Le Temps dans les systèmes contraints

Mémoire présenté le
5 Décembre 2008
pour l'obtention de

l'Habilitation à Diriger des Recherches

en Sciences Mathématiques (spécialité informatique)

par

Samia Bouzefrane

Composition du jury :

Président :

Lionel Seinturier (Professeur – Université de Lille 1)

Rapporteurs :

Françoise Simonot (Professeur – École des Mines de Nancy)
Guy Juanole (Professeur – Université Paul Sabatier de Toulouse)
Jean-Louis Lanet (Professeur – Université de Limoges)

Examinateurs :

Claude Kaiser (Professeur émérite – Conservatoire National des Arts et Métiers)
Pierre Paradinas (Directeur INRIA et Professeur titulaire de chaire CNAM)

Directeur :

David Symplot-Ryl (Professeur – Université de Lille 1)

Université des Sciences et Technologies de Lille
Laboratoire d'Informatique Fondamentale de Lille – UMR 8022
UFR d'IEE – Bât. M3 – 59655 VILLENEUVE D'ASCQ CEDEX
Tél. 33 (0)3 28 77 85 41 – Télécopie : 33 (0)3 28 77 85 37
email : samia.bouzefrane@cnam.fr

I warraw-iw aâzizen Masensen, Taninna d Masilya

A mes chers enfants Masensen, Taninna et Masilya

Remerciements

Je remercie le professeur Lionel Seinturier, de l'université Lille 1 de me faire l'honneur de présider le jury de mon Habilitation à Diriger des Recherches (HDR).

Je remercie chaleureusement les professeurs Jean-Louis Lanet de l'université de Limoges, Françoise Simonot de l'École des Mines de Nancy, et Guy Juanole de l'université Paul Sabatier de Toulouse pour être les rapporteurs de cette HDR, ainsi que le professeur David Simplot-Ryl pour l'avoir dirigée.

Je remercie également les professeurs Claude Kaiser et Pierre Paradinas d'avoir accepté d'être dans le jury. Je ne serai jamais assez reconnaissante à Claude Kaiser qui m'a accueillie dans son équipe « Systèmes temps réel » lors de ma mutation. Travailler avec lui a été un grand enrichissement scientifique; j'ai beaucoup apprécié sa rigueur et son amabilité.

Pierre Paradinas a joué un rôle important dans mes activités de recherche. Notre collaboration a été brève (moins d'une année) mais fructueuse. Il m'a ouvert les portes du monde de *l'embarqué* et plus particulièrement celui de la carte à puce. Il m'a permis de collaborer au projet MESURE qui a été un succès scientifique, ainsi qu'aux enseignements en Master sur la carte à puce. Qu'il trouve ici l'expression de toute ma gratitude. Je dois ma connaissance des plates-formes embarquées *JavaCard* au travail de recherche effectué avec lui dans le cadre de l'encadrement de la thèse de Doctorat de Julien Cordry.

Mes remerciements vont aussi à Eric Gressier-Soudan qui a toujours soutenu mes projets et m'a particulièrement aidée dans les différentes étapes de cette HDR. Je le remercie pour m'avoir fait confiance en acceptant ma collaboration dans ses projets de recherche et d'enseignement. J'espère que notre coopération se poursuivra. Il aurait fait partie du jury de cette HDR s'il n'était pas en congé de maladie. Je lui souhaite un bon rétablissement et un retour en bonne forme.

Je dois mon initiation aux composants au professeur Gérard Florin de par mon implication dans ses cours et en recherche à travers l'encadrement de la thèse de Doctorat de Jean-Paul Etienne. Je l'en remercie vivement.

Je remercie le professeur Stéphane Natkin, responsable de l'équipe MIM1 à laquelle je suis rattachée, qui, bien que n'ayant pas eu l'opportunité de travailler directement avec lui, a toujours accepté de financer mes stagiaires sur ses propres contrats. J'admire sa volonté et son dévouement pour les grands projets autour des jeux vidéo.

[1] *Médias Interactifs et Mobilité*

Durant mon congé de recherche, j'ai travaillé avec Gilles Grimaud, Maître de Conférence à l'université de Lille. Il m'a fait partager son expérience dans le développement autour de l'embarqué, notamment sur les cartes à puce et les réseaux de capteurs et je l'en remercie vivement.

Je remercie la professeure Marie-Christine Costa, directrice du Laboratoire *CEDRIC*, qui a toujours soutenu les membres du laboratoire. Elle a mis ses qualités scientifiques et intellectuelles et toute son énergie dans la défense des intérêts du Laboratoire et de la recherche au CNAM. Des directrices comme elle, la recherche en a besoin.

Je remercie toutes les personnes du Laboratoire CEDRIC ainsi que celles de l'équipe POPS de l'université de Lille, pour leurs encouragements, leur aide ou leur amitié : les membres de l'équipe POPS notamment Isabelle Simplot-Ryl et Anne Rejl, les membres de la spécialité informatique du CNAM, qu'ils soient chercheurs dans le laboratoire CEDRIC ou non. Je pense notamment à Jean-Ferdinand Susini, Pierre Courtieu, David Delaye, Josiane Brudey, Emeraude Kim, Sylvie Verviers, Daniel Enselme, Catherine Coquery, Ivan Boule, Sourour Elloumi, Eric Soutif, Joel Berthelin, Pierre-Henri Cubaud, Jean-Marc Farinone, Hassan Labiah, Viviane Gal, Henri Puglièse, Alexandre Topol, Selma Boumerdassi, Françoise Sailhan, Philippe Augers, Frédéric Lemoine, Jacky Akoka, Isabelle Wattiau, ainsi que toutes les autres personnes des Algécos et du 55 rue Turbigo.

Je remercie également les étudiants qui ont participé à ce projet de HDR: Jean-Paul Etienne, Julien Cordry, Hai Binh Le, Henri Pied, Ernest Tsassong, Nicolas Bouillot, Houas Sadaoui, Skander Essid.

Mes remerciements vont enfin aux personnes qui ont lu et corrigé la version préliminaire de ce document : Claude Kaiser, Eric Gressier-Soudan et mon mari Arezki Bouzefrane ; ainsi qu'à mon frère Mounir Saad pour son assistance technique durant la rédaction de ce document.

Résumé

Ce mémoire de HDR présente mes principaux travaux, dont le dénominateur commun est la recherche de la maîtrise de la ressource *temps* pour, d'une part, ordonnancer efficacement des processus et/ou des requêtes et, d'autre part, caractériser le comportement d'un système observé. Ces deux objectifs sont le fil conducteur des trois chapitres qui constituent ce mémoire.

Dans le premier chapitre, je me suis intéressée aux systèmes qui gèrent des applications manipulant une grande quantité de données fortement soumises à des contraintes de temps, appelés *SGBDs temps réel*. Dans ce cadre, j'ai traité divers problèmes tels que le contrôle de concurrence, la validation, la surcharge, en mettant en exergue, à chaque fois, la prise en compte des contraintes temporelles sans perte de l'intégrité des données.

Le second chapitre traite d'un problème important dans le domaine de *l'embarqué* et du temps réel : la conception d'applications temps réel par assemblage de *composants* en garantissant le respect des contraintes temporelles. Les travaux présentés ici proposent : d'une part, un modèle de composants temps réel qui permet une meilleure compréhension de l'architecture de composants et qui facilite la conception et l'analyse des applications temps réel ; et d'autre part, un système de types étendu avec des automates temporisés qui permet de vérifier la compatibilité et la *substituabilité* des composants de différents points de vue : structurel, comportemental et temporel. Pour prouver la faisabilité du modèle de composants, une implémentation en RTSJ a été réalisée et évaluée.

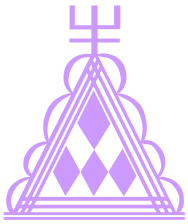
Ces travaux sur les systèmes contraints par le temps m'ont conduite naturellement à m'intéresser aux plates-formes embarquées *JavaCard* comme exemple de systèmes contraints par la ressource mémoire et la puissance de calcul. Dans le dernier chapitre, je présente une méthode de mesure des temps d'exécution et d'élimination du bruit de mesure, en vue de l'évaluation des performances et de la caractérisation de ces plates-formes *JavaCard*. La qualité de l'analyse et la pertinence des résultats obtenus ont permis à ce travail d'être primé dans une conférence industrielle.

Mots Clefs

SGBDs temps réel, contrôle de concurrences, surcharge temporelle, approche composant, systèmes temps réel, automate temporisé, mesure de performances, plate-forme embarquée *JavaCard*.

SOMMAIRE

Introduction générale



Après une thèse sur « l'étude de l'ordonnançabilité des applications temps réel réparties » soutenue à l'université de Poitiers en Janvier 1998, j'ai été recrutée à l'université du Havre où j'avais rejoint l'équipe « *Systèmes de Gestion de Bases de Données Temps Réel (SGBDTR)* ». Pour des raisons liées à la profession de mon mari, j'ai fait une mutation au Conservatoire National des Arts et Métiers (CNAM) à Paris en Septembre 2002 et j'ai rejoint le laboratoire CEDRIC, plus particulièrement l'équipe « *Réseaux Systèmes et Multimédia (RSM)*», devenue « MIM » (*Médias Interactifs et Mobilité*), dirigée à l'époque par Claude Kaiser.

Ce document présente les résultats de mes travaux de recherche entrepris depuis Septembre 1998 à ce jour. Ils résument pour une partie les travaux menés à l'université du Havre en collaboration avec les membres de l'équipe SGBDTR (Laurent Amanton et Bruno Sadeg) ; et pour une seconde partie ceux effectués au laboratoire CEDRIC en collaboration avec l'équipe *MIM* (avec Claude Kaiser, Gérard Florin et Pierre Paradinas).

Ces travaux se déclinent en trois volets matérialisés par les trois chapitres qui composent ce mémoire. Ces chapitres présentent des extraits de travaux qui ont des finalités qui peuvent paraître éloignées, mais ont cependant un dénominateur commun : le temps dans les systèmes contraints à travers divers problèmes et applications. En effet, si le temps a toujours été une ressource précieuse pour le commun des mortels, il l'est davantage dans les systèmes informatiques. Selon les situations, il représente un temps d'exécution, un temps d'accès en mémoire, un temps de communication, autant de variables que l'on essaie de réduire sans cesse, ou bien encore, une échéance, une temporisation, une date de réveil après un délai fixe ou variable, autant de contraintes à respecter, etc.

Globalement nous pouvons classer le temps en deux catégories :

- un temps consommé : temps d'exécution, temps de communication, attente active, etc., c'est-à-dire un temps d'accès, d'utilisation ou d'attente d'une ressource en général,

- un temps à respecter (un quantum, une période, une échéance).

Nous nous sommes intéressés aux systèmes temporellement contraints, c'est à dire des systèmes caractérisés principalement par des échéances à respecter. Parmi ces systèmes, on trouve ceux qui contrôlent des procédés industriels ou des environnements extérieurs, appelés systèmes temps réel, mais aussi ceux qui gèrent des applications classiques manipulant une grande quantité de données

fortement soumises à des contraintes de temps telles que des bases de données à durée de validité limitée appelés SGBDs[2] temps réel.

Outre les systèmes contraints par le temps, il existe des systèmes contraints par des ressources physiques telles que la ressource mémoire, la puissance de calcul ou l'énergie. C'est le cas des systèmes embarqués (*embedded systems*). Dans ce cadre, nous nous sommes intéressés aux plates-formes embarquées *JavaCard* et nous avons cherché à mesurer le temps consommé par chaque opération qui s'exécute sur ce type de plates-formes, pour une meilleure caractérisation de leur efficacité.

Dans ce document, nous nous intéresserons d'une part à la prise en compte des contraintes temporelles dans la conception des SGBDs temps réel et des systèmes temps réel répartis et d'autre part, à la mesure du temps dans les systèmes contraints par la ressource mémoire en prenant l'exemple des plates-formes *JavaCard*.

Le premier chapitre traite de divers problèmes (contrôle de concurrence, validation, contrôle de surcharge) dans les SGBDs temps réel liés au respect des contraintes d'intégrité et de temps. Le deuxième chapitre porte sur la conception de systèmes temps réel répartis à base de *composants* en veillant à respecter les contraintes d'assemblage et de temps. Enfin, le dernier chapitre traite de la mesure du temps dans les plates-formes embarquées *JavaCard*, un exemple de systèmes contraints par la mémoire, en vue de caractériser leur efficacité et de mesurer leurs performances.

Les applications de contrôle de procédés mais aussi les applications les plus courantes aujourd'hui, comme le commerce électronique, manipulent de plus en plus des données ayant une durée de validité limitée, matérialisée par des contraintes temporelles (souvent sous la forme d'échéance) associées aux transactions qui les manipulent. Il devient alors nécessaire de les gérer en utilisant des systèmes adéquats, c'est-à-dire des systèmes capables d'assurer efficacement le traitement des données tout en tenant compte de leurs caractéristiques temporelles. Ces systèmes sont appelés Systèmes de Gestion de Bases de Données (SGBD) temps réel. De tels systèmes ont un double objectif : maintenir la cohérence logique et la cohérence temporelle de la base de données.

Les SGBDs temps réel sont nés de la combinaison des systèmes temps réel et des SGBDs classiques pour, d'une part, utiliser les techniques des SGBDs pour le traitement de grandes quantités de données, et d'autre part, utiliser les techniques des systèmes temps réel notamment les techniques d'ordonnancement pour la prise en compte des contraintes temporelles. Néanmoins pour une meilleure gestion des données temps réel, il s'avère plus judicieux de proposer des protocoles adaptés, garantissant à la fois l'intégrité des données et le respect des contraintes temporelles.

[2] *Systèmes de Gestion de Bases de Données.*

C'est dans ce contexte que nous plaçons la première partie de nos travaux qui s'attachent à résoudre des problèmes tels que le contrôle de concurrence dans un contexte centralisé et dans un contexte réparti ou la validation et le contrôle de surcharge temporelle dans un contexte réparti. En plus de la contrainte causale qui vient se rajouter dans un environnement réparti, nous nous sommes attachés, dans toute cette recherche, à garantir simultanément la cohérence logique des données et les contraintes temporelles des transactions. Ces travaux ont été menés principalement à l'université du Havre et ont été poursuivis au laboratoire CEDRIC en collaboration avec Claude Kaiser, avec un nombre relativement important de publications [1, 2, 4, 5, 17, 19, 20, 21, 22, 23, 24, 25, 26, 27, 31, 34, 37]. L'article [1] est fourni en annexe D. Certaines parties décrites ici sur les SGBDs temps réel sont aussi décrites dans le rapport de HDR de B. Sadeg [S04], étant donné que nous avions travaillé en étroite collaboration.

Ces travaux nous ont permis de proposer de nouveaux concepts tel que le concept d'ε-données mais aussi d'adapter au contexte des SGBDs temps réel des concepts existants par ailleurs tels que le concept de stabilisation [CDKM02] utilisé dans les systèmes temps réel, ou le concept d'ordre causal par phases [AN98, AN99] proposé dans les systèmes répartis. Le but étant de gérer des transactions temps réel tout en veillant au respect de leur échéance et à l'intégrité des données manipulées.

L'autre problème pertinent que nous avons essayé de résoudre est la conception d'applications temps réel par assemblage de *composants* en garantissant le respect des contraintes temporelles. En effet, le fort pouvoir structurant des *composants* permet une gestion efficace de la complexité du système et favorise la réutilisation du logiciel grâce à une séparation claire de leur spécification et de leur implémentation. L'introduction de cette approche dans le développement des systèmes temps réel apporte des bénéfices substantiels. Elle permet notamment de :

- faciliter la conception de systèmes temps réel construits par assemblage de composants préexistants, en réduisant leur complexité,
- accélérer le développement et le déploiement d'applications temps réel ; car, si les spécifications des composants ont été bien définies et l'implémentation conforme à ces spécifications, ils peuvent être réutilisés efficacement dans différentes applications temps réel, du moment que leurs spécifications satisfont les exigences des applications,
- faciliter l'évolution des systèmes temps réel, car la notion de séparation entre spécification et implémentation permet aux composants d'être mis à jour ou remplacés sans aucun redéploiement du système tout entier.

Dans le chapitre 2, nous montrons comment nous avons appliqué les principes de l'approche *composant* dans la conception et la réalisation des systèmes temps réel. Le but principal de ce travail est de proposer une méthodologie de conception permettant la construction de systèmes temps réel par assemblage de composants.

Le travail présenté ici consiste à :

- déterminer quelles sont les caractéristiques d'un composant temps réel. Cela permet dans un premier temps d'établir une spécification bien définie des composants et dans un deuxième temps de déterminer comment vérifier la compatibilité des spécifications des composants lors de la phase d'assemblage,

- établir une relation entre le domaine structurel des composants et le domaine temporel des tâches,

- établir des méthodes visant à vérifier le respect des contraintes temporelles globales en analysant l'ordonnançabilité ainsi que le bon fonctionnement d'un assemblage de composants temps réel.

Ce chapitre reprend globalement les résultats de la thèse[3] de Jean-Paul Etienne que j'ai co-encadrée en collaboration avec Gérard Florin, Professeur au CNAM. Le travail de cette thèse repose sur deux aspects : le temps réel et l'approche composant, qui correspondent à des compétences apportées de manière complémentaire par chaque encadrant. Les résultats de cette recherche ont fait l'objet de plusieurs publications [9, 12, 13, 14, 36]. L'article [9] est fourni en annexe E.

Les travaux présentés dans ce chapitre proposent :

- d'une part, un modèle de composants temps réel qui permet de mieux comprendre l'architecture de composants et de faciliter la conception et l'analyse des applications temps réel

- et d'autre part, un système de types qui sert à vérifier la compatibilité et la substituabilité des composants au niveau syntaxique, mais aussi la compatibilité et la substituabilité via des automates temporisés d'un point de vue comportemental et temporel. Les notions de compatibilité et de substituabilité sont introduites ici pour exprimer respectivement la compatibilité de deux composants durant la phase d'assemblage ou le remplacement d'un composant par un autre durant l'évolution du système. Deux composants sont dits compatibles si les services requis par le composant client peuvent effectivement être réalisés par le composant serveur.

  Les contraintes temporelles globales ne pouvant être vérifiées par le système de types, une analyse d'ordonnançabilité d'une configuration de tâches obtenue après transformation du modèle de composants, est réalisée en utilisant les automates temporisés.
  Par ailleurs, en vue de prouver la faisabilité du modèle de composants temps réel proposé, une traduction du modèle est proposée en RTSJ (Java temps réel).

Si l'objectif des travaux des chapitres 1 et 2 a été le respect des contraintes temporelles, aussi bien dans la conception de protocoles destinés à la gestion de

[3] *Cette thèse a été financée par l'AUF (Agence Universitaire de la Francophonie).*

données temps réel que dans la conception d'applications à base d'assemblage de composants temps réel, le chapitre 3 s'intéresse, quant à lui, à une autre dimension du temps ; sa mesure dans des systèmes contraints par les ressources mémoire comme dans les cartes à puce. L'objectif principal étant de réaliser un ensemble de mesures sur des lots de cartes *JavaCard* appartenant à des domaines applicatifs différents pour caractériser la vivacité de ces plates-formes et mesurer leurs performances.

En effet, le domaine de la carte à microprocesseur ne dispose pas de bancs de mesure de performances ouverts et reconnus. Les grands constructeurs ont développé de tels bancs dans une optique commerciale, qui leur servent principalement à se positionner par rapport à leurs concurrents. Les grands donneurs d'ordre ont également établi des bancs de mesure des performances, souvent simplistes (une séquence de commandes), et très spécifiques à leur domaine d'application. Il n'existe aucun banc de mesure des performances qui soit à la fois accessible à tous et reconnu par les industriels, ce qui rend difficile la prise en compte des performances pour comparer des produits.

Ce chapitre, qui est une synthèse des travaux [7, 8, 10, 11, 35] dont l'article [7] est fourni en annexe F, décrit la méthodologie développée pour effectuer des mesures de performance en se basant sur deux types de programmes :

- des programmes embarqués dans la carte à puce pour exécuter les jeux de test, et
- des programmes qui s'exécutent sur un terminal, permettant de lancer les tests et d'exploiter les résultats obtenus.

Ce travail, réalisé dans le cadre du projet *ANR MESURE* (mai 2006-mars 2008), est la base du travail de thèse de Doctorat de Julien Cordry, que je co-encadre avec Pierre Paradinas, professeur titulaire de la chaire « *Systèmes Embarqués et Mobiles* » au CNAM et directeur du développement technologique à l'INRIA.

Ce projet qui avait pour objectif de mettre à la disposition de la communauté de la carte à puce un outil de mesure des performances des plates-formes *JavaCard* qui soit libre, gratuit et accessible à tous, constitue la première initiative du genre dans le domaine. Ce projet a reçu un écho très favorable de la part des spécialistes de la carte à puce et a reçu le prix Isabelle Attali, décerné par l'INRIA, durant la conférence e-Smart en Septembre 2007. Depuis Janvier 2008, l'outil de mesure est accessible en ligne[4].

Le chapitre 4 conclut ce mémoire en présentant le bilan et les perspectives de mon travail.

Dans ce document, les références bibliographiques présentées sous forme alphabétique sont décrites dans la partie « bibliographie ». Quant à celles présentées sous forme numérique, elles correspondent à des publications personnelles ; elles sont données dans l'annexe B.

[4] *http://mesure.gforge.inria.fr/Fr/Index*

Chapitre 1
Les SGBDs Temps Réel

1.1 Introduction

Les travaux décrits dans ce chapitre sont principalement ceux que j'ai menés à l'université du Havre au sein de l'équipe « SGBDs temps réel ». Ils ont débuté dès mon recrutement dans cette université en Septembre 1998 et se sont poursuivis après ma mutation au CNAM en Septembre 2002 à travers l'étude du problème de surcharge dans les SGBDs temps réel. Dans mes travaux sur les SGBDs temps réel à l'université du Havre, j'avais travaillé avec deux Maîtres de Conférences Laurent Amanton et Bruno Sadeg. Notre préoccupation majeure a été de voir comment prendre en compte les contraintes logique et temporelle des SGBDs temps réel aussi bien dans un contexte centralisé que réparti.

Les applications temps réel aujourd'hui que cela soit dans un contexte strict telles que la gestion des malades, la gestion du moteur d'un avion, ou bien dans un contexte moins strict telles que les applications du commerce électronique, de la bourse en ligne, etc. manipulent une grande quantité de données. Malheureusement, les SGBDs traditionnels ne savent pas gérer de manière efficace les contraintes de temps de ces applications. De même, les systèmes temps réel ne savent pas gérer la cohérence logique des données. Plusieurs problèmes se posent alors, lors de la conception et de la réalisation de ces systèmes. L'objectif de nos travaux est d'appréhender le problème de la gestion des transactions dans les SGBDs temps réel et de proposer des solutions à certains types de problèmes moyennant certaines hypothèses, sans perdre en vue les contraintes temporelles des transactions ainsi que les contraintes d'intégrité des données qu'il faudra respecter.

Dans cette recherche, nous avons essayé de trouver des solutions qui augmentent la concurrence des transactions afin d'augmenter leur chance de s'exécuter dans les temps, tout en garantissant la cohérence des données, que cela soit dans un contexte centralisé ou réparti. L'autre manière abordée ici pour garantir le respect des contraintes temporelles est le maintien dans le système uniquement des transactions temps réel susceptibles de se terminer dans les temps, c'est-à-dire ne garder que celles qui ne génèrent pas de surcharge temporelle.

Par conséquent, ce document commence par traiter les problèmes de contrôle de concurrence dans les SGBDs temps réel. Sachant que les propriétés ACID (Atomicité, Cohérence, Isolation et Durabilité) des SGBDs classiques ne sont pas utilisables telles quelles dans un contexte temps réel, il devient nécessaire de les

revoir [R93]. En effet, il faut ou bien les adapter au contexte temps réel ou bien identifier des propriétés équivalentes pour les SGBD temps réel, à savoir, le respect des échéances des transactions est souvent un critère plus important que celui de la précision des résultats, c'est-à-dire qu'une réponse partielle qui arrive avant l'échéance peut être plus utile pour l'application qu'une réponse précise (ou complète) qui arrive trop tard. Ceci implique que la propriété d'atomicité des transactions peut ne pas être respectée (puisque certaines des actions qui composent la transaction sont annulées si leur échéance est dépassée).

Nous avons alors proposé le concept d'ε-données, proche de *l'epsilon-sérialisabilité*, qui permet de favoriser la ponctualité au détriment de la cohérence des données et qui est applicable aux transactions de lecture qui s'exécutent en concurrence avec des transactions de mise à jour cohérentes. Pour la prise en compte des contraintes temporelles, nous avons défini la notion de Δ-échéance associée à chaque transaction. Selon la valeur de Δ, la transaction peut être plus ou moins stricte. En combinant ces deux notions, nous avons défini une classification des applications temps réel selon qu'elles manipulent des données précises ou non et selon qu'elles tolèrent des dépassements d'échéance ou non. En outre, nous avons proposé un protocole de contrôle de concurrence qui a de bonnes performances en tolérant plus de concurrence grâce à la notion d'ε-données. Ces travaux ont donné lieu à plusieurs publications [4, 31, 32, 34, 37].

Dans un deuxième temps, nous nous sommes intéressés au contexte réparti. La cohérence logique et la cohérence temporelle restent toujours des objectifs à atteindre pour la conception d'un SGBD temps réel réparti. Cependant, les notions de répartition et de communication vont se rajouter en se traduisant par la contrainte causale, une contrainte supplémentaire dont il faudra tenir compte. La prise en compte de ces contraintes a été faite à travers l'étude de la validation et le contrôle de concurrence des transactions temps réel réparties. Plusieurs articles ont été publiés sur ces travaux [2, 22, 23, 24, 25, 26].

En voulant augmenter le taux de transactions pouvant respecter leur échéance, nous avions jusque-là essayé d'augmenter la concurrence entre les transactions moyennant un contrôle qui garantit l'intégrité des données. Pour garantir que les transactions qui s'exécutent pourront respecter leur échéance, l'autre manière de faire est de surveiller les transactions qui sont prêtes à s'exécuter et ne garder que celles dont on est sûr qu'elles pourront s'exécuter dans les temps, les autres, c'est-à-dire celles qui génèreront de la *surcharge temporelle,* seront éliminées. Les travaux sur la surcharge temporelle ont été menés en collaboration avec Claude Kaiser [1, 5, 17, 20, 21]. L'objectif étant d'appliquer le principe de stabilisation de la file des transactions prêtes (à s'exécuter) dans les SGBDs temps réel répartis, que les données réparties soient dupliquées ou non, afin de ne garder dans le système que les transactions qui pourront s'exécuter jusqu'à la fin avec respect de leur échéance.

Ce chapitre commence, dans la section 1.2, par décrire les problèmes liés aux SGBDs temps réel centralisés, auxquels nous nous sommes intéressés en tenant compte des contraintes d'intégrité des données et des contraintes temporelles. La

section 1.3 décrit brièvement les différents problèmes étudiés dans les SGBDs temps réel répartis à travers trois contraintes : logique, causale et temporelle. La section 1.4 conclut ce chapitre.

1.2 Les SGBDs temps réel centralisés

1.2.1 Introduction

Les applications qui s'appuient sur des bases de données et des moniteurs transactionnels contiennent de plus en plus de contraintes de temps qui les rapprochent des systèmes temps réel. C'est le cas des applications temps réel (gestion des malades, gestion du moteur d'un avion, etc.) qui manipulent une grande quantité de données, mais aussi des services fournis en ligne par les banques, les assurances et les entreprises qui font du commerce électronique ou des applications multimédia. Toutes ces applications temps réel utilisent des bases de données et ont en commun la prépondérance du facteur *temps*. En effet, les applications temps réel doivent réagir en tenant compte de l'écoulement du temps. Cette caractéristique fondamentale qui distingue globalement les applications temps réel des autres types d'applications informatiques est exprimée par la définition suivante qui est la plus couramment citée dans la littérature sur le temps réel :

> « *A real-time system is defined as a system whose correctness of the system depends not only on the logical results of computations, but also on the time at which the results are produced* » [SR88].

Malheureusement, les SGBD traditionnels ne permettent pas de répondre de manière efficace aux besoins des applications temps réel [R93, HS01] car ils n'intègrent pas de mécanismes de prise en compte de contraintes temporelles.

Les SGBDs temps réel sont proposés pour gérer une grande quantité de données tout en intégrant les mécanismes de gestion de contraintes de temps. Ces contraintes de temps caractérisent les transactions tout en distinguant différents types comme cela est décrit dans le paragraphe suivant.

1.2.2 Taxonomie des transactions

Selon les conséquences du manquement des échéances des transactions sur l'application et sur son environnement, les transactions dans un SGBD temps réel sont classées en trois catégories [HCL92, PSRS94, R93, DMS99]:

– *Transactions à échéances strictes et critiques* (« *hard deadline transactions* ») : une transaction qui « rate » son échéance peut avoir des conséquences graves sur le système ou sur l'environnement contrôlé. Par exemple [M98], dans un système d'interception de missiles, les transactions qui permettent d'intercepter des missiles adverses sont à échéances strictes et critiques.

– *Transactions à échéances strictes et non critiques* («*firm deadline transactions*») : si une transaction rate son échéance, elle devient inutile pour le système. Ses effets sont

nuls. Elle est donc abandonnée dès qu'elle rate son échéance. En reprenant l'exemple de [DMS99] décrivant un robot qui capte des informations sur des objets défilant sur un convoyeur, si l'opération d'acquisition n'est pas achevée avant la disparition de l'objet du champ de vision du robot, elle est abandonnée et redémarrée pour prendre en compte le prochain objet. On suppose que les objets dont l'acquisition d'informations a été abandonnée sont recyclés plus tard.

– *Transactions à échéances non strictes* (« *soft deadline transactions* ») : si une transaction rate son échéance, le système ne l'abandonne pas immédiatement. En effet, elle peut avoir une certaine utilité pendant un certain temps encore après l'expiration de son échéance, mais la qualité de service qu'elle offre est moindre. Par exemple, dans un système multimédia, on fixe des échéances pour la réception du son et de l'image, afin de garantir une bonne synchronisation de ces deux paramètres au moment de leur présentation à l'utilisateur final. Si le son et l'image arrivent sur deux canaux différents, il peut y avoir des décalages entre la réception des données son et des données image. Quand une image arrive un peu en retard, elle peut toujours être présentée à l'utilisateur si le décalage par rapport au son n'est pas trop important. Ainsi, une réponse à une requête (pour obtenir une image) peut être exploitée au-delà de l'échéance fixée. Il est évident que si le décalage est trop important, le système ne peut plus exploiter l'image reçue, sinon il rend la scène incompréhensible.

1.2.3 La contrainte de cohérence logique

Contrairement à un SGBD traditionnel qui a pour seul objectif de maintenir la cohérence logique de la base de données, un SGBD temps réel doit non seulement satisfaire les règles d'intégrité (cohérence logique), mais aussi la contrainte de cohérence temporelle [SL92]. C'est-à-dire que les transactions agissant sur la base doivent être validées avant l'expiration des échéances qui leur sont fixées. Pour assurer le respect des échéances, le SGBD temps réel doit mettre en œuvre des politiques de contrôle de concurrence et d'ordonnancement des transactions adéquates. Par ailleurs, la nature même des données temps réel (qui ne sont valides que durant des intervalles de temps bien déterminés) fait que certaines propriétés (notamment les propriétés ACID (Atomicité, Cohérence, Isolation, Durabilité) et l'intégrité des données (qui se fonde essentiellement sur l'ordonnancement de transactions sérialisables) doivent être revues pour être applicables dans un contexte temps réel.

1.2.3.1 La sérialisabilité

Un des problèmes posés par la gestion des transactions est le contrôle des accès simultanés aux mêmes données de la base par plusieurs transactions, pour éviter les accès dans des modes incompatibles (écriture/écriture, lecture/écriture). La solution évidente à ces problèmes est de verrouiller les objets accédés jusqu'à ce que la transaction se termine.

