

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Nicolas BERTHIER

Thèse dirigée par **Florence MARANINCHI**
et codirigée par **Laurent MOUNIER**

préparée au sein du laboratoire **VERIMAG**
et de l'**École Doctorale Mathématiques, Sciences et
Technologies de l'Information, Informatique**

Programmation synchrone de pilotes de périphériques pour un contrôle global de ressources dans les systèmes embarqués

Thèse soutenue publiquement le **12 mars 2012**,
devant le jury composé de :

Éric FLEURY

École Normale Supérieure de Lyon, Président

Gilles MULLER

Institut National de Recherche en Informatique et Automatique, Rapporteur

Antoine FRABOULET

Institut National des Sciences Appliquées de Lyon, Examineur

Abdoulaye GAMATIÉ

Centre National de la Recherche Scientifique, Examineur

Florence MARANINCHI

Institut Polytechnique de Grenoble, Directeur de thèse

Laurent MOUNIER

Université Joseph Fourier, Co-Directeur de thèse



REMERCIEMENTS

J'adresse ces premiers remerciements à l'ensemble des membres de mon jury de thèse, pour m'avoir fait l'honneur d'accepter d'évaluer mes travaux. Les commentaires et questions des rapporteurs du manuscrit et des examinateurs, Éric Fleury, Gilles Muller, Antoine Fraboulet et Abdoulaye Gamatié, me seront utiles pour préciser les pistes de recherches futures que j'ai pu identifier pendant la réalisation de ce travail. Je les remercie d'avoir assisté à ma soutenance de plus ou moins loin, et malgré tous les empêchements qui ont pu survenir.

Je remercie également mes directeurs de thèse, Florence Maraninchi et Laurent Mounier, sans qui je n'aurais sans doute pu mener mes travaux. Je leur suis reconnaissant pour le soutien, la confiance, ainsi que la très grande liberté qu'ils m'ont accordés.

Je profite en outre de ces quelques lignes pour remercier les personnes qui m'ont permis de m'introduire dans le monde de la recherche en informatique grenobloise, tout au long de mon cursus universitaire. Je remercie encore Ana Simonet, du laboratoire TIMC-IMAG, qui m'a permis de mettre un premier pied dans une équipe de recherche en 2005. J'ai intégré le laboratoire Verimag l'année suivante, où Guillaume Salagnac et Christophe Rippert m'ont accueilli pour les stages qui suivirent, et je les en remercie à nouveau. Grâce à toutes ces personnes et au fur et à mesure de mes nombreux stages, j'ai pu apprendre à maîtriser les outils qui font aujourd'hui mon quotidien de docteur en informatique. J'ai aussi acquis une vision précise du fonctionnement du monde de la recherche scientifique ; cette précieuse connaissance m'a permis d'éviter bien des écueils pendant ma thèse.

Pour leur accueil et l'ambiance de travail depuis mes premières années de stage jusqu'à aujourd'hui, je remercie l'ensemble de mes collègues du laboratoire Verimag et en particulier les membres des équipes ISE, puis Sychrone. Je tiens également à remercier Jean-Noël Bouvier pour le système informatique très performant qu'il a maintenu pour le laboratoire, malgré les usages parfois peu orthodoxes que j'en ai fait.

Je voudrais de plus remercier Giovanni, Valentin, Romain et Tayeb, valeureux membres du bureau 39 qui ont dû supporter mes soupirs et moments d'absence, notamment lorsque j'étais plongé dans mes réflexions. Je n'oublie pas non plus les autres membres du laboratoire qui m'ont soutenu tout au long de ces années : Chaouki, Claire, Erwan, Guillaume, les trois Juliens, Kevin, Laurie, Mathias, Mathilde, Matthieu, Pascal, Raphaël, Selma, Simplicie, Stéphane, Yvan... et Serge bien sûr. J'ai passé de bons moments avec tous, à la fois au laboratoire qu'en dehors, et eu plaisir à discuter de sujets fort variés. Je poursuis cette liste en remerciant chaleureusement les membres de ma famille qui m'ont encouragé malgré les événements qu'ils connaissent, ainsi que mes amis, qui m'ont toujours apporté leur soutien.

Enfin, je termine cette liste en mentionnant les contributions indirectes aux travaux que j'ai menés à bien et connaissances que j'ai acquises pendant ma thèse, du moins celles qui sont suffisamment importantes à mes yeux pour y figurer. La rédaction du présent manuscrit ainsi que les manipulations et implantations et que j'ai réalisées, ont nécessité des outils appropriés sans lesquels ces travaux

auraient assurément été bien (plus) pénibles. Pour leur existence, je veux donc remercier les innombrables « contributeurs » de logiciels libres, et particulièrement ceux des outils relatifs à \LaTeX , OCaml et des multiples outils GNU (avec une mention encore plus spéciale pour les développeurs modestes et géniaux de GNU Emacs).

