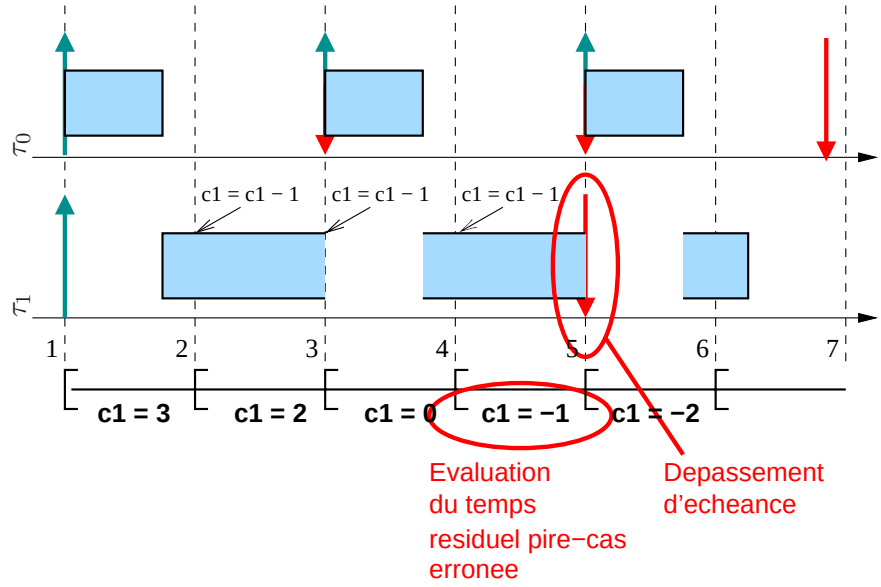


FIG. 8.4: Acceptation de tâche à tort par surestimation des ressources disponibles

8.5.5 Évaluation du temps non utilisé

Lorsqu'une tâche se termine plus tôt que prévu, on souhaite profiter du temps laissé disponible pour ré-accepter d'autres tâches. Pour certains ordonnancements, on a besoin de connaître le temps d'exécution ainsi récupéré. Ce temps non utilisé vaut tout simplement le temps d'exécution restant, mesuré lorsque la tâche se termine. D'après les remarques formulées dans le paragraphe précédent, il faut surestimer le temps pire-cas restant, donc si on part de cette mesure pour définir le temps non utilisé par la terminaison plus tôt du travail, le problème est alors qu'on le surestime aussi.

Pour évaluer de façon sûre le temps non utilisé qu'on peut récupérer suite à la terminaison plus tôt d'une tâche, il faut donc partir d'une vision sous-estimée du temps d'exécution restant. Par exemple, une solution triviale est de considérer que ce temps d'exécution restant vaut toujours 0, auquel cas on est assuré de ne pas pécher par optimisme, mais on ne récupère jamais aucune ressource inutilisée. Dans ARTISST, nous avons choisi l'approche complémentaire à celle vue précédemment : à chaque *tick* d'horloge, on décrémente (de la valeur de la granularité de l'horloge système) le temps d'exécution restant de toutes les tâches qui ont été exécutées au moins une fois pendant la durée du *tick* d'horloge qui vient de s'écouler (en veillant à ce qu'il ne devienne jamais négatif) : `temps_restant = max(temps_restant - granularité_horloge, 0)`.

Dans tous les systèmes à garantie en-ligne que nous évaluons, les tâches sont indépendantes, et nous utilisons des algorithmes d'ordonnancement à priorité qui respectent un ordre conservatif en cas de priorités égales (ordre FIFO). Il suffit donc de considérer la tâche courante au moment du *tick*, car on est assuré qu'elle est la seule du système qui *i)* a pu s'exécuter pendant cet intervalle, et *ii)* existe encore dans le système. En

effet, d'autres tâches ont certes pu s'exécuter, mais alors nécessairement elles avaient plus haute priorité, et donc nécessairement elles se sont terminées durant cet intervalle ; sinon ce serait l'une d'entre elles qui serait la tâche courante.

troisième partie

Cas d'étude

Dans cette partie, nous présentons quelques résultats d'évaluation par simulation avec ARTISST, relatifs à l'impact de la granularité de l'horloge système sur le comportement de différentes politiques d'ordonnancement à acceptation dynamique de tâches apériodiques temps-réel. Dans un premier temps, nous ne tenons pas compte des surcoûts d'exécution du système (chapitre 9), puis ensuite nous donnons quelques résultats en les prenant en compte (chapitre 10).

Chapitre 9

Influence de la granularité de l'horloge système

Dans un premier temps (section 9.1), nous décrivons le protocole de simulation pour évaluer l'influence de la granularité d'horloge système sur le comportement de systèmes monoprocesseur soumis à un flot de requêtes apériodiques avec contraintes temporelles, et ordonnancés par différents ordonnanceurs à garantie en-ligne. Nous présentons ensuite les résultats de deux campagnes d'évaluation : la première pour un *facteur de recouvrement* (décrit en début de section) faible (section 9.2), la seconde pour un facteur de recouvrement plus élevé (section 9.3).

9.1 Dispositif expérimental

Dans tout ce chapitre, tous les systèmes que nous évaluons ont les caractéristiques suivantes :

- ils sont monoprocesseur ;
- les tâches sont synthétiques ;
- les tâches sont indépendantes ;
- les surcoûts d'exécution du support d'exécution simulé sont négligés ;
- la loi d'activation de toutes les tâches est identique pour chaque vague de simulations (indiquée en début de chaque section).

Dans un premier temps, nous considérons que toutes les tâches sont apériodiques temps-réel garanties en-ligne. Dans un deuxième temps, nous rajoutons un ensemble de tâches temps-réel périodiques garanties hors-ligne.

9.1.1 Paramètres de chaque simulation

Chaque simulation prend en entrée trois paramètres :

- l'ordonnanceur ;
- la granularité de l'horloge système (liée à sa fréquence) : varie entre 1ms et 100ms (respectivement 1kHz et 10Hz) ;

– la *charge individuelle moyenne* des travaux, *load*. Il s'agit de la charge individuelle (*i.e.* $u_i = c_i/d_i$, avec c_i le temps d'exécution effectif et d_i l'échéance relative) moyenne des travaux activés dans le système. Cette charge est fournie entre 0.1 et 0.9.

9.1.2 Description de l'environnement

La simulation utilise le module de simulation de système temps-réel ARTISST, auquel sont connectés en entrée deux modules de génération aléatoire d'événements, pour :

– la génération de l'interruption d'horloge système. La loi de distribution probabiliste associée est la loi constante qui définit la granularité d'horloge, fournie en paramètre de chaque simulation.
– Le module de génération des activations de tâches, dont le traitant d'interruption associé active les travaux des tâches. La loi de génération de ces interruptions est définie en début de section pour chaque vague de simulations.

9.1.3 Description du système simulé

Le traitant d'interruption d'horloge se limite à tenir à jour la date système, le temps d'exécution pire-cas restant, et le temps meilleur-cas restant nécessaire au calcul du temps non utilisé en cas de fin de tâche plus tôt (voir section 8.5.5). Le traitant de l'interruption de génération des activations des tâches active les tâches apériodiques temps-réel synthétiques dont :

– le temps d'exécution pire-cas est généré sous la forme d'une loi définie en début de section pour chaque vague de simulation ;
– le temps d'exécution effectif c_i de la tâche (synthétique : ne fait qu'un appel à `hold_cpu()`) est généré sous la forme d'une loi dépendante du temps pire-cas, et définie en début de section pour chaque vague de simulations ;
– l'échéance relative est calculée à partir du temps d'exécution effectif déterminé précédemment, de telle sorte que la charge individuelle moyenne soit égale à la valeur spécifiée : $d_i^* = \frac{c_i}{load}$. L'échéance relative est alors calculée sous la forme d'une distribution centrée sur cette valeur moyenne, et définie en début de section pour chaque vague de simulations.

Suivant les ordonnanceurs étudiés, les travaux ainsi activés peuvent être soumis à un test d'acceptation, être rejetés, ou être mis sur une file d'attente pour réacceptation ultérieure.

9.1.4 Mesures

Pour chaque configuration simulée (ordonnanceur, granularité d'horloge, charge individuelle moyenne des travaux), on mesure le taux de garantie des travaux apériodiques obtenu au bout de 7 jours en temps-réel simulé.

Les résultats sont représentés sous la forme de surfaces (une par ordonnanceur évalué), dont la base est quadrillée par la granularité d'horloge considérée et la charge

moyenne individuelle des travaux, et dans lesquelles chaque point représente une configuration simulée. Des lignes de niveaux sont projetées sur la base (la légende "`GR=...`" figure le taux de garantie associé) pour pouvoir comparer quantitativement les surfaces.

9.1.5 Performances de simulation

Pour chaque configuration, le temps de simulation dépend de la machine hôte et de la granularité d'horloge simulée. Sur un parc hétérogène de 10 machines (4 Sparc Ultra10 440MHz, 1 Sparc Ultra60 monoprocesseur 360MHz, 1 Sparc Ultra60 biprocesseur 2x360MHz avec 2 simulations, 2 Pentium IV 2GHz, 1 Pentium III biprocesseur 2x500MHz avec 2 simulations, 1 Pentium III 450MHz) fonctionnant 12 heures par jour (mise en sommeil pendant les heures ouvrables), les 1200 simulations environ ont nécessité près de deux mois et demi. Les plus courtes (grosse granularité d'horloge) n'ont duré qu'une poignée de minutes sur les machines les plus rapides (Pentium IV), les plus longues (petite granularité d'horloge) pouvaient nécessiter jusqu'à 3 jours pleins sur les machines les moins rapides (Sparc).

9.2 Configuration à faible *facteur de recouvrement*

Dans un premier temps, nous avons sélectionné une loi sur les temps pire-cas d'exécution des travaux telle que, en moyenne, le ratio $\frac{WCET_{moyen}}{délai\_inter\_arrivée_{moyen}}$ soit proche de 1. Nous disons que cette configuration est à *faible facteur de recouvrement*.

Intuitivement, le *facteur de recouvrement* est proportionnel à la probabilité qu'une tâche soit activée pendant qu'une autre est en cours d'exécution. Il s'agit d'une donnée globale au système, qui est complémentaire à la charge individuelle moyenne des travaux *load* (qui représente une mesure locale), dans la mesure où la connaissance de ces deux grandeurs est une indication de la charge globale du système : à charge individuelle moyenne donnée, la charge du système est d'autant plus forte que la probabilité d'activation d'une tâche pendant qu'une autre s'exécute est grande, c'est-à-dire que le facteur de recouvrement est élevé.

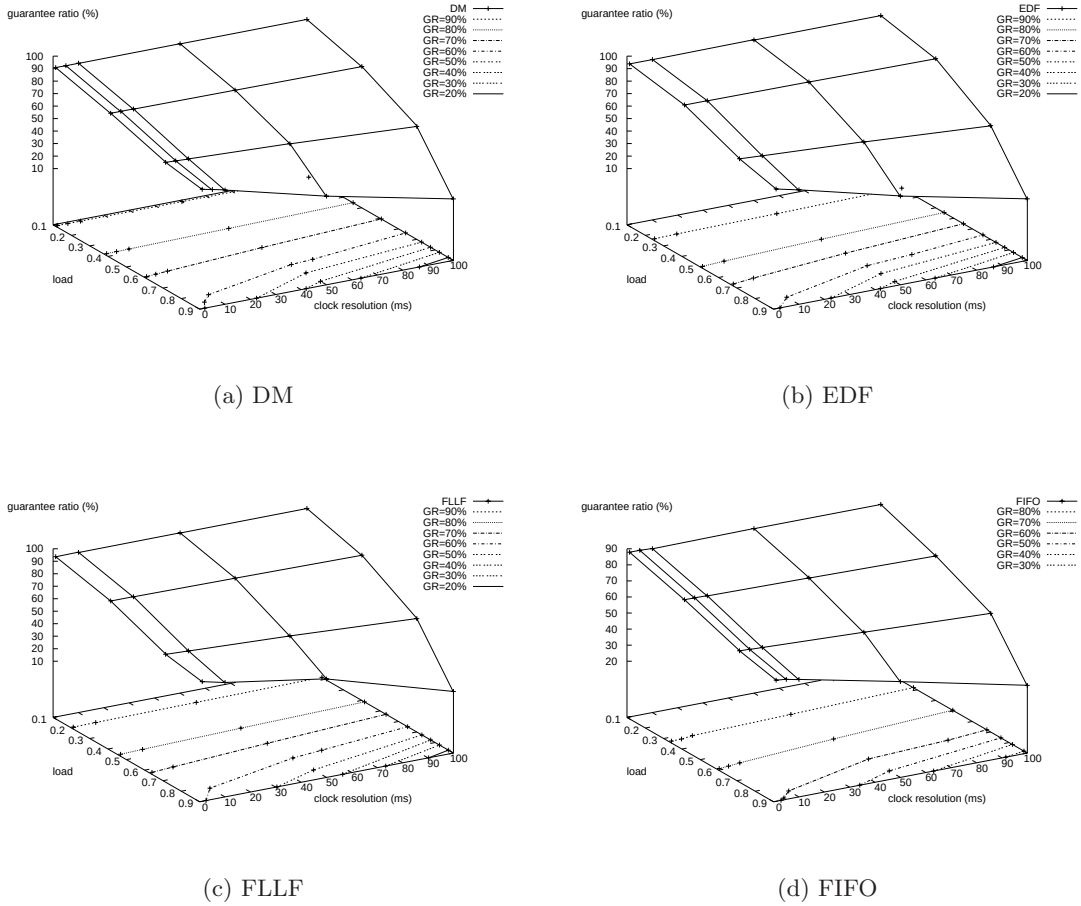
9.2.1 Configurations sans tâche garantie hors-ligne

Les simulations sont toujours paramétrées par l'ordonnanceur, la granularité de l'horloge système, et la charge individuelle moyenne. Les paramètres suivants sont identiques pour toute la vague de simulations de cette section :

- les travaux sont activés suivant la loi normale centrée sur 1 seconde, et d'écart type 450ms ;
- les temps d'exécution pire-cas $WCET$ des travaux des tâches apériodiques sont générés suivant la loi uniforme sur [0.3s, 2s] ;
- les temps d'exécution effectifs c_i dépendent du $WCET$ établi, et sont générés suivant la loi uniforme sur $[WCET - 0.2s, WCET]$;
- l'échéance de chaque travail est générée suivant la loi normale centrée sur $\frac{c_i}{load}$, et d'écart-type 0.1s.

Nous évaluons ici les ordonnanceurs de la famille JFP (voir 8.1 de la partie précédente) avec les quatre affectations de priorités d'ordonnancement DM, EDF, FLLF, ou FIFO (les deux autres priorités nécessaires pour le paramétrage d'une partie des ordonnanceurs de la famille, à savoir celles de réacceptation et de rejet, sont de type FIFO) ; et l'ordonnanceur TPS.

9.2.1.1 Ordonnanceurs à garantie sans réacceptation ni politique de rejet



(a) DM

(b) EDF

(c) FLLF

(d) FIFO

FIG. 9.1: Taux de garantie pour les ordonnanceurs à garantie selon différentes affectations des priorités

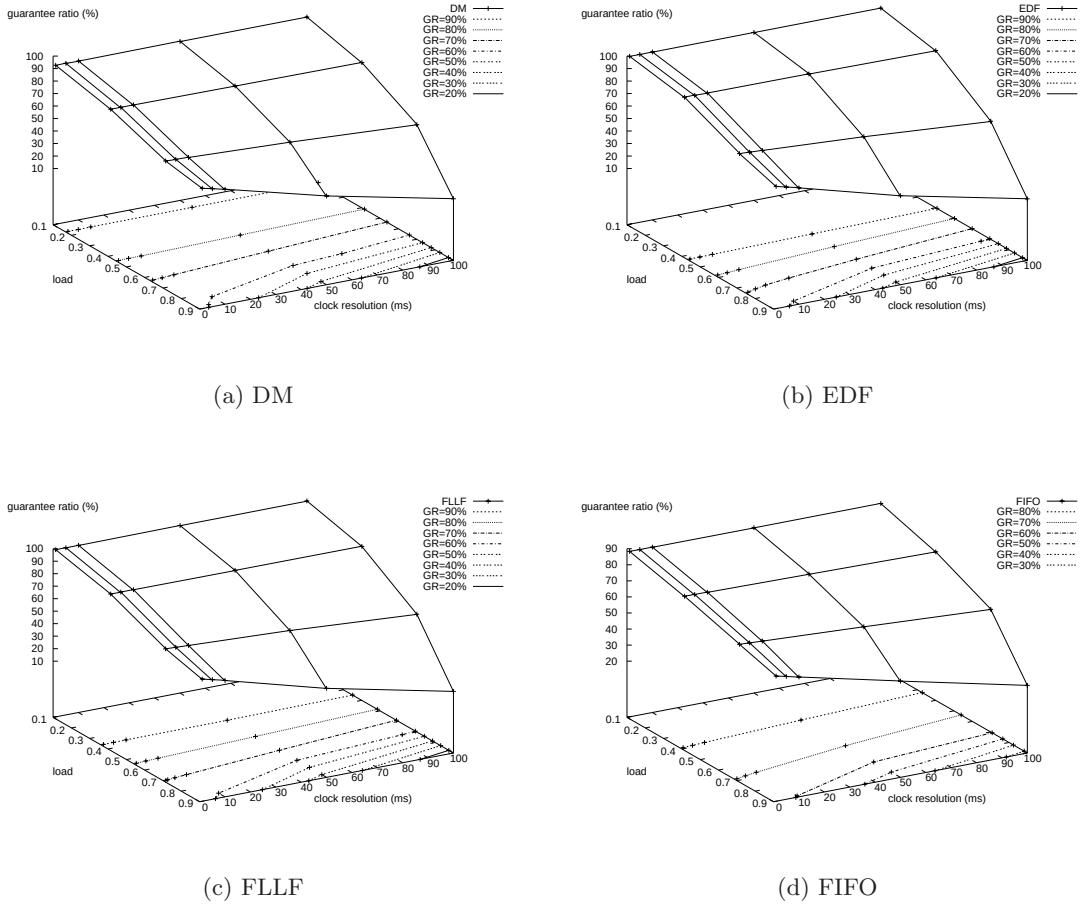
Pour ces politiques d'affectation des priorités, la tendance générale (toute naturelle) est que le taux de garantie décroît d'autant plus que la charge individuelle moyenne est élevée, ou que la granularité de l'horloge système est grosse. À faible charge, la granularité de l'horloge système intervient peu sur les décisions d'acceptation, mais plus cette charge augmente, et plus l'effet de la granularité de l'horloge se ressent

fortement. Par exemple pour l'ordonnanceur EDF, à charge 0.1 la différence des taux de garantie pour les granularités d'horloge 1ms et 100ms est négligeable (de l'ordre de 0.01%) ; elle est par contre de l'ordre de 46% pour la charge 0.9.

Ce comportement, qui continuera de s'appliquer dans les simulations de ce type, montre l'intérêt de prendre la granularité d'horloge en considération dans les travaux en ordonnancement, en particulier lorsque le système est soumis à une charge individuelle moyenne des travaux élevée. L'influence de la charge individuelle moyenne sur le taux de garantie est cependant prépondérante ici, puisqu'on observe, sur EDF par exemple, une chute du taux de garantie d'environ 79% lorsque la charge varie de 0.1 à 0.9 (granularité 100ms).

D'autre part, dans notre configuration, les ordonnancements étudiés sont assez égaux face aux variations de la granularité de l'horloge système. Ils sont par contre inégaux, mais faiblement, face aux variations de la charge individuelle moyenne des travaux : à faible charge, EDF se comporte légèrement mieux que FLLF (meilleur taux de garantie, de l'ordre de 1 à 3% d'écart), qui se comporte mieux que DM (de l'ordre de 2 à 3% d'écart) ; plus la charge augmente, et plus les surfaces se rapprochent (jusqu'à se confondre à 0.2% près pour la charge 0.9).