Cependant, dans un souci de performance, il est intéressant de laisser s'exécuter le maximum de transactions en concurrence, tout en veillant à ne pas remettre en cause la cohérence de la base. Pour cela, des protocoles dits de

contrôle de concurrence ont été proposés. La plupart de ces protocoles ont comme idée de base de veiller à ce que l'exécution des transactions concurrentes soit sérialisable. La sérialisabilité est le critère généralement accepté pour l'exécution correcte des transactions dans un SGBD traditionnel [EGLT76, KM96, XSRT96, XSSRT96, XR04]. La sérialisabilité de deux transactions peut être définie de la manière suivante [BHG87] : *l'exécution de deux transactions est dite sérialisable si l'exécution entrelacée de leurs opérations fournit les mêmes résultats que leur exécution en série (l'une après l'autre).*

1.2.3.2 Principe de l'epsilon-sérialisabilité

Comme nous l'avons déjà dit, il est difficile de maintenir la cohérence logique de la base de données tout en prenant en compte les contraintes de temps. Pour cette raison, certains auteurs ont tenté de relaxer la propriété de sérialisabilité jugée trop restrictive dans un contexte temps réel.

Dans certains cas, la sérialisabilité stricte n'est pas nécessaire et on peut tolérer l'imprécision des données : des résultats approximatifs obtenus à temps peuvent être plus utiles que des résultats précis obtenus tardivement. C'est le cas par exemple des applications multimédia qui peuvent souvent tolérer la perte ou l'altération d'une certaine quantité de données (quelques pixels d'une image, par exemple). Un autre exemple concerne les jeux vidéo massivement multi-joueurs. Dans un jeu de tennis sur Internet [N03], basé sur une architecture client/serveur, lorsqu'un joueur tape sur la balle, le serveur applique un modèle cinématique pour déterminer la trajectoire de la balle jusqu'à rencontrer un nouvel objet (le sol, la raquette d'un autre joueur) et envoyer à chaque client (joueur) les positions successives de la balle à raison d'une trentaine de fois par seconde. Dans l'approche de prédiction d'état (*dead reckogning*), ce calcul est réalisé par les postes clients, le serveur, lui, ne se préoccupe que de la prochaine collision. Dans ce cas, le serveur envoie la position de collision et la nouvelle vitesse de la balle suffisamment tôt même si ces valeurs ne sont pas précises, pour qu'il n'y ait pas de discontinuité dans la trajectoire de la balle, et que son mouvement semble naturel. Faire marcher en arrière une balle de tennis peut être du plus mauvais effet.

Notre travail a initialement été inspiré par les travaux de Pu et al. [PL92, P93] concernant les ε-transactions, qui sont des transactions dont le critère de correction est l'epsilon-sérialisabilité. Concrètement, une transaction de mise à jour peut exporter une certaine incohérence quand elle met à jour une donnée alors que celle-ci est accédée par des transactions de lecture. Réciproquement, une transaction de lecture peut importer une certaine incohérence quand elle lit une donnée alors que les mises à jour sur cette donnée n'ont pas encore été validées. Par conséquent, les transactions de mise à jour comme les transactions de lecture peuvent manipuler des données incohérentes.

Nous avons alors proposé la notion d'ε-données qui permet de favoriser la ponctualité au détriment de la cohérence des données et qui est applicable uniquement aux transactions de lecture qui s'exécutent en concurrence avec des transactions de mise à jour cohérentes.

Le concept d'ε-données [4] traite de l'imprécision des données, qui peut être décrite comme un pourcentage de la valeur courante des données. Par exemple, soit une donnée d = 20. Si d tolère une imprécision ε alors elle est dite ε-donnée. Soit ε = 10%, alors nous tolérons l'utilisation d'une valeur qui est dans l'intervalle [20 - 2, 20 + 2]. De manière générale, à une donnée d sont associées deux valeurs : *afterValue* et *beforeValue*. Le système mémorise la valeur *beforeValue* pour la durée de vie de la transaction qui fait une mise à jour sur la donnée, de sorte que cette valeur puisse être restaurée si la transaction est annulée. En outre, si la valeur *afterValue* est dans [*beforeValue*-ε, *beforeValue*+ε] alors toutes les transactions de lecture qui s'exécutent avant la validation de la transaction de mise à jour, peuvent utiliser la valeur *beforeValue*. Un plus grand nombre de transactions peuvent alors s'exécuter en parallèle, augmentant par conséquent leur chance de respecter leur échéance. Cette conséquence est très importante lorsque l'on sait que dans les domaines bancaires, boursiers, des assurances, etc. comme nous l'avons cité précédemment, 70% des transactions effectuées sont de type lecture. Il faut pouvoir prendre en compte la fraîcheur d'une donnée (précision temporelle) et l'approximation de sa valeur (précision fonctionnelle) ou une combinaison des deux. Pour les données numériques, la notion d'ε-données permet d'exprimer à la fois la fraîcheur d'une donnée et une estimation de son évolution. On peut calculer *afterValue* par un lissage sur les dernières valeurs observées et se servir d'ε pour décider de l'utiliser ou non. Pour des données alphabétiques ou graphiques, il faudrait définir une notion de distance plus complexe, ce qui sortirait du cadre de notre recherche.

Dans un contexte strict, si une transaction d'écriture détient un verrou sur une donnée d au moment où une transaction de lecture demande un verrou de lecture sur d, cette dernière devra attendre ou être annulée. Dans des applications qui manipulent des ε-données, le niveau d'isolation toléré n'est pas maximal, c'est-à-dire, chaque transaction de lecture peut accéder à des données verrouillées en écriture pourvu que la quantité d'incohérence introduite par la transaction d'écriture reste inférieure à une borne ε. Par conséquent, la gestion des verrous décrite ici est quelque peu différente de celle utilisée dans les SGBDs classiques.

Ce principe de contrôle de concurrence basé sur la notion *d'epsilon-donnée* sera utilisé en combinaison avec d'autres protocoles tels que l'ordonnancement ou la validation des transactions que l'on verra dans la suite.

1.2.4 La contrainte temporelle

Les contraintes temporelles peuvent prendre des formes diverses (périodes, échéances absolues ou relatives, etc.) et s'appliquer à divers composants d'une application, c'est-à-dire à des actions (par exemple, une transaction doit se terminer avant une échéance fixée), à des données (par exemple, la validité temporelle des données est limitée à une durée) ou à des événements (par exemple, tel événement doit apparaître x unités de temps après tel autre événement). Pour plus de détails sur l'expression des contraintes temporelles, le lecteur pourra se référer à [M98, R96, XR04].

Dans les SGBD temps réel, les transactions possèdent des contraintes de temps, souvent sous forme d'échéances qui peuvent provenir de deux sources [R96, DMS99, XR04) :
- des contraintes temporelles liées à l'environnement (par exemple, l'action qui corrige la trajectoire d'un robot téléguidé doit s'exécuter avant que le robot n'entre en collision avec un obstacle),

- des contraintes temporelles liées à des choix de conception et d'implantation d'une application (par exemple, durée d'exécution d'une opération sur un processeur choisi ou délai de communication de messages).

Lorsque les données manipulées ont une durée de validité limitée (par exemple des données lues à partir d'un capteur), il est nécessaire de les lire régulièrement afin de maintenir la fraîcheur des données. Les transactions de lecture de ces données sont dites périodiques car elles sont déclenchées périodiquement.
Les transactions temps réel sont en général caractérisées par une date de réveil, une durée d'exécution, une échéance et une période si elles ont périodiques.

Dans les bases de données temps réel, tous les ordonnancements sérialisables ne sont pas acceptables : ceux qui ne respectent pas les contraintes temporelles des transactions sont rejetés. Par conséquent, les techniques de contrôle de concurrence de transactions développées dans les SGBDs traditionnels ne sont pas directement applicables pour les SGBDs temps réel. D'autre part, les algorithmes conçus pour l'ordonnancement des tâches temps réel [SSNB95, CDKM02] ne peuvent pas être appliqués tels quels aux transactions des bases de données pour intégrer les contraintes temporelles.

Dans la suite nous aborderons le problème de contrôle de concurrence et de validation ainsi que les solutions proposées pour la gestion des transactions temps réel.

1.2.4.1 Le contrôle de concurrence des transactions temps réel

Dans beaucoup d'applications, il est préférable que les résultats des transactions, notamment d'interrogation, soient complets, exacts et obtenus avant une échéance donnée. Cependant, si pour certaines raisons, ceci est impossible, il est toléré que ces transactions puissent dépasser leur échéance d'une quantité de temps borné (au-delà de laquelle la transaction est abandonnée), et/ou de se contenter de résultats partiels/imprécis.

Nous avions alors émis l'idée de considérer des transactions pouvant manipuler des données imprécises (ε-donnée) comme décrit précédemment et pouvant avoir deux types d'échéances: une échéance initiale et une échéance étendue (Δ-échéance).

Un protocole de contrôle de concurrence basé sur cette idée a été proposé [32, 34]. Des simulations ont montré que ce protocole améliore les performances des SGBDs temps réel par rapport aux protocoles temps réel déjà existants. Dans la

suite de cette section, nous décrirons le principe de ce protocole basé sur ces notions d'ε-donnée et de Δ-échéance. Mais d'abord, nous proposerons une classification des applications en nous basant sur ces notions, avant de montrer l'impact de ces notions sur la matrice de compatibilité.

1.2.4.2 Classification des applications temps réel

Les notions d'ε-donnée et de Δ-échéance permettent non seulement de relaxer la sérialisabilité des transactions, mais également de contrôler les échéances des transactions. Ces notions permettent de modéliser, par exemple, les applications qui établissent un compromis entre la ponctualité des données et leur précision, moyennant un contrôle, et les dépassements d'échéances.

Une application potentielle de ces notions est le domaine du multimédia sur lequel beaucoup de travaux ont été effectués [WWFW97, BCE03, WFSP01].

Nous avons établi la classification suivante des applications temps réel selon les valeurs des paramètres :

- *Applications ε-Δ*: ce sont des applications qui peuvent tolérer l'obtention de résultats imprécis/incomplets, qui arriveraient légèrement en retard, par exemple les applications multimédia.

- *Applications ε-0* : ce sont des applications qui peuvent tolérer l'obtention de résultats imprécis/incomplets, mais qui doivent arriver avant échéance (dans les temps), par exemple les applications de prise de décision.

- *Applications 0-Δ* : ce sont des applications dont les résultats doivent être précis/complets, même s'ils sont obtenus légèrement en retard, par exemple les applications bancaires, réservations de places d'avion ou de train, etc.

- *Applications 0-0* : ce sont des applications dont les résultats doivent être non seulement précis et complets mais aussi obtenus dans les temps, par exemple le domaine de l'avionique qui est temps réel strict, critique.

1.2.4.3 Matrice de compatibilité

De nouveaux verrous sont utilisés dans le contrôle de concurrence des ε-Δ-transactions, et la nouvelle matrice de compatibilité des verrous est élaborée. Elle est illustrée par le tableau 1.1:

- les colonnes correspondent aux transactions détentrices de verrous et les lignes aux transactions demandeuses de verrous.
- les cases X correspondent à un cas d'incompatibilité de verrous
- les cases OK correspondent à un cas de compatibilité de verrous
- la case V_1 correspond à une compatibilité conditionnelle : cas où une transaction d'écriture détient un verrou sur une donnée, et une transaction d'interrogation sollicite ce verrou. Cette dernière peut acquérir le verrou à condition que l'imprécision qu'elle importe soit inférieure à celle de la donnée.
- la case V_2 correspond à une compatibilité conditionnelle : cas où une transaction d'interrogation détient un verrou sur une donnée, et une

transaction d'écriture sollicite ce verrou. Cette dernière peut acquérir le verrou à condition que l'imprécision qu'elle engendre en accédant à la donnée soit inférieure à celle de la donnée.

Le tableau 1.1 décrit la matrice de compatibilité entre les transactions d'interrogation, notées Q, et les transactions de mise à jour. Nous considérons qu'une transaction de mise à jour est composée d'une partie lecture (notée R) de la donnée et d'une partie écriture (notée W).

	$Q^{\varepsilon,\Delta}$	$R^{\varepsilon,\Delta}$	$W^{\varepsilon,\Delta}$
$Q^{\varepsilon,\Delta}$	OK	OK	V_1
$R^{\varepsilon,\Delta}$	X	X	X
$W^{\varepsilon,\Delta}$	V_2	X	X

Tableau 1.1 : Matrice de compatibilité

1.2.4.4 Principe du protocole de contrôle de concurrence

La majorité des travaux sur la gestion des transactions porte sur le contrôle de concurrence, et concernent surtout les transactions de type strict non critique (*firm*). Ils s'appuient sur les techniques traditionnelles de contrôle de concurrence des transactions dans un SGBD d'une part, et sur les techniques d'ordonnancement des tâches dans les systèmes temps réel d'autre part. Ces travaux se basent aussi bien sur des politiques de contrôle de concurrence optimistes [HCL92, HSRT91, N93] que sur des politiques pessimistes [MN82, HCL92, HJC93].

Dans les politiques optimistes, le conflit n'est détecté qu'au moment de la validation des transactions où la transaction en conflit avec d'autres transactions est abandonnée et redémarrée. Cette politique est dite optimiste car l'hypothèse considérée est qu'il existe une faible probabilité pour que deux transactions rentrent en concurrence sur un même granule de données. Les politiques pessimistes évitent toute exécution concurrente de transactions tant qu'il existe des conflits potentiels, si bien qu'une transaction en conflit est bloquée (mise en attente) jusqu'à la validation des transactions avec lesquelles elle est en conflit.

Le protocole proposé se base sur un gestionnaire de transactions qui contrôle l'exécution des transactions selon la matrice de compatibilité du tableau 1.1. Contrairement aux gestionnaires de transactions traditionnels, celui-ci permet à deux transactions d'accéder simultanément à une même donnée avec des accès, *a priori*, incompatibles. Le gestionnaire intègre pour cela un mécanisme qui contrôle la quantité d'incohérence introduite par ces accès, ainsi que le dépassement éventuel de l'échéance des transactions. Le contrôleur de concurrence utilise le protocole ε-Δ-2PL-HP, une version étendue du protocole 2PL-HP [AG88].

Son principe est le suivant :

Il suppose l'utilisation d'un algorithme d'ordonnancement du type EDF[5] (*Earliest Deadline First*). Quand un conflit d'accès du type V_1 ou V_2 sur une donnée survient, les transactions en conflit T_i et T_j sont autorisées à poursuivre leur exécution à condition que la quantité d'incohérence qu'elles introduisent dans la base soit bornée par le paramètre ε de la donnée d'une part, et que les échéances des deux transactions ne soient pas dépassées, ou dépassées seulement d'une durée inférieure au minimum des Δ (c-à-d min(Δ_i, Δ_j)) de chaque transaction.

La description du protocole peut être trouvée dans [31]. Nous avons démontré que ce protocole permet à davantage de transactions temps réel de respecter leurs échéances, à condition que l'application concernée autorise des imprécisions bornées de résultats et des dépassements limités d'échéances.

1.2.5 Conclusion

Dans cette première partie, nous nous sommes intéressés aux SGBDs temps réel centralisés comme exemple de systèmes contraints. Nous avons montré que deux contraintes les caractérisent, l'une de type logique liée à l'intégrité des données et l'autre de type temporel liée au respect des échéances des transactions. Comme il est difficile de respecter en même temps la cohérence logique et la cohérence temporelle, notre idée a été de relaxer la sérialisabilité des transactions, en proposant la notion d'ε-donnée qui dérive de la notion d'ε-sérialisabilité, mais également de relaxer et de contrôler les échéances des transactions.

Les notions d'ε-donnée et de Δ-échéance nous ont permis de proposer une classification des applications temps réel qui tient compte, d'une part de la tolérance ou non des applications à manipuler des données imprécises, et d'autre part de la possibilité que des résultats puissent arriver ou ne doivent pas arriver en retard.

Enfin, un protocole de contrôle de concurrence des transactions temps réel a été proposé. Nous avons appliqué ces notions dans le domaine du multimédia et une description détaillée est faite dans [31]. L'application de ces notions à un environnement mobile et sans fil pour pallier les déconnexions fréquentes dans les SGBDTR a également été effectuée. Les détails peuvent être trouvés dans [27, 28, 29, 30].

1.3 Les SGBDs Temps Réel répartis

Les systèmes de bases de données distribués sont inhérents aux applications (ex. marchés électroniques). Ils offrent la possibilité aux transactions de partager des données localisées sur des sites distants. La validation des transactions distribuées temps réel selon des critères temporels nécessitera d'assurer la cohérence globale de la base, ainsi que la cohérence locale sur chaque site.

[5] *L'algorithme EDF suppose que la transaction qui est élue est celle qui a l'échéance la plus proche.*

Le modèle le plus commun pour les transactions distribuées est un système composé d'un site maître (le coordonnateur) à qui la transaction globale est soumise, puis subdivisée en sous-transactions, et de plusieurs sites participants auxquels sont soumises ces sous-transactions.

Une littérature abondante existe sur les résultats de travaux de recherche dédiés aux problèmes des SGBDs temps réel distribués. Ces problèmes concernent le contrôle de concurrence [LSH97; LY98, LPSC99, LLH00, B95, U93, UB96], l'assignement d'échéances [CG96], la réplication [U94], l'exécution des transactions imbriquées [CG96, U98]. Mais les travaux sur la validation (*Commit*) des transactions temps réel distribuées [GHRS96, HR00] sont peu nombreux.

1.3.1 La contrainte logique

L'un des principaux problèmes posés par les architectures distribuées est la validation des transactions au niveau global tout en garantissant la cohérence logique de la base de données distribuée. Le problème étant plus complexe que dans un environnement centralisé car la cohérence logique signifie :

- d'une part, la cohérence de la base de données locale à chaque site telle que nous l'avons définie précédemment dans un contexte centralisé ;

- d'autre part, la cohérence globale de la base de données répartie notamment si des parties de la base sont dupliquées sur des sites différents.

Avant de discuter de la validation des transactions dans notre contexte, rappelons d'abord le principe de la validation classique des transactions distribuées [BHG87]. Une transaction *T* qui est lancée sur un site maître appelé coordonnateur, peut avoir besoin d'accéder à des données localisées sur d'autres sites. Dans ce cas, la transaction *T* envoie une sous-transaction sur chaque site distant concerné. A chaque fin d'exécution d'une sous-transaction, celle-ci doit envoyer une réponse OUI/NON selon que son exécution a pu se dérouler normalement ou non. Un protocole de validation atomique est un algorithme utilisé par le coordonnateur et les sites participants tel que : ou bien le coordonnateur et tous les participants valident la transaction, ou bien tous l'abandonnent. Plus précisément, chaque processus doit effectuer exactement un vote OUI ou un vote NON et atteindre exactement une décision : *Commit* ou *Abort*.

Des conditions pour qu'un protocole de validation soit atomique et d'autres considérations relatives à la validation des transactions distribuées peuvent être trouvées dans [BHG87].

Si on devait utiliser ce protocole de validation classique dans un contexte temps réel, il est clair que si l'exécution d'une sous-transaction quelconque est retardée à cause d'un accès conflictuel à une donnée, la décision finale de validation de la transaction globale sera retardée, pouvant alors provoquer le non respect de son échéance. Le protocole que nous avons proposé permet à des transactions conflictuelles de poursuivre leur exécution en travaillant sur des données empruntées, moyennant un contrôle qui garantit la cohérence des

données à la fin des transactions. Ce comportement accélèrera l'exécution, par conséquent la validation des transactions dans les temps.

Ainsi, dans nos travaux sur le *commit* des transactions distribuées temps réel, nous avons utilisé le modèle classique de SGBD distribué avec un site coordonnateur et des sites participants. Les transactions sont soumises au site coordonnateur, qui les analyse et envoie les sous-transactions aux sites participants appropriés, en fonction de la localisation des données sur lesquelles travaille chaque sous-transaction. Nous considérons que les transactions sont de type strict non critique (*firm*) et qu'une sous-transaction possède la même échéance que sa transaction-mère. Le protocole D-ANTICIP (*Descending-ANTICIP*) que nous proposons dans [23] est basé sur le principe de relaxation de la propriété d'isolation comme dans [GHRS96], mais également sur celui de la duplication des sous-transactions comme le protocole de contrôle de concurrence proposé dans le contexte centralisé par Bestavros et Braoudakis [BB95].

L'idée de base du protocole est que, dès qu'un conflit survient entre une sous-transaction WA qui arrive sur un site avec une autre sous-transaction déjà active WE, alors WA est dupliquée de sorte qu'une copie s'exécute avec l'*image avant* des données et l'autre copie est bloquée en attendant que l'*image après* soit disponible. Ces données après mise à jour par WE sont "prêtées" à la copie de WA même si la sous-transaction WE est dans son état d'incertitude. Chaque sous-transaction qui arrive sur un site suit le précédent schéma, augmentant ainsi le nombre de transactions en exécution sur le site.

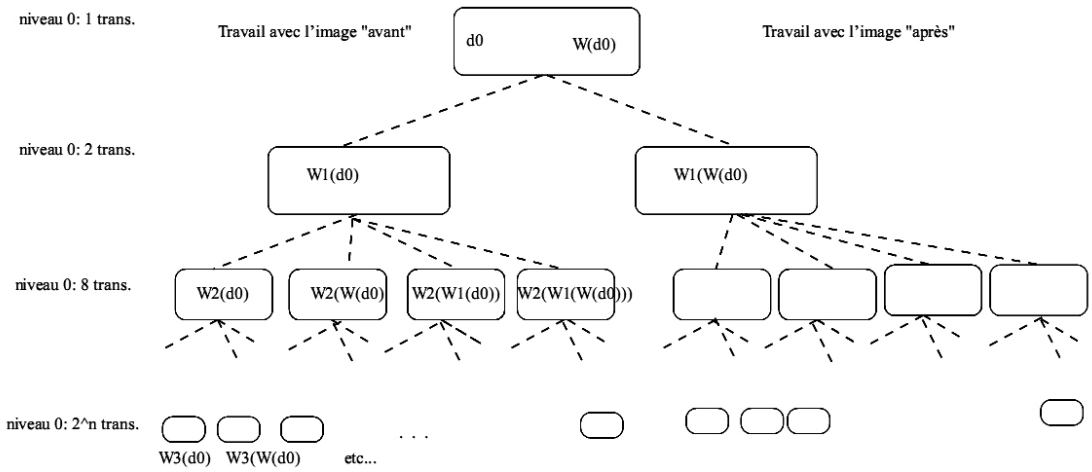


Figure 1.1: Illustration du modèle de copies de sous-transactions

L'exécution des sous-transactions peut être représentée par une arborescence (voir Figure 1.1). La figure 1.2 illustre le cas où le parent d'une sous-transaction a effectué son *Commit*. Il s'agit d'une sous-transaction ayant prêté ses données non encore validées à d'autres sous-transactions actives ou à leurs copies. Dans ce cas, toutes les copies de sous-transactions ayant utilisé les données "avant mise à jour" (l'image avant) sont abandonnées. L'ordre d'abandon est ici descendant.

Toujours dans [23], nous avons effectué des simulations de l'algorithme qui ont montré que les performances obtenues sont meilleures avec le protocole D-ANTICIP par rapport à l'utilisation des protocoles tels que le 2PC.

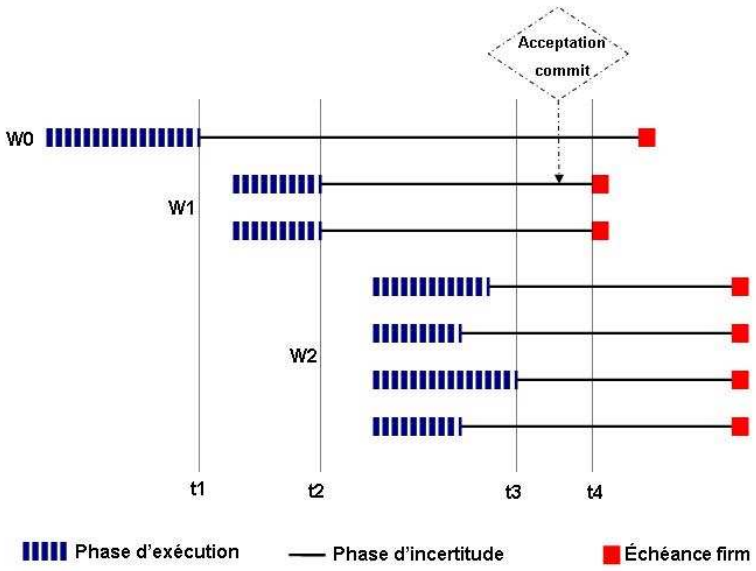


Figure 1.2. Schéma des sous-transactions quand la transaction prêteuse a validé

1.3.2 La contrainte causale

Dans un contexte distribué, la communication est un facteur supplémentaire dont il faut tenir compte. En effet, le problème le plus crucial est l'ordre des communications entre les différentes composantes du système, qui peut être total ou partiel pour satisfaire les contraintes des applications. Un protocole d'ordre causal assure que si deux messages sont causalement reliés et ont la même destination, alors ils sont délivrés à l'application dans le même ordre. Cette propriété est très utile dans le cas de l'exécution de transactions distribuées temps réel ayant, dès leur soumission, des relations de précédence.

Pour construire un ordre, on peut :

- ou bien ordonner les sous-transactions des transactions,
- ou bien couper l'exécution des transactions en plusieurs phases consistantes, donc en plusieurs groupes de transactions ou de sous-transactions. La deuxième voie utilise des protocoles comme la *re-synchronisation*. Basé sur ces notions, un nouvel ordre a été introduit : l'ordre causal par phases [AN99].

Dans les travaux que nous présentons, un protocole de *Commit* des transactions distribuées a été proposé. Les sous-transactions soumises aux sites participants s'exécutent dans des groupes temporels, appelés phases. Ce protocole exploite des résultats obtenus par Amanton et Naimi [AN98, AN99] sur l'ordre causal par phases.

Principe du protocole

Dans un SGBD distribué, une transaction est subdivisée en sous-transactions qui vont s'exécuter sur des serveurs distincts. Dans le travail que nous présentons, nous utilisons un algorithme d'ordre causal [AN96, BMR96, MT96, BSS91] pour sérialiser l'exécution des transactions. La méthode consiste à construire des "coupes" consistantes entre les transactions en conflit.

Le résultat est un protocole, appelé RT-DIS-COM (*Real-Time DIStributed COMmit*), qui permet de gérer les transactions distribuées de type *soft*, subdivisées en sous-transactions (ou actions atomiques) envoyées sur les sites où sont localisées les données auxquelles elles accèdent.

Le principe de RT-DIS-COM est le suivant :

Si dans un SGBD distribué, au sein d'un site, des sous-transactions ne sont pas en conflit, alors elles peuvent être exécutées dans la même phase sur les serveurs. Le site coordonnateur gère ce groupe de sous-transactions de la manière suivante :

– lorsque la transaction courante ne génère pas de conflit avec aucune autre transaction d'un groupe, elle est ajoutée à ce groupe.

– lorsque la transaction courante est susceptible de créer un conflit, le coordonnateur commence une nouvelle phase : toutes les transactions du groupe précédent doivent être terminées avant le début de la transaction courante, et des suivantes. Le groupe est ensuite effacé et réinitialisé avec la nouvelle transaction dans une nouvelle phase.

– lorsqu'une transaction d'un groupe a été validée (*Commit*) ou est abandonnée (*Abort*), elle est retirée du groupe.

Le protocole d'ordre causal par phases garantit que les sous-transactions de deux groupes successifs vont être exécutées sur les serveurs distribués selon leur ordre causal. Ce qui permet d'éviter les conflits et de préserver la cohérence des données.
La description détaillée du protocole ainsi que la démonstration de sa correction sont données dans [25, 26].

1.3.3 La contrainte temporelle

Jusqu'à présent la prise en compte de la contrainte temporelle s'est traduite par l'accélération de l'exécution pour garantir le respect des échéances, notamment en permettant des exécutions simultanées même lorsqu'elles sont conflictuelles moyennant certains contrôles pour garantir simultanément l'intégrité des données.

L'autre manière de tenir compte des contraintes de temps est d'observer les échéances des transactions et de ne garder que celles qui, *a priori*, pourraient s'exécuter dans les temps. Les transactions qui sont éliminées sont celles qui vont générer de la surcharge temporelle.

On parle de *surcharge temporelle* lorsque le temps d'exécution de l'ensemble des transactions dépasse le temps disponible sur le processeur provoquant ainsi le non respect des échéances. Selon [HS01], "l'ordonnancement se focalise sur *quand* exécuter les transactions alors que le gestionnaire de surcharge se charge de trouver *quelle* transaction doit être autorisée à s'exécuter".

Une solution triviale pour résoudre le problème de surcharge est d'augmenter le nombre de ressources dans le système. D'autres politiques proposent de terminer ou de rejeter des transactions durant la surcharge. La terminaison de transactions consiste à annuler des transactions qui ont déjà obtenu le processeur alors que le rejet de transactions consiste à rejeter une transaction dès qu'elle est soumise au système.

De nombreux auteurs ont conçu des algorithmes d'ordonnancement temps réel qui tolèrent la surcharge du système [BLA98, KS95, CDKM02]. Buttazo et *al.* [BSS95] présentent une étude comparative des algorithmes d'ordonnancement où les tâches sont caractérisées non seulement par des échéances mais également par une valeur d'importance. Ils montrent que les meilleurs résultats sont obtenus en ordonnançant par échéance avant la surcharge et par valeur d'importance durant la surcharge.

Toutefois, le contrôle de la surcharge dans les SGBDs temps réel a surtout été étudié dans l'environnement centralisé [HSSA98, BN96, HS01, BSS95, Pang et *al*. 1992, Datta et *al*. [DMKVB96] et très peu de travaux se sont intéressés aux environnements distribués.

Carney et al. dans [Carney et al. 2002] ont construit un SGBD appelé *Aurora*. *Aurora* est une architecture qui gère des applications qui acquièrent des données à partir de sources extérieures (par exemple, des capteurs). *Aurora* intègre un évaluateur de QoS qui gère les performances du système et lance un contrôleur de surcharge lorsqu'il détecte une situation de surcharge qui provoque de faibles performances. Amirijoo et al. 2006 [AHS06] ont proposé une plate-forme de gestion de la qualité de service dans les bases de données temps réel imprécises. Dans leur approche, ils utilisent non seulement l'imprécision des données mais aussi celle des transactions afin d'obtenir une gestion plus efficace de la qualité de service et de la surcharge. L'imprécision des données est basée sur la notion de déviation de la nouvelle valeur à écrire avec la valeur courante de la donnée. Cette notion est similaire à notre concept d'ε-donnée. Par contre, l'imprécision des transactions suppose que toute transaction comprend une partie obligatoire qu'il faut exécuter absolument et dans les délais, et une partie optionnelle qui n'est exécutée que si les ressources nécessaires sont disponibles (telles que processeur, temps).

Dans cette recherche, nous nous sommes attachés à concevoir des protocoles qui bornent les incohérences logiques des données et qui contrôlent les situations de surcharge dans un contexte temps réel distribué. Outre la notion d'ε-donnée décrite précédemment et qui a été utilisée par ces protocoles, ces derniers se

basent sur la notion d'importance associée aux transactions pour réaliser de la stabilisation, c'est-à-dire contrôler la surcharge temporelle des transactions. Avant de parler de ces différentes notions, nous commençons par décrire l'architecture utilisée.

1.3.3.1 L'architecture utilisée

Dans notre modèle, nous considérons uniquement les transactions temps réel strictes non critiques (type « *firm* »). Une transaction est soumise au site maître où est exécuté le processus coordonnateur. Des parties distinctes de la base de données sont réparties sur différents sites. La transaction globale est décomposée en sous-transactions qui sont envoyées aux sites participants, où elles sont exécutées par un processus mandataire appelé *cohorte*. Le coordonnateur sait à quelles cohortes appartiennent les sous-transactions d'une transaction globale.