D'un point de vue beaucoup moins technique, et en bonus aux lect-eur/rice-s assez téméraires pour finir de lire ces quelques paragraphes, je remercie Richard Monvoisin pour l'initiation à la zététiqque que j'ai reçue dans le cadre de mon monitorat. Merci aussi à Axelle, Cyrille et Marion, d'avoir contribué avec moi à l'accroissement de l'ensemble des « zétéclips » qui, je l'espère, participe à sa mesure à l'apprentissage de l'esprit critique. Je veux aussi remercier les intervenants des formations à l'éthique dans la recherche et philosophie des sciences. Toutes ces connaissances m'ont grandement servi pour tenter d'apporter des réponses aux sempiternelles réflexions d'ordre philosophique relatives à l'intérêt de travaux de recherche qui, me semble-t-il, « frappent » beaucoup de thésards. J'espère à l'avenir trouver le temps d'approfondir mon savoir dans les vastes domaines que j'ai pu aborder par l'intermédiaire de ces disciplines.

Historique des modifications :

3 décembre 2012 : Vérification et mise-à-jour d'URL ; Quelques corrections typographiques et orthographiques ; Tentative de « démasculinisation » du/de la lect-eur/rice.

Les URL de ce document ont été vérifiées le 3 décembre 2012.

Détails et crédits des polices de caractères :

La police principale du document est Linux Libertine/Biolinum, publiée sous licence GPL/OFL ;

Le texte à largeur fixe (*e.g.*, listings, URL) utilise la police TX, également sous GPL ;

Les titres de chapitres utilisent Comfortaa, publiée sous licence cc-by-nd 3.0 par Johan Aakerlund.

TABLE DES MATIÈRES

| | |
|--|------------|
| Remerciements | iii |
| Table des matières | vii |
| Figures, listings et tableaux | xi |
| Liste des figures | xi |
| Liste des listings | xi |
| Liste des tableaux | xii |
| | |
| I Introduction | 1 |
| | |
| 1 Contexte et problématique générale | 3 |
| 1 Contexte général | 3 |
| 1.1 Systèmes embarqués | 3 |
| 1.2 Gestion de ressources | 4 |
| 1.3 Motivations | 5 |
| 2 Restriction à la programmation des nœuds de réseaux de capteurs sans fil | 5 |
| 3 Problématique | 5 |
| 4 Contribution | 5 |
| 4.1 Programmation synchrone des pilotes | 6 |
| 4.2 Synthèse d'un contrôleur global | 6 |
| 4.3 Évaluation | 6 |
| 5 Plan et guide de lecture | 6 |
| | |
| 2 Précision du problème : premier débroussaillage | 9 |
| 1 Introduction du chapitre | 10 |
| 2 Les réseaux de capteurs sans fil | 10 |
| 2.1 Applications | 10 |
| 2.2 Plates-formes matérielles typiques | 12 |
| 2.3 Contraintes intrinsèques | 15 |
| 3 Prise en compte de l'énergie dans les réseaux de capteurs sans fil | 17 |
| 3.1 Éléments impactant la consommation énergétique | 17 |
| 3.2 Restriction du problème à la gestion locale des nœuds | 17 |
| 3.3 Vers une gestion avisée des périphériques consommateurs | 18 |
| 4 Caractérisation des batteries et besoins afférents | 18 |
| 4.1 Batterie idéale | 18 |
| 4.2 Première tentative d'investigation : examen de documentations constructeur | 19 |

| | | |
|---|--|-----------|
| 4.3 | Recours aux modèles existants | 19 |
| 4.4 | Bonnes pratiques et besoins | 21 |
| 5 | Gestion des périphériques | 22 |
| 5.1 | Pilotes de périphériques | 22 |
| 5.2 | Implantation des pilotes de périphériques | 23 |
| 5.3 | Vérification statique de propriétés sur les pilotes | 23 |
| 5.4 | Synthèse de pilotes | 24 |
| 5.5 | Quelques observations | 25 |
| 6 | Conclusion du chapitre | 26 |
| II État de l'art | | 27 |
| 3 Gestion de ressources et implantation des nœuds de réseaux de capteurs | | 29 |
| 1 | Principes généraux de gestion du microcontrôleur | 30 |
| 1.1 | Programmation du calculateur | 30 |
| 1.2 | Structuration des programmes : fil et contexte d'exécution | 31 |
| 1.3 | Traitement des requêtes d'interruption matérielles | 32 |
| 2 | Modèles de programmation | 33 |
| 2.1 | Déclenchement de réactions | 33 |
| 2.2 | Boucle de scrutation explicite | 33 |
| 2.3 | Réalisation de la concurrence | 34 |
| 3 | Langages dédiés à l'implantation | 36 |
| 3.1 | Approches langages à composants | 36 |
| 3.2 | Protothreads. | 37 |
| 3.3 | Utilisation d'automates hiérarchiques | 37 |
| 4 | Approches d'implantation des applications | 38 |
| 4.1 | Implantation sur machine nue | 38 |
| 4.2 | Support système pour l'implantation | 38 |
| 4.3 | Machines virtuelles de haut niveau | 43 |
| 4.4 | Discussion | 45 |
| 5 | La gestion des périphériques dans les réseaux de capteurs sans fil | 45 |
| 5.1 | Gestion de ressources décentralisée | 45 |
| 5.2 | Une approche centralisée multifils | 46 |
| 6 | Conclusion du chapitre | 47 |
| 4 Programmation synchrone et synthèse de contrôleur | | 49 |
| 1 | Programmation synchrone | 50 |
| 1.1 | Langages synchrones | 50 |
| 1.2 | Fragment de LUSTRE | 51 |
| 1.3 | Machines de Mealy Booléennes | 54 |
| 2 | Synthèse de contrôleur | 56 |
| 2.1 | Principe | 56 |
| 2.2 | Outils existants | 58 |
| 2.3 | Exemples d'applications | 58 |