9.2.1.2 Ordonnanceur à réacceptation des travaux refusés, sans politique de rejet



(a) DM

(b) EDF

(c) FLLF

(d) FIFO

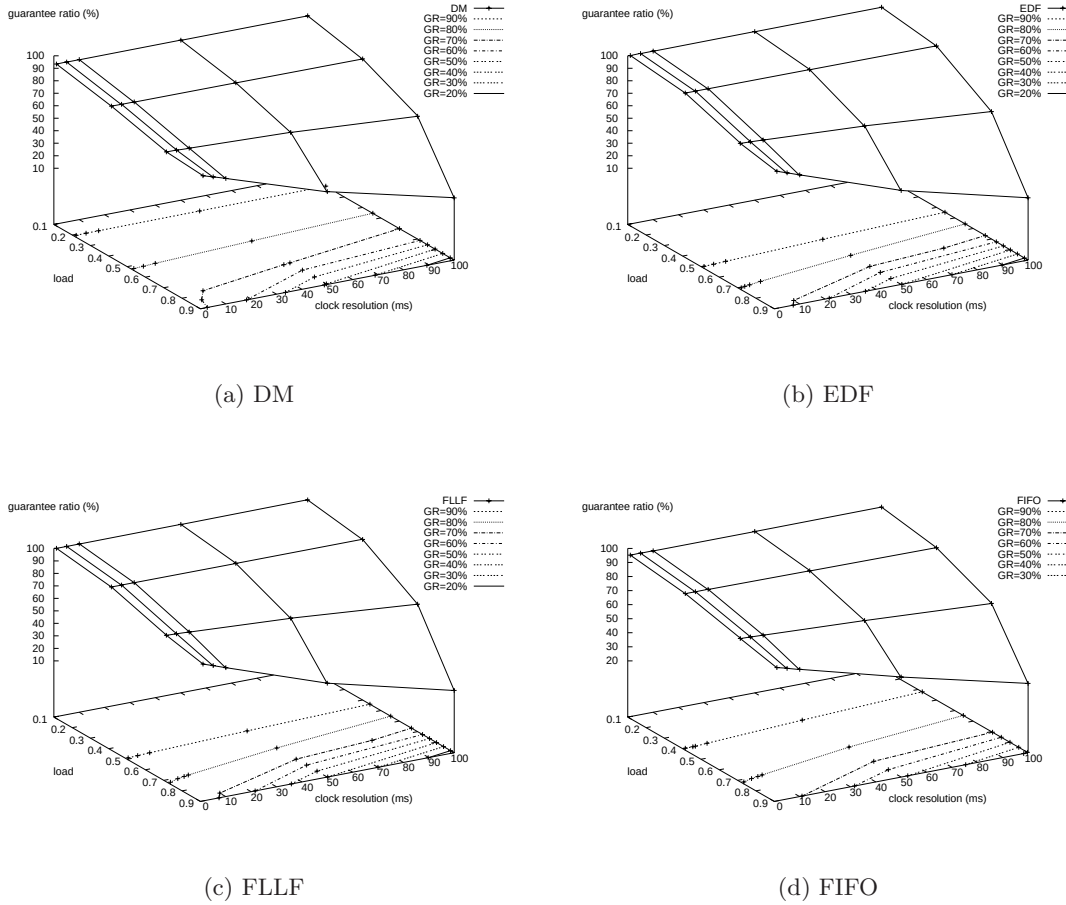
FIG. 9.2: Taux de garantie pour les ordonnanceurs à réacceptation des travaux refusés selon différentes affectations des priorités

Nous observons la même tendance générale que dans le cas avec garantie simple, légèrement translaté puisque le taux de garantie est plus élevé que dans le cas à garantie sans réacceptation : l'écart observé entre les deux surfaces est de 2% à 6% à faible charge suivant les ordonnanceurs (indépendamment de la granularité de l'horloge), et décroît à forte charge de façon dépendante de la granularité d'horloge (0.1% à grosse granularité, 2% à granularité fine).

D'une manière générale, par rapport aux ordonnanceurs sans réacceptation, ceux-ci présentent donc un taux de garantie plus élevé, mais l'influence de la granularité d'horloge se fait légèrement plus sensible, surtout à forte charge (pour EDF à la charge de 0.9 par exemple, l'écart entre les taux d'acceptation pour des granularités d'horloge

de 1 et 100ms est de 48.5% au lieu des 46% précédemment). Et l'influence de la charge individuelle moyenne continue de s'accroître et d'être largement prépondérante (chute de 85% pour EDF entre 0.1 et 0.9 pour la granularité 100ms).

9.2.1.3 Ordonnanceur robuste à politique de rejet simple



(a) DM

(b) EDF

(c) FLLF

(d) FIFO

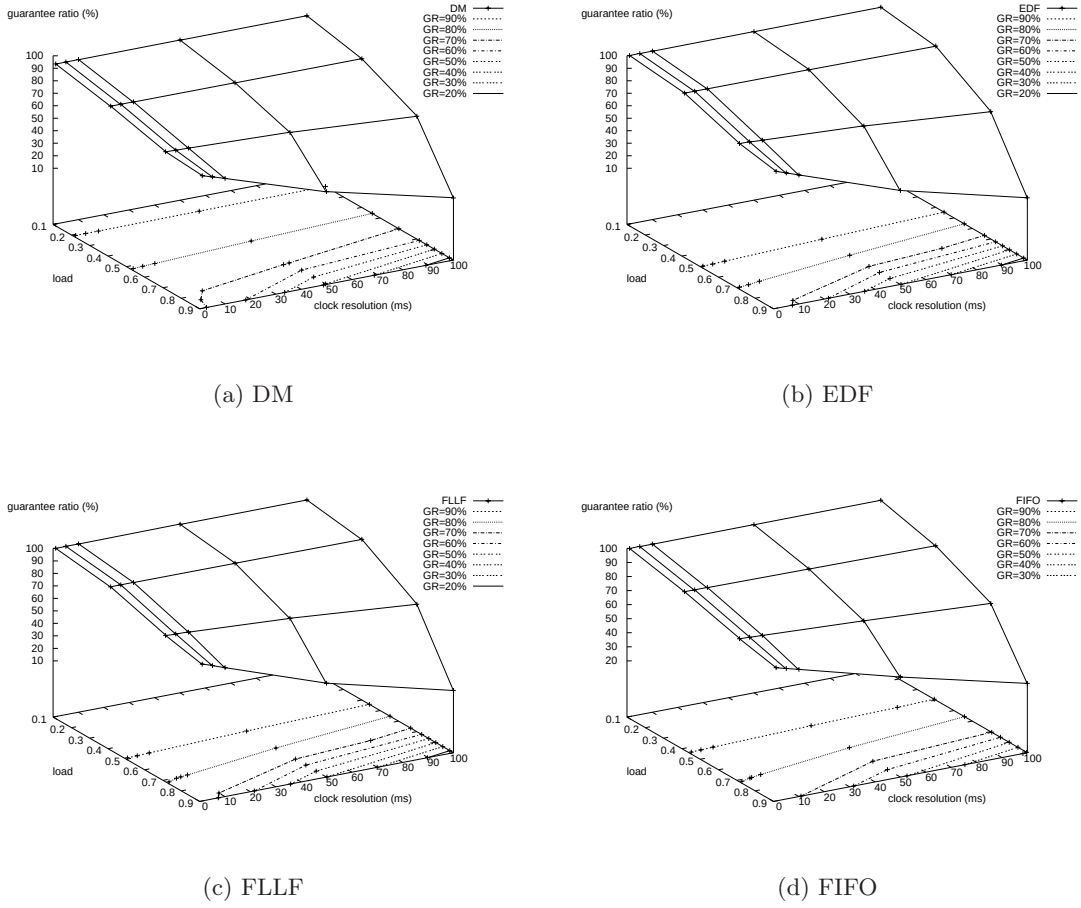
FIG. 9.3: Taux de garantie pour les ordonnanceurs à politique de rejet simple selon différentes affectations des priorités

La tendance générale se conserve à nouveau, et on observe encore une translation par rapport à la surface précédente (réacceptation sans politique de rejet). Pour chaque ordonnanceur, cet écart entre les deux surfaces est faible à faible charge et/ou à granularité grossière d'horloge (entre 1 et 5%). Par contre, dans le cas d'une charge individuelle moyenne élevée (0.9) en présence d'une granularité d'horloge fine (1ms), la différence en faveur de ce type d'ordonnanceur est proche de 10%.

Ainsi, pour ce type d'ordonnanceur, l'influence de la granularité d'horloge est encore

plus sensible que précédemment : par exemple, pour EDF avec une charge de 0.9, entre les granularités 1ms et 100ms, l'écart sur le taux d'acceptation est cette fois de l'ordre de 59.7%, au lieu des 48.5% précédemment constatés.

9.2.1.4 Ordonnanceur robuste à politique de rejet multiple



(a) DM

(b) EDF

(c) FLLF

(d) FIFO

FIG. 9.4: Taux de garantie pour les ordonnanceurs à politique de rejet multiple selon différentes affectations des priorités

Avec la configuration du système choisie, aucune différence remarquable n'apparaît par rapport à la politique à rejet simple, quelle que soit l'affectation des priorités choisie : l'écart entre les deux surfaces est en faveur de la politique à rejet multiple, mais est de l'ordre de 0.01% à 0.05%.

Ceci signifie que dans notre cas, le rejet d'une seule tâche suffit dans la grande majorité des cas. C'est la raison pour laquelle nous avons voulu tester le système dans une autre configuration : nous verrons les résultats obtenus dans la section suivante.

9.2.1.5 Ordonnanceur TPS

Le dispositif expérimental est légèrement modifié pour TPS, puisque le modèle de tâches n'est pas le même que dans les autres systèmes : il s'agit là d'accepter une tâche dont on ne connaît pas le comportement temporel intégral. La phase d'acceptation d'une tâche se limite pour cela à l'acceptation d'une tâche associée et parfaitement caractérisée : l'exception. Pour l'évaluation de TPS, il a été choisi que l'exception ait un temps d'exécution pire-cas formant 10% du temps pire-cas généré pour une tâche JFP équivalente. Il en découle que les résultats de cette évaluation ne sont pas comparables avec les précédents en termes de données chiffrées.

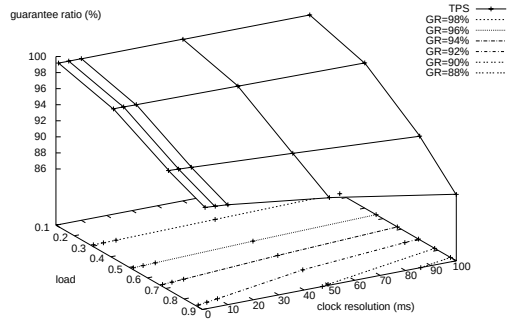


FIG. 9.5: Taux de garantie pour TPS

Cependant, comme dans la famille JFP, la tendance est conservée, à savoir que le taux de garantie est davantage sensible à l'évolution de la charge, qu'à la granularité de l'horloge système ; mais la sensibilité à la granularité de l'horloge s'accroît avec la charge. En effet, à granularité d'horloge de 1ms, la diminution du taux de garantie en fonction de la charge est de l'ordre de 7.5%, et passe à 11.8% pour une granularité d'horloge de 100ms.

9.2.2 Configurations avec tâches garanties hors-ligne

Il s'agit maintenant d'intégrer dans le système un ensemble de tâches garanties hors-ligne :

- les travaux apériodiques ont le même comportement que précédemment ;
- la charge garantie hors-ligne est constituée de deux tâches périodiques :
  - de périodes respectives 8 et 20 secondes,
  - d'échéances relatives 6 et 18 secondes,
  - de temps d'exécution pire-cas 2 et 3 secondes,
  - de temps d'exécution effectifs suivant la loi uniforme entre 0 et le temps pire-cas.

Ces tâches sont générées par deux interruptions spéciales, et leur activation ne dépend donc pas de l'interruption d'horloge.

Puisque le dispositif expérimental est modifié, les résultats des évaluations qui suivent ne sont pas comparables avec les précédents en termes de mesures quantitatives.

Les deux ordonnanceurs que nous comparons (DP et CBS-JFP) sont de type hiérarchique : les tâches apériodiques sont ordonnancées entre elles suivant un ordonnanceur de type JFP qui prend en compte les tâches garanties hors-ligne. Nous nous limitons à une seule spécialisation de cet ordonnancement : EDF avec garantie en-ligne des travaux apériodiques sans réacceptation ni rejet.

9.2.2.1 Ordonnanceur à réquisition de temps-creux `DP`

Les tâches temps-réel garanties hors-ligne sont ordonnancées ici suivant DM.

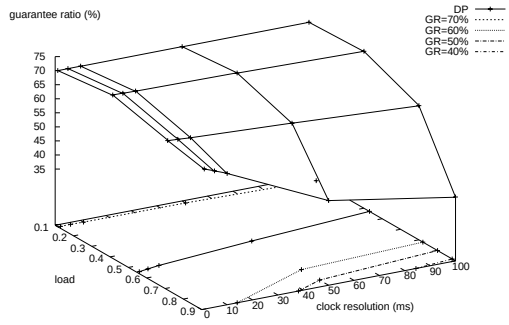


Fig. 9.6: Taux de garantie pour DP

Dans le comportement de cet ordonnanceur, les caractéristiques de l'ordonnanceur EDF à garantie sous-jacent (responsable de l'ordonnancement des tâches apériodiques garanties en-ligne) semblent prépondérantes. En effet, la tendance générale de EDF à garantie se retrouve, cependant que les données chiffrées sont différentes. À charge individuelle moyenne élevée, l'influence de la granularité de l'horloge joue à hauteur de 27% sur le taux de garantie, et cette influence a tendance à s'estomper très sensiblement à partir de la granularité 10ms. L'influence de la charge individuelle moyenne est presque comparable, à hauteur de 32% lorsqu'elle varie de 0.1 à 0.9.

9.2.2.2 Ordonnanceur à base de serveurs `CBS-JFP`

Les tâches garanties hors-ligne et les serveurs sont ordonnancés ici suivant EDF. Les tâches apériodiques sont prises en charge par un unique serveur de capacité 10ms et d'échéance relative 20ms.

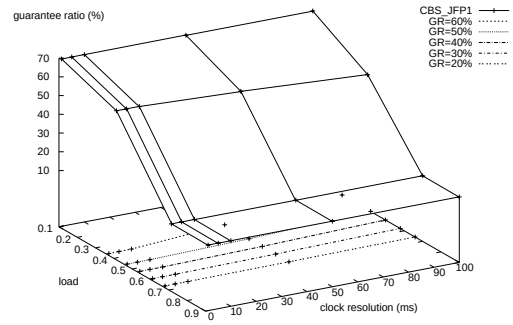


FIG. 9.7: Taux de garantie pour CBS-JFP

La tendance générale observée sur la plupart des autres ordonnanceurs se confirme : le taux de garantie diminue avec la charge (de 54% environ entre les charges 0.1 et 0.9). Cependant, d'une part l'influence de la granularité de l'horloge système apparaît très localement sur son comportement, ce qui est un point de différenciation avec l'ordonnanceur EDF à garantie sous-jacent important : l'impact est de l'ordre de 5% pour la charge individuelle moyenne 0.4, et de 0.1% le reste du temps. Et d'autre part, le taux de garantie diminue fortement avant de se stabiliser à charge élevée (présence d'un palier).

9.3 Configuration à *facteur de recouvrement* plus élevé

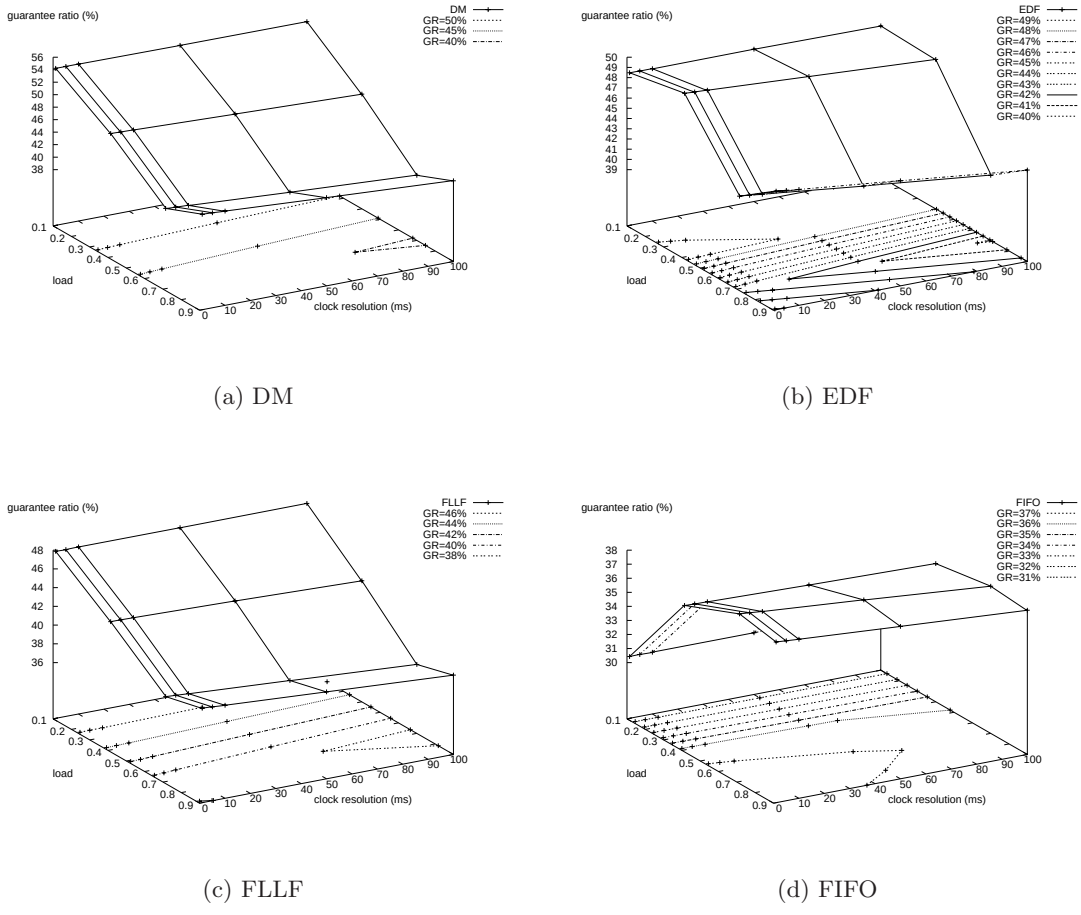
Dans cette section, nous reprenons la même série d'évaluations, avec un rapport $\frac{WCET_{moyen}}{délai\_inter\_arrivée_{moyen}}$ plus élevé, de l'ordre de 3 au lieu de 1, ce qui multiplie d'autant la probabilité qu'une tâche soit activée pendant qu'une autre s'exécute : à charge individuelle moyenne donnée, la charge globale du système est par conséquent plus élevée que précédemment.

Plus précisément, les paramètres suivants sont identiques pour toute la vague de simulations de cette section :

- les travaux sont activés suivant la loi normale centrée sur 1 seconde, et d'écart type 450ms ;
- les temps d'exécution pire-cas $WCET$ des travaux des tâches apériodiques sont générés suivant la loi uniforme sur [0.3s, 6s] (au lieu de [0.3, 2s]) ;
- les temps d'exécution effectifs c_i dépendent du $WCET$ établi, et sont générés suivant la loi uniforme sur $[\max(0, WCET - 2s), WCET]$ (au lieu de $[WCET - 0.2s, WCET]$;
- l'échéance de chaque travail est générée suivant la loi normale centrée sur $\frac{c_i}{load}$, et d'écart-type 1 seconde (au lieu de 0.1s de la section 9.2).

9.3.1 Configurations sans tâche garantie hors-ligne

9.3.1.1 Ordonnanceurs à garantie sans réacceptation ni politique de rejet



(a) DM

(b) EDF

(c) FLLF

(d) FIFO

FIG. 9.8: Taux de garantie pour les ordonnanceurs à garantie selon différentes affectations des priorités

Par rapport à la configuration à faible recouvrement précédente, la première observation est que le taux de garantie est considérablement diminué (de près de 50%), beaucoup moins sensible à la variation de la charge individuelle moyenne des travaux, et moins sensible à la granularité de l'horloge système.