Chaque site participant comporte trois modules (voir Figure 1.3):

- un module d'ordonnancement qui utilise l'algorithme EDF (*Earliest Deadline First*) [LL73], choisi parce que dans les systèmes temps réel une échéance exprime bien l'urgence d'une tâche, que celle-ci soit isolée (apériodique) ou qu'elle fasse partie d'un train de tâches périodiques (dans ce dernier cas, on pourrait traduire l'urgence par un ordonnancement à priorités fixes), et aussi parce que EDF est optimal pour toute configuration de tâches ordonnançables ;
- un module de gestion de surcharge qui contrôle les surcharges à chaque fois qu'une nouvelle sous-transaction est soumise au site participant et
- un module de gestion de données qui applique le concept d'ε-données lorsque des sous-transactions rentrent en conflit pour l'accès aux données du site participant.

Deux types de sous-transactions sont considérés : les sous-transactions de lecture et les sous-transactions de mise à jour. En plus des situations de panne qui ne sont pas traitées ici, notre modèle suppose que les bases de données ne sont pas dupliquées. Un gestionnaire de surcharge qui utilise des données dupliquées est proposé dans [20].

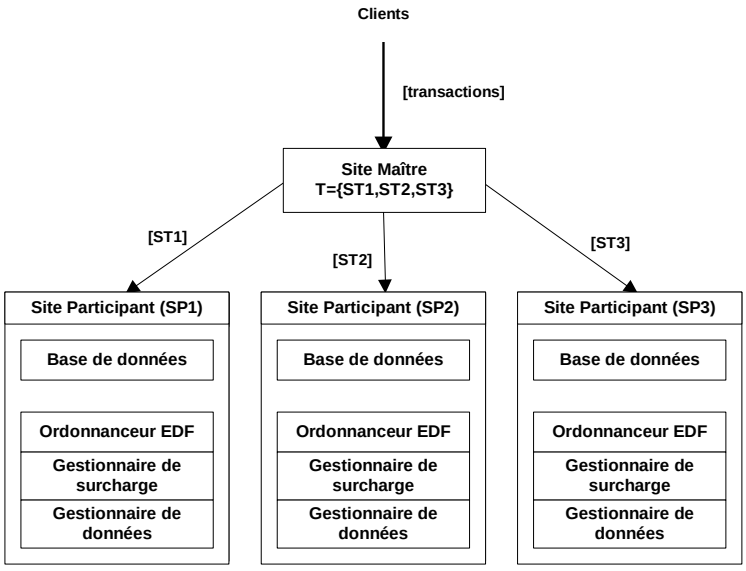


Figure 1.3. *L'architecture du SGBD utilisé*

1.3.3.2 Contrôleur de surcharge temporelle

La soumission d'une nouvelle sous-transaction peut générer une surcharge et par conséquent une faute temporelle, si le temps nécessaire à l'exécution des sous-transactions (y compris celle qui est soumise) excède la capacité de calcul du processeur du site. Nous savons que la politique d'ordonnancement EDF garantit le respect des échéances lorsque les configurations sont ordonnançables. Cependant en cas de surcharge, la politique EDF devient particulièrement mauvaise : d'une part, on constate une cascade de dépassements d'échéance ; d'autre part, les suppressions se font dans le désordre. Notre politique vise à ce que les transactions les plus importantes de l'application subissent moins de fautes temporelles et moins d'annulations dans des situations de surcharge que d'autres transactions. Les transactions sont classées selon un paramètre appelé *valeur d'importance* [20], l'importance étant l'inverse de l'ordre de suppression. Ainsi, d'une part, on favorise les transactions jugées indispensables pour l'application ; et d'autre part, on constate souvent que cette politique augmente le taux de réussite grâce à l'anticipation des suppressions, qui évite que se poursuivent les calculs des transactions qui ne pourront respecter leur échéance.

1.3.3.3 L'importance d'une transaction

Chaque transaction est caractérisée par une échéance qui définit son urgence et par une valeur d'importance qui définit la criticité de son exécution par rapport aux autres transactions de l'application temps réel. La valeur d'importance peut être attribuée par l'utilisateur ou le concepteur de l'application. Plus la valeur est grande pour une transaction, plus cette dernière est indispensable pour l'application. L'importance d'une transaction n'est pas liée à son échéance; ainsi, deux transactions différentes qui ont la même échéance peuvent avoir différentes valeurs d'importance. Les valeurs d´importance peuvent être attribuées par classes de transactions ; chaque classe représentant un domaine applicatif précis.

Considérons quelques exemples d'applications [B05]. Dans le domaine bancaire, les autorisations cartes bancaires (essentiellement des lectures avec un temps de réponse inférieur à 300 ms) sont plus importantes que les opérations *GAB*[6] (avec un temps de réponse également inférieur à 300 ms et 70% de lecture) qui, elles-mêmes, sont plus importantes que les opérations à distance (avec un temps de réponse compris entre 300 et 400 ms et 80% de lecture).

Dans le domaine boursier, les ordres de bourse (avec un temps de réponse inférieur à 300 ms et 70% de lecture) sont plus importants que les opérations effectuées sur un portail Web titre (essentiellement des lectures avec un temps de réponse entre 300 et 400 ms).

Dans le domaine des assurances, les applications caisse (telles que profil contrat, tarification, souscription de contrat, etc.) avec 70% de lecture et un temps de réponse compris entre 400 et 500 ms, sont plus importantes que les applications

[6] *Guichet Automatique Bancaire*

de type back-office (du siège) qui gèrent les contrats (validation, annulation, édition de cartes vertes, etc.) et qui effectuent 70% de mises à jour.

Dans notre modèle, une transaction globale est caractérisée par une date d'arrivée, une échéance et une valeur d'importance. De la même manière, sur un site, une sous-transaction est caractérisée par une date d'arrivée, une durée d'exécution, une échéance et une valeur d'importance. Notons que l'échéance et la valeur d'importance d'une sous-transaction sont héritées de la transaction globale à laquelle elle appartient. De plus, l'échéance d'une transaction doit être supérieure à 2×δ avec δ le temps de communication, afin de garantir que cette échéance n'expire pas avant le lancement de ses sous-transactions au niveau des sites participants et avant l'obtention d'une réponse relative à leur validation ou leur annulation.

1.3.3.4 La stabilisation

La stabilisation vise à contrôler la surcharge du système et consiste à maintenir dans la file des transactions prêtes uniquement les sous-transactions qui respecteront leur échéance. Elle favorise les sous-transactions qui ont des valeurs d'importance élevées lorsqu'une surcharge est détectée sur un site. Pour cela, des sous-transactions plus urgentes, mais de plus basse valeur d'importance qu'une transaction touchée par la surcharge sont supprimées de la file des transactions prêtes et sont annulées jusqu'à ce que la laxité du processeur retrouve une valeur positive et que la surcharge disparaisse pour cette transaction privilégiée. La laxité du processeur, à l'instant *t*, est le temps maximum durant lequel le processeur peut rester oisif, après *t*, sans qu'une transaction ne rate son échéance. Dans le cas particulier où des sous-transactions ont la même valeur d'importance, ce sont les moins urgentes qui sont supprimées en premier.

Une sous-transaction *ST* sur un site participant *s* est définie à l'aide de quatre paramètres temporels: *r* la date de soumission de *ST* dans *s*, *C* sa durée d'exécution, *Dead* l'échéance absolue héritée de la transaction globale de *ST* et *Imp* sa valeur d'importance. Nous notons que $Dead_{ST} - r_{ST}$ est le délai critique de *ST*. $C_{ST}(t)$ représente le temps de calcul restant à l'instant *t* pour terminer l'exécution de *ST*.

Notons deux articles intéressants [XR04, RSD04] qui montrent comment dériver les échéances et les périodes pour les transactions temps réel, à partir des durées de validité des données manipulées.

Soit $readyQueue_s$ la file d'attente des sous-transactions prêtes d'un site participant *s*, triée par ordre croissant des échéances à l'instant *t*:

$$readyQueue_{s,t} = \{ST_0, ST_1, ..., ST_n\} \text{ où } C_{STi}(t) > 0 \ (\forall\ i \in \{1, n\})$$

Nous définissons la laxité du processeur *LP* pour un site *s* à l'instant *t* comme un paramètre qui dépend de la laxité conditionnelle *LC* de chaque sous-transaction de $readyQueue_{s,t}$. Par conséquent :

$$LP(t) = \text{laxité}(readyQueue_{s,t}) = \text{Min}\ \{LC_{STi}(t)\}\ (\forall\ ST_i \in readyQueue_{s,t})$$

$$\text{où } LC_{STi}(t) = Dead_{STi} - t - \sum C_{STj}(t).$$

La somme sur j (j = 0, i) calcule le temps d'exécution restant de toutes les sous-transactions (y compris ST_i) qui sont déclenchées à l'instant t et qui précèdent ST_i dans la file.

Une situation de surcharge est détectée dès que la laxité du site $LP(t)$ est négative. Les sous-transactions en retard sont celles dont la laxité conditionnelle est négative. La valeur de la surcharge est égale à la valeur absolue de la laxité du processeur, $|LP(t)|$. Par conséquent:

- $LP(t) > 0 \Rightarrow Dead_{STi} > t + \sum C_{STj}(t)$ ($\forall$ ST_i) : ceci signifie que tout en permettant l'exécution de (i -1) sous-transactions qui précèdent ST_i dans la file d'attente, ST_i terminera son exécution avant l'expiration de l'échéance.

- $LP(t) < 0 \Rightarrow (\exists\ ST_k)\ Dead_{STk} < t + \sum C_{STj}(t)$: ceci signifie que nous avons, au moins, une sous-transaction prête ST_k pour laquelle l'échéance expire avant que ST_k ne termine son exécution. Par exemple, dans la figure 1.4, ST_3 génèrera une surcharge à l'instant 8.

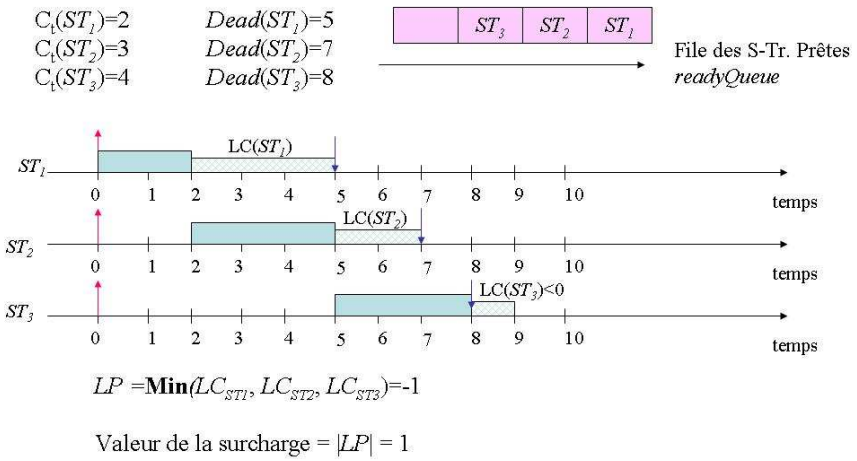


Figure 1.4. *La laxité du Processeur (LP)*

Afin d'avoir dans la file des transactions prêtes uniquement des sous-transactions qui respectent leur échéance, nous devons stabiliser la file d'attente et déduire les sous-transactions à annuler chaque fois qu'une surcharge est détectée. Par conséquent, nous exprimons la notion de stabilisation comme suit:

Nous définissons $readyQueue_{s\,t}{}^{stable}$ comme une stabilisation de $readyQueue_{s,\,t}$ si:

1. $readyQueue_{s,t}{}^{stable} \subseteq readyQueue_{s,t}$,

2. laxité($readyQueue_{s,t}{}^{stable}$) ≥ 0 et

3. Si $\Re$ = $readyQueue_{s,t}$ - $readyQueue_{s,t}{}^{stable}$ alors ($\forall$ $ST \in \Re$) ($\forall$ $\kappa \subseteq readyQueue_{s,t}{}^{stable}$) nous avons: ($\forall$ $ST' \in \kappa$, $Imp_{ST'} < Imp_{ST}$) $\Rightarrow$ laxité(($readyQueue_{s,t}{}^{stable}$ - κ) $\cup$ {ST}) < 0.

Lorsqu'une surcharge est détectée sur un site participant, la routine de stabilisation retire de $readyQueue_{s,t,}$ une par une, les sous-transactions les moins importantes parmi les sous-transactions plus urgentes que la sous-transaction

touchée par la surcharge, et cela jusqu'à ce que la laxité soit positive à nouveau. Ceci est exprimé dans le point 2. Pour expliquer le point 3, considérons le cas de trois sous-transactions ST_1, ST_2 et ST_3 triées selon une importance croissante (ST_3 plus importante que ST_2 et ST_2 plus importante que ST_1). Si l'on détecte une surcharge et si la suppression de ST_1 ne suffit pas pour résorber la surcharge mais qu'au contraire la suppression seule de ST_2 suffit en rendant inutile la suppression de ST_1, alors ST_1 est gardée dans le système.

La stabilisation a été utilisée pour gérer la surcharge temporelle des transactions réparties qui accèdent à une base de données dupliquée [20]. La stabilisation a été combinée avec la notion d'ε-donnée pour donner naissance à un autre protocole [1] de gestion de surcharge temporelle dans un SGBD temps réel réparti. Une plate-forme de simulation a été développée et des expérimentations ont été menées (voir Figure 1.5). Nos expérimentations montrent qu'on peut régler un moniteur transactionnel pour favoriser le respect de la ponctualité demandée pour certaines lectures, y compris en situation de surcharge, à condition d'accepter quelquefois un degré d'incertitude (dont la tolérance maximale est décidée par l'application) sur le résultat fourni à la date demandée.

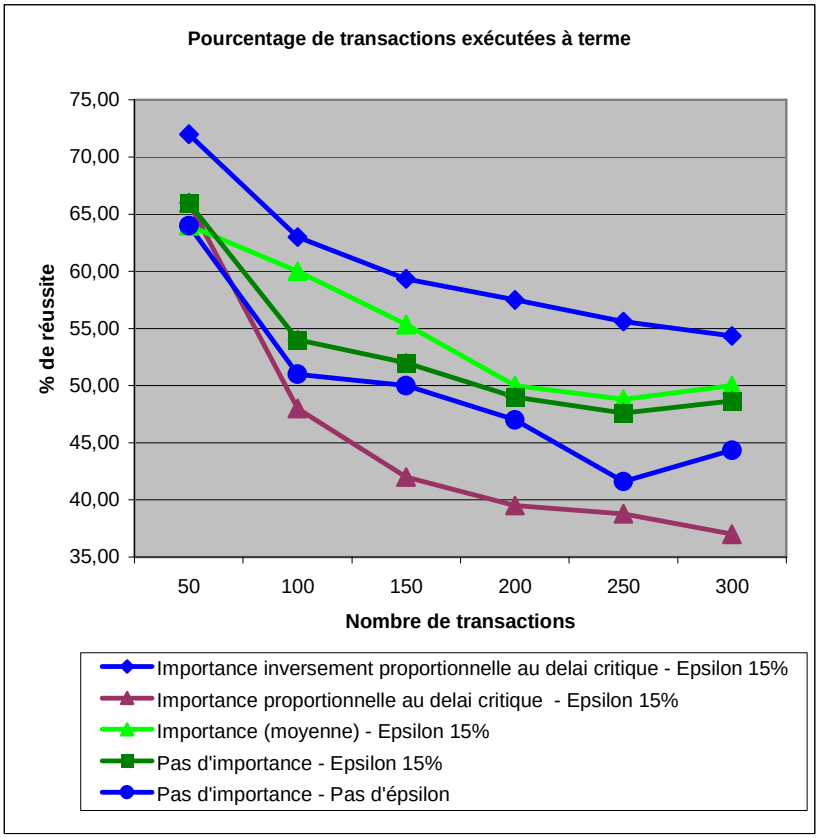


Figure 1.5. *Comparaison de 4 politiques simulées*

1.4 Conclusion

Dans ce chapitre, nous avons passé en revue brièvement les problèmes que nous avons tenté de résoudre en veillant au respect des contraintes temporelles.

Pour augmenter le nombre de transactions pouvant s'exécuter avec respect de leur échéance, trois approches différentes ont été proposées :

- une solution basée sur le concept d'ε-données est proposée pour tolérer plus de concurrence entre les transactions conflictuelles en permettant aux transactions de lecture de poursuivre leur exécution en parallèle avec des transactions d'écriture partageant les mêmes données, moyennant une imprécision bornée par ε. Ce concept a été appliqué aussi bien dans le contrôle de concurrence des transactions centralisées que dans celui des transactions réparties.

- une solution basée sur la duplication des transactions conflictuelles de manière à avoir une transaction T traitant des données avant mise à jour et une autre transaction T' bloquée en attente de disponibilité des données fraîches, c'est-à-dire après mise à jour (T étant la copie de T' avec T et T' en conflit avec une transaction S). La transaction qui valide en premier annule sa copie.

- une troisième solution pour éviter les conflits a pour principe d'exécuter les transactions dans des phases différentes selon leur ordre causal.

En augmentant le degré de concurrence moyennant un contrôle qui garantit l'intégrité des données, toutes ces solutions ont eu pour objectif d'augmenter les chances de respect des contraintes temporelles.

L'autre manière de garantir cet objectif a été de ne garder dans le système que les transactions qui peuvent s'exécuter dans les temps. Toutes les transactions qui risquent de générer de la surcharge temporelle sont détectées et éliminées selon un protocole qui a été proposé.

A travers les problèmes posés ici et les solutions proposées, nous avons voulu montrer que les SGBDs temps réel peuvent être une solution pour la gestion d'applications temps réel manipulant une grande quantité de données. Ainsi, les travaux présentés ici pourraient être intégrés dans les systèmes transactionnels sous forme de « *plug-in* ».

Nous ne prétendons pas avoir résolu tous les problèmes liés aux SGBDs temps réel. En effet, même si certains problèmes ont beaucoup été étudiés (comme la gestion des transactions, la gestion de la mémoire, les entrées/sorties, le recouvrement, etc.), d'autres problèmes restent encore ouverts, tels que : la tolérance aux fautes des transactions distribuées temps réel ou bien les méthodes de conception de bases de données temps réel. Ceci est dû en partie au fait que la communauté des SGBDs temps réel est relativement restreinte. En France, l'équipe des SGBDs temps réel du Havre est principalement la seule équipe qui s'intéresse à cette problématique. En effet, la problématique du temps réel n'intéresse pas forcément la communauté des bases de données ; de même dans la communauté du temps réel, très peu de chercheurs possèdent la culture des bases de données.

Au vu des nombreux travaux de recherche sur les SGBD temps réel, nous pouvons dire que ces systèmes ont maintenant atteint leur stade de maturité. Nous pensons qu'une des manières d'exploiter ces résultats de recherche est de revisiter ces problèmes dans le contexte des SGBDs embarqués. En effet, des SGBDs embarqués comme TinyDB[7] (SGBD pour réseaux de capteurs) ou PicoDB (SGBD pour carte à puce) [BBPV2000] pourraient profiter des recherches menées dans le domaine des SGBDs Temps Réel pour une meilleure maîtrise du développement et du déploiement de ce type de systèmes.

[7] *http://www.tinyos.net*

Chapitre 2

Les systèmes temps réel répartis

2.1 Introduction

Comme tous les domaines de l'informatique, le domaine du temps réel n'est pas épargné par le besoin de développer de plus en plus vite des systèmes de plus en plus complexes. Traditionnellement, l'utilisation de langages de bas niveau était de rigueur dans le développement de tels systèmes afin de garantir un contrôle total de leur comportement.

Cependant, au fil des années, cette complexité accrue, résultant de l'évolution des besoins et l'arrivée de nouvelles technologies, a rendu le développement de ces systèmes beaucoup plus difficile. Cette difficulté à gérer efficacement la complexité freine l'évolution des systèmes et introduit un nombre important d'erreurs dans leur conception et leur réalisation, entraînant ainsi un coût de production beaucoup plus conséquent et un temps de mise en œuvre beaucoup plus long.

Vint alors le besoin d'introduire de nouvelles méthodologies qui faciliteraient la conception et la gestion des systèmes temps réel. Il s'agit surtout de capitaliser le logiciel afin d'améliorer sa productivité. Cependant, une capitalisation efficace du logiciel requiert une méthodologie reposant notamment sur « *l'augmentation du niveau d'abstraction* » afin d'alléger la complexité logicielle et sur « *la réutilisation du logiciel* » pour faciliter et accélérer le développement [DF05] et, aspect plus important dans le domaine du temps réel, pour faciliter la certification des logiciels.

Ainsi la notion première mise en avant dans les nouveaux développements est la notion d'architecture : elle est de plus en plus essentielle puisqu'il s'agit de développer des applications à base de briques élémentaires. Aussi, l'optique d'une réutilisation logicielle demande une caractérisation claire de ce qu'est un « *élément réutilisable* » et aussi des moyens pour vérifier que cet élément satisfait les exigences du système. C'est dans ce contexte que nous plaçons les travaux présentés dans ce chapitre, qui ont pour objectif de proposer une méthodologie pour la conception d'applications temps réel à base de composants. Cette méthodologie va permettre:

- de définir toutes les étapes du cycle de développement, de l'expression des besoins jusqu'au déploiement du système temps réel sur une plate-forme d'exécution, et

- d'apporter un cadre formel afin de permettre une conception rigoureuse et sûre des systèmes temps réel.

Dans le domaine de l'embarqué et du temps réel, différentes approches orientées composant existent. On peut citer : KOALA [OLKM00], COMPARE [IST05], RUBUS [NI02], AUTOCOMP [SF04], PECOS [MSZ01], CLARA [D98], METAH [MetaH98], RTCOM [TNHN04], AADL [SAE04], BIP [BBS06], VEST [S01], PECT [Wa03], ROBOCOP [R03], SAVECCM [HACM04], AUTOSAR [FB et al. 06], TINAP [L08]. Une bonne synthèse de ces approches est aussi présentée dans [D05].

A travers la notion d'activité, CLARA, SAVECCM et AADL, par exemple, modélisent l'architecture à l'aide d'assemblages de tâches concurrentes. Or, à notre connaissance, les modules fonctionnels doivent être les entités de base à utiliser pour la compatibilité et la réutilisabilité. Par ailleurs, en dehors de COMPARE et TINAP, les architectures de haut niveau citées ici produisent à l'exécution du code monolithique. C'est ce que nous avons essayé d'éviter à travers notre plate-forme RTSJ qui est elle-même implantée sous forme d'assemblage de composants. Contrairement à la majorité de ces approches, nous fournissons en plus des outils formels pour vérifier l'assemblage des composants temps réel de différents points de vue, ceci afin d'éviter leur intégration dans le code métier des composants.

La suite de ce chapitre est organisée comme suit. Nous commençons par proposer à la section 2.2 un modèle de composants, en définissant différentes entités logicielles utiles dans un contexte temps réel. Ensuite, afin de vérifier l'assemblage des composants de l'application, un système de types étendu à l'aide d'automates temporisés a été proposé en section 2.4 : il permet de vérifier la compatibilité et la substituabilité des composants, de différents points de vue : syntaxique, comportemental et temporel.
Sachant que cet assemblage, réalisé sur chaque paire de composants, ne permet pas de vérifier les contraintes temporelles globales, telle que l'échéance, une analyse d'ordonnançabilité de la configuration de composants est faite en utilisant les automates temporisés en section 2.5.

Enfin, pour vérifier la faisabilité du modèle de composants proposé, celui-ci a été traduit en RTSJ (spécification pour Java temps réel) et implanté en *jRate*[8] comme décrit à la section 2.6.

2.2 L'architecture à composants

Pour modéliser un système temps réel à l'aide de composants, il est important de mettre en avant les abstractions jugées essentielles dans ce type de conception

[8] *http://jrate.sourceforge.net/*

afin de faciliter la compréhension et l'analyse du système. Ainsi, toute architecture logicielle temps réel, doit pouvoir offrir :

- une description explicite des différents types d'entités logicielles (actives, passives, mécanismes de communication) que nous rencontrons habituellement dans les systèmes temps réel,
- une définition explicite des besoins temporels de chaque entité ou de l'ensemble de l'architecture, ainsi que
- des moyens permettant d'analyser l'ensemble de l'architecture tant au niveau fonctionnel que temporel.

Nous nous plaçons à un niveau conceptuel élevé, à la ADL (« langage de définition d'architecture ») [MT00]. En effet, représenter un composant comme une entité abstraite nous permet d'avoir une meilleure compréhension de l'architecture et facilite la conception et l'analyse des applications.

Dans le modèle que nous proposons, un programme est défini comme un ensemble de définitions d'interfaces et de composants ainsi qu'une configuration, spécifiant les instanciations et les liaisons des entités logicielles. Pour expliquer les diverses abstractions relatives aux systèmes temps réel, nous définissons cinq types de composants : le composant *actif* (*ActiveComponent*), le composant *événementiel* (*EventComponent*), le composant *passif* (*PassiveComponent*), le composant *protégé* (*ProtectedComponent*) et le *composite*. En dehors du composant protégé, les autres types de composants sont définis comme dans PECOS [MSZ01]. En général, un composant est un modèle type qui peut être utilisé pour obtenir plusieurs instances de composants, où l'instance du composant constitue l'entité qui fournit les services réels. La spécification de ces services est faite via des interfaces. Ainsi, tout service offert ou requis par un composant est spécifié à l'aide d'un type d'interface, caractérisé par un nom d'interface et par un numéro de port associé.

2.2.1 Le composant actif

Le composant actif (voir Figure 2.1) est une entité logicielle ayant son propre *thread* de contrôle. Il représente généralement des tâches temps réel et inclut des propriétés temporelles telles que la période, l'échéance et la priorité. Il comporte également une interface de contrôle, utilisée par exemple pour lancer l'exécution du composant s'il dépend des événements externes ou d'autres composants actifs.

```
ActiveComponent   HLWhandler (time deadline, time period, time offset, int priority)
{
      provides   IControl  itc;
      requires    IHLW2   ih;
      requires    IMotor   m;
}
```

Figure 2.1 : *Un composant actif*

2.2.2 Le composant événementiel

Le composant événementiel est un composant actif qui représente une interruption ou une routine de traitement (*handler*) d'événement. Il possède par conséquent une interface de contrôle dont le déclenchement dépend des interruptions matérielles (capteur, horloge, réseau, etc.) ou logicielles.

Dans ce modèle, les composants actifs tout comme les composants événementiels ne fournissent pas directement des services à d'autres composants. Au lieu de cela, ces services peuvent être fournis par l'intermédiaire d'une entité externe via un composant protégé.

En effet, étant une tâche, un composant actif aura son propre *thread* de contrôle. Par conséquent, si un composant actif est autorisé à fournir des services, ces derniers doivent être accessibles en exclusion mutuelle. Ce choix de spécification peut sembler peu commun ; cependant il nous facilitera l'analyse temporelle du système.

2.2.3 Le composant passif

Le composant passif, par opposition au composant actif ou événementiel, n'a pas son propre *thread* de contrôle. Il représente généralement des librairies ou des API de haut niveau utilisées pour accéder au matériel. Lors de son appel, son exécution est effectuée dans le contexte du composant actif/événementiel demandant ses services.

2.2.4 Le composant protégé

Le composant protégé (voir Figure 2.2) est un composant passif équipé de mécanismes d'exclusion mutuelle, avec un protocole à priorité plafond [SLR90] qui garantit un accès borné aux ressources critiques. Un composant protégé peut être vu comme un moniteur Java ou un objet protégé du langage Ada, permettant un accès exclusif à une ressource critique. Le protocole à priorité plafond est utilisé pour éviter le problème de l'inversion de priorités [SLR90], problème connu dans le domaine du temps réel mais aussi celui de l'interblocage.

```
ProtectedComponent  Motor (int ceiling) {
        provides  IMotor  m;
        requires   ICH4    ic;
}
 ProtectedComponent  Motot2 (int ceiling) {
         provides  IMotor  m;
 }

ProtectedComponent  Motor3 (int ceiling) {
        provides  IMotor  m;
        requires   ICH41  ic;
        requires   ILog   log;
 }
```

Figure 2.2. *Un composant protégé*

2.2.5 Le composite

Dans notre modèle, la conception hiérarchique est également présente à travers le concept de composite. Le composite est une entité logicielle qui facilite la réutilisation d'une composition de composants de base ou d'autres composites. Il comporte généralement une liste d'instances de composants et de *bindings* liant les interfaces requises de certains composants aux interfaces offertes d'autres composants. L'instanciation d'un composant se fait en utilisant la clause *new*. Comme dans la figure 2.3, l'instanciation est utilisée seulement avec les primitives d'affectation. Les *bindings* des interfaces de composants sont réalisés à l'aide de la clause de forme (*x.p, y.q*) avec *x.p* le service requis et *y.q* le service offert. Au sein d'un composite, le service *forwarding* est réalisé à l'aide de la clause *forwards* pour exporter des services requis/offerts des sous-composants vers l'extérieur via les interfaces du composite.

Généralement le port requis d'un sous-composant est toujours lié exactement à un seul port offert d'un autre sous-composant ou bien à un port requis d'un composite. C'est une restriction justifiée car, sinon, un composant client ne saura pas à quel port lier l'appel de service. Par contre, un port offert d'un composant peut être lié à un ou plusieurs ports requis.

Comme on peut le voir, selon notre définition, le composite est seulement une entité structurante, qui permet la réutilisation à gros grains, sans toutefois offrir ou demander plus de fonctionnalités que celles offertes ou requises par les composants qu'elle contient. Par conséquent, les interfaces visibles qu'elle comporte représentent uniquement des délégations à celles de ses sous-composants.

```
Composite   HLWSensor {

      requires   IMotor   m ;

      HWSensor   hws :=  new  HWSensor(0s, 5s, 0s, 8) ;
      LWSensor    lws := new  LWSensor (0s, 5s, 2500ms, 8);
      HLWStatus   hlws  :=  HLWStatus (9);
      HLWHandler  hlwh  := new HLWHandler(1s, 0s, 0s, 1);

      bind(hws.ic1, hlwh.itc);
      bind(hws.ih1, hlwh.ih);
      bind(lws.ic1, hlwh.itc);
      bind(lws.ih2, hlws.ih);
      bind(hlwh.ih, hlws.ih);
      forwards(hlwh.m, m);
}
```

Figure 2.3. *Un exemple de composite*

2.3 Spécification des services en utilisant des automates temporisés

Les systèmes temps réel étant réactifs et concurrents par nature, compter seulement sur un aspect syntaxique pour décrire les services requis et offerts des

composants ne suffit pas pour assurer la composition sûre du logiciel. Par conséquent, pour augmenter la fiabilité dans la conception, la spécification des services doit tenir compte non seulement du comportement dynamique, mais aussi des conditions temporelles des composants. Nous avons choisi leur prise en charge à l'aide du formalisme des *automates temporisés* [AD94] d'une part parce que celui-ci nous permet de considérer des modèles d'exécution complexes, et d'autre part parce qu'il nous permet d'établir des estimations temporelles beaucoup plus précises que celles fournies par des analyses classiques [Lj00].

Dans notre modélisation, les descriptions des composants sous forme d'automates temporisés suivent une convention pour représenter les exécutions d'appels de méthode. Chaque appel de méthode est décrit à l'aide de deux actions synchronisées, une pour indiquer le début de son exécution et une pour indiquer sa terminaison. Par exemple, étant donné une méthode *get*, nous aurons comme actions synchronisées *get* et *endget*.

Généralement, la spécification d'un service dans un contrat d'interaction d'un composant dépend du type du service : requis ou offert par le composant. Si un composant offre un service, son exécution est spécifiée comme dans la figure 2.4.