| | |
|---|-----------|
| III Contribution | 61 |
| 5 Proposition d'architecture logicielle : programmation synchrone et contrôle global | 63 |
| 1 Introduction du chapitre | 64 |
| 2 Description générale de l'architecture proposée | 64 |
| 2.1 Constitution de l'invité | 65 |
| 2.2 Rôle de la couche de contrôle | 65 |
| 3 La couche d'adaptation | 65 |
| 3.1 Gestion de la plate-forme matérielle par l'invité | 66 |
| 3.2 Exécution de l'invité | 66 |
| 4 Description générale de la couche de contrôle et définitions | 66 |
| 4.1 Structure de la couche de contrôle : la membrane et le noyau réactif | 67 |
| 4.2 Chemins d'exécution dans la couche de contrôle | 68 |
| 5 Détails de la membrane | 70 |
| 5.1 Gestion globale du microcontrôleur | 70 |
| 5.2 Traitement des requêtes matérielles | 71 |
| 5.3 Traitement des requêtes logicielles | 72 |
| 5.4 Exécution du noyau réactif | 72 |
| 5.5 Exécution des traitants d'interruptions virtuelles | 73 |
| 6 Détails du noyau réactif | 74 |
| 6.1 Automates de pilotes | 74 |
| 6.2 Automate de contrôle | 75 |
| 6.3 Construction du noyau réactif | 75 |
| 7 Un scénario d'exécution simple | 76 |
| 8 Conclusion du chapitre | 79 |
| 6 Outillage et Choix | 81 |
| 1 Introduction du chapitre | 81 |
| 2 Choix d'une plate-forme cible | 81 |
| 3 Implantation de machines de Mealy communicantes | 82 |
| 3.1 Synchronous C | 82 |
| 3.2 ARGOS | 86 |
| 3.3 Remarques à propos de la causalité | 88 |
| 4 Conclusion du chapitre | 90 |
| 7 Implantation et évaluations | 91 |
| 1 Introduction du chapitre | 92 |
| 2 Implantation de la couche de contrôle | 92 |
| 2.1 Construction du noyau réactif | 92 |
| 2.2 Adaptation de systèmes invités | 93 |
| 3 Étude de cas et principes d'adaptation | 94 |
| 3.1 Présentation de l'étude de cas | 94 |
| 3.2 Détails de l'implantation originale | 95 |
| 3.3 Principes de l'adaptation | 97 |
| 3.4 Adaptation effective | 98 |
| 3.5 Discussion sur l'adaptation | 100 |
| 4 Évaluation quantitative | 101 |
| 4.1 Empreinte mémoire | 101 |
| 4.2 Coût calculatoire | 102 |

| | | |
|----------|--|------------|
| 4.3 | Comparaison avec des approches existantes | 102 |
| 5 | Évaluation qualitative et extensibilité | 102 |
| 5.1 | Sur le développement des pilotes | 103 |
| 5.2 | Extensibilité | 103 |
| 6 | Considérations sur les interblocages | 105 |
| 6.1 | Exemples de situations d'interblocage potentiel | 105 |
| 6.2 | Possibilités de détection hors ligne | 106 |
| 6.3 | Conclusion sur les interblocages | 108 |
| 7 | Synthèse de contrôleur automatisée | 108 |
| 7.1 | Sur le non déterminisme des contrôleurs produits | 109 |
| 7.2 | Le cas de BZR/SIGALI | 110 |
| 7.3 | Vers une chaîne d'outils idéale | 111 |
| 8 | Conclusion du chapitre | 112 |
| 8 | Conclusion | 115 |
| 1 | Bilan | 115 |
| 2 | Perspectives | 115 |
| | Bibliographie | 117 |

FIGURES, LISTINGS ET TABLEAUX

Liste des figures

| | | |
|-----|---|-----|
| 2.1 | Représentation de la plate-forme matérielle Wsn430. | 15 |
| 4.1 | Deux machines de Mealy Booléennes Sa et Sb. | 54 |
| 4.2 | Produit synchrone brut des machines Sa et Sb de la figure 4.1. | 55 |
| 4.3 | L'automate produit, après encapsulation. | 56 |
| 4.4 | Deux machines de Mealy isolées, et un contrôleur. | 57 |
| 4.5 | Représentation du comportement du système contrôlé. | 58 |
| 5.1 | Représentation globale de l'approche. | 65 |
| 5.2 | Architecture de la couche de contrôle. | 67 |
| 5.3 | Illustration de deux chemins d'exécution différents dans la couche de contrôle. | 69 |
| 5.4 | Représentation d'un pilote de périphérique, et une version contrôlable de celui-ci. | 74 |
| 5.5 | Pilote de timer original, et version contrôlable. | 76 |
| 5.6 | Automate de pilote contrôlable pour un ADC. | 77 |
| 5.7 | Automate de pilote contrôlable pour un émetteur-récepteur radio. | 77 |
| 5.8 | Automate de contrôle simple pour les pilotes des figures 5.6 et 5.7. | 77 |
| 5.9 | Un exemple d'exécution de la pile logicielle complète. | 78 |
| 6.1 | Illustration du dialogue instantané. | 88 |
| 7.1 | Outils utilisés pour la construction du noyau réactif du prototype de couche de contrôle. | 93 |
| 7.2 | Représentation de l'implantation originale et des pilotes de périphériques qu'elle utilise. | 95 |
| 7.3 | Principe de contrôle des accès directs aux ressources. | 104 |
| 7.4 | Deux extraits d'automates de pilotes incorporables à la couche de contrôle. | 106 |
| 7.5 | Deux pilotes, ainsi que le contrôleur le plus permissif. | 109 |
| 7.6 | Le contrôleur de la figure 7.5, avec oracle. | 110 |
| 7.7 | Chaîne d'outils idéale incorporant un outil de synthèse de contrôleur discret. | 112 |