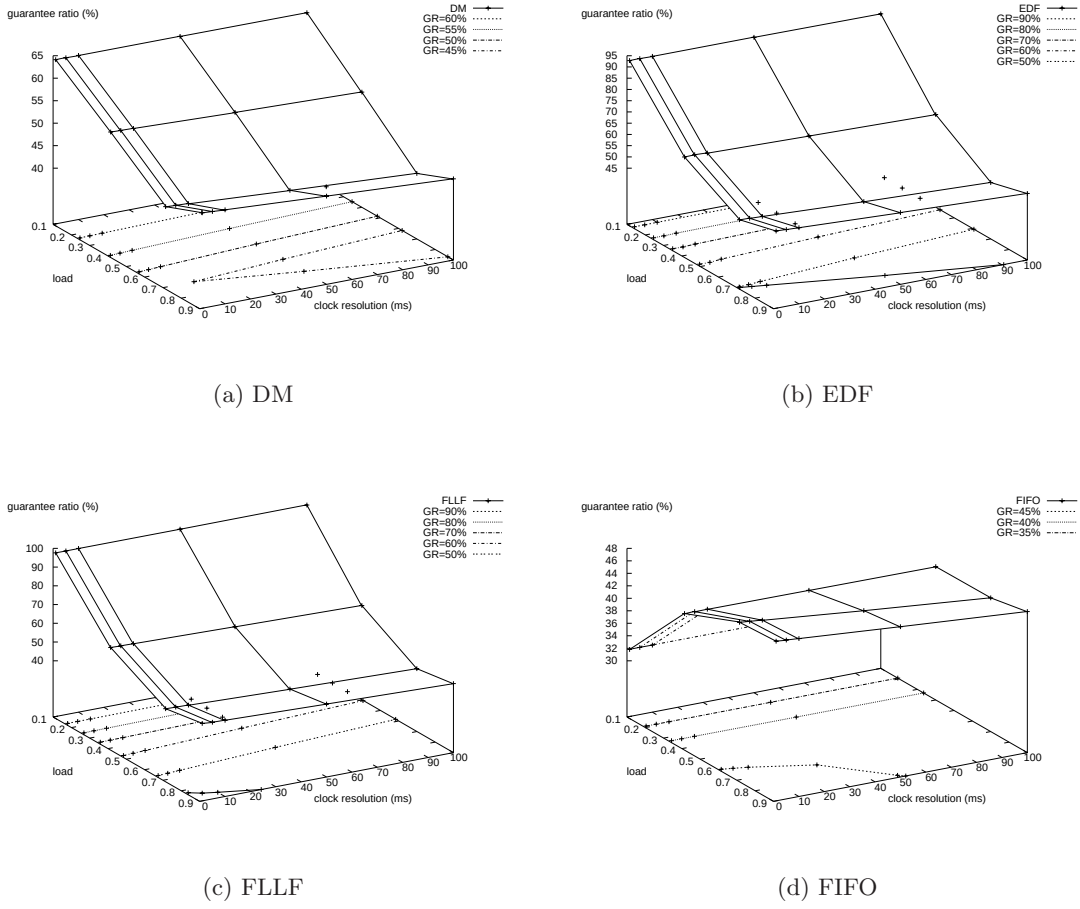
D'une part, la distinction entre les différents ordonnanceurs est cette fois nettement plus marquée : les ordonnanceurs à priorité fixe (DM et FLLF) ont un comportement plus régulier à faible charge, comparé à EDF qui présente un palier avant de décroître. Mais tous ces ordonnanceurs ont en commun qu'à partir d'une charge modérée (0.4), leur comportement est similaire : le taux de garantie décroît avec la charge, avant de se

stabiliser ou de croître légèrement à charge élevée. Cependant, ces fluctuations restent relativement faibles par rapport au cas à recouvrement (amplitude inférieure à 12% pour DM, 9% pour FLLF, 6% pour EDF). Ainsi, à faible charge, EDF et FLLF sont assez proches (indépendamment de la granularité d'horloge, à moins d'1% près), DM se comportant mieux (à 5% près). Quand la charge augmente, EDF se rapproche de DM (à moins d'1% près), alors que DM et FLLF ont tendance à décroître, en conservant un écart de 3 à 5% en faveur de DM.

En ce qui concerne FIFO, d'une part son comportement est maintenant tout à fait remarquable : le taux de garantie d'autant plus faible que la charge est faible, et tend à augmenter avec la charge. Cependant, cette remarque est à relativiser par rapport aux autres ordonnanceurs, puisque le taux de garantie est dans l'ensemble plus uniforme (amplitude des fluctuations bien moins élevée : de l'ordre de 6%), et sensiblement plus faible que pour les ordonnanceurs précédents (10% plus faible en moyenne). Puisque FIFO est par définition non préemptif, il est plus sensible aux lois d'activation effectives, auquel cas les lois probabilistes pour les durées d'exécution et les échéances choisies pourraient expliquer ce comportement.

D'autre part, ces ordonnanceurs sont par contre assez égaux en ce qui concerne leur comportement face à la granularité de l'horloge système : à faible charge, l'écart des taux de garantie observés pour des granularités d'horloge de 1ms et 100ms est de l'ordre de 0.1%, quand il est de l'ordre de 3% à charge élevée (à comparer aux 46% de la configuration à faible recouvrement). Dans la configuration choisie, on peut en conclure que l'impact de la granularité d'horloge est d'autant moins sensible que des travaux ont plus de chance d'être activés pendant l'exécution d'un autre, c'est-à-dire que la charge globale est élevée.

9.3.1.2 Ordonnanceur à réacceptation des travaux refusés, sans politique de rejet



(a) DM

(b) EDF

(c) FLLF

(d) FIFO

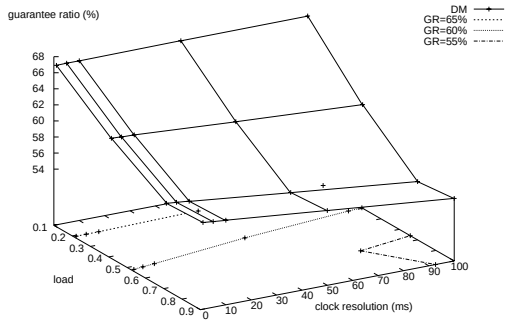
FIG. 9.9: Taux de garantie pour les ordonnanceurs à réacceptation des travaux refusés selon différentes affectations des priorités

Pour ces ordonnanceurs, la différence entre les ordonnanceurs DM, EDF et FLLF, s'estompe, et les tendances sont très similaires : le taux de garantie décroît avec la charge individuelle moyenne, jusqu'à se stabiliser à forte charge. Comparé aux ordonnanceurs sans réacceptation précédents, le taux de garantie est très nettement plus élevé, surtout à faible charge et surtout pour EDF et FLLF où cette différence vaut respectivement 44 et 50% ; pour DM, elle est néanmoins de 10%. À forte charge individuelle moyenne, cette différence s'estompe pour être de l'ordre de 5%. Il en découle cette fois-ci que la performance relative des ordonnanceurs est en faveur de FLLF, suivi de EDF (écart de moins de 3%) puis DM (écart variant de 30% à 2%), et enfin FIFO (écart variant de 30% à 3%). FIFO reste atypique dans cette configuration, et probablement fortement

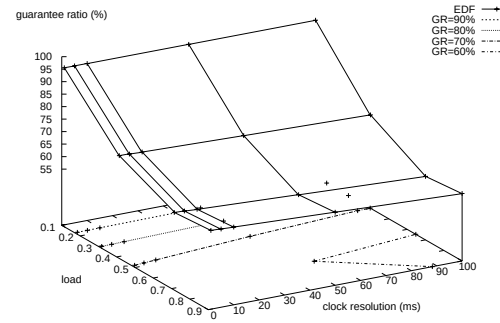
contraint par les lois d'activation.

Pour ces ordonnanceurs, l'influence de la granularité d'horloge est négligeable à faible charge individuelle moyenne (fluctuations inférieures à 1%), plus marquée à forte charge (fluctuations de l'ordre de 4%), mais toujours largement inférieure aux configurations à faible recouvrement.

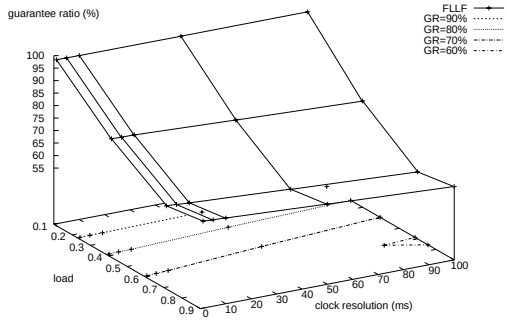
9.3.1.3 Ordonnanceur robuste à politique de rejet simple



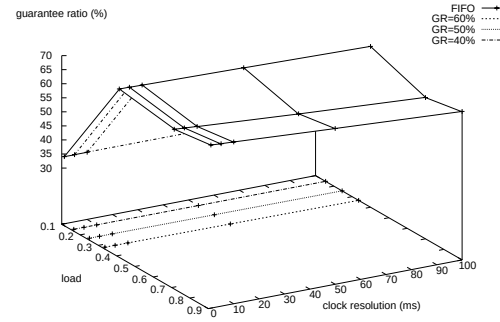
(a) DM



(b) EDF



(c) FLLF



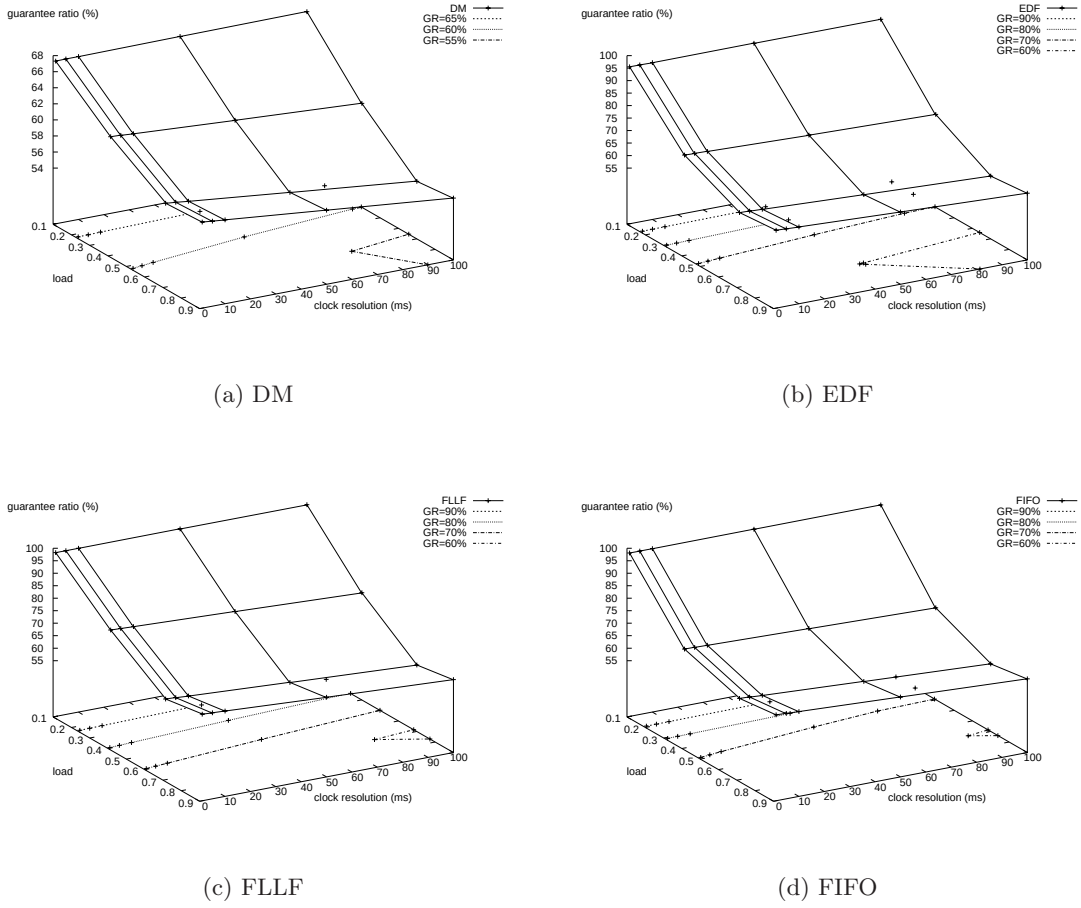
(d) FIFO

FIG. 9.10: Taux de garantie pour les ordonnanceurs à politique de rejet simple selon différentes affectations des priorités

La tendance générale reste identique à l'ordonnanceur précédent. Pour DM, EDF et FLLF, les surfaces sont légèrement translatées par rapport à la politique à réacceptation simple, et d'autant plus fortement que la charge est forte (à forte charge : 15% d'écart pour FLLF, 10% pour EDF et DM) ; cet écart est négligeable à faible charge.

En ce qui concerne l'influence de la granularité d'horloge, elle est comparable aux mesures précédentes (de l'ordre de 5%).

9.3.1.4 **Ordonnanceur robuste à politique de rejet multiple**



(a) DM

(b) EDF

(c) FLLF

(d) FIFO

FIG. 9.11: Taux de garantie pour les ordonnanceurs à politique de rejet multiple selon différentes affectations des priorités

À nouveau, on n'observe aucune différence notable par rapport aux ordonnanceurs précédents, contrairement à ce qu'on envisageait en modifiant le *facteur de recouvrement* (translation entre les deux surfaces inférieure à 0.3% pour DM, EDF, FLLF). Ce qui indique qu'une fois encore, un rejet simple suffit dans l'immense majorité des cas.

Pour FIFO cependant, le changement de comportement est spectaculaire avec cette politique : elle rejoint la tendance observée sur les autres ordonnanceurs, à savoir que le taux de garantie diminue avec la charge. Avec cette politique, l'aspect non-préemptif de FIFO est supplanté par une forme de préemption (caractéristique des autres poli-

tiques évaluées) très autoritaire (l'annulation d'autant de travaux que nécessaire), ce qui pourrait expliquer ce comportement.

9.3.1.5 Ordonnanceur `TPS`

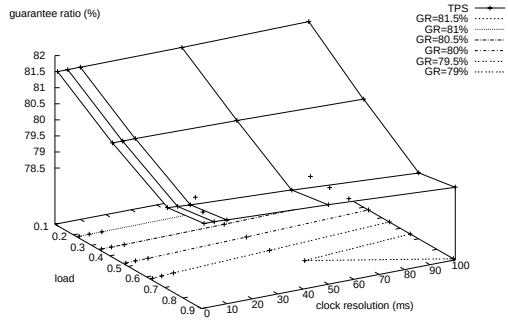


FIG. 9.12: Taux de garantie pour TPS

La tendance générale est conservée par rapport à la configuration à faible recouvrement : le taux de garantie décroît avec la charge individuelle moyenne, et l'influence de la granularité d'horloge est d'autant plus marquée que la charge est élevée.

Mais, de même que pour les ordonnanceurs qui précèdent, à la différence de la configuration à faible recouvrement, on observe que la décroissance du taux de garantie avec la charge est d'amplitude beaucoup plus faible (2%), que le taux de garantie est globalement moins élevé (plus de 15% d'écart), et qu'il est moins sensible à la granularité d'horloge (0.3% au plus, pour la charge la plus élevée). On observe également, comme pour les ordonnanceurs précédents, un palier du taux de garantie à forte charge individuelle moyenne.

9.3.2 Configurations avec tâches garanties hors-ligne

Dans les deux évaluations qui suivent, la charge des tâches apériodiques n'est pas modifiée. Une charge garantie hors-ligne formée des deux tâches périodiques décrites dans la section précédente est ajoutée.

9.3.2.1 **Ordonnanceur** `DP`

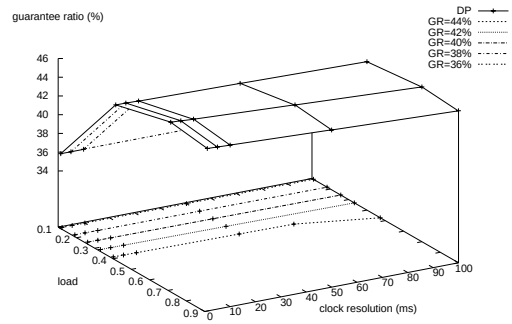


FIG. 9.13: Taux de garantie pour DP

À faible charge individuelle moyenne, on observe une croissance sensible du taux d'utilisation avec la charge, probablement dûe à la présence des tâches garanties hors-ligne, qui s'ajoute au fait que les tâches apériodiques ont une échéance d'autant plus longue que la charge est faible, ce qui nuit à l'acceptation d'autres tâches pendant tout ce long intervalle. Cependant, ce comportement est à nuancer, dans la mesure où son influence est de l'ordre de 9.5% quand la charge passe de 0.1 à 0.9 (granularité d'horloge 1ms), sur un taux de garantie qui a fortement diminué par rapport à la configuration à plus faible recouvrement (de 34% à faible charge).

Cependant, par rapport à la configuration à faible recouvrement, l'influence de la granularité d'horloge est très modeste (moins de 1%), tout comme nous l'avons déjà observé dans les évaluations précédentes.

9.3.2.2 **Ordonnanceur** `CBS-JFP`

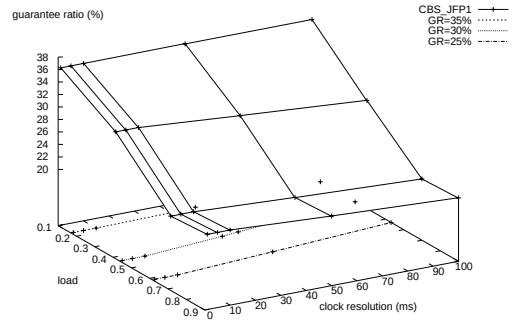


FIG. 9.14: Taux de garantie pour CBS-JFP

Le comportement qualitatif général de cet ordonnanceur dans la présente configuration suggère les mêmes observations que dans la configuration précédente, à faible

recouvrement. La seule différence est d'ordre quantitatif, et provient du fait que le taux de garantie est considérablement (deux fois) plus faible à faible charge, et légérement plus élevé à forte charge (7% d'écart).

9.4 Synthèse

Au travers de ces évaluations, nous pouvons dégager les tendances générales suivantes :

- la granularité d'horloge système a un impact d'autant plus important que la charge individuelle moyenne des travaux soumis est élevée,
- cet impact tend à s'amenuiser à mesure que la probabilité qu'une tâche soit activée pendant qu'une autre s'exécute augmente, c'est-à-dire que la charge globale augmente,
- à facteur de recouvrement faible (*i.e.* charge globale modérée), l'impact sur le taux de garantie des tâches apériodiques peut être considérable (proche de 50%).

Les remarques que nous avons pu faire portaient sur le comportement de ces ordonnanceurs en fonction de la charge individuelle moyenne :

- le taux de garantie décroît généralement avec la charge individuelle moyenne. Les exceptions notables apparaissent lorsque la probabilité qu'une tâche soit activée pendant qu'une autre s'exécute est plus élevée, et sont DP et la famille JFP spécialisée avec l'affectation des priorités FIFO,
- le taux de garantie décroît à mesure que la probabilité qu'une tâche soit activée pendant qu'une autre s'exécute augmente,
- pour un facteur de recouvrement plus élevé, le taux de garantie décroît jusqu´à atteindre un palier constant à charges individuelles moyennes élevées.

Et, comparativement, l'influence de la charge individuelle moyenne sur le taux de garantie est prépondérante, cette prépondérance s'accroissant lorsque la probabilité qu'une tâche soit activée pendant qu'une autre s'exécute augmente, c'est-à-dire lorsque la charge globale augmente.

Enfin, les remarques que nous pouvons faire sur le comportement relatif des différents ordonnanceurs proposés sont les suivantes :

- les ordonnanceurs de la famille `JFP` sont sensiblement égaux vis-à-vis des variations de la granularité d'horloge système,
- les ordonnanceurs de la famille `JFP` sont légèrement inégaux face aux variations de la charge individuelle moyenne : à faible charge individuelle moyenne, EDF se comporte légèrement mieux (quelques pourcents) que FLLF tant que le facteur de recouvrement est faible (*i.e.* charge globale modérée), la tendance s'inversant lorsque le facteur de recouvrement est plus élevé. Et FLLF ou EDF se comportent mieux que DM (quelques pourcents). Plus la charge individuelle moyenne augmente, plus ces comportements se rejoignent (différence inférieure à 1%). Ces sensibilités s'estompent d'autant plus fortement que le facteur de recouvrement est élevé, et que les ordonnanceurs ne proposent pas de politique de réacceptation ou de rejet.

– l'ordonnanceur `DP` est moins sensible que `CBS-JFP` aux variations de la charge individuelle moyenne et de la résolution d'horloge, et présente un taux de garantie supérieur. Cette différence tend cependant à s'estomper à mesure que le facteur de recouvrement (*i.e.* la charge globale) augmente.

Chapitre 10

Évaluation en présence de surcoûts d'exécution du support d'exécution

Dans ce chapitre, nous reprenons la configuration à facteur de recouvrement plus élevé (voir section 9.3 précédente), et nous évaluons le système en tenant compte des surcoûts du support d'exécution.