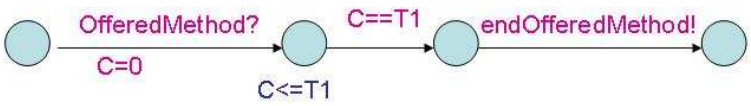


Figure 2.4. *Le service offert d'un composant*

Dans cet exemple, la variable *T1* représente le temps d'exécution de la méthode. Dans l'automate du serveur, l'exécution d'un service offert commence dès la réception de l'appel de méthode, caractérisée par une synchronisation par l'action *OfferedMethod*?. Au même instant, une horloge locale *C* est mise à 0 pour compter le temps durant lequel la méthode sera exécutée. L'automate reste à la même place jusqu'à ce que l'horloge *C* atteigne la valeur de *T1*. Une fois *C=T1*, l'appelant du service est informé de la fin d'exécution via une synchronisation par l'action *endOfferedMethod*!. Dans cette spécification, l'invariant $C \leq T1$ est ajouté à la place qui reçoit *OfferedMethod*? pour forcer la transition, une fois *C=T1*. Si cette contrainte est omise, l'automate risque de rester indéfiniment à la même place.

Pour le composant requérant un service, l'exécution du service est spécifiée comme présentée dans la figure 2.5, de manière symétrique à l'automate temporisé du service offert.

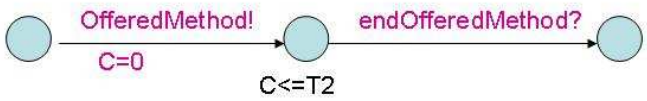


Figure 2.5. *Le service requis d'un composant*

Contrairement à la figure 2.4, l'horloge locale *C* de la figure 2.5 sera utilisée pour déterminer si la contrainte temporelle, (représentée par la variable *T2*) imposée par

l'appelant, sera respectée. Cette vérification sera effectuée par l'invariant $C \leq T2$ au sein de la place destinataire.

Alors que la spécification du temps d'exécution d'un service offert est obligatoire, celle d'un service requis est optionnelle.

2.4 Assemblage de composants : substituabilité et compatibilité

Valider un assemblage de composants temps réel revient à vérifier que la composition du logiciel est possible tout en permettant la réutilisation de ses composants. L'assemblage de composants est traduit à l'aide de deux notions : la *compatibilité* et la *substituabilité* qui sont des notions très importantes qu'il convient de traiter avec n'importe quel langage orienté composant. Car d'une part, avec la notion de substituabilité, la sûreté est garantie en permettant notamment le remplacement de composants uniquement si le nouveau composant est capable d'accomplir au moins les services de l'ancien composant, sans toutefois demander plus de services. D'autre part, avec la notion de compatibilité, la sûreté de composition de logiciel est garantie en permettant à un composant client et à un composant serveur d'être assemblés entre eux uniquement si le serveur fournit au moins tous les services requis par le client.

Dans notre langage, la compatibilité et la substituabilité apparaissent lors des affectations ou des *binding/forwarding* entre composants. Nous avons choisi de gérer ces notions formellement au niveau syntaxique, comportemental et temporel à travers la notion de sous-typage [C88, P02], étendue avec le formalisme d'automates temporisés, comme nous le verrons dans la suite.

P ::= D+ef Arch
Def ::= Idef | Cdef
Idef ::= Interface I {M+}
M ::= τb m($\overline{\tau_b}$ $\overline{x}$ *)*
Cdef ::= ActiveComponent C (Time deadline, Time period, Time offset, Int priority, $\overline{\tau_b}$ $\overline{x}$ *)*
{provides IControl itc ; ed} {stA}
| EventComponent C (Time deadline, Time period, Time offset, Int priority, $\overline{\tau_b}$ $\overline{x}$ *) {ed} {stA}*
| ProtectedComponent C (Int ceiling, $\overline{\tau_b}$ $\overline{x}$ *) {eb} {stP}*
| PassiveComponent C ($\overline{\tau_b}$ $\overline{x}$ *) {eb2} {stP}*
| Composite C {ec} { $\overline{stP}$ *}*
Arch ::= Architecture {ea}
rs ::= requires I p
ps ::= provides I p
ed ::= rs | ed1 ; ed2
eb ::= rs | ps | eb1 ; eb2
ec ::= rs | ps | C x := a | bind(π1 , π2)|forwards(π1 , π2)| ec1 ; ec2
ea ::= C x := a | bind(π1 , π2) | ea1 ; ea2
π ::= p | x.p
a ::= new C ($\overline{cst}$ *)*
cst ::= b | n | r | c | s | t
avec C ∈ ComponentNames b ∈ Bool n ∈ Int r ∈ Real
I ∈ InterfaceNames p ∈PortNames c ∈ Char s ∈ String
M ∈ MethodNames c ∈ Char t ∈ Time x,y ∈ VarNames

$\overline{stP}$ *signifie que stP peut être vide*

Figure 2.6. Syntaxe du langage

2.4.1 Le système de types

L'objectif du système de types est de garantir la sûreté de composition de logiciels, tout en facilitant la réutilisabilité. Ainsi, la flexibilité de réutilisation de logiciel est renforcée via la notion de *sous-typage*, qui définit les conditions de remplacement d'un type *T* par un type *S*.

Dans notre contexte, cela permet d'établir la substituabilité et la compatibilité des composants. Sachant qu'un composant est aussi caractérisé par un automate temporisé décrivant son comportement et ses caractéristiques temporelles, la notion de sous-typage est étendue pour prendre en compte la substituabilité et la compatibilité aux niveaux comportemental et temporel. Ainsi, l'objectif du système de types est double :

- premièrement, il permet d'un point de vue syntaxique de déterminer la compatibilité et la substituabilité des composants, et
- deuxièmement, il permet d'établir à travers le formalisme d'automates temporisés, la substituabilité et la compatibilité du point de vue comportemental et temporel.

Notre langage se base sur le système de types nominal de [P02] pour caractériser les types des interfaces et des composants, dans lesquels le nom et les propriétés structurelles sont utilisés pour les identifier de manière unique. La syntaxe et les types du langage sont présentés à la figure 2.6. Les méta-variables *C* et *I* désignent respectivement les noms des composants et des interfaces; *m* est le nom de méthode ; *p* celui du port (instance de type d'interface); *e* désigne une expression; *x* et *y* sont des noms de paramètres ou des noms de variables d'instances de composants; et *b*, *n*, *r*, *c*, *s*, *t* désignent respectivement des booléens, des nombres naturels, des réels, des caractères, des chaînes de caractères et des constantes de temps.

Dans notre langage, la substituabilité des composants a lieu dans les instructions d'affectation, alors que la compatibilité des composants apparaît lors de la liaison (*binding*) ou de la délégation (*forwarding*) des interfaces des composants.

Le comportement des composants est décrit à l'aide d'une syntaxe de haut niveau (voir figure 2.7). Les méta variables *p*, *m*, *x*, *n* et *t* correspondent respectivement aux numéros de ports, aux noms de méthodes, aux noms de variables locales, aux entiers naturels et aux constantes de temps. Les composants ont en général un comportement cyclique (d'où la clause *repeat*) ou périodique (d'où la clause *period*) caractérisée éventuellement par une échéance (d'où la clause deadline). Si un composant est déclenché par un événement extérieur ou par un composant alors l'appel de service sur une méthode offerte est traduit à l'aide de la clause *receive*. A l'inverse, un appel de service sur une méthode requise se traduit

par la clause *send*. *timeout* est une construction qui exprime le temps maximal pour obtenir un service requis alors que la clause *wait* modélise un calcul interne.

$stA ::= \textbf{period}(t_1, t_2)\ \{stb\} \mid \textbf{repeat}\ \{stb2\}$
$stP ::= \textbf{repeat}\ \{ste\}$
$stb ::= \textbf{deadline}\ (t)\ \{stc\} \mid stc$
$stb2 ::= \textbf{receive}\ p.m\ \overline{(n)}\ \overline{[asg]}\ \{stc\}\ ;\ \textbf{deadline}\ (t)\ \{std\}$
$stc ::= \textbf{wait}\ (t)\ \overline{[asg]} \mid \textbf{send}\ p.m\ \overline{(n)}\ \overline{\{std\}} \mid stc_1\ \textbf{or}\ stc_2$
$\mid stc_1\ ;\ stc_2 \mid \textbf{if}\ expb\ \textbf{then}\ stc_1\ \textbf{else}\ stc_2$
$std ::= \textbf{timeout}\ (t)$
$ste ::= \textbf{receive}\ p.m\ \overline{(n)}\ \overline{[asg]}\ \{stc\} \mid ste_1\ \textbf{or}\ ste_2$
$ste1\ ;\ ste2 \mid \textbf{if}\ expb\ \textbf{then}\ ste_1\ \textbf{else}\ ste_2$
$asg ::= x := exp \mid asg_1\ ,\ asg_2 \mid \varepsilon$
$exp ::= exp_1 \oplus exp_2 \mid (exp) \mid n \mid x$
$expb ::= exp_1 \sim exp_2 \mid expb_1 \wedge expb_2 \mid \neg\ expb \mid (expb)$
avec $\oplus \in \{+, -, \times, /\}$ et $\sim \in \{=, <, >, \geq, \leq\}$

Figure 2.7. *Syntaxe relative au comportement des composants*

2.4.2 Types

La structure de types du langage est basée sur le type des interfaces, le type des composants, le type des méthodes ainsi que sur les types de base τb. Ces différents types définis par l'utilisateur, sont mémorisés dans une table de déclaration notée Θ comme le montre la figure 2.8. Le type d'une interface I (∈ *InterfaceNames*) sert à typer les noms des ports requis et offerts d'un composant. Le type d'un composant C est défini dans la table Θ à l'aide des informations suivantes :

- $C \in$ *ComponentNames*, utilisé pour typer les variables d'instances de composants
- $R \Rightarrow P$, désignant l'ensemble des services requis et offerts par le composant
- $\langle N, l_0, X, V, \Sigma_r, \Sigma_o, E, Inv \rangle$, un automate temporisé, utilisé pour évaluer la compatibilité et la substituabilité des composants d'un point de vue comportemental et temporel.

La table de déclaration Θ sera utilisée pour vérifier les définitions des composants et des interfaces, le type des expressions et les règles de sous-typage pour les composants et les interfaces.

$\Theta ::= \varnothing \mid \Theta[Decl]$
$Decl ::= C_{decl} \mid C_{decl} \times (V_{decl}) \mid I_{decl}$
$C_{decl} ::= C : ctype \times R \times P \times \langle N, l_0, X, V, \Sigma_r, \Sigma_o, E, Inv \rangle$
$R, P ::= \{p_1:I_1, \ldots, p_n:I_n\} \quad n \geq 0$
$V_{decl} ::= x : \tau_b$
$I_{decl} ::= I : \Gamma$
$\Gamma ::= \varnothing \mid \Gamma[M_{decl}]$
$M_{decl} ::= m : \tau_m$
$\tau_m ::= \tau_{b1} \times \ldots \times \tau_{bn} \rightarrow \tau_b \quad n \geq 0$
$\tau_b ::= Nat \mid Bool \mid Int \mid Char \mid String \mid Time \mid Void$
$ctype ::= active \mid passive \mid event \mid protected \mid composite$
avec $C \in ComponentNames \quad I \in InterfaceNames$
$p \in PortNames \quad m \in MethodNames$

Figure 2.8. *Types du langage tels qu'ils sont définis dans la table Θ*

2.4.3 Substituabilité et compatibilité structurelles

Dans ce paragraphe, nous définissons la notion de sous-typage structurel entre composants, interfaces et méthodes. Les règles présentées à la figure 2.9, seront appliquées lors de l'évaluation de la compatibilité des composants au niveau des *bindings* d'interfaces, et lors de la vérification de la substituabilité au niveau des opérations d'affectation. La notion de sous-typage assure la substituabilité des composants en permettant aux composants *sous-typés* d'être utilisables par des composants *super-typés* sans que la différence ne soit décelée par les composants clients.

Ainsi dans notre langage, un composant *C'* est un sous-type d'un autre composant *C* si et seulement si :

- *C'* fournit au moins tous les services offerts par *C*,
- *C'* requiert au plus tous les services requis par *C*.

Ceci est exprimé dans la règle *C-SUB* de la figure 2.9.

Avec la fonction *iTypes* qui retourne, à partir de la table Θ, l'ensemble des types des services requis et offerts d'un composant, la relation de sous-typage entre le composant remplaçant *C'* et le composant remplacé *C* est établie en vérifiant : d'abord que *C* possède un sous-type correspondant dans *C'* pour chaque interface requise, ensuite que pour chaque interface offerte par *C*, il existe un sous-type d'interface correspondant dans *C'*.

La première condition, en vérifiant que le composant remplaçant ne demande pas plus de services, permet de garantir que tout composant qui fournit une interface compatible avec l'interface requise par *C*, sera aussi compatible avec l'interface correspondante requise par *C'*. La deuxième condition assure la continuité du service en garantissant que tout composant client qui requiert un service offert par *C* peut aussi utiliser le service correspondant offert par *C'* sans aucune différence visible. Comme présenté dans la figure 2.9, la vérification de ces deux conditions est réalisée à l'aide de la règle *S-SUB*. Dans les deux cas, *S-SUB* évalue le sous-typage de services en supposant qu'un composant peut fournir ou demander plusieurs services du même type. Cette règle vérifie que dans un ensemble *S*, étant donné une paire *p:I*, celle-ci a un sous-type correspondant *p':I'* dans *S*.

Selon la règle *I-SUB*, l'interface *I'* est un sous-type de l'interface *I*, si *I* possède une signature de méthode correspondante dans *I'* qui soit un sous-type, pour chaque signature de méthode dans *I'*.

Enfin, la règle de sous-typage (*M-SUB*) suppose que les méthodes en relation de sous-typage possèdent le même nombre et les mêmes noms d'arguments. Mais le sous-typage des méthodes suit les règles usuelles de *contravariance* et de *covariance* des types de fonctions [C88]. La contravariance assure la substituabilité de *M* par *M'* concernant les signatures des arguments (paramètres d'entrée). À ce niveau, *M'* est un sous-type de *M* si chaque argument de *M* est un sous-type de l'argument correspondant de *M'*. Ceci garantira que chaque valeur d'entrée spécifiée à l'appel

de *M* sera correctement manipulée par *M'*. La covariance, par contre, manipule le sous-typage des méthodes par rapport à la signature du paramètre de sortie. Dans ce cas, *M'* est un sous-type de *M* si le paramètre retourné par *M'* est un sous-type de celui retourné par *M*. Ceci garantit que les appelants de *M*, interpréteront correctement les valeurs retournées par *M'*.

Le principe de sous-typage d'interface est également utilisé pour évaluer la compatibilité des composants, dans les cas suivants :

- lors des liaisons des interfaces requises et offertes des composants : dans cette situation, la compatibilité est établie en vérifiant que l'interface offerte du composant serveur est un sous-type de l'interface requise du composant client. Ceci garantit que le serveur offre effectivement tous les services requis par le client.

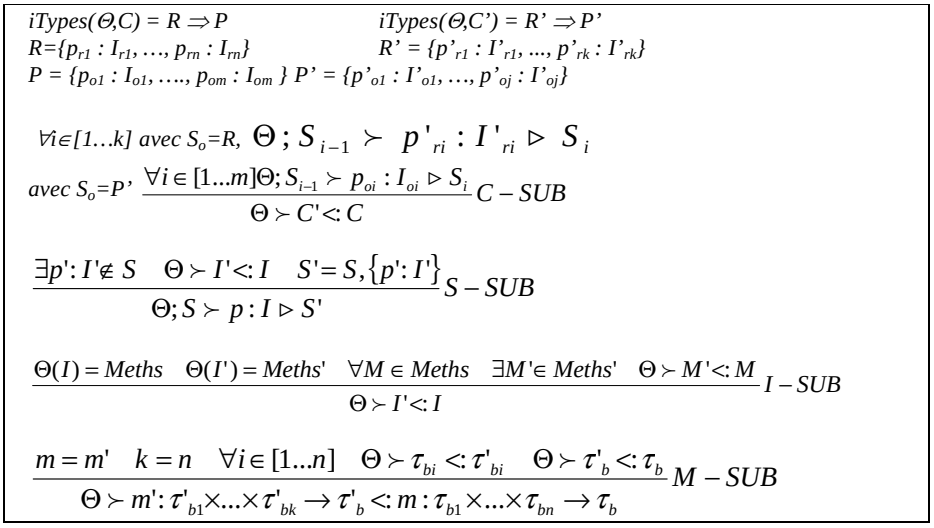


Figure 2.9. *Les règles de sous-typage*

- lors de l'exportation des interfaces des sous-composants vers celles du composite, à travers des forwarding explicites : dans ce cas, si les interfaces du composite et du sous-composant sont des interfaces requises, la compatibilité est évaluée en vérifiant que l'interface du composite est un sous-type de l'interface du sous-composant. Par contre, si les interfaces du composite et du sous-composant sont des interfaces offertes, la compatibilité est établie en vérifiant si l'interface du sous-composant est un sous-type de l'interface du composite.

2.4.4 Compatibilité et substituabilité comportementales

Dans notre modèle, la substituabilité et la compatibilité comportementales des composants sont vérifiées sur une version de l'automate temporisé dépourvu de temps. Ceci est dû au fait que le problème d'inclusion des langages entre les automates temporisés est connu comme étant non décidable [AD94].

Cependant, la compatibilité et la substituabilité temporelles des services offerts et requis sont réalisées sur un automate temporisé muni de contraintes de temps

avec des invariants sur des places utilisant une horloge *C*. Dans notre langage, la relation de sous-typage comportemental est construite sur la base de la notion de *défaillances stables* (*stable failures*) de l'algèbre de processus de Hoare [H85]. Les défaillances stables correspondent à un modèle sémantique, qui décrit le comportement d'un automate/processus comme un ensemble de tuples (*trace*, *refus*), avec *trace* caractérisant une séquence possible d'exécution à travers des actions visibles de l'automate/processus, indépendamment de l'état des variables et des contraintes gardées. Le *refus* (*refusal*) caractérise l'ensemble des actions qui peuvent être refusées après l'exécution d'une *trace* donnée.

Nous modifions le principe de défaillances stables défini dans [H85] pour : d'une part, gérer le sous-typage comportemental avec des sous-types et des super-types vus comme des composants serveur et des composants client. La raison est que les notions de sous-typage comportemental pour les services offerts et les services requis, sont symétriquement inversées. En effet, la substituabilité (sous-typage) comportementale pour les services offerts est effective si le composant sous-type fournit au moins tous les services fournis par son composant super-type et, si le composant sous-type a un comportement plus général que celui du composant super-type. Pour les services requis, le composant sous-type doit demander au plus tous les services requis du composant super-type et, le composant sous-type a un comportement plus restreint que celui du super-type.

Dans notre cas, l'évaluation du sous-typage comportemental des deux points de vue est réalisée en faisant abstraction des transitions qui ne sont pas pertinentes. Dans le cas des services offerts, nous supprimons des transitions qui décrivent les appels de services requis et dans le cas des services requis, nous faisons l'inverse, c'est-à-dire que nous supprimons les transitions des appels de services offerts. D'autre part, il faut gérer la possibilité d'avoir un composant sous-type qui offre plus de services avec un comportement moins restrictif du point de vue du serveur, ou bien qui requiert moins de services mais avec un comportement plus général du point de vue du client. Ceci s'obtient en réajustant les défaillances stables des composants via des opérations d'internalisation (*hiding*), de confinement (*concealment*) [W03] et de produit fermé (*close-product*). Ces opérations vont aider à masquer, et les services additionnels offerts par le composant sous-type lors de l'évaluation du sous-typage du point de vue du serveur, et les services requis additionnels du point de vue du composant super-type lors de l'évaluation du sous-typage du point de vue du client.

En suivant ces extensions, la flexibilité de la substituabilité est réalisée en permettant notamment :

- à un composant *C'* d'offrir plus de services que ceux réellement fournis par le composant *C* qu'il remplace, tout en ayant un comportement moins contraint.

- à un composant *C'* d'avoir moins de services requis que le composant *C* qu'il remplace, tout en ayant un comportement plus contraint.

Par rapport à la compatibilité, une telle extension assure la flexibilité dans la composition du logiciel en permettant :

- à un composant serveur de fournir plus de services que ceux réellement requis par le client composant auquel il est lié. Par conséquent, dans une telle condition, l'utilisation du serveur ne sera pas limitée à un seul type de client.

- à un composant serveur de demander d'autres services (via d'autres ports) afin d'accomplir les services offerts aux composants client.

- à un composant client de demander des services (via d'autres ports requis) autres que ceux offerts par le composant serveur auquel il est lié. Par conséquent, ceci permet à un composant client d'être satisfait par plusieurs composants serveur.

En suivant le formalisme des « défaillances stables », notre objectif est d'évaluer la conformité des composants uniquement à travers les actions visibles. Pour des raisons de simplicité, cette évaluation est réalisée à partir d'une version abstraite de l'automate temporisé des composants. Cette abstraction considère uniquement les actions et les transitions internes de l'automate indépendemment des invariants de places, des gardes ou des opérations de mise à jour sur les horloges et sur les variables entières. Ce qui s'exprime formellement à l'aide de la définition 1.

Définition 1. *Étant donné un automate* $A=\langle N, l_0, X, V, \Sigma_r, \Sigma_o, E, Inv\rangle$, *soit* ***abstract***$(A)$ *qui retourne un nouvel automate* $\langle N', l'_0, \Sigma'_r, \Sigma'_o, E'\rangle$ *dans lequel tous les invariants de place, les contraintes gardées, les opérations de mise à jour sur les horloges et sur les variables entières sont omis.*

abstract$(A) \triangleq \langle N', l_0', \Sigma_r', \Sigma_o', E'\rangle$

E' est donné par la règle d'inférence suivante : $\dfrac{l \xrightarrow{\phi,\psi,\alpha,R,asg}_{E} l'}{l \xrightarrow{\alpha}_{E'} l'} ABS-1$

où : $N' \leftarrow N,\ l'_0 \leftarrow l_0,\ \Sigma_r' \leftarrow \Sigma_r,\ \Sigma_o' \leftarrow \Sigma_o$

La figure 8 illustre l'exemple de l'automate d'un composant `CMotor` et de l'automate correspondant après application de la fonction *abstract*. En nous basant sur le formalisme d'automate abstrait, nous définissons les « *défaillances stables* » comme suit :

Définition 2. *Soit un automate abstrait* $A = \langle N, l_0, \Sigma_r, \Sigma_o, E \rangle$ *tel que :*

- $\Sigma_\tau = \Sigma_r \cup \Sigma_o \cup \{\tau\}$ *l'ensemble des actions visibles et internes de A,*
- $\sigma \in \Sigma_\tau^*$ *une trace de A*
- $Acts \subseteq \Sigma_\tau$ *un ensemble d'actions*

$\sigma \lceil Acts$, *est la trace où toutes les occurrences des actions ne figurant pas dans Acts sont supprimées.*

Définition 3. *Soient un automate abstrait* $A = \langle N, l_0, \Sigma_r, \Sigma_o, E \rangle$ *et* $l_i, l, l' \in N$, $\Sigma_\tau = \Sigma_r \cup \Sigma_o \cup \{\tau\}$ *l'ensemble des actions visibles et internes de A,* $a_i \in \Sigma_\tau$, $\Sigma = \Sigma_\tau \setminus \{\tau\}$ *et* $\sigma \in \Sigma^*$

- $l_0 \xrightarrow{a_1 \ldots a_n} l_n$ *ssi il y a des places* $l_0, \ldots, l_n$, *tel que pour* $i = 0, \ldots, n-1$:

  $a_{i+1} = a_{i+1}$ *si* $l_i \xrightarrow{\alpha_{i+1}}_E l_{i+1}$ *avec* $\alpha_{i+1} = a_{i+1}\eta$, $a_{i+1} \in \Sigma$, $\eta \in \{!, ?\}$

  $a_{i+1} = \tau$ *si* $l_i \xrightarrow{\alpha_{i+1}}_E l_{i+1}$ *avec* $\alpha_{i+1} = \tau$

- $l \overset{\sigma}{\Longrightarrow} l'$ *ssi il existe une trace* $t \in \Sigma_\tau^*$ *tel que* $l \xrightarrow{t} l'$ *et* $\sigma = t \lceil \Sigma$

L'ensemble des traces de A est noté $traces(A) = \{\sigma \in \Sigma^* \mid \exists l \in N,\ l_0 \overset{\sigma}{\Longrightarrow} l\}$. *Une place l est stable, notée stable(l), si elle ne possède pas de transitions* τ *émanant d'elle, c'est-à-dire si* $l \xrightarrow{\tau}$. *L'ensemble des actions permises dans une place stable l est* $next(l) = \{a \in \Sigma \mid \exists l' \in N,\ l \xrightarrow{a} l'\}$. *Les refus d'une place stable l sont donnés par :*
$refusals(l) = \Sigma \setminus next(l)$. *Les défaillances stables de A notées sont* $failures(A) = \{(\sigma, R) \in \Sigma^* \times 2^\Sigma \mid \exists l \in N,\ l_0 \overset{\sigma}{\Longrightarrow} l \wedge stable(l) \wedge R \subseteq refusals(l)\}$.

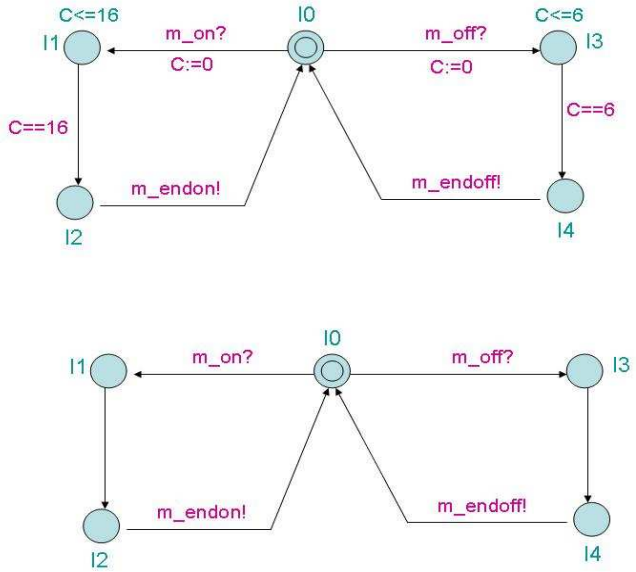


Figure 2.10 *A gauche : automate temporisé d'un composant CMotor et A droite : abstraction de l'automate du composant*

En suivant ces définitions, les traces de l'automate abstrait A, notées *traces*(A), désignent toutes les séquences possibles des actions visibles partant de la place

initiale, alors que ses « défaillances stables » notées *failures*(A) mémorisent l'ensemble des actions visibles qui peuvent être refusées après chaque trace possible.

Dans le formalisme des défaillances stables, les traces considérées sont uniquement celles menant à des places stables, c'est-à-dire des places n'émanant pas de transitions internes (τ). Par conséquent, avec un tel modèle sémantique, le comportement d'un automate est décrit uniquement par rapport à ses places stables. Dans ce cas, un automate sans places « stables » aura un ensemble vide « de défaillances stables». L'automate abstrait du composant de `CMotor`, défini dans la figure 2.10 a les traces suivantes, avec l_1 la place initiale de l'automate et ε la trace vide :

- les traces de l_1 à l_1 = {ε, m_on m_endon, m_off m_endoff}
- les traces de l_1 à l_2 = {m_on, m_off m_endoff m_on}
- les traces de l_1 à l_3 = {m_on, m_off m_endoff m_on}
- les traces de l_1 à l_4 = {m_off, m_on m_endon m_off}
- les traces de l_1 à l_5 = {m_off, m_on m_endon m_off}

Ainsi les traces de `CMotor` sont définies comme suit :
traces(`CMotor`) = {{ε, m_on m_endon, m_off m_endoff,
m_on, m_off m_endoff m_on,
m_off, m_on m_endon m_off }

Les défaillances de `CMotor` sont définies comme suit avec l_1 la place initiale de l'automate abstrait, l_1, l_3, l_5 des places stables et ε une trace vide.

- défaillances de l_1 à l_1 = {(ε, {m_endon, m_endoff}), (m_on m_endon, {m_endon, m_endoff}), (m_off m_endoff, {m_endon, m_endoff})}
- défaillances de l_1 à l_3 = {(m_on, {m_on, m_off, m_endoff}), (m_off m_endoff m_on, {m_on, m_off, m_endoff})}
- défaillances de l_1 à l_5 = {(m_off, {m_on, m_endon, m_off}), (m_on m_endon m_off, {m_on, m_endon, m_off})}

Les défaillances de `CMotor` sont alors égales à :
défailllances(CMotor) = {(ε, {m_endon, m_endoff}), (m_on m_endon, {m_endon, m_endoff}), (m_off m_endoff, {m_endon, m_endoff}), (m_on, {m_on, m_off, m_endoff}), (m_off m_endoff m_on, {m_on, m_off, m_endoff}), (m_off, {m_on, m_endon, m_off}), (m_on m_endon m_off, {m_on, m_endon, m_off})}

A travers la notion de "défaillances stables", nous définissons maintenant les opérations d'internalisation "hiding" et de confinement « concealment ».

Définition 4. *Étant donné un automate abstrait* $A= \langle N, l_0, \Sigma_r, \Sigma_o, E\rangle$ *et* $\Sigma_\tau = \Sigma_r \cup \Sigma_o \cup \{\tau\}$ *l'ensemble des actions internes et visibles de A,* $\Sigma = \Sigma_\tau \setminus \{\tau\}$, *F=failures(A) et* $Acts \subseteq \Sigma$ *un ensemble d'actions, alors :*

- $hide(Acts, F) = \{(\sigma', R') \mid \exists(\sigma, R)\in F, \sigma'=\sigma/(\Sigma \setminus Acts) \wedge R' \subseteq R \wedge Acts \subseteq R\}$
- $conceal(Acts, F) = \{(\sigma', R') \mid \exists(\sigma, R)\in F, \sigma'=\sigma/(\Sigma \setminus Acts) \wedge R' \subseteq R \cup Acts\}$

Le but de l'opération *hide* est d'internaliser un ensemble donné d'actions *Acts* au sein des défaillances d'un automate, les rendant invisibles de l'extérieur. Au sein de la description de l'automate, ceci correspond à renommer en transitions τ toutes les transactions labellisées par des actions appartenant à *Acts*. En général, l'internalisation de quelques transitions rendra des places stables non stables. Par conséquent, étant donné que le comportement d'un automate est décrit uniquement par rapport à ses places stables dans la sémantique de défaillances stables, avoir des places stables devenues non stables entraînera la suppression des paires associées (*trace*, *refus*) des défaillances de l'automate. Néanmoins, pour rendre de tels refus explicites, nous définissons la fonction d'ajustement suivante:

Définition 5. *Étant donné un automate abstrait* $A = \langle N, l_0, \Sigma_r, \Sigma_o, E \rangle$ *et* $\Sigma = \Sigma_r \cup \Sigma_o$ *l'ensemble des actions visibles de A, F=failures(A) et Acts* $\not\subset \Sigma$ *un ensemble d'actions qui n'appartiennent pas à l'alphabet de A :*

$$adjust(Acts, F) \triangleq \{(\sigma, R \cup Acts) \mid (\sigma, R) \in F\}$$

adjust() met à jour les refus R de chaque paire (σ, R) *dans F avec l'ensemble des actions de Acts.*

En suivant, les définitions décrites ici, nous définissons dans les sous-paragraphes suivants la compatibilité et le sous-typage comportementaux pour les composants.

Substituabilité (sous-typage) comportementale

Sachant que les notions de sous-typage comportemental pour les services offerts et les services requis sont symétriquement inverses, elles doivent être gérées séparément. Dans notre formalisme, l'évaluation du sous-typage comportemental indépendamment des différents points de vue est rendue possible en faisant abstraction à partir de l'automate du composant :

- des transitions d'actions concernant les appels de services requis, lors de l'évaluation du sous-typage du point de vue des services offerts ; et
- des transitions d'actions concernant les appels de services offerts, lors de l'évaluation du sous-typage du point de vue des services requis.