Liste des listings

| | | |
|-----|---|----|
| 4.1 | un_sur_deux : un nœud LUSTRE simple. | 52 |
| 4.2 | Composition en LUSTRE de deux nœuds un_sur_deux communicants. | 52 |
| 4.3 | Noyau réactif obtenu par compilation du nœud LUSTRE du listing 4.2. | 53 |
| 5.1 | Deux versions d'une fonction de pilote de convertisseur analogique-digital. | 66 |

| | | |
|-----|--|-----|
| 5.2 | Une procédure <code>main()</code> | 70 |
| 5.3 | Exemple de traitant d'interruption. | 71 |
| 5.4 | La routine <code>on_it()</code> | 72 |
| 5.5 | La fonction <code>on_sw()</code> | 72 |
| 5.6 | La routine <code>react()</code> | 73 |
| 6.1 | Un codage en Synchronous C de la composition parallèle de deux machines de Mealy. | 84 |
| 6.2 | Implantation en ARGOS de la composition parallèle des deux automates de la figure 4.1. | 87 |
| 6.3 | Traduction directe en LUSTRE du programme du listing 6.2. | 87 |
| 6.4 | Traduction directe en LUSTRE du programme de la figure 6.1. | 89 |
| 6.5 | Le programme en LUSTRE du listing 6.4, après transformation des systèmes d'équations. | 89 |
| 7.1 | Exemple d'usage incorrect de l'implantation de l'X-MAC par une couche de routage. | 96 |
| 7.2 | Une routine extraite de l'implantation originale de l'X-MAC. | 97 |
| 7.3 | Adaptation possible du code original du listing 7.2. | 99 |
| 7.4 | Autre adaptation possible du code original du listing 7.2 | 100 |

Liste des tableaux

| | | |
|-----|---|-----|
| 7.1 | Empreinte mémoire typique de quelques systèmes existants. | 102 |
| 7.2 | Comparaison de surcoûts en matière de cycles de calcul. | 102 |

PREMIÈRE PARTIE

INTRODUCTION

INTRODUCTION : CONTEXTE ET PROBLÉMATIQUE GÉNÉRALE

Contenu du chapitre

| | | |
|-----|--|---|
| 1 | Contexte général | 3 |
| 1.1 | Systèmes embarqués | 3 |
| 1.2 | Gestion de ressources | 4 |
| 1.3 | Motivations | 5 |
| 2 | Restriction à la programmation des nœuds de réseaux de capteurs sans fil | 5 |
| 3 | Problématique | 5 |
| 4 | Contribution | 5 |
| 4.1 | Programmation synchrone des pilotes | 6 |
| 4.2 | Synthèse d'un contrôleur global | 6 |
| 4.3 | Évaluation | 6 |
| 5 | Plan et guide de lecture | 6 |

1 Contexte général

Les travaux présentés dans cette thèse ont été menés au laboratoire VERIMAG, dont les thématiques de recherche couvrent la conception des systèmes embarqués. Les principaux axes de développement des différentes équipes visent donc à l'élaboration de méthodes de conception des aspect logiciels et matériels de ces systèmes. Les approches qui y sont conçues sont centrées sur la conception de langages de programmation, la modélisation des systèmes, ou encore la vérification automatique de programmes.

1.1 Systèmes embarqués

Les systèmes embarqués sont issus des mécanismes de contrôle automatique, dans les domaines industriels et les transports par exemple. Leurs parties logicielles et matériels étaient initialement conçus conjointement. Avec la miniaturisation des composants électroniques, ces mécanismes moins encombrants purent être embarqués dans des systèmes de plus en plus variés. Les applications pour lesquelles ils sont employés sont notamment de plus en plus gourmandes en capacités de calculs.

La complexité croissante des tâches à réaliser par ces systèmes impose l'usage de méthodes autorisant la conception modulaire des programmes réalisant ces différents travaux. Chaque tâche est en général spécifiée

et exprimée de manière à simplifier son implantation. Elles sont souvent conçues comme des calculs réalisés de manière concurrente par le système.

Aussi, les contraintes intrinsèques liées à l'environnement d'utilisation des systèmes embarqués imposent des limitations fortes en ce qui concerne leur encombrement, et restreignent par là la quantité de mémoire et la capacité de calcul dont ils disposent. De plus, le respect de contraintes de réactivité est souvent imposé puisque ces systèmes de contrôle fonctionnent de manière couplée avec leur environnement. Par exemple, le temps maximum autorisé pour la prise d'une décision par le calculateur dépend du système physique qu'il contrôle ; un dépassement de cette borne peut avoir des conséquences allant jusqu'à la perte de vies humaines.