10.1 Dispositif expérimental

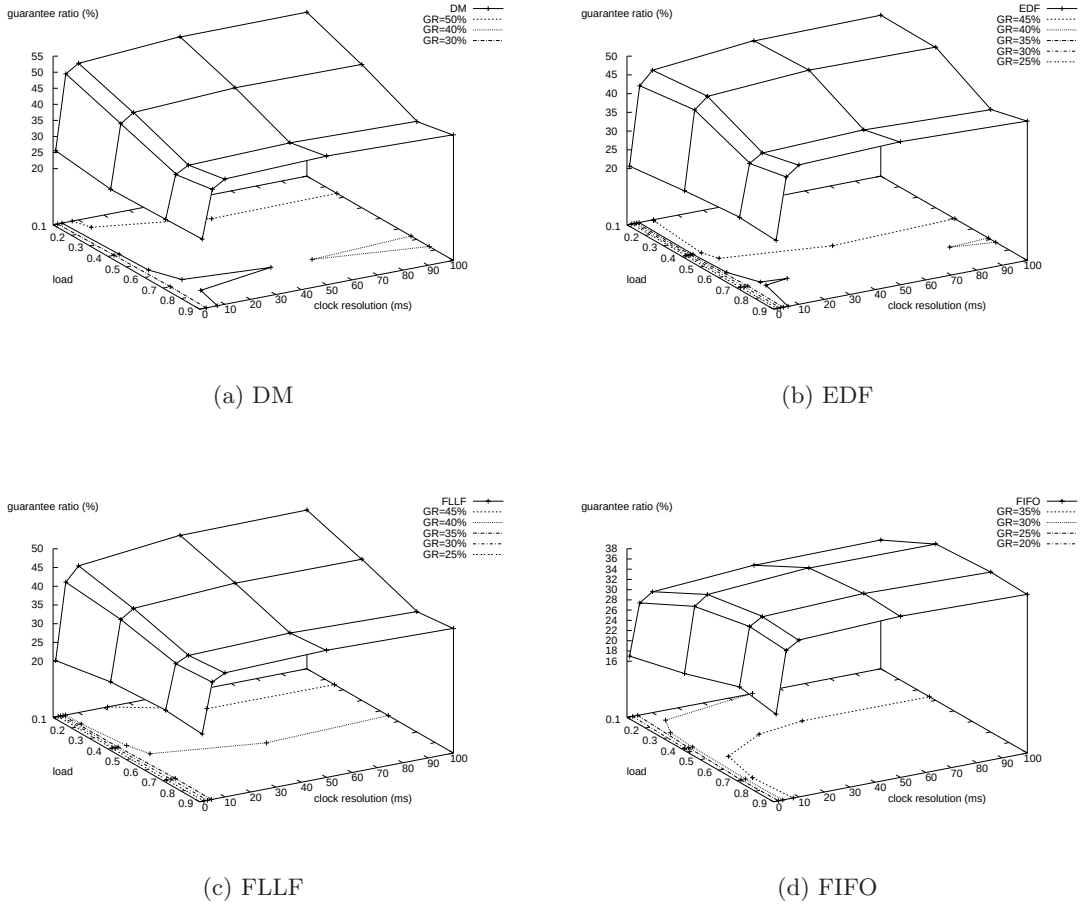
La configuration du système est identique à celle définie en 9.3, à laquelle nous rajoutons les coûts d'exécution du système suivants :

- chaque traitant d'interruption (de tout type) occupe un temps d'exécution donné par la loi uniforme sur [0, 0.5ms] ;
- chaque appel système (par exemple l'activation d'une tâche) occupe un temps d'exécution donné par la loi uniforme sur [0, 0.5ms] ;
- chaque appel à l'ordonnanceur occupe un temps d'exécution donné par la loi uniforme sur [0, 0.5ms].

Ces surcoûts d'exécution sont indépendants de la granularité d'horloge choisie. Ils ne sont pas pris en compte par les tests d'acceptation, les algorithmes de réacceptation, ou les politiques de rejet : il en résulte que des travaux dépassent leur échéance, et sont aussitôt supprimés par le support d'exécution. Dans les simulations qui suivent, nous mesurons par conséquent l'impact de la granularité d'horloge sur les décisions d'acceptation, et également l'impact des travaux acceptés à tort (*i.e.* qui dépassent leur échéance) sur les tests d'acceptation à venir.

10.2 Configurations sans tâche garantie hors-ligne

10.2.1 Ordonnanceurs à garantie sans réacceptation ni politique de rejet



(a) DM

(b) EDF

(c) FLLF

(d) FIFO

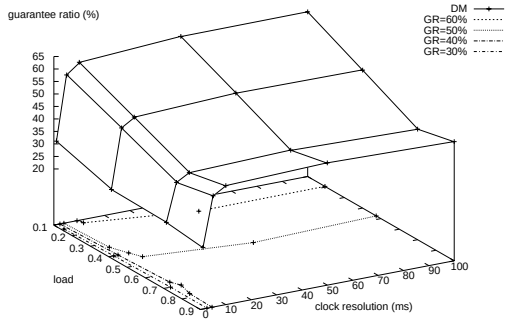
FIG. 10.1: Taux de garantie pour les ordonnanceurs à garantie selon différentes affectations des priorités

La tendance générale, qui continuera de se vérifier par la suite, est que le comportement des ordonnanceurs à granularité d'horloge grossière est comparable aux évaluations qui ne tenaient pas compte des surcoûts d'exécution du système (à moins de 0.3% près). Ceci indique que la marge pour la prise en compte de la granularité de l'horloge suffit à absorber les coûts système.

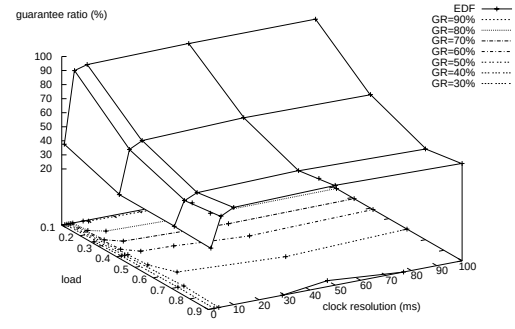
Par contre, comme on peut s'y attendre, le taux de garantie s'effondre lorsque la granularité d'horloge s'approche des surcoûts moyens liés au système : de 15 à 20% suivant les ordonnanceurs lorsque la granularité d'horloge passe de 100ms à 1ms. On

observe que cette décroissance est d'autant plus forte que la charge est faible pour DM, EDF et FLLF (écart de plus de 7% entre les charges 0.1 et 0.9 pour la granularité de 1ms), mais croît avec la charge pour FIFO (de plus de 2%).

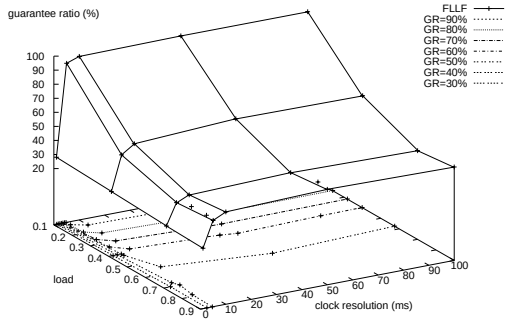
10.2.2 Ordonnanceur à réacceptation des travaux refusés, sans politique de rejet



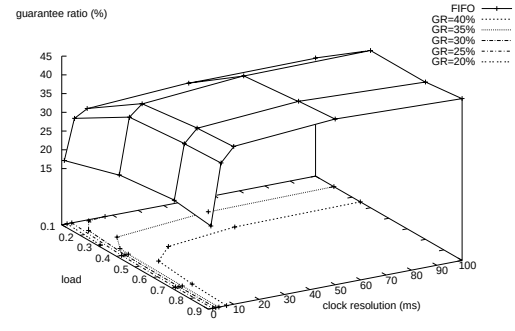
(a) DM



(b) EDF



(c) FLLF

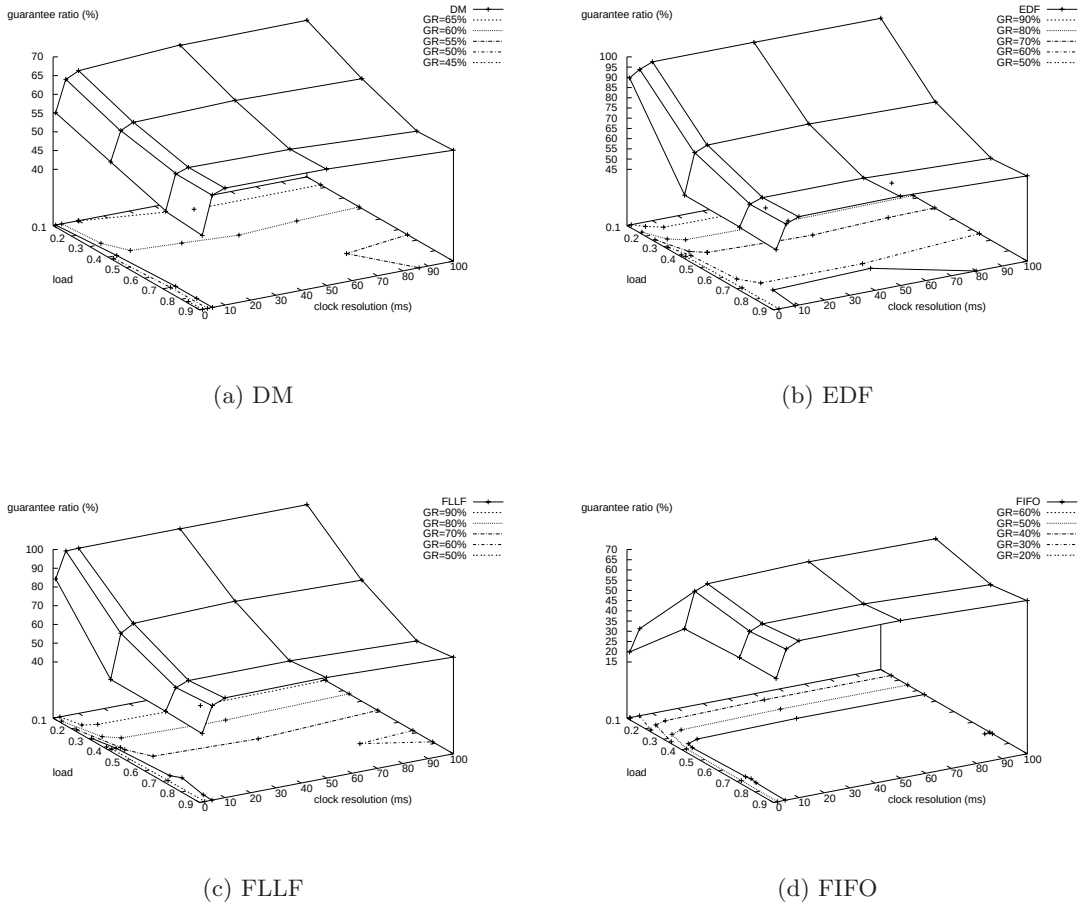


(d) FIFO

FIG. 10.2: Taux de garantie pour les ordonnanceurs à réacceptation des travaux refusés selon différentes affectations des priorités

Les remarques précédentes s'appliquent également pour cette catégorie d'ordonnanceurs, à savoir que l'impact des surcoûts système est négligeable à granularité d'horloge grossière, et s'accroît très sensiblement à mesure qu'elle devient fine. Cependant, par rapport aux ordonnanceurs sans réacceptation, l'écart dû à l'impact des surcoûts système s'amenuise, en particulier à faible charge.

10.2.3 Ordonnanceur robuste à politique de rejet simple



(a) DM

(b) EDF

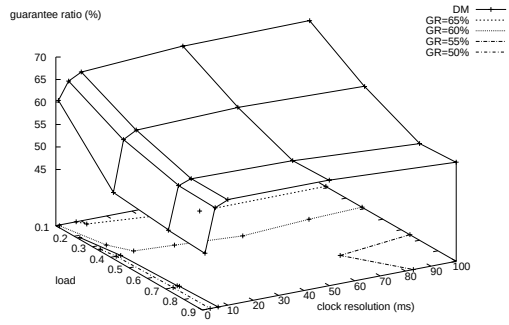
(c) FLLF

(d) FIFO

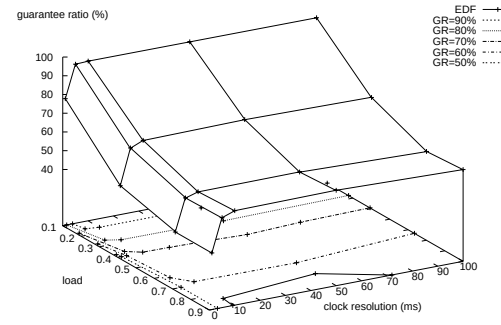
FIG. 10.3: Taux de garantie pour les ordonnanceurs à politique de rejet simple selon différentes affectations des priorités

Les mêmes remarques continuent de s'appliquer, et l'écart dû à l'impact des surcoûts d'exécution du système continue de s'amenuiser.

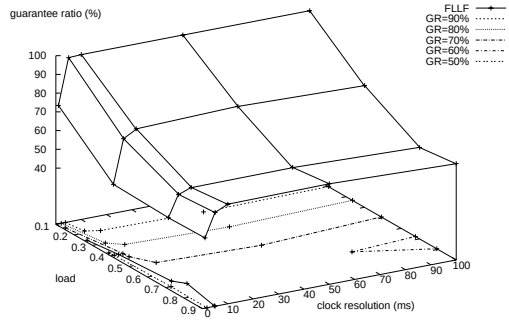
10.2.4 Ordonnanceur robuste à politique de rejet multiple



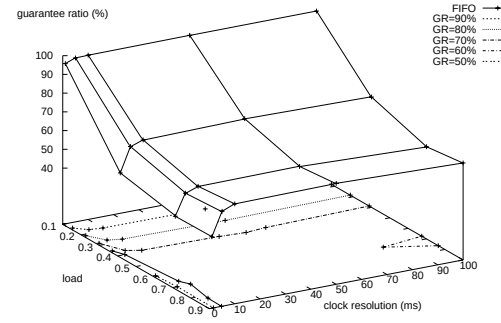
(a) DM



(b) EDF



(c) FLLF



(d) FIFO

FIG. 10.4: Taux de garantie pour les ordonnanceurs à politique de rejet multiple selon différentes affectations des priorités

Par rapport au cas à rejet simple, la seule différence sensible est que le taux de garantie est légèrement plus élevé à faible charge pour DM (5%), EDF et FLLF (3%). La brutale modification du comportement de FIFO observée dans la section 9.3 se constate ici aussi.

10.2.5 Ordonnanceur TPS

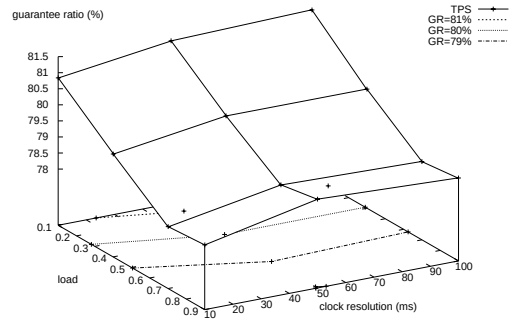


FIG. 10.5: Taux de garantie pour TPS

À nouveau, les surfaces des taux d'acceptation sont très proches du cas sans prise en compte des surcoûts d'exécution à granularité d'horloge grossière. Cependant, contrairement aux ordonnanceurs évalués précédemment, l'écart entre les deux surfaces reste très peu significatif (de l'ordre de 0.1%), même à granularité d'horloge fine, où il est plus prononcé à faible charge, mais reste modeste (moins de 0.5%).

10.3 Configurations avec tâches garanties hors-ligne

10.3.1 Ordonnanceur DP

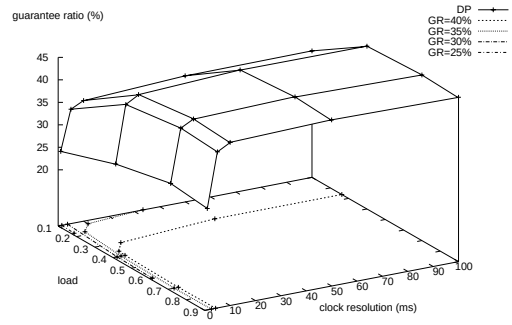


FIG. 10.6: Taux de garantie pour DP

Les mêmes remarques que dans le cas des ordonnanceurs de la famille `JFP` s'appliquent, à savoir que l'impact des surcoûts système est négligeable à granularité d'horloge grossière, et s'accroît très sensiblement à mesure qu'elle devient fine (de l'ordre de 12%). Mais, contrairement à la plupart des ordonnanceurs de la famille `JFP`, hors FIFO, l'écart entre les deux surfaces (sans/avec prise en compte des coûts système) s'accroît

avec la charge individuelle moyenne (passe de 11% à 15% lorsque la charge individuelle moyenne passe de 0.1 à 0.9 pour une granularité d'horloge de 1ms).

10.3.2 Ordonnanceur CBS-JFP

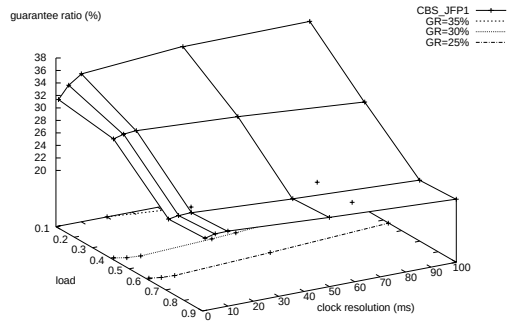


FIG. 10.7: Taux de garantie pour CBS-JFP

On observe que l'influence des surcoûts d'exécution du système est très légère, même à granularité d'horloge fine (inférieure à 3%). Et l'écart est le plus élevé là ou la charge moyenne individuelle est la plus faible.

10.4 Synthèse

Au travers de ces observations, nous pouvons dégager les tendances suivantes :

- l'influence des coûts système sur le taux de garantie est d'autant plus marquée que la granularité d'horloge en est proche. Le comportement avec prise en compte des coûts système rejoint naturellement celui sans prise en compte de ces coûts système lorsqu'ils deviennent négligeables face à la granularité d'horloge,
- cette influence est en général plus élevée à faible charge individuelle moyenne. Les deux exceptions observées sont DP et la famille `JFP` avec l'affectation des priorités suivant FIFO,
- en ce qui concerne la famille `JFP`, les remarques sur les comportements relatifs des ordonnanceurs étudiés sans prise en compte des coûts système s'appliquent ici aussi.

Conclusion

Dans ce document, nous avons commencé par présenter différents modes d'évaluation de systèmes temps-réel : par analyse statique d'une part, et par exécution (réelle ou simulée) d'autre part. En nous intéressant d'abord aux travaux dans le domaine de l'ordonnancement en temps-réel, nous avons dressé un état de l'art de ces techniques formant une des approches d'évaluation par analyse statique. Ces travaux mènent à une vérification sûre des propriétés temporelles du système, au prix d'une complexité ou d'un pessimisme croissant lorsque les contraintes du système se multiplient. En remarquant en plus que ces techniques ne sont plus compatibles avec la non connaissance complète, ou avec un fort dynamisme du comportement temporel de l'environnement ou du système lui-même, nous avons présenté quelques approches d'évaluation par exécution réelle ou simulée. Nous nous sommes tournés vers les approches par simulation, pour lesquelles l'instrumentation du système simulé ne perturbe pas les résultats obtenus (pas d'effet sonde), contrairement à l'instrumentation pour l'évaluation de systèmes par exécution réelle, qui introduit en général un tel effet sonde, à moins que la version du produit évalué soit spécialement conçue pour l'évaluation (ce qui est en général onéreux, ou difficile à réaliser).

Après avoir présenté les atouts et les défauts des outils existants, nous avons proposé une plate-forme de simulation, ARTISST pour "ARTISST is a **R**eal-**Ti**me **S**ystem **S**imulation **T**ool", qui permet l'évaluation de systèmes temps-réel centralisés ou distribués à la fois à partir de modèles abstraits et à partir d'implantations concrètes, en ayant alors pour objectif de reproduire le plus fidèlement possible le comportement effectif des traitements simulés, afin de réduire les erreurs de représentativité du modèle dues à l'écart entre la modélisation simulée et l'implantation effective. Par rapport à l'existant, cette plate-forme :

- n'est pas limitée à un modèle de tâches donné : tous les modèles de tâches sont extensibles par héritage objet ;
- n'est pas limitée à une palette d'ordonnanceurs donnée : il est possible de définir ses propres ordonnanceurs, y compris pour des modèles de tâches incompatibles avec le modèle de taches par défaut ;
- permet de simuler des tâches qui ne sont pas que des enveloppes statistiques abstraites d'occupation de temps processeur : il est possible de leur associer des vrais traitements, et c'est l'exécution de ces traitements qui conditionne la simulation d'occupation du processeur simulé, de manière totalement contrôlée par le développeur ;

– permet de simuler les surcoûts associés à l'exécution du système, y compris ceux de l'ordonnanceur. Et ceci de manière aussi fine que pour les coûts d'exécution des tâches ;
– permet de simuler l'interface de programmation de tout système d'exploitation temps-réel, par héritage objet et enrichissement de l'interface par défaut ;
– permet de réutiliser du code d'application existant : toutes les tâches, tous les services du système simulé (y compris l'ordonnanceur) sont écrits en `C/C++`, et peuvent être portés à d'autres langages. ARTISST se présente sous la forme d'une bibliothèque de fonctions ;
– reproduit la vision approximative (d'erreur quantifiée) et discrète que les systèmes réel ont de l'écoulement du temps, par décorrélation de l'échelle de temps-réel globale avec les échelles de temps système locales à chaque nœud. Cette caractéristique autorise par ailleurs la simulation d'anomalies ou de désynchronisation des horloges de systèmes distribués.