Dans notre formalisme, abstraire l'ensemble d'actions *Acts* à partir de l'automate A implique la suppression de A de toutes les transactions labellisées avec les actions de *Acts*. Toutefois, notre objectif est d'obtenir une spécification d'automate, dans laquelle les caractéristiques comportementales des transitions restantes sont maintenues inchangées. Ainsi, pour obtenir une telle spécification, l'idée générale est de regrouper au sein de la spécification du nouvel automate les places source et destination de chaque transition supprimée afin de rediriger les transitions restantes. Ceci prévient les places puits ou inaccessibles introduites par l'automate résultant, gardant intactes les caractéristiques comportementales des transitions restantes.

Définition 6. *Étant donné un automate abstrait* $A' = \langle N', l'_0, X', V', \Sigma'_r, \Sigma'_o, E', Inv' \rangle$ *l'automate temporisé du composant C' et* $A = \langle N, l_0, X, V, \Sigma_r, \Sigma_o, E, Inv \rangle$ *l'automate du composant C. Soient :*

- AbsA' = abstract(A') l'automate abstrait de C' de la forme $\langle N'_a, l'_{a0}, \Sigma'_{ar}, \Sigma'_{ao}, E_a' \rangle$

- AbsA = abstract(A) l'automate abstrait de C de la forme $\langle N_a, l_{a0}, \Sigma_{ar}, \Sigma_{ao}, E_a \rangle$

- VoidReqA' = removeτ(removeActs(Σ'_{ar}, *AbsA') l'automate abstrait de C', à partir duquel les services requis et les calculs internes de C' ont été omis*

- VoidReqA = removeτ(removeActs($\sum_{ar}$, AbsA) l'automate abstrait de C, à partir duquel les services requis et les calculs internes de C ont été omis
- VoidProA' = removeτ(removeActs($\sum_{ao}$, AbsA') l'automate abstrait de C', à partir duquel les services offerts et les calculs internes de C' ont été omis
- VoidProA = removeτ(removeActs($\sum_{ao}$, AbsA) l'automate abstrait de C, à partir duquel les services offerts et les calculs internes de C ont été omis
- ProCloseProd = removeInaccessibles(closeprod1(VoidReqA, VoidReqA')) un automate caractérisant l'interaction entre VoidReqA et VoidReqA'
- ReqCloseProd = removeInaccessibles(closeprod1(VoidProA, VoidProA')) un automate caractérisant l'interaction entre VoidProA et VoidProA'

A' est un sous-type de A, noté A'<: A, comme suit :

$$\frac{ProCloseProd <:_s Void\ ReqA \quad ReqCloseProd <:_s Void\ ProA'}{A' <: A} A-SUB$$

Avec $A_1 <:_s A_2$ défini comme suit :

$$\frac{isEmpty(A_2) = true}{A_1 <:_s A_2} A-SUB-1$$

$$\frac{isEmpty(A2)=false \quad containsDead(A_1)=false \quad failures(A_2) \subseteq failures(A_1)}{A_1 <:_s A_2} A-SUB-2$$

Figure 2.11. *Règle de sous-typage comportemental*

En suivant la définition 6 et la figure 2.11, du point de vue des services offerts, C' est un sous-type de C si l'interaction entre C' et C (caractérisé par *ProdCloseProd*) est conforme au comportement de C (caractérisé par *VoidReqA*). Du point de vue des services requis, le sous-typage est établi si l'interaction entre C et C' (caractérisé par *ReqCloseProd*) est conforme au comportement de C' (caractérisé par *VoidProA'*). Dans les deux cas, le sous-typage est évalué en suivant les règles A-SUB-1 et A-SUB-2 de la figure 9. La règle A-SUB-1 exprime le fait que *VoidReqA* et *VoidProA'* peuvent être vides, lors de l'évaluation du sous-typage respectivement des points de vue services offerts et services requis. Si *VoidReqA* est vide, ceci veut dire que C n'offre aucun service. Dans ce cas, C' est considéré comme un sous-type de C, indépendamment du fait qu'il offre des services. De même, si *VoidProA'* est vide, ceci signifie que C' ne requiert aucun service. Sous cette condition, sachant que C' ne dépend d'aucun autre composant, il peut être utilisé pour remplacer C.

La règle A-SUB-2 est appliqué à chaque fois que *VoidReqA* ou *VoidReqA'* ne sont pas vides. Dans ce cas, le sous-typage est établi du point de vue des services offerts :

- d'abord, s'il n'y a aucune incompatibilité dans l'interaction entre C et C' (caractérisé par *ProCloseProd*). La satisfaction de cette condition indique que C' est capable de servir n'importe quel client ayant au plus le même comportement général que C (en plus des clients qui ont le même comportement que C'), sans causer de conflit dans l'interaction.
- Ensuite, dans les défaillances de C, caractérisées par *failures* (*VoidreqA*), sont un sous-ensemble des défaillances de l'interaction *ProCloseProd*. Satisfaire cette condition veut dire que n'importe quel client ayant au plus le même comportement général que C peut être servi par C' de la manière qu'il veut et non pas par la manière dictée par C'.

De même, du point de vue des services requis, le sous-typage est établi :

- Premièrement, si l'automate d'interaction *ReqCloseProd* de C et C' ne contient pas de places « mortes ». Ce qui veut dire, que C' peut être servi par n'importe quel serveur ayant au moins le même comportement que C, sans générer de conflit dans l'interaction.
- Deuxièmement, si les défaillances de C', caractérisées par *failures*(*VoidProA'*), sont un sous-ensemble des défaillances de l'interaction *ReqCloseProd*. Ce qui signifie que n'importe quel serveur ayant au moins le même comportement que C peut servir C' de la manière demandée par C'.

Compatibilité comportementale

La compatibilité comportementale est décrite dans la définition suivante.

Définition 7. *Soient C et C' deux composants, avec C requérant les services fournis par C', Θ la table de déclaration, $Pro_{meths} = meths(\Theta, OI(\Theta, C')) \setminus meths(\Theta, RI(\Theta, C))$ l'ensemble des méthodes offertes par C' et dont n'a pas besoin C, $Pro_b = begins(Pro_{meths})$ les actions début des méthodes de Pro_{meths}, $Pro_e = ends(Pro_{meths})$ les actions de fin de méthodes dans Pro_{meths}, $Req_{Acts} = acts(RI(\Theta,C'))$ l'ensemble des actions dont a besoin C', $hacts = Req_{Acts} \cup Pro_e$, l'ensemble des actions qui doivent être cachées, A(C) l'automate temporisé de C et A(C') l'automate de C'. C est compatible avec C', noté $C \approx_c C'$ ssi : $CL_{Prob}(H_{hacts}(failures(A(C) \times A(C')))) \subseteq failures(A(C))$*

La compatibilité comportementale vérifie si les services requis par le composant client C sont toujours accomplis par C', tout en tenant compte des services additionnels offerts (CL : concealment) par C' et en omettant les services requis (H : hiding) de C'.

2.4.5 Compatibilité et sous-typage (substituabilité) d'un point de vue temporel

Un composant C' est un sous-type d'un autre composant C d'un point de vue temporel, si tous les services offerts et requis que C' partage avec C sont respectivement plus restrictifs et moins restrictifs en temps que les services de C.

Un composant client C est temporellement compatible avec un composant serveur C' si tous les services requis de C, et qui sont manipulés par C', sont moins restrictifs en temps que les services offerts par ce dernier. Le sous-typage temporel tient compte uniquement des temps d'exécution et des conditions temporelles des services offerts et requis des composants comme dans la figure 2.12.

$$A(C') <: A(C) \quad iTypes(\Theta,C)=R \Rightarrow P \quad iTypes(\Theta,C')=R' \Rightarrow P'$$

$$\forall l_r' \in R', \exists l_r \in R \quad \Theta \succ I_r <: l'_r$$

$$\forall m' \in methds(l_r') \; \exists m \in methds(I_r) \; m=m' \wedge exec(A(C), m) \leq exec(A(C'), m')$$

$$\forall I_o \in P, \exists l_o' \in P' \quad \Theta \succ I_o <: l'_o$$

$$\frac{\forall m \in meths(I_o) \quad \exists m' \in meths(I'_o) \quad m'=m \quad \wedge \quad exec(A(C'),m') \leq exec(A(C),m)}{\Theta \succ C' <: C}$$

Figure 2.12. *Règle de sous-typage combinée*

Pour vérifier d'autres contraintes temporelles, que l'on peut qualifier de contraintes globales, comme la période ou l'échéance des composants actifs/événementiels, nous devons considérer l'ensemble de la configuration des composants. C'est l'objet du paragraphe suivant.

2.5 Vérification des contraintes temporelles globales par l'analyse d'ordonnançabilité

La vérification des contraintes globales telle que l'échéance nécessite de considérer toute la configuration de composants en évaluant l'ordonnançabilité du système. Jusqu'à présent, nous avons fait l'hypothèse d'un parallélisme maximal en considérant que chaque processus (actif/événementiel) a un processeur dédié. Car dans le cas contraire, nous aurions eu à faire lors des analyses à des espaces d'états trop grands à cause des contraintes d'ordonnançabilité. Nous gardons ainsi séparée cette analyse en effectuant une abstraction des spécifications fonctionnelles de l'assemblage de composants. Ce filtrage nous permet de considérer uniquement les composants actifs et événementiels du système, les propriétés temporelles associées à leur exécution ainsi que les actions de contrôle (actions d'accès aux ressources critiques, mécanismes de *wait/signal* ainsi que l'instanciation et la terminaison des tâches) régissant leur exécution, offrant donc uniquement une vision temporelle du système.

Nous effectuons l'analyse d'ordonnançabilité de l'assemblage de composants à l'aide d'automates temporisés [AD94] car ce formalisme possède la syntaxe et la sémantique nécessaires à la prise en compte des contraintes d'ordonnancement et de ressources et a été déjà appliqué avec succès pour la modélisation et l'analyse des systèmes temps réel.

Dans ce travail, une plate-forme d'ordonnancement basée sur les automates temporisés a été développée, pour fournir respectivement un modèle de tâches plus fin et plus précis en comparaison avec les analyses classiques telle que celles du *pire cas* comme RMA[9] [Lj00], tout en étant efficace en termes de complexité de *model checking*. L'efficacité est réalisée grâce à l'utilisation d'une seule horloge pour compter les temps d'exécution, les échéances et les périodes de la configuration de tâches, contrairement à d'autres solutions qui utilisent plusieurs horloges générant par conséquent un espace d'états exponentiel rendant la décidabilité de l'analyse garantie uniquement dans des situations restreintes [GCO01, LA03, FPY02, FMPY03].

La transformation de la configuration de composants en modèle de tâches est effectuée tout d'abord en répertoriant tous les composants actifs et événementiels faisant partie de l'assemblage. Ensuite, suivant la spécification interactionnelle de chaque composant, nous identifions les actions de contrôle (*signal, acquire, release,*

[9] *Rate Monotonic Analysis*

wait) et les contraintes fonctionnelles susceptibles d'influencer leur exécution, faisant ainsi abstraction des détails fonctionnels.

Nous obtenons donc un graphe d'interaction réduit, composé uniquement d'actions régissant l'exécution du composant. Nous comptabilisons par la suite les temps d'exécution sur les transitions entre les actions restantes tout en prenant en compte ceux des actions écartées. Les temps d'exécution considérés durant cette phase proviennent tous exclusivement d'opérations offertes, car la prise en compte des contraintes temporelles des opérations requises n'a pas de sens.

Nous ne gérons pas la préemption explicitement comme dans [GCO01], car ici chaque tâche comprend des états indiquant son état prêt, en attente ou en exécution. Ce qui est géré dans chaque phase à l'aide de l'action synchronisée *okSch*[*id*] émise par l'ordonnanceur, avec *id* l'identifiant unique de la tâche. Plus précisément, l'action est utilisée pour indiquer que la tâche a fini une phase particulière et qu'elle est ordonnancée pour entamer une nouvelle phase.

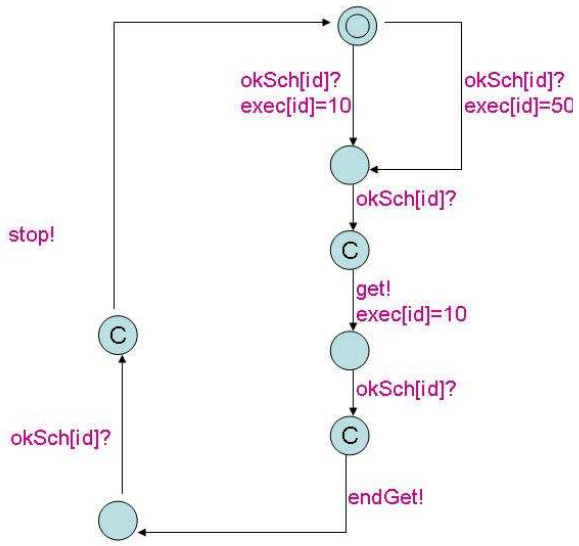


Figure 2.13 : *Automate d'une tâche simple*

En général, une tâche commence par attendre l'appel de l'ordonnanceur *okSch*[*id*] qui doit l'informer de son début d'exécution. Dans l'exemple de la figure 2.13, le début d'exécution est illustré par l'exécution non déterministe d'un des deux calculs internes possibles, chacun ayant un temps d'exécution distinct. Ceci est décrit par deux transitions *okSch*[*id*]? émanant de la place initiale. La tâche va demeurer dans la place destinataire jusqu' à recevoir un signal de l'ordonnanceur lui indiquant la fin de la phase courante en faisant un ré-ordonnançant des tâches.

Dans cet exemple, l'accès à la ressource est caractérisé par l'exécution atomique de deux actions (*get*!, *endGet*!), indiquant respectivement le début et la fin de l'opération. Des transitions atomiques sont spécifiées dans l'automate en marquant les places à partir desquelles elles émanent avec un '*C*' (pour *Committed*), dans lesquelles le temps ne s'écoule pas. Cette modélisation correspond à l'exécution d'un appel de méthode sur un objet protégé à la ADA [L98]. En recevant *okSch*[*id*]

après avoir effectué l'action de début, cela signifie que la tâche a pu accéder à l'objet protégé et exécuter sa méthode. Une fois cette partie accomplie, la tâche se synchronise de manière atomique sur *endGet*! pour indiquer la libération de l'objet protégé. Ensuite, elle se met en attente une fois de plus du signal *okSch*[*id*]. Pour indiquer la terminaison, la tâche se synchronise de manière atomique sur l'action *stop*, dès la réception de *okSch*[*id*].

2.5.1 Réajustement des contraintes de temps

Comme notre approche repose sur une horloge unique, le respect des contraintes temporelles dans le modèle nécessitera, en dehors de l'initialisation de l'horloge du système, le réajustement des contraintes de temps actives, afin de gérer le temps restant. Dans notre cas, ces contraintes actives représentent l'échéance, la période et la date de réveil des tâches, ainsi que le temps d'exécution de la tâche courante.

Le réajustement est défini selon une procédure de remise à zéro (voir procédure *reset*() de la figure 2.14) , appelée par des transitions qui possèdent des contraintes de garde du type *c==period[id]*, *c==dead[id]*, et *c==offset[id]*, où *c* représente l'horloge du système et les tables globales *period*, *dead* et *offset* représentent les temps restants pour atteindre respectivement la période, l'échéance et le réveil des tâches.

```
void reset(int val) {
 int i ;
 for (i=0;i<MaxTask;i++){
          if (dead[i]>val] dead[i]-=val;
          else dead[i]=0;

          if (period[i]>val] period[i]-=val;
          else period[i]=0;

          if (offset[i]>val] offset[i]-=val;
          else offset[i]=0;
 }
 if (tail >0) {
          if (exec[queue[0]]>0 &&
                   status[queue[0]]==RUNNING)
             exec[queue[0]-=val;
          else {
                   if (status[queue[0]]==RUNNING)
                            exec[queue[0]]=0;
          }
 }
 C=0; // reset system clock
 }
```

```
bool inQueue(int d) {
          int tmp=0;
          bool found =false;

          while (tmp<tail && !found) {
                   if (queue[tmp]==id) found=true;
                   tmp++;
          }
          return found;
 }
```

Figure 2.14. *La procédure reset() et la fonction inQueue()*

2.5.2 Gestion de la violation de l'échéance et de la période

La gestion de la violation de l'échéance et de la période est faite grâce à une fonction *inQueue*(*id*) (voir figure 2.14). Elle est utilisée pour vérifier, dès l'expiration de l'échéance ou la période, si la tâche s'exécute encore dans le système ou non. Dans le premier cas, la transition (avec la contrainte gardée *inQueue(id)*) se

synchronise avec l'ordonnanceur sur une action d'erreur indiquant la violation de la période ou de l'échéance.

Dans le second cas (transition avec contrainte gardée !*inQueue*(*id*), seule une mise à jour est faite pour réajuster les temps actifs. Si la contrainte de temps est la période, alors la transition va aussi se synchroniser avec l'ordonnanceur sur l'action de démarrage pour que la tâche se mette sur le prochain réveil.

2.5.3 Modélisation de l'ordonnanceur

L'automate de l'ordonnanceur coordonne l'ordonnancement des tâches ainsi que la gestion de leurs états à travers sa synchronisation avec des automates de niveau système (automates de démarrage, automates de ressources, automate *wait/signal*). De plus, il est responsable de l'exécution de la tâche courante. Typiquement, l'ordonnanceur effectue un ré-ordonnancement de la file d'attente des tâches, à chaque fois qu'il y a synchronisation sur une des actions suivantes :

- *start*, indique le réveil de la nouvelle tâche,
- *stop*, caractérise la terminaison de la tâche en cours d'exécution,
- *suspend*, indique la suspension de la tâche courante,
- *resume*, désigne la reprise de la tâche en attente,
- *reSch*, qui désigne simplement une requête de ré-ordonnancement.

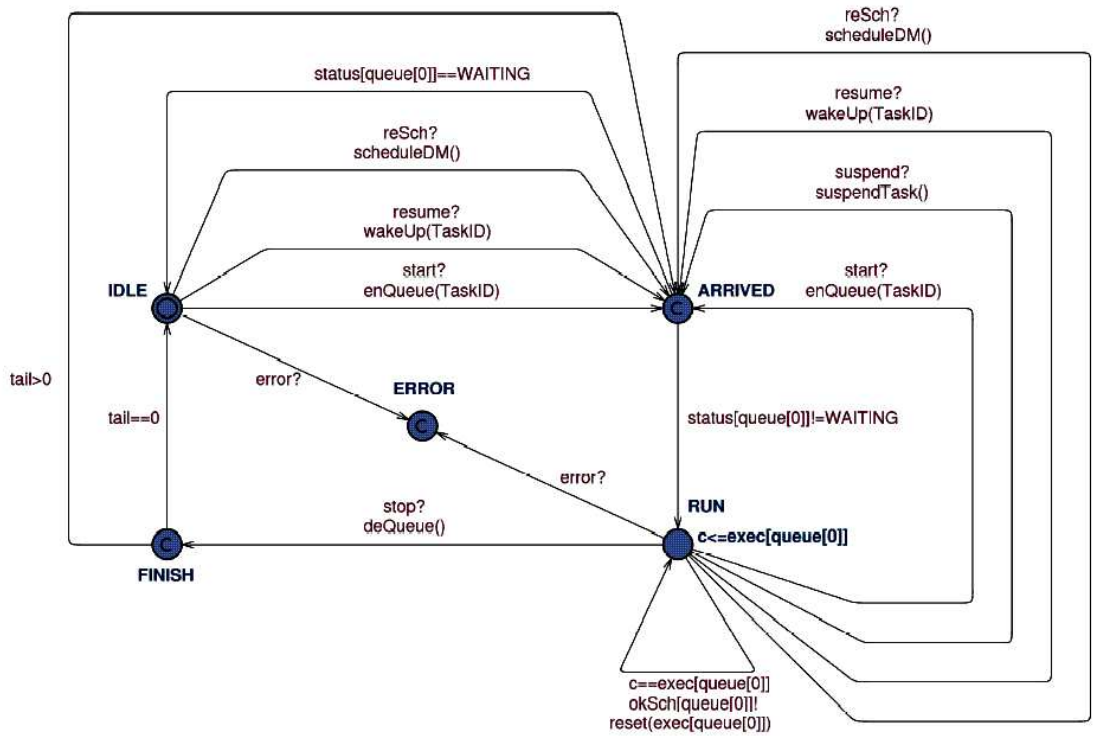


Figure 2.15. *Automate de l'ordonnanceur*

Comme dans la figure 2.15, l'ordonnanceur comprend principalement cinq états :

- État *IDLE*. *IDLE* est une place qui désigne l'état initial oisif de l'ordonnanceur. Il indique le fait qu'aucune tâche du système n'est prête à s'exécuter. L'ordonnanceur quittera cette place ou bien lorsqu'une nouvelle tâche arrive (action *start*), ou bien lorsqu'il reçoit une requête de reprise (action *resume*) en vue d'un ré-ordonnancement.

- État *ARRIVED*. *ARRIVED* est une place temporaire dans laquelle l'ordonnanceur se trouve, uniquement, lorsqu'il reçoit des actions du système provoquant le ré-ordonnancement des tâches. Dans ce cas, l'ordonnanceur a le choix entre deux options : passer à l'état *IDLE* ou passer à l'état *RUN*.

- État *RUN*. *RUN* est une place dans laquelle l'ordonnanceur se trouve lorsque la tâche la plus prioritaire s'exécute. Cette tâche correspond à la tête de liste *queue*[0].

- État *FINISH*. *FINISH* est une place temporaire de l'ordonnanceur qui caractérise la fin d'exécution de la tâche courante. A partir de cette place, selon la situation, l'ordonnanceur peut transiter ou bien vers l'état *ARRIVED* ou bien vers l'état *IDLE*. Ces places sont caractérisées par les transitions ayant respectivement les contraintes gardées suivantes : *tail*>0 et *tail*==0, avec *tail* qui contient la taille réelle de la file *queue*.

- État *ERROR*. *ERROR* est une place dans laquelle l'ordonnanceur se trouve lorsqu'une une tâche rate son échéance ou sa période.

2.5.4 Vérification

Dans ce paragraphe, nous montrons comment le modèle *checker* de UPPAAL peut être utilisé pour prouver la correction de la plate-forme d'ordonnancement et l'analyse des performances de l'application.

2.5.4.1 Vérifier l'absence d'interblocage et l'ordonnançabilité de l'application

Vérifier la correction de la plate-forme consiste à s'assurer d'abord de l'absence de l'interblocage. Ceci est exprimé à l'aide du langage UPPAAL:

A[] not deadlock

Ce qui signifie que le système est invariablement non interbloqué. Si cette propriété n'est pas vérifiée, cela peut vouloir dire que la spécification est mauvaise ou bien que la configuration de tâches n'est pas ordonnançable.

Pour distinguer ces deux situations, l'étape suivante consiste à vérifier que l'interblocage est la conséquence de l'automate ordonnanceur qui a atteint l'état *ERROR*. Ceci est exprimé comme suit :

A[] not Scheduler.ERROR

Au cas où cette propriété serait satisfaite alors que l'absence d'interblocage n'est pas assurée, elle exprimerait le fait qu'il y ait une erreur dans la spécification. Sinon, elle indiquerait seulement que l'ensemble des tâches n'est pas ordonnançable.

2.5.4.2 Établir le temps de réponse d'une tâche

Évaluer le temps de réponse d'une tâche consiste à mesurer le temps qui s'est écoulé entre l'instant de démarrage et de l'instant de fin de la tâche. Dans notre plate-forme, cette évaluation est réalisée à l'aide d'une horloge supplémentaire *rt*, qui est réinitialisée à chaque réveil de la tâche. Comme les réveils des tâches sont effectués à l'aide d'automates de démarrage, la mise à jour de l'horloge est effectuée au sein de l'automate de démarrage.

Plus précisément, ceci est réalisé dans la partie mise à jour des transitions de synchronisation, caractérisant le réveil de la tâche. Le pire temps de réponse d'une tâche s'évalue en déterminant une constante de temps *TC*, qui vérifie ce qui suit :

A[] (TaskI.STOP imply rt <= TC-1) fails
A[] (TaskI.STOP imply rt <= TC) is satisfied

Le meilleur temps d'exécution d'une tâche *TaskI* se déduit en vérifiant que la tâche se termine toujours après un temps limite spécifié. Dans ce cas, le meilleur temps correspond au maximum de temps satisfaisant la contrainte *rt>=TC*.

A[] (TaskI.STOP imply rt >= TC+1) fails
A[] (TaskI.STOP imply rt >= TC) is satisfied

Dans ce document, nous avons omis volontairement de parler de la modélisation des ressources et des mécanismes de synchronisation *wait* et *signal*. Le lecteur intéressé pourra trouver une description détaillée dans la thèse de Jean-Paul Etienne [E09].

2.6 Le modèle de composants temps réel et RTSJ

Dans cette section, nous présentons une traduction de notre modèle en *Real-Time Specification for Java* (RTSJ) [BBG+00]. L'objectif de cette expérimentation est double :

- le premier objectif est de démontrer la modularité et les capacités de réutilisation du modèle de composants proposé, à travers des applications basées sur RTSJ.

- le second concerne plutôt la spécification RTSJ elle-même.

En effet, pour profiter des avantages de Java dans le développement des systèmes embarqués et/ou temps réel, la spécification RTSJ fournit un modèle de gestion mémoire qui augmente la prédictibilité des applications Java. Néanmoins, l'utilisation de ce type de mémoires complique le développement d'applications

temps réel. En effet, ces zones mémoire n'étant pas récupérées par le ramasse-miettes, le développeur d'applications doit gérer lui-même l'allocation mémoire. Ce qui n'est pas facile à cause des règles de référence mémoire définies dans RTSJ [BBG+00].

Notre plate-forme RTSJ offre aux développeurs la possibilité de découpler la logique de l'application des complexités de gestion mémoire de RTSJ. En effet, en construisant une application avec cette plate-forme, les développeurs ont uniquement besoin de fournir la logique de l'application sous forme de composants *à la* Java (*POJO,Plain-Old-Java-Object*) ; la plate-forme générant automatiquement des codes RTSJ spécifiques à la gestion mémoire. Ceux-ci suivent quelques modèles de conception RTSJ [BN03, PFHV04, RZPCK05] pour, d'une part, assurer la non-violation des règles de référence mémoire de RTSJ, et d'autre part, éviter les fuites mémoire.

A notre connaissance, des recherches concernant l'applicabilité de l'approche composant en RTSJ ont été menées uniquement par [CGPK06]. Cependant leur modèle composant n'est pas hiérarchique. Mais plus récemment, il y a eu des travaux pour introduire de la distribution en RTSJ [MPLMS08].

2.6.1 Introduction à RTSJ

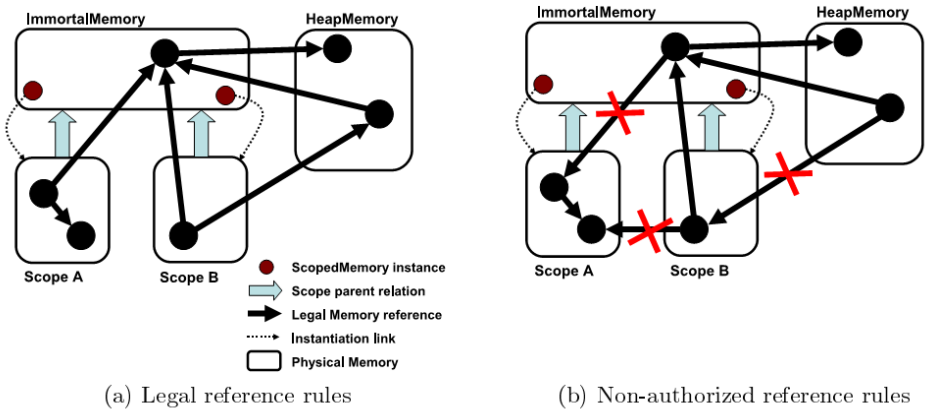
Real-Time Specification for Java (RTSJ) [BBG+00] améliore les mécanismes d'ordonnancement de Java à travers notamment la définition de deux nouveaux types de threads, *RealtimeThread* et *NoHeapRealtimeThread*, qui ont une sémantique d'ordonnancement plus précise que les threads standard de Java. Le thread *NoHeapRealtimeThread* est conçu pour offrir un support d'exécution pour un contexte temps réel strict. Ceci est garanti grâce à la non-interaction de *NoHeapRealtimeThread* avec les activités de gestion mémoire du ramasse-miettes. Selon l'approche RTSJ, ceci est réalisé de deux manières. D'une part, le *NoHeapRealtimeThread* est conçu pour s'exécuter par défaut avec un niveau de priorité plus élevé que celui du ramasse-miettes. Empêchant par conséquent la préemption du thread à chaque invocation du ramasse-miettes. D'autre part, pour éviter l'interaction avec le ramasse-miettes, le *NoHeapRealtimeThread* n'est ni autorisé à s'exécuter dans le tas, ni à allouer ou à référencer des objets du tas.

Pour permettre à des applications temps réel de s'exécuter sans subir des retards imprévisibles dus au ramasse-miettes, RTSJ définit en plus deux nouveaux types de zones mémoire, appelés mémoire *permanente* (*ImmortalMemory*) et mémoire *à portée* (*ScopedMemory*), qui n'ont aucune interaction avec le ramasse-miettes.

Cependant, l'introduction de ces régions mémoire complique le développement des applications temps réel car celles-ci ne sont pas gérées par le ramasse-miettes mais par le développeur. De plus, RTSJ fournit des caractéristiques inhabituelles pour la gestion mémoire [BBG+00] [12]: un objet alloué dans une mémoire à portée est accessible s'il y a une thread active (via la méthode *run*()) au sein de la mémoire à portée, sinon cette mémoire est libérée, désallouant l'objet. D'autre part, les objets

alloués dans la mémoire permanente restent toute la durée de vie de l'application. De plus, les règles de références mémoire (voir figure 2.16), c'est-à-dire la règle de « durée de vie » et la règle « à parent unique» impliquent des mécanismes qui interdisent d'avoir une portée de longue vie qui détient une référence à un objet alloué dans une portée de durée de vie plus courte.

De plus, une portée mère ne peut pas détenir une référence directement ou indirectement à un objet alloué dans une de ses portées filles. Toutes ces restrictions forcent le développeur à être très attentif quant à la gestion de l'empreinte de l'application sans causer de perte mémoire, et sans violer les règles de référence mémoire de RTSJ.



(a) Legal reference rules (b) Non-authorized reference rules

Figure 2.16. *Les règles de référence mémoire*

2.6.2 Traduction du modèle de composant en RTSJ

Bien que les modèles de conception (*design patterns*) de RTSJ [BN03, PFHV04, RZPCK05] permettent une manipulation efficace de la mémoire, le développeur doit néanmoins se préoccuper de la manière d'intégrer ces modèles dans la structure de son application. Ici, le défi est double :

- en premier lieu, minimiser la possibilité d'avoir des fuites mémoire : l'empreinte mémoire de l'application devra être structurée de manière à maintenir les objets de longue durée et les objets de courte durée, séparément dans des portées ancêtres et filles respectivement.

- deuxièmement, les modèles de conception de RTSJ doivent être adaptés à la structure de l'application à concevoir.

Selon les caractéristiques de l'application, effectuer ces tâches peut être très onéreux et sujet aux erreurs. Par conséquent, nous avons cherché à fournir une preuve par concept à travers la plate-forme RTSJ développée qui :

- offre un cadre pour renforcer la modularité et la réutilisabilité dans le développement d'applications temps réel, basé sur notre modèle de composants ;

- donne aux programmeurs la possibilité de développer des applications temps réel, sans se préoccuper de la complexité de gestion mémoire induite par RTSJ. Les développeurs ont uniquement besoin de fournir la logique de l'application sous forme de composants *POJO* (*Plain-Old-Java Objects*). La plate-forme générant automatiquement les codes RTSJ appropriés ;

- et qui minimise la possibilité d'avoir des fuites mémoire durant le développement d'applications.