Autonomie. Dans le cas de systèmes autonomes, aux contraintes précédentes s'ajoutent les aspects liés à la prise en compte des limites d'alimentation énergétique. Il est en effet important de chercher à exploiter efficacement les sources d'énergies que ces systèmes embarquent. Il est de plus nécessaire de pouvoir garantir un temps minimum de fonctionnement du système avant son déploiement. Bien entendu, les concepteurs des plates-formes matérielles destinées aux applications autonomes prennent en compte la consommation des composants qu'ils intègrent. Cependant, pour que ceux-ci soient exploités efficacement, le logiciel qui gère ces différents éléments matériels doit également être conçu en regard des contraintes d'autonomie.

1.2 Gestion de ressources

Une *ressource* est par exemple le temps de disponibilité d'un processeur, la mémoire disponible pour le système, ou encore la quantité d'énergie dont il dispose pour fonctionner. D'un point de vue purement logiciel, un bus est également une ressource s'il relie plusieurs composants du système, et que des accès concurrents sont possibles au moyen de ce bus. Des composants sont aussi partagés si plusieurs tâches y accèdent concurrentement.

Si plusieurs tâches s'exécutent sur le système, alors leurs utilisations des ressources ne supportant pas d'accès concurrents doivent être exclusives entre elles. C'est par exemple le cas d'un processeur mono-cœur, dont le temps d'exécution doit être partagé entre les différents calculs à réaliser.

Concurrence des calculs. La gestion efficace du temps de disponibilité du processeur est un problème central dans les systèmes informatisés en général, mais les conséquences de leur mauvaise gestion ne sont parfois notables que par une dégradation des performances. En revanche, les contraintes intrinsèques imposées dans le contexte des systèmes embarqués exacerbent ces effets, et la gestion correcte du processeur est un élément central lors de la conception de logiciels sûrs.

Dans le domaine des systèmes embarqués critiques, des solutions ont été développées pour assurer la gestion du partage du processeur entre différents travaux concurrents. Les *langages synchrones* par exemple, constituent une solution complète au problème de la l'implantation efficace de calculs exprimés de manière concurrente. Dans ce cadre, les divers tâches peuvent généralement être ordonnancées statiquement, mais d'autres solutions ont été développées pour leur implantation avec à l'aide d'un support système minimal [30].

Aspects énergétiques. Concernant l'énergie, il est important de noter que les composants matériels qui composent les plates-formes consomment également cette ressource en fonction de leur mode de fonctionnement. Aussi, nous verrons que l'état de chaque composant matériel pris indépendamment ne suffit pas à étudier l'impact de chacun sur la durée de vie d'un système alimenté par batterie. Il peut être par exemple intéressant de mettre en œuvre une politique de gestion de l'énergie destinée à éviter les pics de consommation. Or, une telle politique ne peut être mise en place qu'à partir d'une connaissance de l'état global de la plate-forme matérielle.

1.3 Motivations

L'objectif de cette thèse est de proposer une solution au problème de la gestion de ressources qui, sur le modèle de l'énergie, sont partagées par plusieurs composants des plates-formes matérielles.

2 Restriction à la programmation des nœuds de réseaux de capteurs sans fil

Pour étudier le problème de la gestion des ressources dans les systèmes embarqués, nous nous plaçons dans le cadre de l'implantation des nœuds de réseaux de capteurs sans fil, qui en sont une classe représentative. En effet, les nœuds de ces systèmes disposent de composants matériels très contraints en ressources, que ce soit la quantité de mémoire, la capacité de calcul, ou encore l'énergie électrique disponible pour son fonctionnement. Par rapport à d'autres classes de systèmes embarqués comme les *systèmes-sur-puce*, ils présentent en outre les propriétés suivantes :

- L'architecture des composants matériels qui composent les nœuds de ces réseaux est restée comparativement plus simple que celles, plus puissantes, qui sont actuellement développées pour d'autres systèmes embarqués (e.g., téléphones). Cet aspect permet alors de proposer une solution simple et générique pour la gestion efficace des ressources, sans se limiter aux systèmes où un support matériel existe déjà (ce qui restreindrait alors notre champ de recherche). Par exemple, les architectures des plates-formes de réseaux de capteurs sans fil ne présentent pas de composants conformes à la norme ACPI [87], en général complexes et plus adaptés aux ordinateurs portables qu'aux systèmes embarqués contraints ;
- Ces réseaux doivent souvent respecter des contraintes énergétiques fortes couplées à une demande de garantie sur son temps de fonctionnement effectif. L'énergie disponible constitue un cas typique de ressource limitée à prendre en compte à tous les niveaux pendant la conception du système, depuis le choix du matériel au design des protocoles de communication et des applications. Ainsi, des protocoles de communication sont conçus de manière à limiter la quantité de calcul nécessaire sur les nœuds, ou bien intègrent des informations (mesurées ou déduites) relatives à l'énergie restante.