Associé à la réalisation de cette plate-forme, nous avons implanté une série d'ordonnanceurs, qui ont été évoqués lors de l'état de l'art sur le domaine. Nous avons en particulier intégré toute une famille d'ordonnanceurs à priorités génériques, sous la forme d'une hiérarchie d'objets qui peuvent être spécialisés pour un grand nombre d'affectations de priorités statiques ou dynamiques. Cette famille présente une approche uniforme à l'ordonnancement soit en présence de tâches toutes garanties hors-ligne, soit en présence de tâches toutes apériodiques garanties en-ligne. Dans le deuxième cas, nous proposons plusieurs ordonnanceurs au sein de cette famille se caractérisant par des politiques de repêchage des tâches refusées, ou par des politiques de rejet différentes.

Enfin, nous avons étudié l'influence de la granularité de l'échelle de temps perçue par le système simulé (horloge système) sur le comportement d'une partie des ordonnanceurs pour tâches apériodiques avec garantie en-ligne réalisés. Pour cela, nous avons commencé par donner quelques recommandations de conception des ordonnanceurs à garantie dans le cas général afin de prendre en compte correctement cette granularité d'horloge. Puis nous avons soumis ces ordonnanceurs ainsi modifiés à une série d'évaluations de type charge synthétique, sans et avec simulation des surcoûts d'exécution du support d'exécution simulé.

Une première perspective à ce travail est de proposer une infrastructure intégrée au simulateur ARTISST permettant d'extraire les paramètres temporels du système et de l'application par analyse des séquences d'instructions, et par injection de ces résultats d'analyse dans le code source (sous forme d'appels à `hold_cpu()`). L'objectif serait d'utiliser ces informations afin de générer des simulations plus pertinentes et représentatives du comportement effectif, en limitant les interventions de l'utilisateur sur le code, et sans avoir à simuler à l'échelle de l'instruction.

Une autre perspective est d'étendre le mode de réalisation d'ordonnanceurs modulaires et spécialisables présenté dans ce travail (famille `JFP`). Il paraîtrait intéressant de développer une approche plus globale à cette technique, en définissant une hiérarchie d'objets génériques permettant la composition d'ordonnanceurs, afin de proposer une

“boîte à outils” type permettant de réutiliser du code d’ordonnancement, de la même manière qu’on peut maintenant réutiliser, personnaliser et composer des algorithmes génériques grâce à des bibliothèques adaptées. Ce travail a déjà été initialisé dans Artisst sous la forme de deux compositions hiérarchiques d’ordonnanceurs (CBS-JFP et DP), dont les implantations actuelles forment cependant des hiérarchies objets distinctes, bien qu’assez proches. Or, moyennant la définition d’une interface simple permettant le calcul des temps résiduels qui tienne compte de la hiérarchie d’ordonnanceurs, il aurait été possible de réellement composer ces ordonnanceurs, c’est-à-dire de réutiliser par exemple le code de JFP dans CBS-JFP. Une perspective serait de concevoir une approche adaptée à un ensemble plus vaste d’ordonnanceurs, en utilisant notamment des résultats en ordonnancement de systèmes ouverts, tels que ceux de [MF02], ou en proposant une technique reposant sur le calcul des temps résiduels.

Une troisième perspective, plus orientée vers l’architecture des systèmes d’exploitation, consiste à étudier l’influence de l’interface du système d’exploitation sur une même application. Car, pour un cahier des charges donné, les services disponibles auprès du système d’exploitation conditionnent la manière dont l’application est conçue, et aussi son comportement. Par exemple, on pourrait s’intéresser à l’influence des fonctions de gestion des délais sur la manière dont l’application peut être conçue, et se comporte.

Enfin, une autre perspective serait d’étendre Artisst en ajoutant les éléments nécessaires pour l’évaluation de systèmes avec tâches dépendantes (protocoles d’accès aux ressources), et/ou distribués (modélisation de protocoles réseau). À plus long terme, une perspective connexe est d’intégrer Artisst dans une démarche de réalisation de systèmes temps-réel centralisés et distribués complète, proposant à la fois modélisation, vérification et simulation, en s’appuyant sur les travaux menés par la communauté du temps-réel.

Publications et logiciels en relation avec le travail de thèse

Logiciel ARTISST et documents reliés

Logiciel ARTISST et tutoriel

Bibliothèque de simulation* ARTISST *:

Disponible sous licence libre `LGPL` à l'adresse
`http://www.irisa.fr/aces/software/artisst`, et est accompagnée du tutoriel [Dec02] en anglais. Un dépôt à l'Agence pour la Protection des Programmes est effectué.

Tutoriel :

D. Decotigny. *Using Artisst: a Tutorial*, 1999-2002.
Un guide didactique pour réaliser des simulations simples avec ARTISST en centralisé et en distribué autour de 3 exemples.

Publications liées à la thèse

D. Decotigny and I. Puaut. Artisst: An extensible and modular simulation tool for real-time systems. *Proceedings of the International Symposium on Object-oriented Real-time Distributed Computing*, May 2002.
Bref aperçu des caractéristiques d'ARTISST, avec un exemple d'utilisation au travers d'une petite campagne de simulations.

D. Decotigny and I. Puaut. Artisst: An extensible framework for the simulation of real-time systems. *Proceedings of the Conférence Internationale sur les Systèmes Temps-Réel (RTS'02)*, March 2002.
Description synthétique et technique des éléments composant la plate-forme ARTISST.

D. Decotigny and I. Puaut. Artisst: An extensible and modular simulation tool for real-time systems. Tech. Rep. 1423, IRISA, Nov 2001,
`ftp://ftp.irisa.fr/techreports/2001/PI-1423.ps.gz`.
Description technique des éléments et des caractéristiques d'ARTISST, accompagnée d'un exemple de campagne de simulations.

Autres publications

I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. *Proc. of the 23rd IEEE International Real-Time Systems Symposium*, December 2002, http://www.irisa.fr/aces/doc/ps02/rtss02.ps.gz.
Deux algorithmes de sélection de contenu de cache pour des caches verrouillés, qui optimisent la faisabilité d'un ensemble de tâches temps-réel périodiques.

P. Chevochot, I. Puaut, G. Cabillic, A. Colin, D. Decotigny, and M. Banâtre. Hades: a distributed system for dependable hard real-time applications built from cots components. Tech. Rep. PI1357, IRISA, October 2000,
ftp://ftp.irisa.fr:/techreports/2000/PI-1357.ps.gz.
Brève description des caractéristiques de la plate-forme temps-réel distribuée tolérante aux fautes HADES, des techniques de réplication automatiques, et évaluation du respect de l'hypothèse de silence sur défaillance au travers d'une campagne d'injection de fautes mettant en œuvre et comparant plusieurs mécanismes de détection de fautes.

A. Colin, D. Decotigny, P. Chevochot, and I. Puaut. Are cots suitable for building distributed fault-tolerant real-time systems ? *Workshop on parallel and distributed real-time systems (WPDRTS'00), LNCS 1800*, pp. 699-705, May 2000,
http://www.irisa.fr/solidor/doc/ps00/wpdrts00.ps.gz.
Présentation de l'outil d'estimation de temps d'exécution pire-cas par analyse statique du code source HEPTANE et de la plate-forme HADES.

E. Anceaume, G. Cabillic, P. Chevochot, D. Decotigny, A. Colin, and I. Puaut. Un support d'exécution flexible pour applications distribuées temps-réel dur. *Première Conférence Française sur les Systèmes d'Exploitation (CFSE'1)*, pp. 109-120, June 1999, http://www.irisa.fr/solidor/doc/ps99/cfse1hades.ps.gz.
Présentation du support d'exécution de HADES, de ses propriétés, et de sa mise en œuvre au-dessus du noyau Chorus.

D. Decotigny. *Bibliographie d'introduction à l'ordonnancement dans les systèmes informatiques temps-réel*, 1999-2002,
http://david.decotigny.free.fr/rt/intro-ordo/.
Une bibliographie du domaine de l'ordonnancement temps-réel, plus détaillée que dans le chapitre 3.

Annexe A

Précisions techniques, et extraits de codes de simulations utilisant Artisst

Dans les sections qui suivent, nous commençons par résumer les étapes que l'utilisateur d'Artisst doit suivre pour mettre en place une simulation, avant de donner quelques informations techniques sur quatre éléments qui interviennent dans la réalisation, à savoir :

- Les messages Artisst disponibles par défaut ;
- Les fonctions de rappel des modules et l'interface de programmation utile à la programmation de modules (avec exemple) ;
- L'interface de programmation pour la construction et l'utilisation d'un circuit de simulation (avec exemple) ;
- L'interface de programmation utile à la programmation d'applications temps-réel et les fonctions de rappel à personnaliser pour produire son propre système d'exploitation temps-réel (avec exemple). Nous fournissons à la fin l'automate exécuté par le module pour faire la liaison entre le circuit de simulation et le système temps-réel simulé.

Tous les renvois font référence à la partie II.

1.1 Étapes de mise en place d'une simulation

Lorsque l'utilisateur souhaite mettre en place une simulation, il doit implanter le ou les systèmes à évaluer, et le/les intégrer dans un circuit de simulation afin d'exploiter les messages d'instrumentation. Nous décrivons ici très brièvement les quelques étapes de réalisation qui aboutissent à une simulation prête à être menée.

Le système à évaluer est géré par le module de simulation de système temps-réel décrit dans la section 7.2 précédente, qui joue le rôle de support d'exécution. L'utilisa-

teur doit implanter l'application, et doit compléter, le cas échéant, le système d'exploitation simulé, avant de pouvoir lancer des simulations.

1.1.1 Application

L'application qui est simulée est constituée de travaux de tâches, qui correspondent à du code quelconque, qui peut par exemple être directement issu du code réel d'une implantation en cours de production. Afin de rendre compte du comportement temporel des traitements, ce code doit faire appel à la primitive de simulation d'occupation du temps processeur (`hold_cpu()`). Il peut aussi faire appel à l'interface de programmation du système d'exploitation simulé (voir 7.2.4.3), et à toute autre fonction de bibliothèque disponible sur la machine hôte de la simulation.

En plus de cette phase de codage, l'étape d'implantation de l'application comprend une phase de renseignement du modèle de tâches, qui est intimement lié aux services du système d'exploitation, et en particulier à l'ordonnanceur.

1.1.2 Système d'exploitation simulé

1.1.2.1 Modèle et statut d'exécution des tâches

Suivant les services système intégrés dans le système (par exemple : services de gestion de ressources), ou les ordonnanceurs envisagés, la première étape consiste à définir le modèle de tâches adapté, ainsi que le statut d'exécution. Ceci consiste en une extension (héritage) des structures de données par défaut.

1.1.2.2 Services système

La bibliothèque Artisst définit l'interface de programmation du système d'exploitation simulé. L'utilisateur peut choisir de la personnaliser (par implantation ou surcharge des fonctions de rappel associées), ou de l'étendre (par définition de la fonction de rappel `syscall()`). Le code qui est ajouté peut faire tous les appels à l'interface de programmation du système d'exploitation simulé, ce qui n'est pas forcément possible dans les systèmes réels, et également faire appel aux primitives de simulation (en particulier `hold_cpu()`). Toutes les personnalisations ou extensions peuvent exploiter les extensions aux modèle de tâche et au statut d'exécution des tâches effectuées par l'utilisateur.

D'autre part, par défaut tous les services système sont interruptibles, mais il est possible de demander à ce que tous les services système soient non-interruptibles lors de la compilation.

1.1.2.3 Traitants d'interruption

Les traitants d'interruption suivent les mêmes recommandations que les travaux des tâches de l'application en ce qui concerne le code qui est exécuté.

La phase de codage est précédée d'une phase de configuration à la compilation, qui vise à indiquer les règles de préemption : par défaut, la règle de préemption par priorité prévaut (voir 7.2.3.3), mais il est possible de demander à appliquer la règle "LIFO".

1.1.2.4 Fonction de rappel d'ordonnancement

La fonction de rappel d'ordonnancement est automatiquement appelée par le système d'exploitation simulé, donc l'utilisateur a à s'occuper de la coder ou d'en réutiliser une parmi celles fournies. Si l'utilisateur choisit de créer sa propre fonction de rappel d'ordonnancement, le principe est toujours d'utiliser la liste des tâches en cours pour établir la décision d'ordonnancement. Les mêmes règles que pour le codage des services système s'appliquent. En particulier, comme pour eux, la fonction de rappel d'ordonnancement repose directement sur le modèle de tâches, et le statut d'exécution des travaux.

Par défaut, la fonction de rappel d'ordonnancement est réentrante, mais il est possible d'indiquer qu'elle ne doit pas l'être lors de la compilation.

1.1.3 Environnement d'évaluation

La construction du circuit de simulation est une étape simple, et qui fait appel aux deux schémas typiques donnés dans la section 7.1.3, suivant que le système simulé comprend un seul nœud, ou plusieurs. La démarche de l'utilisateur consiste en la construction (ou la réutilisation) des différents modules, et en la connexion des modules les uns aux autres : ARTISST fournit la bibliothèque de fonctions idoine.

Une étape fondamentale concerne la simulation de l'environnement : on peut choisir d'utiliser les modules autonomes de génération d'événements fournis avec ARTISST (générateur aléatoire, lecteur de traces), ou d'implanter son propre module de modélisation d'environnement, qui pourra par exemple être inséré dans une boucle fermée avec le ou les module(s) de simulation de système temps-réel.

La dernière étape est la définition de la boucle d'événements, qui consiste en une consommation des messages depuis un module particulier du circuit. Cette méthode permet de récupérer à chaque pas de simulation la date temps-réel courante (*via* l'estampille sur le message consommé), pour éventuellement stopper la simulation au bout d'un temps de simulation donné. Cette méthode permet également de faire cohabiter une simulation ARTISST avec le fonctionnement d'une bibliothèque d'éléments graphiques, comme `gtk` qui est utilisé dans le module de représentation graphique sous forme de chronogramme, car ces bibliothèques possèdent elles aussi leur propre boucle d'événements (graphiques).

1.2 Messages ARTISST définis par défaut

Catégorie	Nom (*i.e.* type) du message	Signification
Gestion des interruptions (section 7.2.3.3)	INTERRUPT_SET_MASK	Modification du masque de filtrage global
	INTERRUPT_RAISED	Une interruption simulée est levée. Le module de simulation de système précise dans le message si elle a été refusée par le masque de filtrage global. Si non, elle est mise en attente d'être prise en compte
	INTERRUPT_POSTPONE	Une interruption levée n'est pas prise en compte par le contexte d'exécution courant (filtrage local)
	INTERRUPT_WAKEUP_POSTPONED	Une interruption précédemment non prise en compte est réactivée par le contexte courant (filtrage local)
	INTERRUPT_ENTER	Indiquent le début et la fin du traitement associé à une interruption
	INTERRUPT_LEAVE	
Gestion des travaux (section 7.2.4.3)	TASK_ACTIVATION	Création d'un travail
	BEGIN_TASK_EXECUTION	Première exécution d'un travail précédemment créé
	SUSPEND_TASK_EXECUTION	Préemption de l'exécution d'un travail par un autre ou interruption par un traitant d'interruption
	RESUME_TASK_EXECUTION	Reprise de l'exécution d'un travail suite à une préemption ou à une interruption
	CANCEL_TASK_EXECUTION	Un travail est abandonné
	END_TASK_EXECUTION	Un travail termine son exécution
Instrumentation du système	KERNEL_SERVICE_ENTER	Un travail de tâche ou un traitant d'interruption fait appel à un service du support d'exécution
	KERNEL_SERVICE_LEAVE	
	SCHEDULER_ENTER	Début/fin de l'appel à l'ordonnanceur
	SCHEDULER_LEAVE	
Réseau	NET_SEND	Envoi d'un message réseau
Extensions utilisateur	CUSTOM_MESSAGE	Génère un message personnalisé dans le flot de messages de simulation

TAB. A.1: Messages ARTISST définis par défaut

1.3 Implantation des modules ARTISST

1.3.1 Fonctions de rappel

Nous donnons ci-dessous la liste des fonctions de rappel qui forment l'interface de chaque module (le module associé est noté `myself` dans la suite). Toutes ou une partie d'entre elles seulement peuvent être définies, suivant les modes de fonctionnement supportés par le module.

Fonction de rappel en mode soumission

`push(myself, src_module, message)` **:** le message `message` en provenance du module `src_module` est soumis au module courant `myself`.

Fonctions de rappel en mode consommation

`front(myself, message)` **:** remplit le contenu de `message` avec celui du message le plus proche (chronologiquement) de la date actuelle que le module courant `myself` peut générer. Le message **reste** dans la file des messages délivrables par `myself` ;

`pull(myself, message)` **:** remplit le contenu de `message` avec le message le plus proche (chronologiquement) de la date actuelle que le module courant `myself` a pu générer. Le message est **enlevé** de la file des messages délivrables par `myself`.

Autres fonctions de rappel

`aux_module_registered(myself, autre_module, mode)` et `aux_module_unregistered(myself, autre_module, mode)` : avis auprès du module courant `myself` du (dé)branchement du module `autre_module` sur le module **initiateur** `myself` en mode `mode` (soumission/consommation) ;

`registered_as_module_for_aux(myself, autre_module, mode)` et `unregistered_as_module_for_aux(myself, autre_module, mode)` : avis auprès du module courant `myself` du (dé)branchement du module **initiateur** `autre_module` sur le module `myself` en mode `mode` (soumission/consommation).

1.3.2 Fonctions utiles pour l'implantation des fonctions de rappel des modules

`st_inout_new_module(callbacks, custom)` **:** Créé un nouveau module en lui associant les fonctions de rappel fournies dans la structure `callbacks`, et les données propres `custom` (pour définir des modules personnalisés) ;

`st_inout_message_pending_from_input(myself, message, src_list...)` **:** consulte les modules `src_list` en entrée de `myself` pour connaître le message le plus proche (chronologiquement) disponible en mode consommation, et remplit `message` avec son contenu. Le message **reste** dans les files des messages délivrables des modules `src_list`. Si `src_list` n'est pas précisée, consulte tous les modules en entrée de `myself` ;

`st_inout_pull_message_from_input(myself, message)` **:** récupère le message le plus proche (chronologiquement) parmi ceux disponibles en mode consommation auprès des modules `src_list` en entrée de `myself`, et remplit `message` avec son contenu. Le message est **enlevé** de la file des messages délivrables par le module de `src_list` concerné. Si `src_list` n'est pas précisée, prend en compte tous les modules en entrée de `myself` ;

`st_inout_push_message_on_output(myself, message, dst_list)` **:** soumet le message `message` aux modules connectés en mode soumission au module `myself` et figurant sur la liste `dst_list`. Si `dst_list` n'est pas précisée, effectue la soumission vers tous les modules en sortie de `myself`.

1.3.3 Exemple de code d'un module simple

L'extrait de code suivant est l'implantation des fonctions de rappel du module "multiprise" (voir 7.4 de la partie II) dont le fonctionnement est le suivant :

- Quand on lui soumet un message, il le diffuse à tous les modules qui lui sont connectés en mode soumission ;
- Quand on récupère un message (mode consommation), il le récupère (mode consommation) depuis tous ses modules qui lui sont connectés en mode consommation, et le diffuse à tous les modules qui lui sont connectés en mode soumission.

```
static st_bool iogw_next_message (struct st_inout_descr_t *myself,
                                  struct st_message_t *message)
{ return st_inout_message_pending_from_input(myself, message, NULL); }

static st_bool iogw_message_pull (struct st_inout_descr_t *myself,
                                  struct st_message_t *message)
{
  if (! st_inout_pull_message_from_input(myself, message, NULL))
    return FALSE;
  return st_inout_push_message_on_output(myself, message, NULL);
}

static st_int iogw_push_message (struct st_inout_descr_t *myself,
                                 struct st_inout_descr_t *d_src
                                 struct st_message_t *message)
{ return (! st_inout_push_message_on_output(myself, message, NULL)); }
```

L'extrait de code suivant présente la fonction qui a été rajoutée (définissant une interface de programmation minimale pour les modules de ce type) pour créer des modules de ce type, en utilisant les fonctions de rappel ci-dessus :

```
static struct st_inout_module_t _iogw_module_callbacks = {
  .name = "In->Out Gateway",
  .input_type = ST_INPUT_TWO_PHASE,
  .input = { two_phase:
             { .front = iogw_next_message,
               .pull  = iogw_message_pull } },
  .push = iogw_push_message
};

struct st_inout_descr_t * st_iogw_new()
{
  return st_inout_new_module(& _iogw_module_callbacks, NULL);
}
```

1.4 Implantation d'un circuit de simulation

1.4.1 Interface de gestion du circuit de simulation ARTISST

Pour construire puis gérer le circuit de simulation, ARTISST propose les fonctions suivantes :

`st_inout_register_input_module(initiateur, src)` et `st_inout_register_output_module(initiateur, dst)` : branche/débranche le module `src` au module `initiateur` pour signifier une connexion en mode consommation dont `initiateur` est l'initiateur ;

`st_inout_register_output_module(initiateur, dst)` et `st_inout_unregister_output_module(initiateur, dst)` : branche/débranche le module `dst` au module `initiateur` pour signifier une connexion en mode soumission dont `initiateur` est l'initiateur ;

`st_inout_message_pending(evt_loop, message)` **:** consulte le module `evt_loop` pour connaître le message le plus proche (chronologiquement) disponible en mode consommation, et remplit `message` avec son contenu. Le message **reste** dans la file des messages délivrables par `evt_loop` ;

`st_inout_pull_message(evt_loop, message)` **:** récupère le message le plus proche (chronologiquement) depuis (mode consommation) le module `evt_loop`, et remplit `message` avec son contenu. Le message est **enlevé** de la file des messages délivrables par `evt_loop` ;

`st_inout_push_message(dst, message)` **:** soumet le message `message` au module `dst`.