Dans cette recherche, nous adoptons une structure mémoire enrichie à l'aide de modèles de conception, où chaque composant est alloué au sein d'une portée distincte. Cette politique d'allocation nous permet de contrôler la durée de vie de chaque composant individuellement, tout en fournissant un moyen efficace pour gérer l'empreinte mémoire de l'application. La portée reste occupée tant que le composant correspondant est actif. Une fois que le composant s'est terminé, la zone mémoire devient disponible encore une fois pour réallocation. Ainsi, cette mémoire structurante nous permet de changer la configuration de l'application durant l'exécution tout en garantissant la disponibilité mémoire.

Cependant, l'adoption d'une telle structure mémoire soulève un problème important, qui consiste à trouver comment effectuer les *bindings* des interfaces sans violer les règles de références mémoire de RTSJ [RZPCK05] [W04]. Ce problème est résolu grâce à l'incorporation au sein des *composites* de code basé sur les modèles de conception de RTSJ [BN03] [RZPCK05] [PFHV04], pour garantir l'accessibilité des composants.

2.6.2.1 Structure du composant RTSJ

Notre plate-forme RTSJ orientée composant comporte principalement trois classes abstraites, correspondant respectivement aux entités active, passive et composite (figure 2.17).

La classe abstraite *Component* implémente l'interface *IComponent* qui fournit des méthodes pour l'initialisation et la terminaison du composant. Elle intègre en plus le *thread* **Wedge** dont le rôle est de garder active la portée associée au composant. Comme cela est décrit dans [RZPCK05], le *thread Wedge* est un thread qui s'endort sans aucun traitement spécifique durant la vie du composant jusqu'à son appel via la méthode *terminate()* pour libérer la mémoire.

La classe abstraite *ActiveComponent* étend la classe *Component* avec l'encapsulation d'un *RealtimeThread* avec une portée qui lui est associée, permettant l'allocation temporaire d'objets. *ActiveComponent* redéfinit la méthode *terminate()* de la classe *Component* pour inclure le code nécessaire pour arrêter l'exécution de *RealtimeThread*, dès la fin du composant actif.

En général, implémenter un composant passif ou actif revient à implémenter une classe qui étend la classe abstraite correspondante (voir figure 2.17). L'écriture de la logique du composant ne nécessite aucune connaissance sur la gestion mémoire de RTSJ. Le code nécessaire est généré au niveau du composite. De plus,

comme la structure des classes d'implémentation est générée automatiquement à partir des spécifications abstraites du modèle de composants, le développeur se contentera de remplir le corps du squelette des méthodes des classes.

Par exemple, lors de l'implémentation d'un composant actif, le développeur complètera la méthode *run*() de *RealtimeThread* du composant avec la logique de ce dernier. Le composant et la structure de son thread temps réel, tout comme ses paramètres temporels et d'ordonnancement, sont générés automatiquement suivant la spécification abstraite du composant. Toutefois, une attention particulière doit être portée à l'utilisation des librairies Java standard [BN03]. Le développeur doit s'assurer que ces librairies ne causent pas de fuite mémoire au sein du contexte d'allocation du composant.

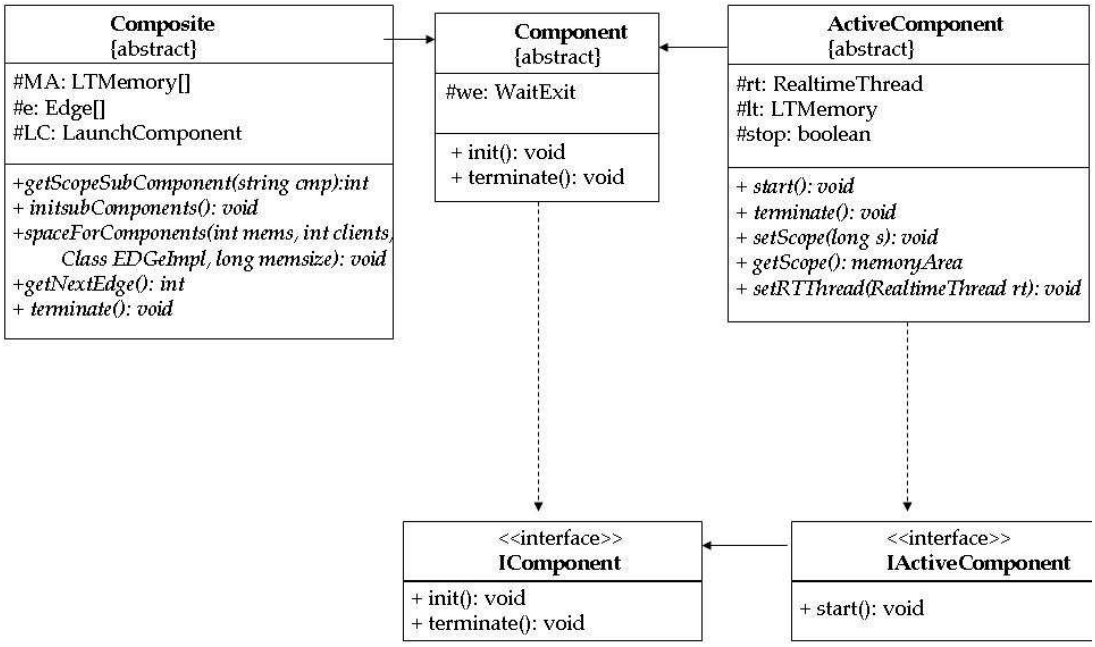


Figure 2.17. *Structure de la plateforme RTSJ*

2.6.2.2 Spécifier les services offerts et requis

Dans notre plate-forme RTSJ, chaque service offert par un composant est caractérisé par un ensemble de signatures exposées à travers une interface. En suivant la syntaxe de Java, ceci est réalisé avec le mot clé *implements*. Dans l'exemple de la figure 2.18 (voir programme 1), étant donné le composant *BufferOne*, le service qu'il offre est défini à travers l'interface *IBuffer* qu'il implémente.

Pour séparer l'implémentation de la spécification, le composant est référencé et accédé uniquement via ses interfaces. Pour augmenter l'adaptabilité et la réutilisation des composants, les services requis suivent la même philosophie d'implémentation. Comme Java ne permet pas de définir explicitement les services requis, une telle spécification est rendue possible grâce à l'utilisation du modèle de conception d'*inversion de contrôle* (d*ependency injection*) défini dans [F04]. Pour augmenter la réutilisation, le modèle d'inversion de contrôle permet en particulier

à une classe de référencer ses services requis uniquement via les interfaces privées de la classe.

Dans notre cas, nous utilisons une forme d'inversion de contrôle appelée *interface injection*. Comme dans la figure 2.18 (voir programme 2), tout composant qui dépend d'un ou de plusieurs composants externes doit implémenter l'interface *IBindController*, qui fournit les méthodes nécessaires à la liaison (*bindInterface*) de ses interfaces privées avec les références concrètes de ces composants, ou bien à la suppression de cette liaison (*unbindInterface*).

Généralement, un composant offrant un ou plusieurs services implémentera une ou plusieurs interfaces, et un composant requérant un ou plusieurs services implémentera une ou plusieurs interfaces privées correspondantes, qui seront liées lors de la liaison avec des implémentations concrètes.

```
public interface IBuffer {
 public void put(int i);
 public int get();
 public boolean full();
 public boolean empty();
}
```

```
public class BufferOne extends
Component implements IBuffer {
 private int[] list;
 private int head;
 private int tail;
 private int numofelements;

public BufferOne()
 {
    list = new int[1];
    head=0;
    tail=0;
    numofelements=0;
 }
 public void put(int i)
 {
    list[tail]=i;
    tail=(tail+1)%list.length;
    numofelements++;
 }
...
}
```

```
public class Producer extends
ActiveComponent implements IBindController {
 private IBuffer b;
 private IMutex m;
 ...
 public Producer()
 {
        SchedulingParameters sp = new
        PriorityParameters(...);
        PeriodicParameters pp = new
        PeriodicParameters(...);
        SizeEstimator s1 = new SizeEstimator();
        s1.reserve(MyHeap.class, 1);
        setScope(s1.getEstimate());
        setRTThread(new MyHeap(sp,pp,getScope()));
 }
 public void bindInterface(String itfName,
 Object o) {
        if (itfName.compareTo("IBuffer")==0) {
        b = (IBuffer)o;
        } ...
 }
public void unbindInterface(String itfName)
{
        if (itfName.compareTo("IBuffer")==0) {
        b = null;
        } ...
}
private class MyHeap extends NoHeapRealtimeThread
 {
        private boolean con;
        ...
        public MyHeap(SchedulingParameters sp,
        PeriodicParameters pp, MemoryArea ma)
        { super(sp,pp,ma);
        ...
 }
 public void run() {
        int i=0;
        while(con && !stop) {
        m.acquire();
        if(!b.full()) { b.put(i); ... }
        else {...}
        ...
        con=waitForNextPeriod();
        }
 }
```

Programme 1: composant passif	} Programme 2: composant actif

Figure 2.18. *Exemples de classes d'implémentation d'un composant passif et un composant actif*

2.6.2.3 Structure du composite RTSJ

Dans notre modèle, le composite est une entité unique structurante qui permet la réutilisation à gros grain du composant. En effet, il joue un rôle central dans la gestion des contextes d'allocation et de l'accessibilité de ses sous-composants. Il est ainsi responsable de :

- la création et de la gestion des portées mémoire associées aux sous-composants ;
- le contrôle de la durée de vie des sous-composants : l'instanciation, l'initialisation, le lancement (pour les composants actifs) et leur terminaison ;
- la gestion des invocations de méthodes des sous-composants serveur, impliquant la traversée de portées ;
- la gestion des liaisons d'interfaces des sous-composants.

Comme le montre la figure 2.19, étant à la base un composant, la classe abstraite *Composite* étend la classe *Component*. Elle gère les portées des sous-composants et leur accessibilité à travers :

- une liste de portées mémoire, chacune étant associée à un sous-composant distinct,
- un pool de *threads Edge*, qui offrent des mécanismes pour l'accessibilité aux sous-composants en traversant leurs portées,
- un thread *LaunchComponent* pour gérer le cycle de vie des sous-composants.

Initialement, le composite fait appel à la méthode *spaceForComponents* en précisant :

- le nombre de portées nécessaires, pour initialiser les sous-composants ainsi que leur taille,
- la classe d'implémentation de Edge qui contient les mécanismes nécessaires pour gérer l'accessibilité à ses sous-composants,
- et le nombre d'instances de la classe Edge nécessaires pour gérer le nombre d'accès concurrents potentiels.

Les classes de type *Edge* sont inspirées des modèles de conception de *Handoff* [PFHV04] et de ceux l'invocation inter-portée (*CrossScope Invocation*) de [RZPCK05] utilisées ici, pour permettre à des composants appartenant à des portées différentes de communiquer sans violer les règles de références mémoire de RTSJ.

```
public class MyComposite extends Composite
implements IBuffer, IMutex, ...
{
 ...
 public void init()
 {
 super.init();
 spaceForComponents(5, 4, MyEdge.class, 8000);
 initSubComponents();
 LC.setCName("Producer1");
 LC.setExeOpt(LaunchComponent.LC_START);
 MA[getScopeSubComponent("Producer1")].enter(LC);
 ...
 }
public void initSubComponents()
{ //INITIALIZING COMPONENTS
 LC.setExeOpt(LaunchComponent.LC_INIT);
 LC.setCName("Producer1");
 MA[getScopeSubComponent("Producer1")].enter(LC);
 LC.setCName("SynchronizedBuffer");
 MA[getScopeSubComponent("IBuffer")].enter(LC);
 //BINDING INTERFACES
 LC.setExeOpt(LaunchComponent.LC_BIND);
 LC.setCName("IBuffer");
 LC.setObjectToBind(this);
MA[getScopeSubComponent("Producer1")].enter(LC);
 ...
 }

protected int getScopeSubComponent(String cmp)
 {
 int tmp=-1;
 if (cmp.compareTo("IBuffer")==0) {tmp=0;}
 else if (cmp.compareTo("Producer1")==0){tmp=1;}
 ...
 return tmp;
 }
public void put(int i) //IBuffer interface method
 {
 int tmp = getNextEdge();
 e[tmp].setItfOpt("IBuffer", "put");
 ((MyEdge)e[tmp]).setParamInt(i);
MemoryArea.getMemoryArea(this).executeInArea(e[tmp]);
 e[tmp].unlock();
 }
 ...
}
```

Figure 2.19. *Exemple de composite*

2.6.2.4 Gestion de sous-composants

La gestion des sous-composants est réalisée avec le composant *LaunchComponent*, qui est une classe membre du composite. *LaunchComponent* est un *RealtimeThread* de plus haute priorité que celles des composants actifs de l'application. Il est appelé par la méthode *initSubComponents*() pour initialiser, démarrer, terminer et lier les interfaces des sous-composants. La méthode *terminate*() libère les sous-composants avec leurs portées mémoire.

Un extrait de son comportement est donné dans la figure 2.20.

```
public class LaunchComponent
extends RealtimeThread {
…
private String str;
private int exeopt;
private Object bindobject;
…
public void setCName(String st) {
   str=st;
}
public void setExeOpt(int opt){
    exeopt=opt;
}
public void setObjectToBind(Object o) {
    bindobject=o;
}

public void run() {
  if (exeopt==LC_INIT) {
    try{
      ComponentFactory CF = ComponentFactory.
                                  getInstance();
     ScopedMemory  sm =
              ScopedMemory)RealtimeThread.
                   getCurrentMemoryArea();
    sm.setPortal(sm.newInstance(CF.getClass(str)));
   ((IComponent)sm.getPortal()).init();
}
catch(ClassNotFoundException cnfe) { …}
}
else if (exeopt==LC_START){
     ((IActiveComponent)((ScopedMemory)RealtimeThread.
getCurrentMemoryArea()).getPortal()).start();
}
else if (exeopt==LC_TERMINATE){
   ((IComponent)((ScopedMemory)RealtimeThread.
     getCurrentMemoryArea()).getPortal()).terminate();
}
else if (exeopt==LC_BIND){
    ((IBindController)((ScopedMemory)RealtimeThread.
    getCurrentMemoryArea()).getPortal()).
    bindInterface(str,bindobject);
  }

 }

}
```

Figure 2.20. *La classe d'implémentation du composant LaunchComponent*

2.7 Conclusion

La complexité croissante des systèmes temps réel a conduit au développement de nouvelles méthodologies pour réduire la complexité du logiciel et fournir un support facilitant sa réutilisation. Durant ces dernières années, l'approche *composant* a émergé comme le nouveau paradigme pour assurer une plus grande maîtrise, réutilisation et fiabilité du logiciel. A travers ce travail de recherche, nous avons conçu un modèle de composants, qui fournit les abstractions nécessaires et les moyens pour concevoir, analyser et valider les systèmes temps réel.

Nous avons donc proposé un système de types, étendu avec les automates temporisés, permettant de vérifier la compatibilité et la substituabilité des composants de différents points de vue : structurel, comportemental et temporel. Par ailleurs, l'analyse d'ordonnançabilité de la configuration de composants nous a permis de nous assurer du respect des contraintes temporelles globales.

Nous avons ensuite étudié l'utilisation de RTSJ pour implémenter de manière structurée notre modèle de composants. Nous avons alors montré comment notre plate-forme peut fournir un modèle de programmation facilitant le développement temps réel en utilisant RTSJ. En effet, à travers notre plate-forme, le développeur ne doit plus se préoccuper des complexités de la gestion mémoire de RTSJ, tout en tirant profit des avantages offerts par le paradigme des composants.

Ce travail s'est effectué dans le cadre de la thèse de Jean-Paul Etienne. Et le fait que cette thèse ait été encadrée par deux personnes : Gérard Florin et moi-même,

ayant des compétences différentes voire complémentaires pour le sujet traité, a permis de combiner l'approche temps réel avec l'approche composant en confrontant des concepts et des points de vue qui étaient de temps en temps différents. Par rapport à l'objectif initial qui consistait à définir un modèle de composants pour les applications temps réel, nous estimons que cet objectif est atteint. En effet, à notre connaissance, jusqu'à présent il n'existe pas de travaux qui utilisent la notion de sous-typage pour vérifier la compatibilité des composants de différents points de vue : syntaxique, comportemental et temporel. En plus de RTSJ, une traduction de notre modèle de composants a été faite vers celui de TinyOS, dans le cadre d'un stage de Master[10].

En France, les travaux qui ont été menés sur les ADL temps réel [DF06] se concentrent davantage sur les aspects liés au temps réel que sur la notion de compatibilité des composants.

Le projet REVE[11] qui se donne comme objectif de construire un modèle de composant, un support d'exécution et un système de typage pour les applications embarquées du domaine du temps réel tel que le domaine du transport ferroviaire, s'intéresse davantage à la vérification fonctionnelle des composants qu'aux notions de concurrence et d'interaction telles que nous les définissons. Les travaux en cours du projet REVE se sont notamment inspirés de nos travaux sur RTSJ pour proposer un modèle de composants en RTSJ [PMS08].

Cependant, certains points n'ont pas été traités dans cette recherche et pourraient donc faire l'objet de perspectives, tels que :

- l'utilisation d'un ordonnancement distribué lors de l'analyse d'ordonnançabilité ou
- la vérification fonctionnelle de l'assemblage.

D'autre part, le modèle proposé ici étant générique, il n'est sûrement pas adapté à un domaine d'application particulier contrairement à des modèles qui existent dans le domaine de l'avionique AADL [SAE04] ou de l'automobile AUTOSAR [FB et al. 06] par exemple. C'est pour cette raison, que j'ai initié une recherche autour de l'utilisation de l'approche composant en ciblant des plateformes embarquées JavaCard. C'est le travail de thèse de LE Hai Binh. Cette problématique me semble intéressante et mérite d'être traitée pour proposer un modèle de composants qui tienne compte des contraintes d'énergie, de mémoire et de puissance de calcul afin de définir une architecture de composants adaptés aux systèmes contraints par ce type de ressources. L'exemple de TinyOS, un système d'exploitation dédié aux réseaux de capteurs et orienté composant, confirme bien que l'utilisation de cette approche est intéressante.

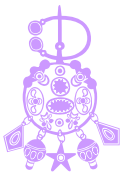
[10] *Stage de Master SEM de Mr Sadaoui Houas, soutenu en septembre 2006*
[11] *http://reve.futurs.inria.fr/*

Chapitre 3

Les systèmes embarqués contraints par les ressources

exemple des plates-formes *JavaCard*

3.1 Introduction



La ressource « temps » est une ressource bien étudiée mais de nouveaux enjeux sont apparus avec l'émergence des systèmes enfouis comme support des applications grand public. Après avoir étudié les facettes principales de l'ingénierie des applications temps réel, il nous a semblé indispensable d'étudier des systèmes très contraints en ressource mémoire et en puissance de calcul avec les cartes à puce. L'arrivée de Pierre Paradinas de Gemplus au CNAM a pu concrétiser l'émergence de ces travaux de recherche, à travers le co-encadrement de la thèse de Julien Cordry et mon implication dans le projet MESURE.

Dans ce chapitre, nous présenterons les résultats de cette recherche et nous montrerons comment mesurer le temps d'exécution pour caractériser les plates-formes embarquées *JavaCard* et calculer leur performance pour ainsi déduire leur efficacité. En effet, Les plates-formes ouvertes utilisées aujourd'hui dans les grandes applications de la téléphonie mobile, du paiement et des documents électroniques (carte d'identité, passeport,...) sont construites sur des plates-formes ouvertes *JavaCard*. Néanmoins ce domaine ne dispose pas de bancs de mesure de performances ouverts et reconnus.

Par ailleurs, la mesure de performances et l'évaluation des caractéristiques d'une carte sont un sujet qui se révèle complexe, pour diverses raisons :

- En raison de la complexité grandissante des plates-formes, la performance d'un point d'entrée de l'API ou d'une caractéristique de la machine virtuelle ne peut pas être déterminée par une simple mesure, mais nécessite plusieurs mesures couvrant les divers contextes d'utilisation ;

- Afin de déterminer une notation globale relative à une application ou à une catégorie d'applications, il est indispensable de pondérer les différentes mesures effectuées. La détermination de ces coefficients de pondération est une tâche difficile, et cependant essentielle : une mauvaise définition de ces valeurs rendrait le banc de mesure inutilisable ;

- Les caractéristiques des cartes ne sont pas uniformes, ce qui rend difficile une comparaison globale.

Notre objectif, très ambitieux, dans ce travail a été de mettre à la disposition de l'industrie de la carte à puce un outil ouvert permettant d'évaluer et de comparer les performances et les caractéristiques des cartes basées sur la technologie *JavaCard* , permettant ainsi d'avoir un nouvel élément de comparaison des produits en plus de ceux existants, comme le prix ou la sécurité.

Ce travail a été réalisé dans le cadre du projet ANR MESURE (mai 2006-mars 2008) et a permis le financement de deux années de thèse de Julien Cordry sur ce sujet. Le projet MESURE a reçu un accueil très favorable de la part de l'industrie de la carte à puce. En effet, une présentation de ce travail avec démonstration de l'outil de mesure à la conférence *e-Smart 2007,* qui a réuni plus de 500 personnes du monde de la carte à puce, s'est vue décerner le prix Isabelle Attali[12]. Notre travail a fait l'objet d'un certain nombre de publications [7, 8, 10, 11, 35] dont l'article [7] en annexe F, publié à la conférence *CARDIS* en septembre 2008.

Nous commencerons par décrire, dans la section 3.2, les caractéristiques des cartes à puce, notamment en tant que systèmes à ressources mémoire limitées. Nous expliquerons ensuite, en section 3.3, la méthode de mesure pour laquelle nous avons opté et nous rappellerons les objectifs à atteindre. Les principes de la technologie *JavaCard* ainsi que les caractéristiques des plates-formes *JavaCard* seront décrits dans la section 3.4. Dans la section 3.5, nous détaillerons la méthodologie adoptée pour la mesure du temps d'exécution moyen. En nous basant sur des plates-formes *JavaCard 2.2*, nous montrerons, entre autres, comment éliminer le bruit et isoler le code du *byte-code* à mesurer ; mais aussi comment mesurer un temps moyen pour chaque opération afin de déduire une note de performance, selon les domaines d'application ou *profils,* ou indépendamment de ceux-ci. Puis nous ferons un état de l'art des tentatives de mesure de performance existantes avant de conclure, en section 3.7, en recensant les fonctionnalités qui seraient réutilisables dans des *JavaCard 3.0,* dont la spécification a été publiée par SUN MicroSystems en Mars 2008.

---

[12] Créé en 2005, en hommage à la directrice de l'INRIA décédée lors du terrible tsunami de 2004, ce prix distingue la communication scientifique la plus innovante présentée lors de la conférence annuelle Smart Event.

3.2 Les systèmes contraints par la ressource mémoire : exemple de la carte à puce

Nous nous intéressons ici à un autre type de système caractérisé non pas par une contrainte de nature temporelle mais plutôt une contrainte liée à l'empreinte mémoire limitée.

En effet, la carte à puce représente une des plus petites plates-formes en utilisation aujourd'hui. Les cartes à puce sont de petits ordinateurs aux ressources physiques très réduites :

- un processeur 8, 16 ou 32 bits, avec une vitesse d'horloge de 5 à 40 MHz,
- une mémoire ROM de 32 à 128 Ko,
- une EEPROM de 16 ou 64 Ko,
- et une mémoire RAM entre 3 et 5 Ko.

Dans ce contexte, le grand défi des développeurs *JavaCard* a été d'embarquer une machine virtuelle *Java* sur une carte à puce tout en laissant suffisamment d'espace mémoire pour les applications. La solution proposée par SUN [C00, JC08] a été de supporter uniquement un sous-ensemble du langage *Java* et de décomposer la *JVM* en deux parties : une partie sur la carte (*on card*) et une partie hors carte (*off card*).

Dans notre recherche , nous nous sommes intéressés aux plates-formes embarquées *JavaCard* comme exemple de systèmes contraints par la ressource mémoire, et notre problématique a été de mesurer le temps des opérations qui s'exécutent sur ce type de plates-formes afin de caractériser leur efficacité et de déduire leurs performances.

Dans la suite de cette section, nous décrirons les caractéristiques de la technologie JavaCard pour déduire les fonctionnalités mesurées ainsi que la méthodologie développée pour atteindre cet objectif.

3.3 Méthodes de mesure

Il existe trois approches pour définir une méthode de mesure.

- La première consiste à simuler le système (ici la plate-forme JavaCard). Le modèle de simulation doit intégrer un programme qui représente les caractéristiques de la carte. Cependant, il est difficile de simuler toutes les fonctionnalités de la carte comme, par exemple, la mémoire cache. Par conséquent, l'utilisation d'un tel modèle ne produira pas des résultats d'une grande précision.

- La seconde approche consiste à construire un modèle analytique du système. Mais celle-ci bute contre la difficulté de décrire mathématiquement toutes les caractéristiques des cartes à puce.

- La dernière approche consiste à mesurer le système lui-même. Selon David J. Lilja [Ld00], cette approche est moins flexible que les autres, mais c'est la plus précise avec un haut niveau de confiance compte tenu des mesures effectuées. C'est cette dernière solution que nous avons retenue.

La méthodologie proposée s'attachera à résoudre les points suivants :

- déterminer toutes les fonctionnalités dont il faut tenir compte, ainsi que les différents contextes d'utilisation qui ont un impact sur les performances de ces fonctionnalités ;

- transformer les mesures brutes en une note (ou des notes) représentant la performance globale d'une plate-forme dans un contexte donné ;

- définir des profils car une carte utilisée dans un profil n'aura pas besoin des mêmes performances dans un autre profil (une carte de paiement utilisée dans un profil transport ne nécessite pas des performances en termes de vitesse de la transaction).

3.4 Les plates-formes JavaCard

La carte à puce communique avec le monde extérieur via le protocole *APDU* (*ISO 7816–4 standard*) en mode maître-esclave. C'est le terminal (lecteur de cartes) qui initialise la communication en envoyant une commande *APDU* à la carte et celle-ci retourne une réponse *APDU*, avec un contenu éventuellement vide.

Dans le cas des plates-formes *JavaCard*, la mémoire ROM héberge la *JavaCard Virtual Machine* (*JCVM*) qui implémente un sous-ensemble du langage *Java*, et qui permet aux applets *JavaCard* de s'exécuter. Le langage *JavaCard 2.2* présente quelques limitations par rapport au langage *Java* ; en effet, les fonctionnalités suivantes ne sont pas supportées :

- les types de données : long, double, float,
- les caractères et les chaînes de caractères,
- les tableaux multi-dimensionnels,
- le chargement dynamique de classes,
- les threads.

Les autres caractéristiques telles que la notion d'interface, d'héritage, de méthodes virtuelles, de surcharge, de création dynamique d'objets sont supportées par le langage *JavaCard* [JC06].

La technologie *JavaCard* définit trois éléments destinés à être utilisés sur une carte à puce (voir Figure 3.1) :

- la *JavaCard Virtual Machine (JCVM):* elle fournit un sous-ensemble du langage *Java* (comme cela est décrit plus haut) ainsi qu'une machine virtuelle *Java* appropriée, compte tenu des ressources limitées des cartes à puce ;

- le *JavaCard Runtime Environment (JCRE): c'*est l'environnement d'exécution des plates-formes *JavaCard*. Il est responsable de la gestion des ressources de la carte, de la communication avec le terminal, de l'exécution des applets et du système de sécurité ;

- la *JavaCard Application Programming Interface (API):* elle fournit un support pour la gestion des *APDUs*, des Application IDentifiers (AIDs), de routines système, des codes PIN, etc.

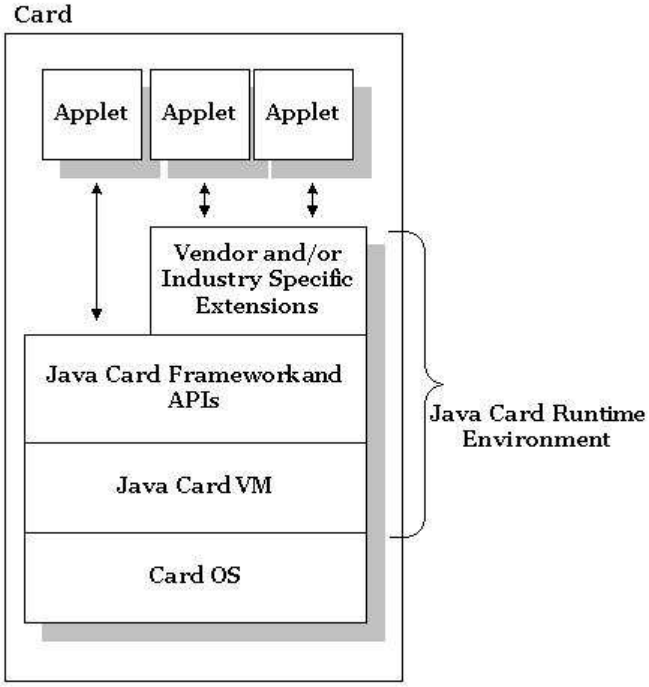


Figure 3.1. Une plate-forme JavaCard

Ce travail de recherche entre dans le cadre du projet *MESURE*, projet ANR, ayant pour objectif de développer un ensemble d'outils *open source* de mesure de performance des plates-formes *JavaCard*.

Les fonctionnalités mesurées ici sont relatives à une phase d'utilisation normale des applications *JavaCard*. Sont exclues des fonctionnalités telles que l'installation, la personnalisation ou la suppression d'une application car elles ne sont pertinentes ni du point de vue utilisateur, ni du point vue performance.

L'évaluation de performances s'effectue sur trois niveaux différents:

- *Niveau VM* : pour mesurer le temps d'exécution des instructions de base de la machine virtuelle, ainsi que le temps d'exécution des mécanismes

sous-jacents de la VM comme, par exemple, la lecture ou l'écriture en mémoire.

- *Niveau API* : pour évaluer le fonctionnement des services proposés par les librairies disponibles sur le système embarqué (méthodes de l'API, celles de *JavaCard* et *GlobalPlatform*).

- *Niveau JCRE* : pour évaluer des services non fonctionnels comme la gestion des transactions, l'invocation de méthodes dans les applets, etc.

L'ensemble des tests est fourni pour mesurer n'importe quelle plate-forme *JavaCard* reliée à n'importe quel lecteur. Les tests doivent par conséquent retourner des résultats précis même s'ils ne sont pas exécutés sur des lecteurs précis. Nous atteignons cet objectif en éliminant la faiblesse potentielle du lecteur en termes de délai, de variance et de prédictibilité et en contrôlant le bruit généré par l'équipement de mesure (le lecteur de carte et le PC).

L'élimination du bruit se fait en calculant une valeur moyenne extraite à partir d'échantillons multiples. Par conséquent, il est important d'effectuer chaque test plusieurs fois et d'utiliser une base de calculs statistiques pour filtrer les résultats. D'autre part, il est nécessaire d'exécuter plusieurs fois l'opération à mesurer sur une durée minimale (par défaut égale à 1 seconde) en espérant obtenir des résultats précis.

Nous ne considérerons pas des caractéristiques telles que celles liées aux entrées/sorties et à la consommation d'énergie car leur mesure génère les problèmes suivants :

- pour une carte donnée, l'utilisation de lecteurs différents peut engendrer des débits d'E/S différents,

- chaque partie d'une commande *APDU* est gérée différemment selon le lecteur de cartes. Les 5 premiers octets sont lus d'abord ; les données suivantes peuvent être transmises de différentes manières : avec ou sans un accusé de réception pour chaque octet, avec ou sans délai avant d'envoyer le SW (*Status Word*).