Il existe une très grande variété d'applications et de systèmes embarqués dédiés à la programmation des nœuds de réseaux de capteurs sans fil. Cependant, nous verrons que les architectures et méthodes d'implantation de ceux-ci ne permettent pas la prise en compte de l'état global

3 Problématique

L'exploitation efficace des différents modes de fonctionnement des périphériques n'est pas simple et requiert une connaissance globale de l'état du système. Nous nous proposons d'apporter une réponse au problème du contrôle global des ressources des plates-formes matérielles, notamment en recherchant une solution à la gestion de l'énergie au niveau des nœuds de réseaux de capteurs sans fil.

À partir d'une réponse permettant la mise en place de politiques de gestion de l'énergie, nous devrions pouvoir généraliser l'approche pour exploiter la connaissance globale requise. Enfin, nous recherchons une solution permettant la réutilisation, dans la mesure du possible, de la grande quantité de codes et applications existants.

4 Contribution

Sur le modèle de la *para-virtualisation* [159], nous proposons l'introduction d'une couche de contrôle regroupant l'ensemble des pilotes de périphériques.

Cette approche permet de réutiliser l'ensemble des codes existants avec un minimum de modifications. Au surplus, une solution basée sur un principe de virtualisation nous permet d'identifier ce que serait une plate-forme idéale dans le cadre de l'implantation d'un contrôle global des ressources.

4.1 Programmation synchrone des pilotes

Pour extraire une connaissance globale de l'état du système, nous distinguons une partie dédiée au contrôle dans la structure des pilotes. Cette partie contrôle reflète fidèlement les modes de fonctionnement et changements d'états possibles du périphérique géré.

Ces comportements sont exprimés sous la forme de machines de Mealy Booléennes communicantes, et programmées à l'aide d'un langage synchrone. Par ce moyen, une information fidèle portant sur l'état global de la plate-forme devient disponible.

4.2 Synthèse d'un contrôleur global

À partir de cette nouvelle structuration des pilotes de périphériques, nous proposons une méthode d'implantation de contrôle global de ressources. Cette solution est basée sur un principe de synthèse de contrôleur, qui permet d'assurer des propriétés globales sur un ensemble de modèles, sans modifier le comportement intrinsèque de chacun d'eux. Dans notre cas, les modèles contrôlés sont les machines de Mealy des pilotes, et le contrôleur à construire permet d'empêcher les changements de modes de fonctionnement des périphériques menant la plate-forme dans un état global violant des propriétés spécifiées. Ces dernières assurent par exemple des exclusion d'accès à des bus, ou encore reflètent une politique de gestion de l'énergie visant à réduire les pics de consommation.

4.3 Évaluation

Nous évaluons l'architecture logicielle que nous proposons à l'aide d'un prototype de couche de contrôle.

Pour programmer la partie contrôle des pilotes, nous proposons l'usage du langage synchrone ARGOS [114]. Ce dernier permet d'exprimer des programmes constitués de machines de Mealy communicantes. Nous implantons un compilateur pour ce langage, afin de traduire ces compositions d'automates dans un code C sans effectuer leur produit. Cet outil nous permet de simplifier l'écriture du prototype de couche de contrôle. Ce prototype est exécutable conjointement avec un noyau multifils préemptif, et un système d'exploitation supportant l'expression de tâches concurrentes selon un modèle d'exécution monofil dirigé par des évènements.

Nous évaluons ensuite l'impact sur les couches logicielles supérieures par une étude de cas. Elle consiste en la modification de code de niveau système existant afin d'évaluer l'impact de la couche de contrôle.

Quelques-unes des applications et extensions de l'approche sont enfin évoquées, afin d'en apprécier la flexibilité. Nous discutons également les possibilités d'automatisation de la synthèse du contrôleur global.

Cette contribution a fait l'objet d'une publication dans les actes d'une conférence internationale [16], ainsi que d'un article de journal [17].

5 Plan et guide de lecture

Pour clore cette partie introductive, nous précisons dans le second chapitre le problème auquel nous nous attelons. Une présentation détaillée des réseaux de capteurs sans fil et des ses caractéristiques affine le domaine de notre recherche. Nous étudions ensuite succinctement les différents niveaux de prise en compte de la consommation énergétique dans ces réseaux. Après une étude du comportement des batteries destinée à identifier de bonnes pratiques pour les utiliser efficacement, nous examinons les méthodes actuelles de développement des pilotes de périphériques. Un résumé du problème vient clore la partie introductive.

Dans le chapitre 3, nous abordons les différentes méthodes et propositions existantes de d'implantation et de gestion des ressources dans les réseaux de capteurs sans fil. Seuls les éléments présentés à la section 5 est nécessaire pour aborder la contribution de cette thèse.

Le chapitre 4 introduit les concepts nécessaires à la compréhension de la contribution, en l'occurrence les principes de la programmation synchrone, ainsi que la synthèse de contrôleur ; le/la lect-eur/rice famili-er/ère avec ces deux notions pourra se reporter directement à la partie III page 63.

La contribution est présentée dans la dernière partie de cette thèse. Dans le chapitre 5, nous présentons en détail l'architecture de la couche de contrôle que nous proposons.

Le chapitre 6 décrit les choix effectués pour l'implantation de notre prototype de couche de contrôle, notamment en ce qui concerne les outils pour la programmation des parties contrôle des pilotes de périphériques.