1.4.2 Exemple de code de création et d'utilisation du circuit de simulation

```
/* Les variables de type module */
struct st_inout_descr_t
  *mod_env,     /* Module de simulation d'environnement */
  *mod_rtsys,   /* Module de simulation de systeme temps-reel */
  *mod_gw       /* Module ``prise multiple''. C'est la boucle
                   d'evenements de notre circuit */
  *mod_traces,  /* Module de generation de traces au format texte */
  *mod_gantt;   /* Module d'affichage graphique de type chronogramme */

/* Le messsage recupere depuis la boucle de simulation */
struct st_message_t message;

/* Creation des modules */
mod_env     = creer_simulateur_environnement();
mod_rtsys   = creer_simulateur_systeme();
mod_gw      = st_iogw_new();
mod_traces  = st_outfile_new("traces.log");
mod_gantt   = st_gantt_chart_new();

/*
 * Connexion des modules :
```

```
 *   Env ->| RTSys ->| GW |-> Traces+Gantt
 */
st_inout_register_input_module(mod_rtsys, mod_env);
st_inout_register_input_module(mod_gw, mod_rtsys);
st_inout_register_output_module(mod_gw, mod_traces);
st_inout_register_output_module(mod_gw, mod_gantt);

/* Le coeur de la simulation */
while (st_inout_pull_message(mod_gw, & message))
{
  /* Besoin de ne rien faire d'autre : mod_gw s'occupe de diffuser les
     messages qu'on recupere, aux modules qui lui sont connectes en mode
     soumission */
}
```

1.5 Module de simulation de système temps-réel `rtsys`

Le module de simulation de système fait l'interface entre le circuit de simulation et le système simulé. Il est créé par l'unique fonction suivante :

`st_rtsys_new(fct_rappel, sys_status, max_jobs, custom, init_task_model, init_task_status, init_task_arg)` : créé un nouveau module ARTISST spécialisé dans la simulation de système temps-réel. Les fonctions de rappel spécifiques à ce module sont définies par `fct_rappel`, le statut de simulation (*i.e.* synthèse des statuts d'exécution passés de tous les travaux) du module est désigné par `sys_status`, et le nombre maximum de travaux simultanés est défini par `max_jobs`. Il est possible d'associer une donnée `custom` propre au module, et une tâche démarrée automatiquement au lancement de la simulation (modèle `init_task_model`, statut d'exécution `init_task_status`, et arguments passés à la nouvelle tâche `init_task_arg`).

Vis-à-vis du circuit de simulation, les fonctions de rappel automatiquement appelées du module ainsi créé sont exactement celles définies pour tous les modules (voir section 1.3). Mais vis-à-vis du système simulé, une interface de programmation spécifique est définie pour l'application et le système d'exploitation simulés (section 1.5.1), qui se charge d'appeler automatiquement les fonctions de rappel spécifiques associées évoquées ci-dessus (`fct_rappel`) et personnalisables (section 1.5.2).

1.5.1 Interface de programmation disponible pour le système simulé

Une fois le module de simulation de système créé, le système simulé (application et support d'exécution) dispose de l'interface de programmation suivante auprès du module de simulation désigné par `rtsys` ci-dessous :

`st_rtsys_activate_task(rtsys, modele, statut, arg)` : création d'un travail de tâche dont le modèle est défini par la structure `modele`, et dont le statut associé que `rtsys` doit tenir à jour est désigné par `statut`. Un argument personnalié `arg` (pointeur) peut être associé à ce travail ;

st_rtsys_hold_cpu(rtsys, duree) : simulation d'occupation du temps processeur par le contexte courant (tâche. traitant d'interruption) pendant la durée duree en tenant compte des préemptions ;

st_rtsys_syscall(rtsys, id, arg) : simule un passage en mode noyau (pour la prise en compte précise des coûts système), et l'appel au service système désigné par id avec le paramètre arg (pointeur) ;

st_rtsys_cancel_task(rtsys, job_id, code_retour) : suppression du travail job_id en précisant au système le code de retour code_retour. Les codes de retour reconnus par défaut sont : ST_TERM_STATUS_DEADLINE_MISS (indique un dépassement d'échéance) et ST_TERM_STATUS_WCET_OVERRUN (dépassement de temps d'exécution pire-cas). D'autres codes de retour quelconques sont autorisés ;

st_rtsys_yield(rtsys) : force une décision d'ordonnancement ;

st_rtsys_get_date(rtsys) : consultation de l'horloge système ;

st_rtsys_change_date(rtsys, op, val) : modification de la date système. Suivant la valeur de op il peut s'agir d'un positionnement à la date val absolue, d'une incrémentation ou d'une décrémentation de val unités de temps ;

st_rtsys_set_irq_mask(rtsys, masque) et st_rtsys_get_irq_mask(rtsys, masque) : modification/consultation du masque de filtrage global des interruptions ;

st_rtsys_set_local_interrupts_state(rtsys, etat) et st_rtsys_get_local_interrupts_state(rtsys) : modification/consultation du filtrage local des interruptions du contexte d'exécution courant ;

st_rtsys_post_message(rtsys, message) : génère un message de simulation message.

1.5.2 Fonctions de rappel du système d'exploitation simulé

Le support d'exécution simulé peut personnaliser les fonctions de rappel suivantes du module de simulation de système désigné par myself ci-dessous :

isr(myself, level) : traitant d'interruptions unique pour tous les niveaux (doit être réentrant). Appelé à chaque fois qu'une interruption doit être traitée, en précisant le niveau (paramètre level) ;

syscall(myself, id, arg) : traitant d'appel système unique pour tous les appels système définis (doit être réentrant). Appelé à chaque fois que la fonction st_rtsys_syscall() ci-dessus est appelée ;

next_task_id reschedule(myself) : fonction de rappel d'ordonnancement en charge de retourner next_task_id, l'identifiant du prochain travail à exécuter ;

notify_task_activation(myself, job_id) : signale l'activation du travail job_id ;

notify_task_termination(myself, job_id, code_retour) : signale la fin du travail job_id, soit normalement (code retour positionné à ST_TERM_STATUS_NORMAL_END), soit suite à une suppression par st_rtsys_cancel_task().

1.5.3 Exemple de système simulé

Nous présentons un exemple de système simulé dont le fonctionnement est le suivant :

- Les tâches sont ordonnancées suivant EDF (ordonnanceur fourni par défaut) sans garantie ;
- Une interruption de niveau 0 correspond à l'interruption d'horloge, qui provoque une augmentation de l'horloge système de 10ms ;
- Une interruption de niveau 1 correspond à l'activation de la tâche dont le modèle est `task_model` (non décrit ci-dessous) ;
- L'exécution de chaque traitant d'interruption occupe 1ms de temps simulé.

1.5.3.1 Version `C`

La version `C` repose sur l'initialisation correcte des fonctions de rappel. Dans l'exemple, les seules fonctions de rappel fournies sont celles de traitement d'interruption (`isr`) et d'ordonnancement (utilise la fonction `st_rtsys_edf()` fournie par défaut). La macro `ST_TIME_CONST(var, day, hour, min, sec, ms, us, ns)` initialise `var` pour être la représentation du temps définie par les paramètres `day`, `hour`, ...

```
/* Modele de tache */
extern struct st_task_model_t task_model; /* Defini ailleurs */

/* Fonction de rappel de traitant d'interruptions */
static void isr(struct st_rtsys_descr_t *sd,
                st_int irq_level)
{
  st_time_t dt;

  if (irq_level == 0) {
    /* Interruption d'horloge => mise a jour de la date systeme */
    ST_TIME_CONST(dt, 0,0,0,0,10,0,0); /* 10ms */
    st_rtsys_change_date(sd, INC_DATE, & dt);

  } else if (irq_level == 1) {
    /* Interruption 1 => creation d'un nouveau travail de tache */
    struct st_task_status_t *st = malloc (sizeof(struct st_task_status_t));
    st_rtsys_activate_task(sd, & task_model, st, 0);
  }

  ST_TIME_CONST(dt, 0,0,0,0,1,0,0); /* 1ms */
  st_rtsys_hold_cpu(sd, & dt);
}

st_inout_descr_t * creer_simulateur_systeme()
{
  struct st_rtsys_callbacks_t rtsys_cb;
  struct st_rtsys_status_t sstatus;

  /* Creation du simulateur de systeme avec st_rtsys_edf comme
     fonction de rappel reschedule() */
  memset(& rtsys_cb, 0x0, sizeof(struct st_rtsys_callbacks_t));
  rtsys_cb.isr = isr;
  rtsys_cb.reschedule = st_rtsys_edf;
```

```
  return st_rtsys_new(& rtsys_cb, & sstatus, 10, NULL, NULL, NULL, NULL);
}
```

1.5.3.2 Version C++

La version C++ encapsule les fonctions de rappel et l'interface de programmation décrite ci-dessus dans une même classe. Elle utilise l'héritage depuis l'ordonnanceur EDF simple de la famille JFP décrite en 8.1.3 de la partie II. Pour définir et manipuler des temps, on utilise la class st_Time fournie, qui définit les objets statiques st_Time::hour, st_Time::min, ... st_Time::ms et l'opérateur parenthèses "(nombre)" de multiplication qui permet d'écrire le temps 10ms sous la forme "st_Time::ms(10)".

```
extern MyTaskModel task_model; // Defini ailleurs

class Demo_RTSys : public st_Sched_EDF {
public:
  /* Constructeur */
  Demo_RTSys(st_int max_tasks,
             const st_Time & clock_resolution,
             JFPTask * init_task,
             void * init_task_args = NULL)
    : st_Sched_EDF(max_tasks, clock_resolution,
                   init_task, init_task_args)
  { };

protected:

  /* Fonction de rappel de traitement d'interruption */
  virtual void cb_isr(st_int irq_level)
  {
    if (irq_level == 0) {
      /* Interruption d'horloge => mise a jour de la date systeme */
      change_date(INC_DATE, st_Time::ms(10)); // 10ms

    } else if (irq_level == 1) {
      /* Interruption 1 => creation d'un nouveau travail de tache */
      activate_task(new JFPTask(& task_model));
    }

    hold_cpu(st_Time::ms(1)); // 1ms
  }
};

st_inout_descr_t * creer_simulateur_systeme()
{
  /* Nouveau module de simulation de systeme avec 10 travaux maximum,
     de resolution d'horloge systeme 10ms */
  return new Demo_RTSys(10, st_Time::ms(10));
}
```

1.5.4 Diagramme de transition

Le diagramme de transition suivant (figure A.1) représente les différents états de l'automate qui régit les aller-retours entre le système simulé, et le contexte d'exécution (dit *originel*) du circuit de simulation. Il correspond à l'algorithme 7.1 décrit dans la partie II, mais apporte des informations sur les messages d'instrumentation intermédiaires qui sont générés (les étiquettes sur les transitions, qui ne correspondent donc pas à des événements qui les tireraient). On peut distinguer 4 régions dans ce diagramme :

- En haut : le corps de l'automate, chargé de déterminer si l'événement à traiter suivant est la fin de la simulation d'occupation processeur par le contexte d'exécution courant, ou si c'est un message de simulation provenant de l'environnement simulé ;
- À gauche : l'instrumentation des début, préemption, reprise (*i.e. allers* vers le contexte d'exécution correspondant), annulation et terminaison de travaux de tâches ;
- À droite : l'instrumentation des levée, masquage global, mise en attente locale, traitement (*i.e. allers* vers le contexte d'exécution correspondant) et fin de traitement des interruptions ;
- En bas : les transitions correspondant au *retour* depuis le contexte d'exécution d'une tâche ou d'un traitant d'interruptions, vers le contexte d'exécution originel.

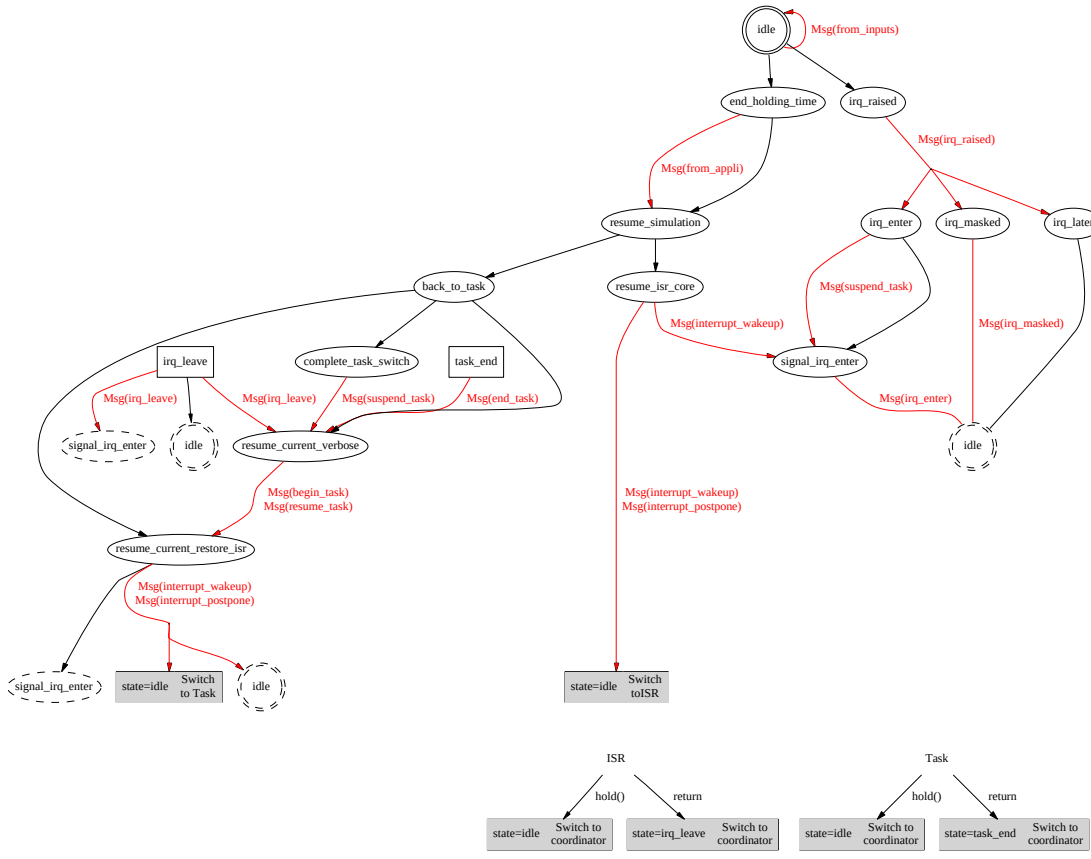


FIG. A.1: Diagramme de transition de l'automate du module de simulation de système temps-réel

Articles référencés

[AB98a] Luca Abeni and Giorgio C. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real-Time System Symposium*, pages 4–13, 1998.

[AB98b] Alia Atlas and Azer Bestavros. An omniscient scheduling oracle for systems with harmonic periods. Technical Report 1998-014, Sept 1998.

[AB98c] Alia K. Atlas and Azer Bestavros. Design and implementation of statistical rate monotonic scheduling in kurt linux. Technical Report 98-013, Boston University, Sept 1998.

[AB98d] Alia K. Atlas and Azer Bestavros. Statistical rate monotonic scheduling. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Dec 1998.

[AB99] S. Aldarmi and A. Burns. Dynamic value-density for scheduling real-time systems. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, Dec 1999.

[ABR+93] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, Sep 1993.

[ABRW91] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atalanta, 1991.

[ABRW94] N. C. Audsley, Alan Burns, M. F. Richardson, and Andy J. Wellings. STRESS: a simulator for hard real-time systems. *SP&E*, 24(6):543–564, 1994.

[ALB99] L. Abeni, G. Lipari, and G. Buttazzo. Constant bandwidth vs. proportional share resource allocation. In *Proceedings of the International Conference on Multimedia Computing and Systems*, volume II, pages 107–111, Jun 1999.

[AP97] E. Anceaume and I. Puaut. A taxonomy of clock synchronization algorithms. Technical Report PI1103, IRISA, July 1997.

[ATB93] N.A. Audsley, K. Tindell, and A. Burns. The end of the line for static cyclic scheduling? In *Proceedings of the Fifth Euromicro Workshop on Real-time Systems*, pages 36–41, Jun 1993.

[Aud91] N. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS164, U York, Nov 1991.

[Bak91] T.P. Baker. Stack-based scheduling of realtime processes. *Journal of Real-Time Systems*, 3(1):67–100, 1991.

[Bar97] Michael Barabanov. A linux-based real-time operating system. Master's thesis, New Mexico Institute of Mining and Technology, Jun 1997.

[Bar98] Sanjoy Baruah. A general model for recurring real-time tasks. In *Proceedings of the Real-Time Systems Symposium*, pages 114–122, 1998.

[Bat98] IJ Bates. *Scheduling and Timing Analysis for Safety Critical Real-Time Systems*. PhD thesis, University of York, Nov 1998.

[BBL01] G. Bernat, A. Burns, and A. Llamosí. Weakly hard real-time systems. In IEEE, editor, *IEEE Transactions on Computers*, number 50(4), pages 308–321, Apr 2001. An extended version is also available as a technical report YCS-99-320 September 1999.

[BCG+99] Sanjoy Baruah, Deji Chen, Sergey Gorinsky, , and Aloysius Mok. Generalized multiframe tasks. *Real-Time Systems*, 1(17):5–22, Jul 1999.

[BCM99] S. Baruah, D. Chen, and A. Mok. Static-priority scheduling of multiframe tasks. In *Proceedings of the Euromicro Conference on Real-Time Systems*, Jun 1999.

[BCP02] G. Bernat, A. Colin, and S. Petters. Wcet analysis of probabilistic hard real-time systems. In *23rd IEEE Real-Time Systems Symposium*, pages 279–188, Dec 2002.

[BG92] Gerard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BGG+94] A. Benveniste, T. Gautier, P. Le Guernic, G. Berry, F. Mignard, P. Caspi, N. Halbwachs, P. Couronné, F. Dupont, C. LeMaire, J.P. Paris, and Y. Sorel. Synchronous technology for real-time systems. In *RTS'94*, pages 105–122, 1994.

[BH98] Sanjoy Baruah and Mary Ellen Hickey. Competitive on-line scheduling of imprecise computations. *IEEE Transactions on Computers*, 47(9):1027–1032, Sep 1998.

[BHR93] Sanjoy K. Baruah, Rodney R. Howell, and Louis E. Rosier. Feasibility problems for recurring tasks on one processor. *Theoretical Computer Science*, 118(1):3–20, 1993.

[BHS01] Sanjoy Baruah, Jayant Haritsa, and Nitin Sharma. On-line scheduling to maximize task completions. *Combinatorial Mathematics and Combinatorial Computing*, (39):65–78, 2001.

[BKM+91] S. Baruah, G. Koren, B. Mishra, D. Mao, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang. On the competitiveness of on-line real-time task scheduling. In *Proceedings of the 12th Real-Time Systems Symposium*, pages 106–115, Dec 1991.

[BMR90] Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *IEEE Real-Time Systems Symposium*, pages 182–190, 1990.

[BPB+00] A. Burns, D. Prasad, A. Bondavalli, F. Di Giandomenico, K. Ramamritham, J. Stankovic, and L. Stringini. The meaning and role of value in scheduling flexible real-time systems. *Journal of Systems Architecture*, 46:305–325, 2000.

[BS93] G. Buttazzo and J. Stankovic. Red: Robust earliest deadline scheduling. In *Proceedings of the International Workshop on Responsive Computing Systems*, 1993.

[BSS94] A. Bondavalli, J. Stankovic, and L. Strigini. Adaptable fault tolerance for real-time systems. Technical Report UM-CS-1994-039, University of Massachusetts, Amherst, Computer Science, May, 1994.