- le pilote de la carte utilisé par le PC induit plus de délai sur les mesures que le lecteur de carte lui-même.

3.5 Framework général de mesure de performances

Nous décrivons ici la méthodologie utilisée pour mesurer le temps d'exécution des opérations élémentaires *JavaCard*.

Le *framework* a été conçu pour réaliser les objectifs du projet *MESURE*, et est composé de plusieurs modules qui sont décrits ici (voir Figure 3.2). L'objectif de la

première étape est de trouver les paramètres optimaux à utiliser pour effectuer correctement les tests. Ceux-ci couvrent les opérations de la VM et les méthodes API. Les résultats obtenus sont filtrés en éliminant les mesures non pertinentes.

La mesure des temps d'exécution est réalisée à l'envoi des commandes *APDU* du PC à la carte via le lecteur. Chaque test (*run*) est exécuté Y fois pour assurer la fiabilité des temps d'exécution collectés, et au sein de la méthode *run*() une boucle est exécutée L fois. L est codé sur l'octet P_2 des commandes *APDUs* envoyées à l'application tournant sur la carte. La taille de la boucle sur la carte est $L=P_2^2$.

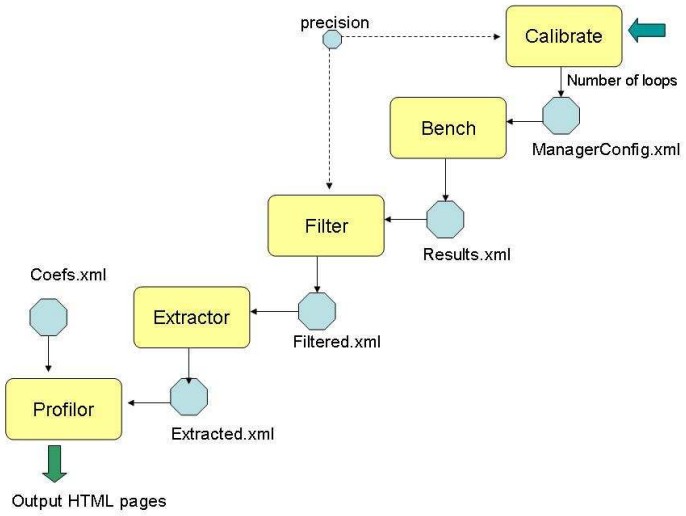


Figure 3.2. Les modules du framework de mesure

3.5.1 Le module *Calibrate*

Mesurer le temps d'exécution des différents byte-codes et entrées API peut s'avérer laborieux. D'un autre côté, il est nécessaire d'être suffisamment précis dans la mesure du temps. Ce module *Calibrate* calcule des paramètres tels que le nombre de boucles selon une précision donnée. Pour des mesures fiables, nous comparons la valeur de la mesure avec l'écart-type. Pour cela, le module aura en entrée le rapport de la mesure moyenne et de l'écart-type ainsi qu'une valeur minimale, égale à 1 seconde, correspondant au temps des mesures par défaut. Avec ces valeurs d'entrée, des tests sont effectuées avec différentes tailles de boucle pour approcher la valeur idéale.

3.5.2 Le module *Bench*

Le terminal (PC) charge les applets sur la carte, gère le lecteur de cartes et collecte les résultats en envoyant des commandes *APDUs* à la carte et en recevant les réponses correspondantes. Comme il n'existe pas de *timer* sur la carte, le code qui effectue la mesure est exécuté sur la carte mais est mesuré sur le PC avec l'horloge de ce dernier.

Pour un nombre de cycles défini par le module *Calibrate*, le module *Bench* effectue les mesures en calculant le temps d'exécution moyen pour :

- les byte-codes de la VM
- les méthodes de l'API
- les mécanismes JCRE (tels que le mécanisme de transactions).

3.5.3 Le module *Filter*

Des erreurs expérimentales peuvent générer du bruit dans les mesures brutes. Ce bruit provoque de l'imprécision dans les valeurs mesurées, rendant difficile l'interprétation des résultats. Dans le contexte des cartes à puce, le bruit est généré lors du parcours dans les deux sens de la plate-forme, du lecteur de cartes et du terminal (voir Figure 3.3).

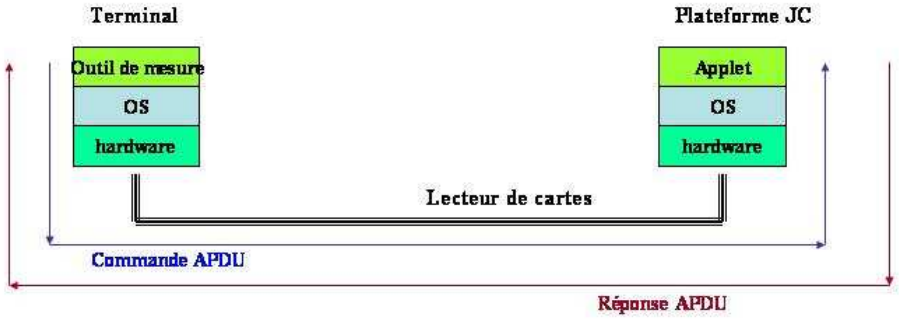


Figure 3.3. Eléments parcourus lors du lancement de la mesure

De plus, comme les mesures respectent une distribution normale gaussienne (voir Figure 3.4), un intervalle de confiance [μ-δ, μ+δ] à l'intérieur duquel le niveau de confiance de 1-*a* est utilisé pour éliminer les mesures qui sont en dehors de l'intervalle de confiance, *a* étant la précision temporelle.

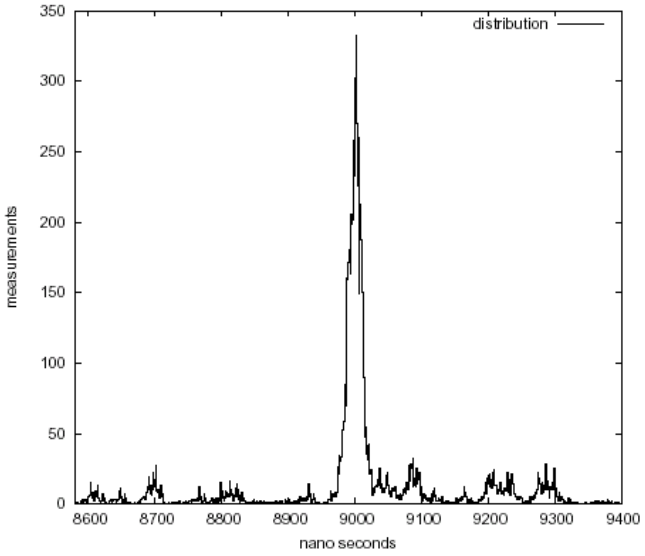


Figure 3.4. La distribution de 1000 temps d'exécution mesurés

3.5.4 Le module *Extractor*

Le module *Extractor* opère sur des byte-codes uniquement. Il sert à isoler le temps d'exécution des opérations visées parmi une masse de mesures brutes. Pour minimiser l'effet de ces interférences, nous avons besoin d'isoler le temps d'exécution de ces opérations tout en s'assurant que le temps d'exécution à mesurer est suffisamment important.

Pour isoler le temps d'exécution, nous avons besoin de calculer le temps d'exécution isolé de n'importe quel byte-code dont dépend le byte-code qui nous intéresse. Par exemple, intéressons-nous au byte-code *sadd* ; les byte-codes qui doivent s'exécuter avant *sadd* sont ceux qui doivent charger les opérandes de *sadd* dans la pile, tels que deux *sspush*. Ainsi, nous soustrayons du temps total mesuré pour un byte-code *s* le temps d'exécution de la boucle à vide ainsi que le temps des byte-codes auxiliaires dont dépend le byte-code *s*, pour déduire le temps d'exécution de *s*.

Comme présenté dans la figure 3.5, le test réel est effectué dans la méthode *run*() pour s'assurer que la pile est vidée après chaque invocation, garantissant ainsi la disponibilité de la mémoire.

Applet du framework	Cas de test

process() {
i=0
while i<= L
do
{
run()
i = i + 1
}
}
	run() {
	op_0
	op_1
	.
	.
	.
	op_{n-1}
	op_n
	}

Figure 3.5. Test du byte-code op_0

L représente la taille de la boucle, op_n représente l'opération en question et op_i (i=0, n-1) représentent les byte-codes auxiliaires nécessaires à l'exécution de op_n.

Pour calculer le temps d'exécution moyen de op_n, nous avons besoin de résoudre les équations suivantes :

Où :

¬ $M(op_i)$ est le temps d'exécution isolé de l'opération op_i.

¬ $m_L(op_i)$ est le temps d'exécution global de l'opération op_i étant donné une boucle de taille *L*, incluant les interférences dues aux opérations exécutées sur le terminal ou bien sur la plate-forme. Ces opérations correspondent aux opérations auxiliaires et à celles qui sont spécifiques à l'OS ou la JVM. La moyenne est calculée à travers un nombre significatif de tests.

EmptyLoop : représente l'exécution d'un cas de test où la méthode *run*() ne fait rien.

La formule présentée ci-dessus signifie qu'avant de calculer ¬ $M(op_n)$, nous devons calculer ¬ $M(op_i)$ pour i=0, n-1. Le système d'équations peut être résolu comme un ensemble d'opérations qui ne dépend pas d'autres opérations.

3.5.5 Le module *Profiler*

Pour définir des références de performance, notre outil de mesure fournit des mesures adaptées à un domaine d'application parmi les suivants :

- domaine bancaire,
- domaine transport,
- domaine d'identité.

Une machine virtuelle *JavaCard* est instrumentée pour compter les différentes opérations exécutées au lancement d'un script pour une application donnée. Plus précisément, la machine virtuelle est simulée sur un PC. L'instrumentation est une méthode simple à implémenter en comparaison avec les méthodes basées sur l'analyse statique du code, et peut atteindre un niveau de précision élevé. Elle nécessite une bonne connaissance des applications. Le simulateur donne des informations utiles tels que :

- pour les méthodes API
  - les types et les valeurs des paramètres de méthodes,
  - la taille des vecteurs passés en paramètre.

- pour les byte-codes
  - le type et la durée des vecteurs pour des byte-codes qui manipulent des tableaux,
  - l'état de la transaction lors de l'appel du byte-code.

Ainsi à partir des données de la machine virtuelle instrumentée, nous attribuons pour chaque domaine d'applications un nombre représentant la performance d'un nombre représentatif d'applets du domaine sur la carte testée. Chacun de ces nombres sera utilisé pour calculer une note globale et pour pondérer un domaine d'applications. Ces poids sont calculés en fonction du nombre d'occurrences d'une caractéristique *JavaCard* durant une utilisation normale d'applets standard pour un domaine donné. Par exemple, si l'on désire tester une carte du domaine transport, on utilise les valeurs statistiques collectées à partir d'un ensemble d'applets représentatives du domaine pour évaluer l'impact de chaque caractéristique mesurée sur la carte.

Considérons la mesure de la caractéristique f sur la carte c pour un domaine d'applications d. Pour un ensemble de n_M mesures extraites $M^1_{c,f}$, …, $M^{nM}_{c,f}$ considérées comme significatives pour la caractéristique f, on peut déterminer une

moyenne $\neg M_{c,f}$ modélisant la performance de *f*. Etant donné n_C cartes pour lesquelles *f* a été mesuré, il est nécessaire de déterminer le temps d'exécution moyen R_f qui servira de base de référence pour tous les tests. Ainsi la note $N_{c,f}$ d'une carte *c* pour une caractéristique *f* est la relation entre R_f et $\neg M_{c,f}$:

$$N_{c,f} = \frac{R_f}{M_{c,f}}$$

Cependant, cette note n'est pas pondérée. Pour chaque paire (caractéristique *f* , domaine *d*), on associe un coefficient $\alpha_{f,d}$ qui modélise l'importance de *f* dans *d*. Plus une caractéristique est utilisée au sein d'applications typiques du domaine, plus le coefficient est grand :

$$\alpha_{f,d} = \frac{\beta_{f,d}}{\sum_{i=1}^{n_F} \beta_{i,d}}$$

Où :

- $\beta_{f,d}$ est le nombre total d'occurrences de la caractéristique *f* des applications représentatives du domaine *d.*
- n_F est le nombre total de caractéristiques impliqué dans le test.

Ainsi le coefficient $\alpha_{f,d}$ représente le rapport d'occurrences de *f* parmi toutes les caractéristiques. Étant donné une caractéristique *f*, une carte *c* et un domaine *d*, la note pondérée $W_{c,f,d}$ est calculée comme suit :

$$W_{c,f,d} = N_{c,f} \times \alpha_{f,d}$$

La note globale $P_{c,d}$ pour une carte *c* d'un domaine *d* est la somme de toutes les notes de la carte ainsi pondérées. Une note indépendante du domaine général pour une carte est calculée comme la moyenne des notes des différents domaines.

3.6 Travaux existants

Il n'existe actuellement aucun outil de mesure de performances qui puisse être utilisé pour démontrer l'efficacité d'une machine virtuelle *JavaCard* et qui puisse fournir des critères de comparaison des performances des plates-formes *JavaCard*.

En effet, s'il existe de nombreux benchmarks développés pour mesurer les performances des machines virtuelles *Java*, très peu de travaux se sont intéressés aux cartes à puce.

La première initiative a été celle de Castellà-Roca et *al*. dans [CDHP00] qui étudient les performances des cartes de paiement *JavaCard* sans PKI. Même s'ils considèrent des plates-formes de différents fabricants, la couverture des mesures faites est insuffisante.

Un travail plus récent et plus complet est celui de Erdmann dans [E04] qui distingue différents domaines d'applications et qui mesure des opérations d'entrée/sortie, des fonctions cryptographiques, de consommation d'énergie, de

JCRE. L'outil n'est malheureusement pas disponible. D'autre part, la seule plate-forme considérée est celle de *Infineon Technologies*.

Le travail de Fischer dans [F06] compare les performances d'une applet *JavaCard* avec celles fournies par une application native.

Un autre travail intéressant est celui mené par le groupe IBM *BlueZ* secure systems [R05] pour tester les performances de l'algorithme de cryptage *DES*, des opérations de lecture/écriture dans les mémoires RAM et EEPROM.

Chaumette et al. [CGKSV04] montrent les performances des grilles de plates-formes *JavaCard* en fonction de l'évolutivité des grilles et de différentes cartes.

3.7 Conclusion

Dans cette partie, nous nous sommes intéressée à la mesure du temps d'exécution correspondant à la métrique temporelle utilisée ici pour mesurer et comparer les performances des plates-formes embarquées *JavaCard*. Ces mesures ont concerné les byte-codes de la machine virtuelle, les méthodes de l'API ainsi que des mécanismes du JCRE.

Nous avons développé une méthodologie qui montre comment faire des mesures précises et extraire celles qui sont pertinentes. L'outil de mesure est accessible gratuitement en ligne[13] depuis mars 2008. Les plates-formes considérées ici peuvent provenir de fabricants différents mais doivent embarquer des *JVM 2.2*.

Depuis Mars 2008, les spécifications de la *JavaCard 3.0* sont publiées par SUN MicroSystem [JC08]. Deux éditions sont proposées :

- une édition connectée : orientée Web, qui fait de la carte un serveur d'application ;
- une édition classique : qui améliore la version *JavaCard 2.2*, ce qui signifie que la majorité des caractéristiques du projet *MESURE* pourront être réutilisées avec cette nouvelle édition.

Par rapport à l'objectif initial du projet, datant de mai 2006, qui était d'offrir un outil de mesure de performance pour les plates-formes JavaCard public et ouvert, nous pouvons conclure que cet objectif est atteint. Cet outil de mesure de performances est le premier outil libre, accessible à tous dans une communauté où les logiciels dédiés à la carte à puce sont encore propriétaires. Ce projet pré-compétitif a reçu un écho favorable auprès de la communauté de la carte à puce, à travers le prix Isabelle Attali qui lui a été décerné durant la conférence e-Smart 2007. Depuis la mise en ligne de l'outil en Janvier 2008, les mails que nous recevons montrent bien que notre outil est utilisé par les développeurs de la carte à puce. Je

[13] *http://mesure.gforge.inria.fr/Fr/Index*

pense en particulier à SUN MicroSystems qui propose les spécifications JavaCard, mais aussi à d'autres, tels que : Athena Card Systems ou Viaccess.

Même si la plus part des cartes à puce utilisent le standard JavaCard, il n'en demeure pas moins qu'il existe des cartes dotées de systèmes d'exploitation propriétaires ou de machines virtuelles (.Net pour carte à puce, l'édition connectée de JavaCard 3.0) qui ne peuvent être mesurées avec notre outil.

L'autre aspect important non traité ici est la sécurité. En effet, en faisant abstraction du facteur sécurité, une plate-forme qui offre un temps d'exécution faible pour une opération mesurée n'est pas forcément plus efficace qu'une autre plate-forme qui offre un temps d'exécution plus grand pour la même opération, si cette dernière intègre un calcul cryptographique par exemple. Par conséquent, une suite logique de notre travail consisterait à voir comment mesurer la sécurité dans les plates-formes JavaCard.

Jusqu'à présent, nous avons ciblé des plates-formes JavaCard avec contact, c'est-à-dire, reliées au terminal via un lecteur. Une extension possible de notre travail serait de voir comment adapter l'outil pour mesurer des plates-formes JavaCard sans contact en tenant compte de leurs caractéristiques spécifiques.

Conclusion et Perspectives

4.1 Bilans d'activité

4.1.1 Bilan scientifique

Dans ce mémoire, nous avons traité trois aspects :

1- ***la gestion des transactions temps réel***

Tous les protocoles proposés ici ont eu pour objectif de respecter les contraintes de temps des transactions. Pour cela, différents mécanismes ont été utilisés :

- augmenter le degré de concurrence des transactions en tolérant la manipulation de données imprécises, ou bien en dupliquant les transactions conflictuelles, ou encore en évitant les conflits en plaçant les transactions conflictuelles dans des phases différentes. Pour cela, nous avons introduit la notion d'ε-données pour autoriser plus de concurrence entre transactions conflictuelles et ainsi augmenter leur chance de se terminer dans les temps, tout en contrôlant l'intégrité des données. Des concepts existants tel que la notion d'ordre causal par phases, ont été adaptés au contexte des SGBDs temps réel.

- éliminer les transactions qui ne respecteraient pas leur échéance ; en d'autres termes, éliminer celles qui provoqueraient de la surcharge temporelle. Dans ce cas, nous avons utilisé la notion de stabilisation déjà utilisée dans les systèmes temps réel [CDKM02] pour maintenir dans le système uniquement les transactions susceptibles de se terminer dans les temps.

2- ***la conception d'applications temps réel par assemblage de composants***

Dans cette problématique, nous avons eu pour objectif de vérifier :

- d'une part, le respect des contraintes de temps (temps d'exécution par exemple) pour chaque paire de composants à assembler en appliquant le principe de compatibilité et substituabilité temporelles, après vérification de l'assemblage du point de vue structurel et comportemental. Pour cela, nous avons eu besoin de définir un système de types basé sur la notion de sous-typage, qui permet une composition sûre et une réutilisation facile du logiciel.

- Et d'autre part, les contraintes de temps globales (échéance de bout en bout par exemple) pour l'ensemble de l'assemblage de composants en analysant, à l'aide des automates temporisés, l'ordonnançabilité d'une configuration de tâches obtenue à partir de cet assemblage.

Pour réaliser cet assemblage, un modèle de composants temps réel a été proposé, et sa faisabilité a été vérifiée en le traduisant en RTSJ (Java temps réel) et en TinyOS.

3- ***la mesure des performances des plates-formes JavaCard*** a eu pour objectif d'utiliser une méthodologie pour mesurer le temps d'exécution des opérations qui caractérisent l'efficacité et la vivacité de chaque plate-forme *JavaCard*.

Bien que portant sur des domaines assez différents, ces travaux traitent d'une problématique commune autour de la mesure et la maîtrise du temps. Maîtriser le temps d'un système s'est traduit ici par:

- le respect de l'échéance d'entités d'exécution (transaction, tâche, composant) de ce système ;

- la minimisation du temps de réponse, par exemple lors de l'assemblage des composants (le service offert doit offrir un temps d'exécution plus petit que celui du service requis) ;

- le contrôle du dépassement d'échéance de manière à le maintenir acceptable ; c'est le cas des transactions de type « *firm* » qui tolèrent un dépassement d'échéance d'une valeur Δ.

La mesure du temps dans un système a concerné :

- la mesure du temps d'exécution des opérations pouvant caractériser le système en essayant d'éliminer *l'overhead* dû à l'interaction avec le système (c'est l'étude du chapitre 3),

- la mesure du temps de réponse global impliquant plusieurs entités (composants dans notre cas) afin de le comparer avec l'échéance de bout en bout par exemple (c'est l'analyse d'ordonnançabilité présentée au chapitre 2).

4.1.2 Bilan quantitatif

Le bilan de ce projet peut être évalué suivant plusieurs critères : publications, formation, projets industriels.

4.1.2.1 Bilan des publications pendant l'HDR

Les publications sont une manière d'évaluer la qualité d'un travail de recherche. Les travaux de la HDR ont généré un certain nombre de publications dont le

tableau ci-dessous donne un aperçu global. Plus de détails sont donnés dans l'annexe B.

Type de publication	Revue internationale	Revue française	Conférence internationale	Conférence nationale
Quantité	**4**	**2**	**27**	**3**

4.1.2.2 Bilan en terme de formation

En terme de formation, ce projet de HDR a été le support de trois thèses de Doctorat et a permis de former des étudiants de DEA/Master ainsi que des élèves-ingénieurs dont des encadrements à distance d'étudiants de l'université de Tizi-Ouzou (Kabylie).

Un doctorant (Jean-Paul Etienne, boursier de l'AUF) est sur le point de soutenir sa thèse sur l'utilisation de l'approche *composant* dans la conception d'applications temps réel. Un autre doctorant (Julien Cordry) travaille sur la mesure des performances des plates-formes JavaCard et sa soutenance est prévue dans un an. Tandis qu'une troisième thèse (Hai Binh LE) a commencé en décembre 2007 sur l'interopérabilité des systèmes de gestion d'identités numériques.

Certains des étudiants en DEA/Master et élèves-ingénieurs poursuivent des recherches en thèse, tandis que d'autres ont entamé une carrière professionnelle. En plus du tableau suivant qui donne un bilan quantitatif, le détail de mes activités d'encadrement est donné en annexe A.

Type de diplôme	Thèse de Doctorat	DEA/Master M2	Ingénieur CNAM	Ingénieur univ. Tizi Ouzou
Quantité	**3**	**7**	**1**	**2**

4.1.2.3 Bilan des coopérations industrielles

Le travail mené durant le projet de HDR peut être qualifié d'académique dans le sens où il y a eu peu de coopérations avec le monde industriel.

Au moment où j'avais rejoint l'équipe de SGBD temps réel, celle-ci avait une année d'existence. Après les premières années qui m'ont permis de maîtriser la problématique et de publier dans le domaine, j'ai participé au montage d'un projet *ACI Jeunes Chercheurs*. Ce projet ayant été accepté au moment où je quittais Le Havre, je n'ai pas pu y contribuer énormément par la suite.

A mon arrivée au CNAM, j'ai travaillé avec Claude Kaiser, puis avec Gérard Florin sur de nouvelles problématiques. Mais comme tous deux étaient à quelques années de partir en retraite, l'opportunité d'initier un nouveau projet en coopération avec des industriels ne s'était pas présentée. Il me fallut donc un peu

de temps pour consolider mes nouveaux contacts et maîtriser les nouvelles thématiques sur lesquelles je commençai à travailler.

L'arrivée de Pierre Paradinas au CNAM pour diriger la nouvelle chaire de « Systèmes Embarqués et Mobiles » mais aussi mon intérêt grandissant pour les systèmes embarqués ont concrétisé l'émergence de travaux de recherche sur la carte à puce au sein du CNAM. Une collaboration avec Pierre Paradinas a ainsi vu le jour à travers le projet *MESURE* et le co-encadrement de la thèse de Julien Cordry. Le projet *MESURE* constitue ma première réelle expérience dans un projet en liaison avec l'industrie. La réalisation de ce projet a été menée avec l'entreprise Trusted Labs et l'équipe POPS de l'INRIA de Lille.

Depuis un an, je participe au projet FC² sur la *fédération de cercles de confiance*[14] pour la gestion d'identités numériques, avec un grand nombre de partenaires industriels. Je co-encadre la thèse de doctorat de Hai Binh LE avec le professeur Pierre Paradinas dans le cadre de ce projet.

Mon expérience sur ces deux projets m'a donné l'assurance et la maîtrise nécessaires pour susciter l'intérêt des industriels pour une coopération sur des projets reposant sur mes activités de recherche et mettre à profit des sources de financement telles que les bourses CIFRE.

4.2 Perspectives

La prolifération des systèmes embarqués, et plus précisément des systèmes informatiques minimalistes, est aujourd'hui une réalité qui guide autant les investissements industriels que les recherches scientifiques. La carte à puce qui assure désormais le rôle de représentant électronique de l'individu est l'exemple type de tels systèmes. Les *tags* RFID ou les réseaux de capteurs sont d'autres exemples de systèmes embarqués qui vont être de plus en plus déployés ; ils constituent ce que l'on appelle déjà *l'informatique ubiquitaire*.

Dans les travaux présentés ici, la contrainte principale dont il fallait tenir compte a été la contrainte temporelle. Or, dans les systèmes embarqués, les contraintes peuvent être d'une autre nature comme la capacité mémoire limitée, la puissance de calcul réduite ou encore la consommation d'énergie ; autant de contraintes dont il est essentiel de tenir compte.

Après cette rétrospective de tous mes travaux de recherche, qui ont reçu une reconnaissance de la communauté scientifique, je peux tirer quelques conclusions générales sur les domaines de recherche que j'ai abordés, sur les problématiques ouvertes, et sur celles qui restent non traitées.

[14] *http://www.fc2consortium.org/projet.html*

Ma réflexion aboutit à une volonté d'élargissement de mon domaine de recherche à celui de *l'embarqué*, tout en mettant à profit mon expérience dans des thématiques de recherche variées.

J'axerai donc mon étude des systèmes contraints sur les points suivants :

1. La gestion des données dans l'embarqué

Ma large culture dans les SGBDs temps réel me permet d'affirmer que de nombreux problèmes traités dans les SGBDs temps réel peuvent être revus ou adaptés au contexte des SGBDs embarqués. En effet, des méthodes de similarité de données peuvent être appliquées, comme dans TinyDB par exemple, pour éviter de lire toutes les données provenant de capteurs.

De manière plus générale, les problématiques de caractérisation des données (durée de validité), de qualité de service des données (imprécision), de caractérisation des requêtes (obligatoire/optionnelle, contrainte de temps, contrainte mémoire, etc.), de distribution des données ont déjà fait l'objet de travaux dans les SGBDs temps réel et peuvent s'appliquer au domaine de l'embarqué.

2. Les méthodes de conception des systèmes embarqués

Après mon travail sur l'utilisation de l'approche *composant* dans un contexte temps réel, je suis persuadée que cette approche a encore de l'avenir, notamment dans les systèmes embarqués. Et la problématique aujourd'hui est de proposer un modèle de composants qui prenne en compte les contraintes de ressources (mémoire, énergie, puissance de calcul, etc.). L'utilisation de l'approche *composant* dans le domaine du ferroviaire à travers le projet REVE ou encore dans le domaine de l'automobile [SE02] me conforte dans mon choix.

C'est pourquoi nous projetons, dans la thèse de Hai Binh LE, d'utiliser l'approche *composant* pour la conception d'applications dédiées à l'embarqué (carte à puce en particulier) en veillant au respect des contraintes de ressources. Ce travail de thèse a démarré il y a un an; il est financé par le projet FC² dont l'objectif est la conception d'une plate-forme complète permettant le développement sécurisé de nouveaux services électroniques basés sur la gestion transparente et fédérée d'identités.

3. La mesure des performances des systèmes embarqués

Mon expérience dans la mesure des performances des plates-formes *JavaCard* a été très encourageante de par l'écho favorable de l'industrie pour notre initiative mais aussi à travers les contacts industriels et académiques que j'ai tissés durant les activités menées dans le cadre du projet *MESURE*. Travailler sur la carte à puce a été un challenge passionnant. Car à l'heure où la carte à puce se pose de plus en plus comme le représentant électronique du citoyen (passeport électronique, carte identité, carte santé, carte de paiement, carte d'accès, etc.), l'étudier a été très motivant, attractif et enrichissant.

Cette expérience m'amène à faire le constat suivant :

Dans le domaine des systèmes embarqués, la mesure des performances et l'évaluation des caractéristiques d'une plate-forme sont un sujet qui se révèle complexe, pour différentes raisons :

-la complexité grandissante des plates-formes,

-les caractéristiques des plates-formes ne sont pas uniformes, chaque plate-forme possédant ses propres contraintes de ressources dont il faut tenir compte (taille mémoire, consommation d'énergie, etc.)

-les technologies utilisées pour le développement sur ces plates-formes sont nouvelles.

Partant de ce constat, deux nouvelles perspectives de recherche constituent une suite logique à mes travaux sur la mesure des performances des plates-formes *JavaCard* :

- D'une part, la mesure de la sécurité, aspect dont nous avons volontairement fait abstraction durant nos travaux sur le projet *MESURE*, constitue une problématique importante que nous pouvons étudier aujourd'hui pour compléter notre travail à travers une recherche plus générale visant la mesure des performances et de canaux cachés dans les systèmes embarqués (cartes avec ou sans contact, RFID[15], réseaux de capteurs). Cette démarche peut s'inscrire principalement en amont du processus de mise sur le marché et permettra de « *mesurer* » la sécurité et de détecter et corriger au plus tôt les éventuelles failles. Cette approche est complémentaire aux processus classiques d'établissement de la confiance (i.e. l'évaluation/certification) lorsque ceux-ci existent car certaines plates-formes comme les capteurs de terrain ne sont pas encore couramment pris en compte par les processus d'évaluation et de certification sécuritaires. Des chercheurs du laboratoire XLIM, compétents dans le domaine de la sécurité et des cartes à puce, sont intéressés pour entamer ensemble ce travail de recherche.

- D'autre part, la publication récente de la spécification *JavaCard 3.0* par SUN, qui intègre une édition orientée serveur Web (*Connected Edition*) avec les premières implémentations qui commencent à voir le jour, et un projet comme *JITS*[16], qui fournit en particulier un serveur Web pour les réseaux de capteurs, sont quelques raisons qui nous laissent penser que le moment est propice pour évaluer et comparer les performances et les caractéristiques des serveurs Web embarqués. Les plates-formes ubiquitaires qui pourront être visées ne cibleront pas uniquement les cartes à puce mais d'autres exemples de systèmes ubiquitaires tels que les réseaux de capteurs ou les RFID, car ces plates-formes ne disposent pas pour la plupart de bancs de mesure des performances. Une meilleure

[15] *RFID = Radio Frequency IDentification*

[16] *Projet de l'équipe POPS (Lille)*

compréhension des comportements d'une plateforme donnée en termes de performances permettra aux industriels d'optimiser les performances de ces plates-formes pour des environnements spécifiques. Ces travaux pourraient être menés avec l'équipe POPS qui possède déjà une première expérience dans le domaine du Web embarqué à travers le projet *JITS*.

Je projette donc d'orienter mes activités de recherche vers les systèmes embarqués pouvant être contraints par des ressources autres que le temps, tout en profitant de mon expérience acquise dans le domaine du temps réel.

Bien que les principales perspectives présentées ici tournent autour des cartes à puce, je compte étendre mes travaux de recherche à d'autres types de systèmes. Mon implication actuelle dans le montage d'une bourse CIFRE avec AIRBUS pour la détermination du WCET (Worst Case Execution Time) pour les systèmes embarqués de l'avionique, basés sur de nouvelles architectures et technologies de processeurs, participe de cette ouverture.