Le chapitre 7 décrit l'implantation que nous avons réalisée, ainsi que l'étude de cas destinée à évaluer les modifications d'un code existant pour qu'il s'exécute conjointement avec la couche de contrôle. Une évaluation quantitative y est également présentée, suivies d'une appréciation de son extensibilité et de considérations relatives à son impact sur le développement des applications. Nous achevons ce chapitre en examinant les limites et perspectives d'utilisation d'outils existants pour l'automatisation de la synthèse du contrôleur global et l'implantation de la couche de contrôle.

Nous concluons ce manuscrit au chapitre 8, en y évoquant quelques perspectives et pistes issues des résultats de notre recherche de solution au problème du contrôle global de ressources.

PRÉCISION DU PROBLÈME : PREMIER DÉBROUSSAILLAGE

Contenu du chapitre

| | | |
|-------|--|----|
| 1 | Introduction du chapitre | 10 |
| 2 | Les réseaux de capteurs sans fil | 10 |
| 2.1 | Applications | 10 |
| 2.1.1 | Exemples typiques | 11 |
| 2.1.2 | Sur le parallélisme intrinsèque des programmes | 12 |
| 2.2 | Plates-formes matérielles typiques | 12 |
| 2.2.1 | Caractéristiques des microcontrôleurs | 12 |
| 2.2.2 | Détails d'une plate-forme | 14 |
| 2.3 | Contraintes intrinsèques | 15 |
| 2.3.1 | Réactivité | 16 |
| 2.3.2 | Parcimonie d'usage des ressources | 16 |
| 2.3.3 | Fiabilité des nœuds | 16 |
| 3 | Prise en compte de l'énergie dans les réseaux de capteurs sans fil | 17 |
| 3.1 | Éléments impactant la consommation énergétique | 17 |
| 3.2 | Restriction du problème à la gestion locale des nœuds | 17 |
| 3.3 | Vers une gestion avisée des périphériques consommateurs | 18 |
| 4 | Caractérisation des batteries et besoins afférents | 18 |
| 4.1 | Batterie idéale | 18 |
| 4.2 | Première tentative d'investigation : examen de documentations constructeur | 19 |
| 4.3 | Recours aux modèles existants | 19 |
| 4.4 | Bonnes pratiques et besoins | 21 |
| 5 | Gestion des périphériques | 22 |
| 5.1 | Pilotes de périphériques | 22 |
| 5.1.1 | Code opérant | 22 |
| 5.1.2 | Code de contrôle | 22 |
| 5.2 | Implantation des pilotes de périphériques | 23 |
| 5.3 | Vérification statique de propriétés sur les pilotes | 23 |
| 5.4 | Synthèse de pilotes | 24 |
| 5.4.1 | Synthèse du code opérant | 24 |
| 5.4.2 | Synthèse de pilotes complets | 25 |
| 5.5 | Quelques observations | 25 |

1 Introduction du chapitre

Dans ce chapitre, nous précisons le problème de la prise en compte des ressources dans la programmation des systèmes embarqués, en la réduisant à une problématique d'implantation logicielle de nœuds de réseaux de capteurs sans fil. Nous verrons notamment que cette restriction du champ de notre recherche de solution se fait sans perte de généralité, c'est-à-dire que les principes d'une solution au problème pour ces plates-formes s'appliquent également aux systèmes embarqués en général.

Après une présentation des réseaux de capteurs sans fil, de leurs applications potentielles et de leurs caractéristiques intrinsèques, nous abordons en détail l'architecture matérielle des nœuds qui les composent. Nous verrons alors que les stratégies d'augmentation de la durée de vie d'une plate-forme doivent être construites à partir de considérations sur le comportement de sa source d'alimentation en fonction de la demande d'énergie. De plus, une mise en œuvre de mécanismes de prise de décision logiciels conformément à ces stratégies implique la possibilité d'avoir une connaissance de l'état de chaque composant matériel influençant la consommation énergétique, même si celle-ci ne sert que lors de la phase de conception. Nous constatons cependant que les techniques usuelles de conception de logiciels de gestion des périphériques ne permettent bien souvent que des solutions *ad hoc* pour l'obtention de cette connaissance globale. De ces observations, nous dégagons une problématique précise pour laquelle nous détaillerons une proposition de solution dans la suite de cette thèse.

2 Les réseaux de capteurs sans fil

Le principe de construction et les applications des réseaux de capteurs sans fil sont dérivés des réseaux de capteurs filaires initialement employés pour le contrôle des chaînes de production. Le développement des technologies de communication sans fils et la baisse du prix et de la taille des capteurs et microcontrôleurs ont permis l'émergence des réseaux de capteurs sans fil, moins nécessiteux en matière d'infrastructures de déploiement. Peu à peu, de nouvelles applications de ces réseaux ont vu le jour, accompagnées par la diversification des composants matériels adaptés à ces usages.

Ces réseaux comportent de quelques dizaines à plusieurs centaines de *nœuds*, chacun possédant un composant programmable exécutant une application. Le plus souvent, tous les nœuds exécutent le même programme pour simplifier le déploiement du *code* exécutable.

2.1 Applications

Les usages typiques des réseaux de capteurs sans fil regroupent principalement les applications de monitoring de l'environnement, ou certaines tâches de contrôle d'usines.