[BSS95] G. Buttazzo, M. Spuri, and F. Sensini. Value vs. deadline scheduling in overload conditions. In *Proceedings of the 15th Real-Time System Symposium*, pages 90–99, Dec 1995.

[BSY88] D. F. Bacon, J. Schwartz, and Y. Yemini. Nest: A network simulation and prototyping tool. In *Proceedings of the USENIX Winter 1988 Technical Conference*, pages 71–78, Berkeley, CA, 1988. USENIX Association.

[Bur93] A. Burns. Fixed priority scheduling with deadlines prior to completion. Technical Report YCS212, 1993.

[Bur94] A. Burns. Preemptive priority based scheduling: An appropriate engineering approach. Technical Report 214, 1994.

[But97] G.C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer academic, 1997.

[BWH93] A. Burns, A. J. Wellings, and A. D. Hutcheon. The impact of an ada runtime system's performance charactersitics on scheduling models. In *Proceedings of the 12th Ada Europe conference*, 1993.

[CA97] Seonho Choi and Ashok K. Agrawala. Scheduling aperiodic and sporadic tasks in hard real-time systems. Technical Report CS-TR-3794, 1997.

[Cas00] Paolo Castelpietra. *Modélisation modulaire pour l'évaluation des performances d'application embarquées dans l'automobile*. Diplôme Supérieur de Formation Industrielle par la Recherche, Mention Firtech, Sept 2000.

[CB97] M. Caccamo and G. Buttazzo. Exploiting skips in periodic tasks for enhancing aperiodic responsiveness. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, 1997.

[CCS02] L. Cucu, R. Cocif, and Y. Sorel. Real-time scheduling for systems with precedence, periodicity and latency constraints. In *Proceedings of the Conférence Internationale sur les Systèmes Temps-Réel*, Mar 2002.

[CFBW93] C.Bailey, E. Fyfe, A. Burns, and A. J. Wellings. The olympus attitude and orbital control system, a case study in hard real-time system design and inplementation. Technical report, University of York, 1993.

[CGF97] Cubert, Goktekin, and P. A. Fishwick. Moose: architecture of an object-oriented multimodeling simulation system. In *Proceedings of Enabling Technology for Simulation Science*, Apr 1997.

[CJ98] George Candea and Michael B. Jones. Vassal: Loadable scheduler support for multi-policy scheduling. In *Second USENIX Windows NT Symposium*, pages 157–166, Seattle, WA, August 1998. USENIX.

[CLB99] Marco Caccamo, Giuseppe Lipari, and Giorgio Buttazzo. Sharing resources among periodic and aperiodic tasks with dynamic deadlines. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, 1999.

[Col01] A. Colin. *Estimation de temps d'exécution au pire cas par analyse statique et application aux systèmes d'exploitation temps-réel*. PhD thesis, Université de Rennes I, Octobre 2001.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[Coo83] Stephen A. Cook. An overview of computational complexity. *Communications of the ACM*, 26(6):400–408, 1983.

[CP99a] P. Chevochot and I. Puaut. An approach for fault-tolerance in hard real-time distributed systems. In *Proc. 18th IEEE Symposium on Reliable Distributed Systems (SRDS'99) (short paper)*, pages 292–293, Lausanne, Switzerland, October 1999.

[CP99b] P. Chevochot and I. Puaut. Tolérance aux fautes dans les systèmes répartis temps-réel strict. *Techniques et Sciences Informatiques (TSI)*, 18(8):837–870, October 1999.

[CP00] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems, Special issue on worst-case execution time analysis*, 18(2):249–274, April 2000.

[CP01a] A. Colin and I. Puaut. Analyse de temps d'exécution au pire cas du système d'exploitation temps-réel RTEMS. In *Seconde Conférence Française sur les Systèmes d'Exploitation (CFSE2)*, pages 73–84, Paris, France, April 2001.

[CP01b] A. Colin and I. Puaut. A modular and retargetable framework for tree-based wcet analysis. In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, pages 37–44, Delft, The Netherlands, June 2001.

[CP01c] A. Colin and I. Puaut. Worst-case execution time analysis of the RTEMS real-time operating system. In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, pages 191–198, Delft, The Netherlands, June 2001.

[CPC+00] P. Chevochot, I. Puaut, G. Cabillic, A. Colin, D. Decotigny, and M. Banâtre. Hades: a distributed system for dependable hard real-time applications built from COTS components. Technical Report PI1357, IRISA, October 2000.

[CPRS03] A. Colin, I. Puaut, C. Rochange, and P. Sainrat. Calcul de majorants de pire temps d'exécution : état de l'art. *Techniques et Sciences Informatiques (TSI)*, 22(5):649–675, 2003.

[CSSLC00] Paolo Castelpietra, Ye-Quiong Song, Françoise Simonot-Lion, and O. Cayrol. Performance evaluation of a multiple networked in-vehicle embedded architecture. In *3rd IEEE International Workshop on Factory Communication Systems*, Sept 2000.

[Dav93] R. Davis. Approximate slack stealing algorithms for fixed priority preemptive systems. Technical Report 216, U. York, Dec 1993.

[Dav94] R. Davis. Dual priority scheduling: A means of providing flexibility in hard real-time systems. Technical Report YCS230, University of York, UK, May 1994.

[DC01] L. David and F. Cottet. Traitement de la gigue de tâches dépendantes dans un contexte d'ordonnancement temps réel en ligne. In *Real Time Systems, Paris*, 2001.

[Dec02] David Decotigny. *Using Artisst: a Tutorial*, 1999-2002.

[Del96] J. Delacroix. Towards a stable earliest deadline scheduling algorithm. *Real-Time Systems*, 10:263–291, 1996.

[DFP01] Radu Dobrin, Gerhard Fohler, and Peter Puschner. Translating offline schedules into task attributes for fixed priority scheduling. *22nd IEEE Real-Time Systems Symposium, December 2001, London, United Kingdom*, Dec. 2001.

[DFW02] W. Dinkel, M. Frisbie, and J. Woltersdorf. *Kurt-Linux User Manual*. UC Kansas at Lawrence, Mar 2002.

[DG01] Loic Dachary and Philippe Gerum. *CarbonKernel User Manual*, 1.3 edition, July 2001.

[DHGP01] José María Drake, Michael González Harbour, José Javier Gutiérrez, and José Carlos Palencia. Mast: Modeling and analysis suite for real time applications. In *13th Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, June 2001.

[DK98] J. Delacroix and C. Kaiser. Un modèle de tâches temps réel pour la résorption contrôlée des surcharges. In *RTS 98*, Paris, Jan 1998.

[DM00] J. Delacroix and C. Ménival. Integration d'un contrôle de charge par importance au sein du système rt-linux. In *RTS 2000*, Paris, 2000.

[Dru96] John Drummond. Establishing a real-time distributed benchmark. In *Proceedings of the International Workshop on Parallel and Distributed Real-Time Systems*, Apr 1996.

[DTB93] R. I. Davis, K. W. Tindell, and A. Burns. Scheduling slack time in fixed priority preemptive systems. In *Proceedings of the Real-Time Systems Symposium*, pages 222–231, 1993.

[EC99] J. Eker and A. Cervin. A matlab toolbox for real-time and control systems co-design. In *Proceedings of the IEEE International Conference on Real-Time Computing Systems and Applications*, 1999.

[EJ00] Cecilia Ekelin and Jan Jonsson. Solving embedded system scheduling problems using constraint programming. In *IEEE RTSS*, May 2000.

[fIT93] IEEE Standard for Information Technology. *Portable Operating System Interface 1003.1b & 1003.1c*. IEEE, 1993.

[FNSB94] P. Fishwick, N. Narayanan, J. Sticklen, and A. Bonarini. A multi-model approach to reasoning and simulation. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(10), 1994.

[Foh94] Gerhard Fohler. *Flexibility in Statically Scheduled Hard Real-Time Systems*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 1994.

[GAGB01] Paolo Gai, Luca Abeni, Massimiliano Giorgi, and Giorgio Buttazzo. A new kernel approach for modular real-time systems development. In *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, June 2001.

[Gar99] M. K. Gardner. *Probabilistic Analysis and Scheduling of Critical Soft Real-Time Systems*. PhD thesis, University of Illinois at Urbana-Champaign, Sep 1999.

[GH98] J. Goossens and Ch. Hernalsteen. A tool for statistical analysis of hard real-time systems. In *Proceedings of The 31st Annual Simulation Symposium*, pages 58–65, Apr 1998.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of Np-Completeness*. Number ISBN 0716710455. W H Freeman & Co., 1979.

[GL95] Donald W. Gillies and Jane W.-S. Liu. Scheduling tasks with AND/OR precedence constraints. *SIAM J. Comput.*, 24(4):797–810, 1995.

[GL99] Mark K. Gardner and Jane W. S. Liu. Performance of algorithms for scheduling real-time systems with overrun and overload. In *Proceedings of the Eleventh Euromicro Conference on Real-Time Systems*, Jun 1999.

[GM99] J. Goossens and C. Macq. Performance analysis of various scheduling algorithms for real-time systems composed of aperiodic and periodic tasks. In *CISSAS'99*, 1999.

[GMR95] L. George, P. Muhletahler, and N. Rivierre. Optimality and non-preemptive real-time scheduling revisited. Technical Report 2516, INRIA, 1995.

[GNM99] Bruno Gaujal, Nicolas Navet, and Jorn Migge. Dual-priority versus background scheduling: A path-wise comparison. Technical-report, Inria, Institut National de Recherche en Informatique et en Automatique, July 1999.

[GP96] R. Gopalakrishnan and Guru M. Parulkar. Bringing real-time scheduling theory and practice closer for multimedia computing. In *Measurement and Modeling of Computer Systems*, pages 1–12, 1996.

[GRS96] L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive real-time uniprocessor scheduling. Technical Report 2966, INRIA Rocquencourt, sep 1996.

[GS96] M. Gergeleit and H. Streich. Taskpair-scheduling with optimistic case execution times – an example for an adaptive real-time system. In *Second International Workshop on Object-oriented Real-time Dependable Systems, Laguna Beach, CA*, GMD, Germany, February 1,2 1996.

[GSSR97] O. Gonzalez, H. Shrikumar, John A. Stankovic, and K. Ramamritham. Adaptive fault tolerance and graceful degradation under dynamic hard real-time constraints. In *Proceedings of the 18th IEEE Real-Time Systems Symposium, San Francisco, California*. U. Mass, December 1997.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[Hei93] P. Heidmann. A statistical model for designers of rate monotonic systems. In *Proceedings of the Software Engineering Institute's Second Annual Rate Monotonic User's Forum*, Nov 1993.

[Heu01] Arnd Christian Heursch. Preemption concepts, rhealstone benchmark and scheduler analysis of linux 2.4. In *Proceedings of the Real-Time & Embedded Computing Expo & Conference*, 2001.

[HMT90] Chetto H., Silly M., and Bouchentouf T. Dynamic scheduling of real-time task under precedence constraints. *Real Time Systems*, (2):181–194, 1990.

[HS] Hans A. Hansson and Mikael Sjödin. An off-line scheduler and system simulator for the basement distributed real-time system.

[HS90] D. Haban and K.G. Shin. Application of real-time monitoring to scheduling tasks with random execution times. *IEEE Transactions on Software Engineering*, 16(2):1374–1389, 1990.

[HV95] Rodney R. Howell and Muralidhar K. Venkatrao. On non-preemptive scheduling of recurring tasks using inserted idle times. *Information and Computation*, 117(1):50–62, 1995.

[IF99] Damir Isovic and Gerhard Fohler. Online handling of firm aperiodic tasks in time triggered systems. In *11th Euromicro Conference on Real-Time Systems*, York, UK, June 1999.

[IMR96] Hong Inki, Potkonjak Miodrag, and Karri Ramesh. Hetergeneous BISR-approach using system level synthesis flexibility. Technical Report 960047, University of California, Los Angeles, Computer Science Department, December 31, 1996.

[JD90] Xu J. and Parnas D. Scheduling processes with release times, deadlines, precedence, and exclusion, relations. *IEEE Transactions on Software Engineering*, 16(3):360–369, 1990.

[Jef92] Kevin Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *IEEE Real-Time Systems Symposium*, pages 89–99, 1992.

[JG99] Kevin Jeffay and Steve Goddard. A theory of rate-based execution. In *Proceedings of the of the 20th IEEE Real-Time Systems Symposium*, Dec 1999.

[JG01] K. Jeffay and S.M. Goddard. Rate-based resource allocation models for embedded systems. In *Proceedings of the First International Workshop on Embedded Software*, pages 204–222, Oct 2001.

[JHA02] Rohit Jain, Christopher J. Hughes, and Sarita V. Adve. Soft real- time scheduling on simultaneous multithreaded processors. In *23rd IEEE Real-Time Systems Symposium*, pages 134–145, Dec 2002.

[Jon98] J. Jonsson. GAST : A flexible and extensible tool for evaluating multiprocessor assignment and scheduling techniques. In *11th Conf. on Parallel Processing, Minneapolis, Minnesota*, pages 441–451, Aug 1998.

[JP86] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[JRR97] Michael B. Jones, Daniela Rosu, and Marcel-Catalin Rosu. Cpu reservations and time constraints: Efficient, predictable scheduling of independent activities. In *ACM Symposium on Operating Systems Principles*, Oct 1997.

[JS93] K. Jeffay and D. L. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, pages 212–221, Dec 1993.

[JSM91] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In IEEE, editor, *Proceedings of the 12 th IEEE Symposium on Real-Time Systems (December 1991)*, pages 129–139, december 1991.

[JV96] Jan Jonsson and Jonas Vasell. Evaluation and comparison of task allocation and scheduling methods for distributed real-time systems. In IEEE, editor, *IEEE Workshop on Real-Time Applications (RTAW)*, Montreal, Canada, Oct 1996.

[JV97] Jan Jonsson and Jonas Vasell. Feast: A framework for evaluation of allocation and scheduling techniques for real-time systems. In *National Swedish Conference on Real-Time Systems*. U. Chalmers, Aug 1997.

[Kai81] Claude Kaiser. De l'utilisation de la priorité en presence d'exclusion mutuelle. Technical Report 84, INRIA, 1981.

[Kai01] Laurent Kaiser. *Contribution à l'analyse des TIOSMs pour la vérification de propriétés temporelles de systèmes complexes*. PhD thesis, Institut National Polytechnique de Lorraine, Mar 2001.

[KAS93] Daniel I. Katcher, Hiroshi Arakawa, and Jay K. Strosnider. Engineering and analysis of fixed priority schedulers. *Software Engineering*, 19(9):920–934, 1993.

[Kat98] J.P. Katoen. Concepts, algorithms, and tools for model checking. Lecture notes of the course "Mechanised Validation of Parallel Systems", 1998.

[KFG+92] H. Kopetz, G. Fohler, G. Grünsteidl, H. Kantz, G. Pospischil, P. Puschner, J. Reisinger, R. Schlatterbeck, W. Schütz, A. Vrchotichy, and R. Zainlinger. The programmer's view of MARS. In Robert Werner, editor, *Proceedings of the Real-Time Systems Symposium - 1992*, pages 223–226, Phoenix, Arizona, USA, December 1992. IEEE Computer Society Press.

[KNH+97] Hermann Kopetz, Roman Nossal, René Hexel, Andreas Krüger, Dietmar Millinger, Roman Pallierer, Christopher Temple, and Markus Krug. Mode handling in the time-triggered architecture. *IFAC DCCS 97, June 1997, Seoul, Korea*, Jun. 1997.

[KNT99] M. Kerboeuf, D. Nowak, and J.-P. Talpin. The steam-boiler problem in signal-coq. Technical Report 3773, INRIA, Oct 1999.

[KS93] G. Koren and D. Shasha. D^{over}: An optimal online scheduling algorithm for overloaded realtime systems. In *Proceedings IEEE Real-Time Systems Symposium*, pages 290–299, 1993.

[KS95] G. Koren and D. Shasha. Skip-over: algorithms and complexity for overloaded systems that allow skips. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, 1995.

[KSSR96] H. Kaneko, J. A. Stankovic, S. Sen, and K. Ramamritham. Integrated scheduling of multimedia and hard real-time tasks. Technical Report UM-CS-1996-045, University of Massachusetts, Amherst, Computer Science, December, 1996.

[Law73] E.L. Lawler. Optimal sequencing of a single machine subject to precedence constraints. *Management Science*, 19, 1973.

[Law78] E.L. Lawler. Sequencing jobs to minimize total weighted completion time. *Annals of Discrete Mathematics*, 2:75–90, 1978.

[LB00a] G. Lipari and G.C. Buttazzo. Schedulability analysis of periodic and aperiodic tasks with resource constraints. *Journal of System Architectures*, 46:327–338, 2000.

[LB00b] Giuseppe Lipari and Sanjoy Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Proceedings of the Real-Time Technology and Applications Symposium*, pages 166–175, May 2000.

[LB01] G. Lipari and S.K. Baruah. A hierarchical extension to the constant bandwidth server framework. In *Proceedings of the Real-Time Technology and Application Symposium*, 2001.

[LD01] N. Nissanke L. David, F. Cottet. Jitter control in on-line scheduling of dependent real-time tasks. In *22nd IEEE Real-Time Systems Symposium*, pages 49–58, 2001.

[Liu00] Jane W.S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LLD+96] J. W. S. Liu, C. L. Liu, Z. Deng, T. S. Tia, J. Sun, M. Storch, D. Hull, J. L. Redondo, R. Bettati, and A. Silberman. PERTS: A prototyping environment for real-time systems. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):161–177, 1996.

[LLS+91] Jane W.-S. Liu, Kwei-Jay Lin, Wei Kuan Shih, Albert Chuang shi Yu, Jen-Yao Chung, and Wei Zhao. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5):58–68, 1991.

[LMA88] S. T. Levi, D. Mosse, and A. K. Agrawala. Allocation of real-time computations under fault-tolerance constraints. In *Proceedings IEEE Real-Time Systems Symposium*, pages 161–170, 1988.

[LRD+93] J. W. S. Liu, J. L. Redondo, Z. Deng, T. S. Tia, R. Bettati, A. Silberman, M. Storch, R. Ha, and W. K. Shih. PERTS: A prototyping environment for real-time systems. Technical Report UIUCDCS-R-93-1802, Department of Computer Science, University of Illinois at Urbana-Champaaign, May 1993.

[LSA+00] C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. H. Son, and M. Marley. Performance specifications and metrics for adaptive real-time systems. In *Proceedings of the 21th IEEE Real-Time Systems Symposium*, Dec 2000.

[LSD89] John P. Lehoczky, Lui Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium*, pages 166–171, 1989.

[LT94] John P. Lehoczky and Sandra R. Thuel. Scheduling periodic and aperiodic tasks using the slack stealing algorithm. *Advances in Real-Time Systems*, pages 172–195, 1994.

[LW82] J. Leung and J.W. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4), 1982.

[Man98] G. Manimaran. *Resource management with dynamic scheduling in parallel and distributed real-time systems*. PhD thesis, Indian Institute of Technology, Madras, jan 1998.

[MB97] C. McElhone and A. Burns. Scheduling optional computations for adaptive real-time systems. Technical Report YCS289, 1997.

[MC96a] Aloysius K. Mok and Deji Chen. A general model for real-time tasks. Technical Report CS-TR-96-24, 1996.

[MC96b] Aloysius K. Mok and Deji Chen. A multiframe model for real-time tasks. In *Proceedings of the Real-Time Systems Symposium*, 1996.

[McE94] C. McElhone. Adapting and evaluating algorithms for dynamic schedulability testing. Technical report, AC York, Feb 1994.

[MF02] Aloysius K. Mok and Alex Xiang Feng. Real-time virtual resource: A timely abstraction for embedded systems. In *Second International Conference on Embedded Software, EMSOFT 2002, LNCS 2491*, Oct 2002.

[Mig99] Jörn Migge. *L'ordonnancement sous contraintes temps-réel : un modèle à base de trajectoires*. PhD thesis, INRIA Sophia Antipolis, Nov 1999.

[MMM00a] G. Manimaran, A. Manikutty, and C Siva Ram Murthy. Dharma: A tool for evaluating dynamic scheduling algorithms for real-time multiprocessor systems. *Journal of Systems and Software*, 50(2):131–149, Feb 2000.

[MMM00b] A. Mittal, G. Manimaran, and C. Siva Ram Murthy. Integrated dynamic scheduling of hard and qos degradable real-time tasks in multiprocessor systems. *to appear in Journal of Systems Architecture*, 2000.

[Mok83] A.K. Mok. *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, Massachusetts Institute of Technology, Jun 1983.

[MRW92] A. D. Malony, D. A. Reed, and H. A. G. Wijshoff. Performance measurement intrusion and perturbation analysis. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):433–450, Jul 1992.