Bibliographie

[AD94] : Rajeev Alur and David L. Dill. A theory of timed automata. Theoretical Computer Science, 126(2):183-235, 1994.

[AG88] Abbot, R. and Garcia-Molina, H. "Scheduling Real-Time Transactions : A Performance Evaluation". In Proc. of the 14th Intl. Conf. on Very Large Data Bases (VLDB), pages 1–12, Los Angeles, California, 1988.

[AHS06] Amirijoo M., J. Hansson et S. Hyuk Son, "Specification and Management of QoS in Real-Time Databases Supporting Imprecise Computations", *IEEE Transactions on Computers*, Vol. 55, No. 3, pp. 304-319, March 2006.

[AN96] Amanton L. and Naimi M. "Total and causal ordering implementation in distributed computations". In 8th Int. Conf. of Computing and Information (ICCI96), volume 2, 1996.

[AN98] Amanton,L. and Naimi, M. "Algorithme d'ordonnancement causal dynamique par phases avec initiateurs multiples ». In Conf. Int. sur les Nouvelles Technologies de la REpartition (NOTERE98), Montréal, Canada, 1998.

[AN99] Amanton, L. and Naimi, M. "A Causal-Phase Ordering Protocol with Multi-Initiators for Overlapped Broadcasts". In 24th IEEE Annual Conference on Local Computer Networks (LCN99), Boston, Massachusetts, USA, 1999.

[B95] Braoudakis S. "Concurrency Control Protocols for Real-Time Databases". PhD thesis, Boston University, USA, 1995.

[B05] Berteau L., « entretien de décembre 2005 » avec Laurent Berteau, ingénieur systèmes et réseaux chez Euro Information Développements (filiale du Crédit Mutuel), professeur du cours Systèmes et Applications Réparties au CNAM Strasbourg.

[BB95] Bestavros, A. and Braoudakis, S. "Value-cognizant Speculative Concurrency Control". In Proc. of the International Conference on Very Large Databases (VLDB), 1995.

[BBG+00]: Gregory Bollella, Benjamin Brosgol, James Gosling, Peter Dibble, Steve Furr, and Mark Turnbull. The Real-Time Specification for Java. Addison Wesley Longman, 1st edition, January 2000.

[BBPV2000] : C. Bobineau, L. Bouganim, P. Pucheral, P. Valduriez, « PicoDBMS : scaling down data base techniques for the smart card », 26th intern. Conf. on Very Large Databases, Egypt 2000.

[BBS 2006] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in bip. In SEFM '06 : Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[BCE03] Bertino E., Catarci T., El magarmid, A. K., and Hacid M.-S. "Quality of Service Specification in Video Databases". IEEE Multimedia, 10(4): 71-81, 2003.

[BHG87] Bernstein P., V. Hadzilacos et N. Goodman, "Concurrency Control and Recovery in Database Systems", Addison Wesley, 1987.

[BLA98] Buttazo G. C., G. Lipari et L. Abeni, "Elastic Task Model for Adaptive Rate Control", *in Proc. of IEEE Real-Time Systems Symposium*, Madrid, dec. 1998.

[BMR96] Baldoni, R., Mostefaoui, A., and Raynal, M. "Causal multicast in unreliable networks for multimedia applications". In 15th IEEE International Conference on Computers, 1996.

[BN96] Bestavros A. and S. Nagy,"Value-cognizant Admission Control for RTDB systems", Proc. of the 17th Real-Time Systems Symp., pp. 230-239, IEEE Computer Society, dec. 1996.

[BN03] : Edward G. Benowitz and Albert F. Niessner. A Patterns Catalog for RTSJ Software Designs. In OTM Workshops, pages 497-507, 2003.

[BSS91] Birman K., Shiper A., and Stephenson P. "Light weight causal and atomic group multicast". ACM Transactions on Computer Systems, 9(3):272-314, 1991.

[BSS95] Buttazo G., M. Spuri and F. Sensini, "Value vs. Deadline Scheduling in Overload Conditions", *Proc. of the 15th R-T Systems Symp.*, Italy, pp. 90-99, dec. 1995.

[C88]: L. Cardelli. Structural Subtyping and the Notion of Power Type. In Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, pages 70–79, San Diego, California, 1988.

[Carney et al. 02] Carney D., U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, et S. Zdonik. "Monitoring streams: a new class of data management applications". In *Proc. 28th International Conference on Very Large Data Bases*, pages 215-226. Morgan Kaufmann Publishers, September 2002.

[CDHP00] Jordi Castellà-Roca,Josep Domingo-Ferrer, Jordi Herrera-Joancomartî and Jordi Planes, " A Performance Comparison of JavaCards for Micropayment Implementation", CARDIS,19-38, 2000.

[CDKM02] Cottet F., Delacroix J., Kaiser C. and Mammeri Z., "Scheduling in Real-Time Systems", Wiley Ed., 261 pages, 2002.

[CG96] Chen, Y. and Gruenwald, L. "Effects of Deadline Propagationon Scheduling Nested Transactions in Distributed Real-Time Database Systems". Information Systems, 21(1), 1996.

[CGKSV04] Chaumette S., Grange P., Karray A., Sauveron D. and Vignéras P., « Secure distributed computing on a JavaCard Grid », LaBRI, Université Bordeaux 1, 1331-04, 2004.

[CGPK06]: Juan A. Colmenares, Shruti Gorappa, Mark Panahi, and Raymond Klefstad. A Component Framework for Real-time Java. 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06), 2006.

[D98] Emmanuel Durand. Description et vérification d'architecture temps réel : CLARA et les réseaux de Pétri temporels. PhD thesis, École Centrale de Nantes, Nantes, 1998.

[D05] Anne-Marie Déplanche. Rapport de synthèse - Action Spécifique CNRS 195 - Composants et Architectures Temps Réel. Technical Report RI2005_1, IRCCyN, March 2005.

[DF05] Anne-Marie Déplanche and Sébastien Faucou. Les langages de description d'architecture pour le temps réel. In actes École d'Été Temps Réel (ETR'05), 2005.

[DF06] : Anne-Marie Déplanche and Sébastien Faucou. Description d'architecture pour le temps réel : l'approche AADL. Chapitre 10 du Traité IC2 Systèmes Temps Réel 1 : Techniques de Description et de Vérification - Théorie et Outils, Hermès Science, 2006.

[DMKVB96] Datta A., Mukherjee S., Konana P., Viguier and Bajaj A., "Multi-calss transaction scheduling and overload management in firm real-time database systems", Information Systems, vol. 21, n° 1, pages 29-54, 1996.

[DMS99] Duvallet C., Z. Mammeri & B. Sadeg, "Analyse des protocoles de contrôle de concurrence et des propriétés ACID dans les SGBD temps réel", Revue TSI, 1999.

[E04] Erdmann M., "Benchmarking von JavaCard", LudwigMaximilians-universitât Mûnchen, Institut fûr Informatik, May 2004.

[E09] : Jean-Paul Etienne, « Utilisation de l'approche composant pour la conception et la réalisation d'applications temps réel », Thèse de Doctorat, CNAM, soutenance prévue en Janvier 2009.

[EGLT76] Eswaran K. P., J. N. Gray, R. A. Lorie et I. L. Traiger, "The notion of consistency and predicate locks in a database system", CACM, 9(11), pp. 624-633, 1976.

[F04]: Martin Fowler. Module assembly. IEEE Software, 21(2):65–67, 2004.

[F06]:Fischer M., "Vergleich von Java und Native-Chipkarten Toolchains, Benchmarking, Messumgebung", LudwigMaximilians- universitât Mûnchen, Institut fûr Informatik, 2006.

[FB et al. 06] Helmut Fennel, Stephan Bunzel, and al. Achievements and exploitation of the autosar development partnership, 2006.

[FMPY03] : Elena Fersman, Leonid Mokrushin1, Paul Pettersson, and Wang Yi. Schedulability Analysis using Two Clocks. In TACAS '03: Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, volume 2619/2003, pages 224-239. Springer Berlin / Heidelberg, 2003.

[FPY02] : Elena Fersman, Paul Pettersson, and Wang Yi. Timed Automata with Asynchronous Processes: Schedulability and Decidability. In TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 67-82, London, UK, 2002. Springer-Verlag.

[GCO01]: Thorsten Gerdsmeier and Rachel Cardell-Oliver. Analysis of Scheduling Behaviour using Generic Timed Automata. Electronic Notes in Theoretical Computer Science, 42, 2001.

[GHRS96] Gupta, R., Haritsa, J., Ramamritham, K., and Sechadri, S. "Commit Processing in Distributed Real-Time Data base Systems". In proc. of the 17th IEEE Real-Time Systems Symposium (RTSS'1996), pages 220-229, 1996.

[H85] C. A. R. Hoare. Communicating Sequential Processes. Prentice-Hall, April 1985.

[HACM04] Hans Hansson, Mikael Åkerholm, Ivica Crnkovic, and Martin Törngren. Saveccm - a component model for safety-critical real-time systems. In Euromicro

Conference, Special Session ComponentModels for Dependable Systems, Rennes, France, September 2004.
[HCL92] Haritsa J.R., Carey M.J. and Livny M., « Data access scheduling in firm real-time database systems », J. of RTS, vol.4, n° 3, p. 203-241, 1992.
[HJC93] Hong D., Johson T. and Chakravarthy S., « Real-time transactions scheduling: cost conscious approach», ACM SIGMOD. Int. Conf. on Management of Data, p. 197-206, 1993.
[HS01] Hansson J. and S. H. Son,"Real-Time Database Systems: Architecture and Techniques", K. Lam and T. Kuo (eds.), Kluwer Academic Publishers, pp. 125-140, 2001.
[HSRT91] Huang J., Stankovic J.A. and Ramamritham K., Towsley D., « On using priority inheritance in real-time database », 12th RTS Symposium, p. 552-561, 1991.
[HSSA98] Hansson J., S. H. Son, J.A. Stankovic and S. F. Andler,"Dynamic Transaction Scheduling and Reallocation in Overloaded Real-Time Database Systems", *Proc. of the 5th Conference on Real-Time Computing Systems and Applications (RTCSA'98)*, pp. 293-302, IEEE Computer Press, 1998.
[HR00] Haritsa, J. and Ramamritham, K. "Adding PEP to Real-Time Distributed Commit Processing". In 21th Real-Time Systems Symposium (RTSS 2000), Orlando, Florida, 2000.
[HRG00] Haritsa, J., Ramamritham,K., and Gupta, R. "The PROMPT Real-Time Commit Protocol". IEEE Transactions on Parallel and Distributed Systems, 11(2): 160-181, 2000.
[IST05] ISTCompare. IST FP6 - COMPARE document D2.3-1 - framework architecture and artefact, July 2005.
[JC06] JavaCard 2.2.2 Specification (mars 2006): http://java.sun.com/javacard/specs.html
[JC08] JavaCard 3.0 Specification (mars 2008) : http://java.sun.com/javacard/3.0/
[KM96] Kuo T-W. and Mok A.K., "Real-time database - similarity semantics and ressource scheduling", *ACM SIGMOD Record*, vol. 25, n° 1, p. 18-22, 1996.
[KS95] Koren G. and D. Shasha, "Dover: An Optimal On-Line Scheduling Algorithm for Overloaded Uniprocessor Real-Time Systems", SISAM J. Comput., 24(2), pp.318-339, 1995.
[L98] Pascal Ledru. Distributed Programming in Ada With Protected Objects. Dissertation.com, December 1998.
[L08] Frédéric Loiret, Tinap : Modèle et infrastructure d'exécution orienté composant pour applications multi-tâches à contraintes temps réel souples et embarquées, Thèse de Doctorat, Université de Lille 1, Mai 2008.
[LA03]: [LA03] Kristina Lundqvist and Lars Asplund. A Ravenscar-Compliant Run-time Kernel for Safety-Critical Systems. Real-Time Systems, 24(1):29-54, 2003.
[Ld00] David J. Lilja, "Measuring Computer Performance : A Practitioner's Guide", 278 pages, Cambridge University Press, July 20, 2000.
[Lj00] Jane W. S. Liu. Real-Time Systems. Prentice Hall, April 2000.
[LL73] Liu L. & J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", Journal of the ACM, Vol. 20, n°1, pp. 46-61, 1973.
[LLH00] Lam, K.-W., Lee, V., and Hung, S.-L. "Transaction Scheduling in Distributed Real-Time Systems". Real-Time Systems, 19: 169-193, 2000.
[LPSC99] Lam K.-Y., Pang C.-L., Son S., and Cao J. "Resolving Executing-Committing Conflicts in Distributed Real-Time Database Systems". Journal of Information Systems, 42(8): 674-692, 1999.
[LSH97] Lam, K.-W., Son, S. H., and Hung, S.-L. "A Priority Ceiling Protocol with Dynamic Adjustment of Serialization Order". In Proc. of ICDE, pages 552-561, 1997.
[LY98] Lam K. and Yan, W. "On Using Similarity for Concurrency Control in Real-Time Database Lam, Systems". Journal of Systems and Software, 43(3): 223-232, 1998.
[M98] Mammeri Z., « Expression et dérivation des contraintes temporelles dans les applications temps réel », APII-JESA, vol. 32, n° 5-6, p. 609-644, 1998.
[MetaH98] Honeywell Technology Center. MetaH Users Manual - version 1.27, 1998.
[MN82] Menasce D. and Nakanishi T., « Optimistic versus pessimistic concurrency control mechanisms in database management systems », Information Systems, vol. 7, n° 1, 1982.
[MPLMS08] Michal Malohlaval, Aleš Plšek, Frédéric Loiret, Philippe Merle, Lionel Seinturier, « Introducing Distribution into a RTSJ-based Component Framework", 22nd Junior Researcher Workshop on Real-Time Computing (JRWRTC 2008), Rennes : France.
[MSZ01] Peter Müller, Christian Stich, and Christian Zeidler. Components@work : Component technology for embedded systems. In 27th International Workshop on Component-Based Software Engineering, EUROMICRO, 2001.
[MT96] Mostefaoui A. and Theel O. "Reduction of Timestamp Sizes for Causal Event Ordering". Technical Report 1062, IRISA, Rennes, France, 1996.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. IEEE Trans. Software Eng., 26(1):70-93, 2000.
[N93] Nakazato H., « Issues on synchronizing and scheduling tasks in real-time database systems », PhD thesis, University of Illinois at Urbana-Champaign, 1993.
[N03] Natkin S., "Architectures et technologies informatiques pour jouer à un million de joueurs ». Les Cahiers du Numérique, 4(2), pp.15-36, Hermes, 2003.
[NI02] C. Norström & D. Isovic. Components in real-time systems. In The 8th Int. Conf. On Real-Time Computing Systems and Applications (RTCSA'2002), 2002.
[OLKM00] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. In IEEE Computer, volume 33, pages 78–85, 2000.
[P93] Pu, C. Relaxing the Limitations of Serializable Transactions in Distributed Systems. In Proceeding of ACM SIGOPS European Workshop and Operation, 1993.
[P02] : B. C. Pierce. Types and Programming Languages. MIT Press, Februrary 2002.
[PFHV04] : F. Pizlo, J. M. Fox, D. Holmes, and J. Vitek. Real-Time Java Scoped Memory:
Design Patterns and Semantics. Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04), pages 101-110, 2004.
[PL92] Pu, C. and Leff, A. Autonomous Transaction Execution with Epsilon Sarializability. In Proceeding of 1992 RIDE Workshop on Transaction and Query Processing. IEEE Computer Society.
[PLC92] Pang H., M. Livny and M. J. Carey, "Transaction scheduling in multi-class real-time database systems", in Proc. of the R-T Systems Symp., pages 23-34, IEEE Computer Society Press, dec. 1992.
[PMS08] : Ales Plsek, Philippe Merle, Lionel Seinturier, "A Real-Time Java Component Model", ISORC'2008, Florida.
[PSRS94] Purimetla B., Sivasankaran R.M., Ramamritham K. and Stankovic J.A., « Real-time databases: issues and applications », Principles of RTS, ed. Printice Hill, 1994.
[R96] Ramamritham., « Where do time constraints come from and where do they go ? », Int. J. of Database Management, vol. 7, n° 2, p. 4-10, 1996.
[R93] Ramamritham, K. "Real-Time Databases", J. of Distributed and Parallel Databases, Vol. 1, n° 2, pp. 199, 226, 1993.
[R03] Robust open component based software architecture for configurable devices project. Technical report, 2003.
[RSD04] Ramamritham K., S. H. Son and L. C. Dipippo, "Real-time databases and data services", Real-*time systems*, pp. 179-215, *Kluwer Academic Publishers,* 2004.
[R05] Rehioui K., "JavaCard Performance Test Framework", Université de Nice, Sophia-Antipolis, IBM Research internship, september, 2005.
[RZPCK05]: K. Raman, Y. Zhang, M. Panahi, J. A. Colmenares, and R. Klefstad. Patterns and Tools for Achieving Predictability and Performance with Real-time Java. 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05), pages 247–253, 2005.
[S01] J. Stankovic. Vest : A toolset for constructing and analyzing component based embedded systems. In Springer-Verlag Lecture Notes, pages 390–402, 2001.
[S04] Sadeg B., "Contribution à la gestion des transactions dans les SGBD temps réel ", Mémoire de HDR : http://scott.univ-lehavre.fr/~sadeg/hdr/hdr.html
[SAE04] SAE. Architecture Analysis & Design Language (AADL). As-2 Embedded Computing Systems Committee SAE, nov 2004. SAE Standards AS5506.
[SE02] : Françoise Simonot-Lion , Jean-Pierre Elloy, « An Architecture Description Language for In-Vehicle Embedded System Development", 15th Triennial World Congress of the International Federation of Automatic Control Barcelona 2002.
[SF04] M. A. K. Sandstrom, Johan Fredriksson. Introducing a component technology for safety critical embedded real-time systems. In International Symposium on Component-based Software Engineering (CBSE7), 2004.
[SL92] Son S.H. and Liu J.W.S., « How well can data temporal consistency be maintained ? », IEEE Symp. on Computer-Aided Control System Design, 1992.
[SR88] Stankovic J.A. and Ramamritham K., « Hard real-time systems: a tutorial », IEEE Computer Society Press, 1988.
[SRL90]: L. Sha, R. Rajkumar, J. P. Lehoczhy, « Priority inheritance protocols: an approach to real-time synchronization », IEEE Transactions on Computers, Vol. n°39, n°9, sept. 1990, p.1175-1185.
[SSNB95] Stankovic, J., Spuri, M., Natale, M. D., and Buttazzo, G. "Implications of Classical Scheduling Results in Real-Time Systems". IEEE Computer, 28(6) :16–25, 1995.

[TNHN04] Aleksandra Tešanovic, Dag Nyström, Jörgen Hansson, and Christer Norström. Aspects and components in real-time system development : Towards reconfigurable and reusable software. Journal of Embedded Computing, 1(1), feb 2004.
[U93] Ulusoy, O. "Lock-Based Concurrency Control in Distributed Real-Time Database Systems". Journal of Data Management, 4(2), 1993.
[U94] Ulusoy O. "Processing Real-Time Transactions in Replicated Database Systems". International Journal of Parallel and Distributed Databases, 2(4):405-436, 1994.
[U98] Ulusoy O. "Transaction Processing in Distributed Active Real-Time Database Systems". Journal of Systems and Software (Special Issue on Active Real-Time Databases), 42(3), 1998.
[UB96] Ulusoy O. and Buchman A. "Exploiting Main Memory DBMS Feature to Improve Real-Time Concurrency Control Protocols". ACM SIGMOD Record, 25(1):123-151, 1996.
[UPPAAL07] : http://www.uppaal.com/ (accédé en 2007)
[W03] Heike Wehrheim. Behavioral subtyping relations for active objects. Form. Methods Syst. Des., 23(2):143-170, 2003.
[Wa03] Kurt C. Wallnau. Volume iii : A technology for predictable assembly from certifiable components. Technical report, CMU/SEI-2003-TR-009, 2003.
[W04]: A. Wellings. Concurrent and Real-Time Programming in Java. John Wiley and Sons, September 2004.
[WFSP01] Wu L., Faloutsos C., Sycara K., and PayneT. "Multimedia Queries by Example and Relevance Feedback". IEEE Data Engineering Bulletin, 24 (3): 14-21, 2001.
[WWFW97] Wang, J. Z., Wiederhold, G., Firschein, O., and Wei, S. X. "Content-based image indexing and searching using daubechies' wavelets". Int. J. on Digital Libraries,1(4): 311-328, 1997.
[XR04] Xiong M. and K. Ramamritham, "Deriving deadlines and periods for real-time update transactions", IEEE Transactions on Computers, Vol. 35 n°5, pp.567-583, 2004.
[XSRT96] Xiong M., J. A. Stankovic, K. Ramamritham, D. Towsley et R. M. Sivasankara, "Maintaining Temporal Consistency : issues and algorithms", 1st Int. Workshop on RTDBS: Issues and Applications, pp. 1-6, California, 1996.
[XSSRT96] Xiong M., Sivasankaran R.M., Stankovic J.A., Ramamritham K. et Towsley D., "Scheduling transactions with temporal constraints: exploiting data semantics", 17th IEEE Real-Time Systems Symposium, Washington, p. 240-251, 1996.

Annexes

Annexe : Publications

1. Ouvrages

- **Samia Bouzefrane, Les Systèmes d'Exploitation – Unix, Linux et Windows XP avec C et Java**, Dunod Éditions, 566 pages, octobre 2003, ISBN 2 10 007189 0.

Cet ouvrage, destiné aux étudiants de 2ème et 3ème cycles, présente de manière pédagogique les concepts fondamentaux des systèmes d'exploitation, ainsi que leurs applications. Chaque chapitre pose d'abord la problématique du thème traité, puis présente les concepts de base conduisant aux solutions théoriques, pour finir par des exercices corrigés, proposés avec de nombreux programmes testés en C et en Java.

- **S. Saad-Bouzefrane, Lexique d'informatique (Français- Anglais- Berbère)**, L'Harmattan Éd, sept. 1996, Paris, ISBN 2-7384-4650-7.

2. Revues internationales

[1] S. **Bouzefrane**, J.-P. Etienne and C. Kaiser, "Handling Overload and Data-relaxation Control in Distributed Real-Time Database Systems", in International Journal of Computers and Their Application, Vol 15, n°3, Sept. 2008.

[2] L. Amanton, B. Sadeg, S. **Bouzefrane**. « RT-DIS-COM: A Protocol to Manage Real-Time Distributed Transaction Commit", IEEE Intl. Symp. on Parallel and Distributed Computing (ISPDC'02), Iasi, Romania. Paru dans le journal INFORMATICA, Scientific Annals of "Alexandru Ioan Ciza" TOME XI, pp. 337-348, 2002.

[3] Sadeg B., Amanton L. and **Saad-Bouzefrane S**., "Disconnexion Tolerance in Real-Time Mobile Databases", Special Issue of INFORMATION: An International Journal, Vol. 4, N° 4, pp. 445-456, 2001.

[4] **Saad-Bouzefrane S**. and Sadeg B., "Relaxing the Correctness Criteria in Real-Time DBMSs", International Journal of Computers and their Applications, Vol. 7, N° 4, pp. 209-217, 2000.

3. Revues françaises

[5] **Bouzefrane S.**, J.-P. Etienne, C. Kaiser « Gestion de la surcharge dans les systèmes de gestion de base de données temps réel », Technique et Science Informatique, Edition Lavoisier, RSTI - TSI. Volume 27 – n° 7/2008, pages 879 à 910, sept. 2008.

[6] **Saad-Bouzefrane S**. and Cottet F., « Méthodologie d'analyse temporelle des applications temps réel réparties », revue APII JESA, Ed. Hermes, Vol. 33, N° 3, pp. 251-284, avril 1999.

4. Communications internationales

[6bis] Haï-Binh LE, **Samia Bouzefrane**, « Identity management systems and interoperability in a heterogeneous environment », Intern. Conf. on Advanced Technologies for Communications, Hanoï, oct. 2008.

[7] **Samia Bouzefrane**, Julien Cordry, Hervé Meunier, and Pierre Paradinas, "Evaluation of JavaCard Performance", Eighth Smart Card Research and Advanced Application Conference (CARDIS), London Sept 2008.

[8] **Samia Bouzefrane**, Julien Cordry, Pierre Paradinas, Gilles Grimaud, « An Open-Source Tool to Benchmark JavaCard Platforms", e-Smart Conference, Nice (France), sept. 2008.

[9] Jean-Paul Etienne and **Samia Bouzefrane**, "A Typed Composition Language for Real-time Systems", *the* 10th IEEE High Assurance Systems Engineering Symposium (HASE'2007), nov. 2007, Texas, USA.

[10] Pierre Paradinas, Julien Cordry, **Samia Bouzefrane**, « How to measure the performance of the JavaCard Platforms », e-Smart Conference 2007, sept. 2007, Nice (France).

[11] Pierre Paradinas, Julien Cordry, **Samia Bouzefrane**, « Performance Evaluation of JavaCard Bytecodes », Workshop in Information Security Theory and Practices, 9-11 May 2007, Greece.

[12] J.-P. Etienne, J. Cordry et **S. Bouzefrane**. Applying the CBSE Paradigm in the Real Time Specification for Java . In 4th International Workshop on Java Technologies for Real-time and Embedded Systems, Paris, 11-13 Oct, pp. 218-226, 2006, Vol. 177, ACM.

[13] J.-P. Etienne et **S. Bouzefrane**. Applying the CBSE paradigm on real-time systems. In (WFCS'06) 6th IEEE International Workshop on Factory Communication Systems, Torino, 28-30 June, 2006.

[14] J.-P. Etienne et **S. Bouzefrane**. "Vers une approche par composants pour la modélisation d'applications temps réel ». In (MOSIM'06) 6ème Conférence Francophone de Modélisation et Simulation, Rabat, 3-5 avril, pp. 1-10, Lavoisier, 2006.

[15] J. Cordry, N. Bouillot et **S. Bouzefrane**. "Performing Real-Time Scheduling in an Interactive Audio-Streaming Application". In 7th International Conference on Enterprise Information Systems, pp. 140-147, Miami, Mai 2005.

[16] J. Cordry, N. Bouillot et **S. Bouzefrane**. "BOSSA et le Concert Virtuel Réparti, intégration et paramétrage souple d'une politique d'ordonnancement spécifique pour une application multimédia distribuée ». In RTS'05 (13th International Conference on Real-Time Systems), Paris 5 Avril, pp. 18-39, 2005.

[17] J. P. Etienne and **S. Saad-Bouzefrane**, "A Java Platform to Control Real-Time Transactions Overload" IEEE ISORC'04 Conference, Autriche, mai 2004.

[18] **S. Saad-Bouzefrane** and S. Bourenane, "A transactional multi-mode model to handle overload in distributed RTDBSs" Conférence ICEIS'2004, Porto, avril 2004.

[19] **S. Saad-Bouzefrane**, "How to Manage Real-Time Transactions Overload in epsilon-data applications ?", Work In Progress Session, EuroMicro Real-Time Systems Conference, juillet 2003, Porto, Portugal.

[20] **S. Saad-Bouzefrane**, C. Kaiser, "Distributed Overload Control for Real-Time Replicated Database Systems", (Full paper), International Conference on

Enterprise Information Systems (ICEIS'2003) Conference, avril 2003, Angers France.

[21] **S. Saad-Bouzefrane**, C. Kaiser, "How to Manage Replicated Real-Time Databases in an Overloaded Distributed System?", (Full paper), Conférence Real Time Systems (RTS'2003), Paris avril 2003.

[22] B. Sadeg, J. Haubert, L. Amanton and **S. Bouzefrane** , "WEP: An Adaptation of 1-PC Protocol to Distributed Real-Time Transactions", IEEE Intl. Symp. on Signal Proc. and Information Technology (IEEE-ISSPIT'02), Marrakech, Morocco, 2002.

[23] Sadeg B., **Saad-Bouzefrane S**. and Amanton L. , "D_ANTICIP: a Protocol Suitable for Distributed Real-Time Transactions", 4th Int. Conf. on Enterprise Information Systems - Databases and Informations Systems topic (ICEIS'02), Universidad de Castilla-La mancha, Ciudad Real – Spain, avril 2002.

[24] Sadeg B., Amanton L. and **Saad-Bouzefrane S**., "Real-Time Transactions Management in a Distributed Environment", 20th IASTED Applided Informatics: Databases and Applications Symposium (AI 2002), Innsbruck, Austria, 2002.

[25] Amanton L., Sadeg B., **Saad-Bouzefrane S**., "The Causal Phase Ordering Protocol to Manage Soft Real Time Distributed Transactions", In ISCA 14th Int. Conference on Computer Application in Industry and Engineering (CAINE-2001), Nov. 27-29, Las Vegas, Nevada, USA.

[26] Sadeg B., Amanton L., **Saad-Bouzefrane S**., "A Causal-Phase Protocol to Order Soft Real-Time Transactions in a Distributed Database", In 26th annual IEEE Conference on Local Computer Networks (LCN-2001), Nov. 15-16, Tampa, Florida, USA.

[27] Amanton L., Sadeg B., **Saad-Bouzefrane S**., Applying EDF Principle to Real-Time Transactions in a Wireless Environment, In Intl. Symp. on Systems and Programs, (ISPS-2001), May 14-16, Algiers, Algeria.

[28] **Saad-Bouzefrane S.,** Sadeg B., Amanton L., "Soft Real-Time Transactions Scheduling in a Wireless Environment", In IEEE Int. Symp. on Object-Oriented Real-Time Distributed Computing, (IEEE ISORC-2001), 2-4 Mai, Magdeburg, Germany.

[29] Sadeg B., **Saad-Bouzefrane S**., Amanton L., "Management of Real-Time Query Transactions in Mobile Computing Environment", In Int. Conf. On Real-Time and Embedded Systems (RTS-2001), 6-8 mars, Paris.

[30] Amanton L., Sadeg B., **Saad-Bouzefrane S**., "Disconnection Tolerance in Soft Real-Time Mobile Databases", In Int. Conf. on Computers and Their Applications (CATA-2001), 28-30 mars, Seattle, Washington, USA.

[31] **Saad-Bouzefrane S**., Sadeg B., "How to Manage Real-Time Transactions in Non-Strict Context?", 3rd IEEE ASSET-2000, (Application-Specific Systems & Software Eng. & Technology) Communication publiée dans les actes, 24-25 mars, Richardson, Texas, USA.

[32] **Saad-Bouzefrane S**., Sadeg B., "Relaxing the Correstness Criteria in Real-Time DBMSs", CATA-2000 (Computers And Their Applications), Communication publiée dans les actes, 28-31 mars, New-Orleans, Louisiane, USA.

[33] Kordjani-Taibi H., **Saad-Bouzefrane S**., "Modelling technique for evaluating Client/Server DBMS performance", Inter. Conference on Soft. Eng. Applied to networking and parallel and distributed computing, 2000, Reims.

[34] Sadeg B., **Saad-Bouzefrane S**., "Enhancing Concurrency Control and Scheduling of Real-Time Transactions", WIP-Int. Real-Time System Symposium, dec. 1-3, 1999., Phoenix, Arizona, USA.

5. Communications françaises

[35] **S. Bouzefrane**, J. Cordry et P. Paradinas. "A methodology for testing JavaCard performance ". In Conférence Française en Systèmes d'Exploitation (CFSE), Suisse, 2008.

[36] J.-P. Etienne et **S. Bouzefrane**. "Utilisation de l'approche par composants pour la conception d'applications temps réel ». In (RJCITR'05) Premières Rencontres des Jeunes Chercheurs en Informatique Temps Réel, Nancy, septembre, 2005.

[37] Sadeg B., **Saad-Bouzefrane S**., "Gestion des Transactions temps réel à échéance non stricte », 8th international Conference on Real-Time Systems (RTS-2000), Communication publiée dans les actes, 28-30 mars, Paris.