[MS95] M. Marucheck and J. Strosnider. An evaluation of the graceful degradation properties of real-time schedulers. In *25th Annual International Symposium on Fault-Tolerant Computing*, June 1995.

[MVP+96] Richard C. Metzger, Brian VanVoorst, Luiz S. Pires, Rakesh Jha, Wing Au, Minesh Amin, David A. Castanon, and Vipin Kumar. The c3i parallel benchmark suite - introduction and preliminary results. In *Proceedings of the SUPERCOMPUTING '96*, 1996.

[MZ93] C. E. Moron and H. Zedan. Adaptable scheduler using milestones for hard real-time systems. Technical Report 191, UYork, 1993.

[Nav99] Nicolas Navet. *Évaluation de performances temporelles et optimisation de l'ordonnancement de tâches et messages*. PhD thesis, Institut National Polytechnique de Lorraine, Nov 1999.

[Net97] E. Nett. Real-time behaviour in a heterogeneous environment. In *Third International Workshop on Object-oriented Real-time Dependable Systems, Newport Beach, CA*, GMD, Germany, February 6-7 1997.

[NG97a] E. Nett and M. Gergeleit. Preserving real-time behavior in dynamic distributed systems. In *IASTED International Conference on Artificial Intelligence and Soft Computing, The Bahamas*, GMD, Germany, December 8–10 1997.

[NG97b] E. Nett and M. Gergeleit. Preserving real-time behavior un dynamic distributed systems. In IEEE, editor, *IEEE IASTED International Conference on Intelligent Information Systems*, Grand Bahama Island, The Bahams, December 1997.

[NGM98] E. Nett, M. Gergeleit, and M. Mock. An adaptive approach to object-oriented real-time computing. In *Proceedings of ISORC'98, Kyoto, Japan*, GMD, Germany, April 20–22 1998.

[NGM01] E. Nett, M. Gergeleit, and M. Mock. Enhancing oo middleware to become time-aware. *RTS Journal*, ??:??, ? 2001.

[NGS97] E. Nett, M. Gergeleit, and H. Streich. Flexible resource scheduling and control in an adaptive real-time environment. In *IASTED International Conference on Artificial Intelligence and Soft Computing, Banff, Canada*, GMD, Germany, July 27 –August1 1997.

[Nic] David M. Nicol. Discrete-event simulation in performance evaluation. In *Performance Evaluation: Origins and Directions, LNCS 1769*.

[Nic98] Guillem Bernat Nicolau. *Specification and Analysis of Weakly Hard Real-Time Systems*. PhD thesis, Universitat de les Illes Balears, Jan 1998.

[Ost92] J. Ostroff. Formal methods for the specification and design of real-time safety critical systems. *Journal of Systems and Software*, 18(1), Apr 1992.

[PD02] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proc. of the 23rd IEEE International Real-Time Systems Symposium*, Austin, TX, USA, December 2002.

[Poo96] A Pool. Benchmarkingh of real-time high-performance computing systems. In *Proceedings of the International Workshop on Parallel and Distributed Real-Time Systems*, Apr 1996.

[Pua02] I. Puaut. Real-time performance of dynamic memory allocation algorithms. In *Proc. of the 14th Euromicro Conference on Real-Time Systems*, Vienna, Austria, June 2002.

[RH01] Mario Aldea Rivas and Michael González Harbour. Posix-compatible application-defined scheduling in marte os. Technical report, Universidad de Cantabria. SP., 2001.

[RJMO98] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shui Oikawa. Resource kernels: A resource-centric approach to real-time systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, Jan 1998.

[RRGC02] Michael Richard, P. Richard, E. Grolleau, and F. Cottet. Contraintes de précédences et ordonnancement mono-processeur. In *Proceedings of the Conférence Internationale sur les Systèmes Temps-Réel*, Mar 2002.

[RS94] K. Ramamritham and J. Stankovic. Scheduling algorithms and operating systems support for real-time systems. In IEEE, editor, *Proceedings of the IEEE*, volume 82, January 1994.

[RS01] J. Regehr and J. Stankovic. Hls: A framework for composing soft real-time schedulers. In *Proceedings of the Real-Time Systems Symposium*, pages 3–14, Dec 2001.

[RSH00] John Regehr, Jack Stankovic, and Marty Humphrey. The case for hierarchical schedulers with performance guarantees. Technical Report CS-2000-07, CS Virginia, 14, 2000.

[SA00] K. Subramani and Ashok Agrawala. A dual interpretation of "standard constraints" in parametric scheduling. In *Proceedings of the Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 121–133, Sep 2000.

[SAWJ+96] Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy Baruah, Johannes Gehrke, and C. Greg Plaxton. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1996.

[SB94] M. Spuri and G. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the 15th IEEE Real-Time System Symposium*, pages 2–21, 1994.

[SBS95] M. Spuri, G. Buttazzo, and F. Sensini. Robust aperiodic scheduling under dynamic priority systems. In *Proc. of the IEEE Real-Time Systems Symposium, Pisa, Italy*, dec 1995.

[SdSA95] M. Saksena, J. da Silva, and A. K. Agrawala. Design and implementation of maruti-ii. *Advances in Real-Time Systems*, pages 72–102, 1995.

[SG97a] H. Streich and M. Gergeleit. On the design of a dynamic distributed real-time environment. In *11th International Parallel Processing Symposium, University of Geneva, Switzerland*, GMD, Germany, April 1-5 1997. University of Geneva, Switzerland.

[SG97b] H. Streich and M. Gergeleit. On the design of a dynamic distributed real-time environment. In IEEE, editor, *IEEE Workshop Parallel Distributed Real-Time Systems (WPDRTS)*, Geneva, Switzerland, 1997.

[SGL97] Jun Sun, Mark K. Gardner, , and Jane W. S. Liu. Bounding completion times of jobs with arbitrary release times, variable execution times, and resource sharing. *IEEE Transactions on Software Engineering*, 23(10):603–615, Oct 1997.

[SL94] Matthew F. Storch and Jane W.-S. Liu. A simulation environment for distributed real-time systems. In *Proceedings of the SCS Simulation Multiconference*, April 1994.

[SL95] WK Shih and WS Liu. Algorithms for scheduling imprecise computations with timing constraints to minimize maximum error. *Proceedings of the IEEE Transactions on Computers*, 44(3), March 1995.

[SL96] Matthew F. Storch and Jane W.-S. Liu. DRTSS: A simulation framework for complex real-time systems. In *Proceedings, Real-Time Technology and Applications Symposium*, pages 160–169. IEEE, June 1996.

[SL99] Françoise Simonot-Lion. Une contribution à la modélisation et à la validation d'architectures temps-réel, Dec 1999. Habilitation à diriger des recherches, LORIA.

[SLS95] J.K. Strosnider, J.P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions On Computers*, 44(1):73–91, Jan 1995.

[SLS98] John A. Stankovic, Chenyang Lu, and Sang H. Son. The case for feedback control real-time scheduling. Technical Report CS-98-35, Department of Computer Science, University of Virginia, November 27 1998.

[SNF98] Kristian Sandström, Christer Norström, and Gerhard Fohler. Handling interrupts with static scheduling in an automotive vehicle control system. In *Proceedings of the International Conferance on Real-Time Computing Systems and Applications*, pages 158–165, Oct 1998.

[SP97] Steven Sommer and John Potter. Admissibility tests for interrupted earliest deadline first scheduling with priority inheritance. Technical Report C/TR97-10, U Macquarie, Australia, Jun 1997.

[Spr90] B. Sprunt. *Aperiodic task scheduling for real-time systems*. PhD thesis, Carnegie Mellon University, Aug 1990.

[Spu96] M. Spuri. Analysis of deadline scheduled real-time systems. Technical Report 2772, INRIA Rocquencourt, jan 1996.

[SRG89] W. Schwabl, J. Reisinger, and G. Grunsteidl. A survey of mars. Technical report, Institut fur Technische Informatik, Technical University Wein, 1989.

[SRL90] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sept 1990.

[SRN+98] John A. Stankovic, Krithi Ramamritham, Douglas Niehaus, Marty Humphrey, and Gary Wallace. The spring system: Integrated support for complex real-time systems. Technical Report CS-98-18, Department of Computer Science, University of Virginia, August 1 1998.

[SS93] Marco Spuri and John A. Stankovic. How to integrate precedence constraints and shared resources in real-time scheduling. Technical Report UM-CS-1993-019, U. Mass, 1993.

[SS02] A. Seznec and N. Sendrier. Havege : Hardware volative entropy gathering and expansion unpredictable random number generation at user level. In *Workshop on Random Number Generators and Highly Uniform Point Sets*, Montréal, June 2002.

[SSDB94] J. Stankovic, M. Spuri, M. DiNatale, and G. Buttazzo. Implications of classical scheduling results for real-time systems. Technical Report UM-CS-1994-089, U. Mass, 1994.

[SSNB94] J.A. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo. Implications of classical scheduling results for real-time systems. Technical report, Scuola Superiore S.Anna, Pisa - Italy, June 1994.

[Sta88] John A. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, 21(10):10–19, 1988.

[Sta93] John A. Stankovic. Reflective real-time systems. Technical Report UM-CS-1993-056, University of Massachusetts, Amherst, Computer Science, June, 1993.

[Str95] H. Streich. Taskpair-scheduling: An approach for dynamic real-time systems. *Int. Journal of Mini & Microcomputers*, 17, No. 2:77–83, 1995.

[SWR+99a] Behrooz Shirazi, Lonnie Welch, Binoy Ravindran, Charles Cavanaugh, Bharath Yanamula, Russ Brucks, , and Eui nam Huh. Dynbench: A dynamic benchmark suite for distributed real-time systems. In *Proceedings of the Workshop on Embedded HPC Systems and Applications*, 1999.

[SWR+99b] Behrooz Shirazi, Lonnie R. Welch, Binoy Ravindran, Charles Cavanaugh, Barath Yanamula, Russ Brucks, and Eui nam Huh. Dynbench: A dynamic benchmark suite for distributed real-time systems. In *IPPS/SPDP Workshops*, pages 1335–1349, 1999.

[TBW92a] K. Tindell, A. Burns, and A.J. Wellings. Allocating real-time tasks (an np-hard problem made easy). *Real-Time Systems*, 4(2):145–165, Jun 1992.

[TBW92b] Ken Tindell, Alan Burns, and Andy J. Wellings. Mode changes in priority pre-emptively scheduled systems. In *IEEE Real-Time Systems Symposium*, pages 100–109, 1992.

[TC94] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming - Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems)*, 40:117–134, 1994.

[TDS+95] T. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L. Wu, and J. Liu. Probabilistic performance guarantee for real-time tasks with varying computation times. In *Proceedings of the Real-Time Technology and Applications Symposium*, pages 164–173, may 1995.

[Tin92a] K. Tindell. An extendible approach for analyzing fixed priority hard real-time tasks. Technical Report YCS189, 1992.

[Tin92b] K. Tindell. Using offset information to analyse static priority pre-emptively scheduled task sets. Technical Report YCS182, Aug 1992.

[Tin93] K.W. Tindell. *Fixed Priority Scheduling of Hard Real-Time Systems*. PhD thesis, University of York, Dec 1993.

[Tin94] K. Tindell. Adding time-offsets to schedulability analysis. Technical Report YCS221, 1994.

[TLS95] T. Tia, J. Liu, and M. Shankar. Algorithms and optimality of scheduling aperiodic requests in fixed-priority preemptive systems. *Journal of Real-Time Systems*, 1995.

[TLSH94] T. Tia, W. Liu, J. Sun, and R. Ha. A linear-time optimal acceptance test for scheduling of hard real-time tasks, 1994.

[TNR90a] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Toward a Predictable Real-Time System. In *Proceedings of USENIX Mach Workshop*, pages 73–82, October 1990.

[TNR90b] Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao. Real-time Mach: Towards a predictable real-time system. In *Proceedings of the USENIX Mach Workshop*, pages 73–82, 1990.

[VGH96] S. De Vroey, J. Goossens, and Ch. Hernalsteen. A generic simulator of real-time scheduling algorithms. In *Proceedings of the 29th Simulation Symposium*, pages 242–249, Apr 1996.

[VJP97] Brian VanVoorst, Rakesh Jha, and Luiz Pires. A real-time parallel benchmark suite. *SIAM's Parallel Processing for Scientific Computing*, 1997.

[Wei89] N Weiderman. Hartstone: Synthetic benchmark requirements for hard real-time applications. Technical Report CMU/SEI-89-TR-023, CMU/SEI, 1989.

[WL99a] Yu-Chung Wang and Kwei-Jay Lin. Implementing a general real-time scheduling framework in the RED-linux real-time kernel. In *IEEE Real-Time Systems Symposium*, pages 246–255, 1999.

[WL99b] Yu-Chung Wang and Kwei-Jay Lin. Providing real-time support in the linux kernel. In *IEEE Real-Time Technology and Applications Symposium*, 1999.

[WS99a] Y. Wang and M. Saksena. Scheduling fixed priority tasks with preemption threshold. In *Proceedings of the IEEE International Conference on Real-Time Computing Systems and Applications*, 1999.

[WS99b] Lonnie R. Welch and Behrooz A. Shirazi. A dynamic real-time benchmark for assessment of qos and resource management technology. In *Real-Time Technology and Applications Symposium*, Jun 1999.

[WS02] Yun Wang and Manas Saksena. Scheduling fixed-priority tasks with preemption threshold: An attractive technology?, 2002. Regehr's readings.

[Zub98] Khawar M. Zuberi. *Real-Time Operating System Services for Networked Embedded Systems*. PhD thesis, University of Michigan, 1998.

Liens référencés

[1] ACSL.
http://www.acslsim.com.

[2] ADEVS.
http://www.acims.arizona.edu/DEVS_HLA/devs_hla.html
et http://www.ece.arizona.edu/~nutaro/.

[3] AKSL.
http://www.topology.org/src/aksl/.

[4] C++ Sim.
http://cxxsim.ncl.ac.uk/.

[5] Carbonkernel.
http://www.carbonkernel.org.

[6] CMU Sew.
http://home1.gte.net/kevinb1/SEW/SEW.html.

[7] CNCL.
http://www.comnets.rwth-aachen.de/doc/cncl.html.

[8] DSLib.
http://www.rtlinux.org/dsl/.

[9] ERS.
http://www-sop.inria.fr/mistral/soft/ers.html.

[10] GPSS/H.
http://www.meridian-marketing.com/GPSS_H
et http://www.wolverinesoftware.com/simulate.htm.

[11] Hyperformix Workbench.
http://www.hyperformix.com/products/workbench.htm.

[12] Javasim.
http://cxxsim.ncl.ac.uk.

[13] Lookheed Martin CSIM.
http://www.atl.external.lmco.com/proj/csim.

[14] LUCAS.
http://www.control.lth.se/research/projects2000/realtime.html.

[15] Mesquite CSim.
http://www.mesquite.com.

[16] Metasim.
http://metasim.sssup.it.

[17] Modline.
http://www.simulog.fr.

[18] Moose.
http://www.cise.ufl.edu/~fishwick/moose.html.

[19] MQXSim.
http://www.psti.com/Rtossim.html.

[20] OMNET++.
http://www.hit.bme.hu/phd/vargaa/omnetpp/.

[21] OPNet.
http://www.opnet.com.

[22] OSE Softkernel.
http://www.enea.com.

[23] Parsec.
http://pcl.cs.ucla.edu/projects/parsec/.

[24] PERTSS/DRTSS.
http://pertsserver.cs.uiuc.edu/software/.

[25] PSim.
http://science.kennesaw.edu/~jgarrido/psim.html.

[26] Ptolemy.
http://www.ptolemy.eecs.berkeley.edu.

[27] Ptolemy II.
http://ptolemy.eecs.berkeley.edu/ptolemyII/summary.htm.

[28] Quelques listes de références vers des simulateurs.
http://ubmail.ubalt.edu/~harsham/ref/RefSim.htm,
http://www.pscs.umich.edu/education/websites.html,
http://www.idsia.ch/~andrea/simtools.html,
http://www.topology.org/soft/sim.html.

[29] RAPIDS.
http://www.ecs.umass.edu/ece/realtime/recovery/simulator/.

[30] Sim++.
http://www.cis.ufl.edu/~fishwick/simpack/simpack.html.

[31] SimICS.
http://www.virtutech.com/simics/simics.html.

[32] SimOS.
http://simos.stanford.edu/.

[33] Simpack.
http://www.cis.ufl.edu/~fishwick/simpack/simpack.html.

[34] Simplescalar.
http://www.cs.wisc.edu/~mscalar/simplescalar.html.

[35] SimScript.
http://www.caciasl.com/simscript.cfm.

[36] SoftPC.
http://www.xsim.com/bib/index1.d/Index.html#Tool-SoftPC.

[37] Spacebell Schedsim.
http://www.spacebel.be/schedsim.htm.

[38] SPIM.
http://www.cs.wisc.edu/~larus/spim.html.

[39] SRMS Workbench.
http://cs-www.bu.edu/groups/realtime/SRMSworkbench/.

[40] TAPS.
http://www.kadak.com.

[41] The RTAI Programming Guide.
http://www.rtai.org
et http://www.aero.polimi.it/~rtai/documentation/index.html.

[42] Tripacific RAPID RMA.
http://www.tripac.com/html/prod-fact-rrm.html.

[43] UTSA Scheduling simulator.
http://vip.cs.utsa.edu/nsf/.

[44] VxSim.
http://www.wrs.com/products/html/vxsim_ds.html.

[45] Pierluigi Crescenzi, Viggo Kann, Magnús Halldórsson, Marek Karpinski, and Gerhard Woeginger, A compendium of np optimization problems.
http://www.nada.kth.se/~viggo/wwwcompendium/.

[46] Ralf S. Engelschall, GNU Portable threads.
http://www.gnu.org/software/pth.

[47] Red Hat, Sid.
http://sources.redhat.com/sid/.

[48] Kevin Lawton, Bochs.
http://bochs.sf.net.

[49] E. Lefevre, Scheduling simulator.
http://www.chez.com/elefevre/english/scheduler/scheduler.html.

[50] Glenn Reeves, What really happened on mars? Jan 1998.
http://catless.ncl.ac.uk/Risks/19.54.html#subj6.

[51] I. Ripoll, Interactive EDF Checker.
http://bernia.disca.upv.es/~iripoll/str/fuentes/iedfc/.

[52] Frank Singhoff, The Cheddar project.
http://beru.univ-brest.fr/~singhoff/cheddar/.

[53] Telelogic, ObjectGeode.
http://www.telelogic.se//objectgeode/default.asp.

[54] Timesys, Timewiz.
http://www.timesys.com/products/timewiz.html.

[55] Washington University, Nachos.
http://www.cs.washington.edu/homes/tom/nachos/.

[56] TRON Association Version-Up Working Group, Industrial Real-time Operating system Nucleus (ITRON).
http://tron.um.u-tokyo.ac.jp/TRON/ITRON/home-e.html.

Liste des tableaux

Table des figures

Index

S

T

U

V

W

Résumé

Un système informatique est temps-réel lorsque ses traitements doivent vérifier des propriétés d'ordre à la fois logique et temporel. Dans ce travail, nous proposons un outil de simulation pour l'évaluation de tels systèmes. Il peut venir compléter les méthodes sûres d'analyse statique, en particulier lorsque le comportement temporel du système ou de son environnement est insuffisamment caractérisé. L'outil met l'accent sur la faculté de personnalisation du système simulé, la grande fidélité des comportements temporels reproduits grâce à une granularité de simulation ajustable, la possibilité de réutiliser du code d'application existant, et l'efficacité de simulation. Nous présentons aussi un modèle objet générique pour l'ordonnancement dynamique couvrant un grand nombre d'ordonnanceurs existants, et qui a été évalué grâce à l'outil. Nous détaillons enfin les moyens de prise en compte de la granularité de l'horloge système dans ces algorithmes, ainsi qu'une évaluation de leur impact.

Mots-clefs : simulation, systèmes temps-réel, évaluation, centralisé, distribué, coûts système.

Abstract

A real-time computing system is one whose correctness depends on both the logical and the temporal properties of its computations. In this thesis, we propose a simulation tool to evaluate such systems. It may be used as a complement to safe static analysis methods, especially when the temporal behavior of the system or that of its environment is not fully characterized. We made this tool as customizable as possible, provided it with the capacity to reuse existing application code, tried to make it efficient, and allowed the simulated temporal behavior to be as close to the effective one as possible, thanks to the ability to adjust the timing resolution of the simulation. We also introduce a generic objet model for dynamic real-time scheduling, that can adapt to a wide variety of existing schedulers, and that has been assessed by the tool. We finally describe the way the system clock granularity can be taken into account in these algorithms, and evaluate its impact.

Keywords : simulation, real-time systems, evaluation, centralized, distributed, system overheads.