Les réseaux destinés au monitoring ont souvent en commun de contenir un nœud particulier, appelé *puits*, et connecté à Internet (ou une station de base elle-même possiblement connectée à Internet) et disposant d'une source d'énergie virtuellement illimitée. Puisque la plupart des nœuds du réseau ne sont pas à portée de transmission radio du puits, il est parfois nécessaire de construire une infrastructure réseau permettant à tout nœud susceptible d'initier une transmission de message de l'envoyer jusqu'au puits si l'application le requiert. Dans ces situations, l'un des rôles de chaque nœud du réseau est donc de relayer certains messages jusqu'à ce nœud particulier, ou au contraire de diffuser une requête, commande ou configuration initiée par lui, vers tous les autres nœuds du réseau.

D'autres applications pour lesquelles les nœuds sont mobiles, requièrent par exemple des données sur le positionnement relatif ou absolu des nœuds.

2.1.1 Exemples typiques

Nous citons ici quelques applications originales afin de mettre en avant l'hétérogénéité des usages et situations.

Monitoring scientifique. Parmi les premiers déploiements effectifs, Szewczyk *et al.* [148] étudièrent le comportement d'une variété d'oiseaux sur une île, et Juang *et al.* [95] analysèrent celui de zèbres. Ces applications ont la particularité d'autoriser une période d'échantillonnage des capteurs relativement longue.

Toujours dans le domaine du monitoring scientifique, le projet « Permafrost » [18] dispose des capteurs sur des zones escarpées de montagne en Suisse, afin de surveiller les variations des propriétés géologiques de ces terrains en fonctions de données climatiques comme la température. Pour une telle application, on remarquera que les très basses températures auxquelles sont exposés les nœuds lors du déploiement en haute montagne, impactent le choix des composants matériels. En effet, les conditions environnementales peu usuelles pour des systèmes informatisés auxquelles sont confrontés les nœuds imposent des contraintes non négligeables pour ces choix.

Le monitoring à but scientifique, comme la surveillance de l'activité de volcans décrit par Werner-Allen *et al.* [158] impose parfois des contraintes importantes par rapport au débit des données à récupérer de l'environnement et leur précision. La datation des mesures est aussi un problème récurrent de ce genre d'applications.

Transmission d'alarmes. Un autre exemple typique d'application de réseaux de capteurs sans fil est présenté dans la thèse de Samper [141]. Idéalement, le but de ce type de réseau est d'être déployé au dessus d'une forêt (à partir d'un avion par exemple), et de permettre la transmission d'un message d'alerte vers un poste de secours en cas de température anormale.

Avec une telle application, il est également important de maximiser la probabilité qu'un message d'alerte atteigne effectivement le puits. Une garantie concernant le temps maximum écoulé entre le premier envoi de l'alerte et la réception par la station de base est aussi souhaitable. Dans ce réseau, on notera que la durée de vie des nœuds peu éventuellement ne pas excéder le temps jusqu'à l'occurrence d'un premier incendie, puisque les nœuds qui détectent effectivement peuvent dépenser beaucoup d'énergie à envoyer une alarme si la température est très élevée ; la conception de protocoles de communication très adaptés à ce genre de situations est donc envisageable, voire souhaitable.

Une autre application du même genre sert à détecter un nuage de pollution. Au contraire du cas précédent, la durée de vie du réseau peut s'étendre après la transmission d'une alerte, et les protocoles de gestion de l'infrastructure de communication doivent être conçus en conséquence.

Études d'interactions sociales. Un exemple d'application d'étude d'interactions sociales par réseaux de capteurs sans fil est celle de Friggeri *et al.* [70]. Les auteurs conduisent une expérience de déploiement dans le cadre de l'étude de l'exposition à certaines maladies des personnels soignants d'un service d'hôpital. Pour cela, des nœuds fixes sont disposés dans les différentes chambres du service, et mesurent les temps de présence des personnels portant sur eux des nœuds mobiles. Dans ce cas, les nœuds fixes sont reliés au courant du secteur et écoutent le médium radio en permanence, et les nœuds mobiles émettent périodiquement des messages. Bien que ces propriétés particulières simplifient les protocoles de détection de proximité des nœuds, les résultats des mesures de cette expérience montrèrent que l'usage du signal radio pour effectuer cette mesure est peu précis.

Enclot virtuel. Une autre application originale des réseaux de capteurs sans fil est la « construction » d'un enclot à vaches virtuel par Jurdak *et al.* [96]. Réalisé en Australie, ce travail consiste à remplacer les centaines de kilomètres de clôtures qui nécessitent un entretien régulier, par un réseau de capteurs et actionneurs disposés au cou des vaches. Chacun est également muni d'un récepteur GPS afin de déterminer

la position des vaches, et de savoir leur positionnement par rapport à l'enclot. À mesure que la vache s'approche d'une limite virtuelle, son nœud émet un signal audio sensé la dissuader d'avancer plus.

Le composant GPS consommant une quantité significative d'énergie lorsqu'il est en fonctionnement, les auteurs proposent une méthode de localisation adaptative des nœuds permettant de réduire l'usage de ce périphérique. Une difficulté peu commune que les concepteurs de ce réseau doivent gérer est l'association de la mobilité de tous ses nœuds et le besoin de positionnement absolu. Malgré cela, les propriétés sur le déplacement des vaches (en l'occurrence, qu'elles se déplacent lentement et restent généralement en troupeaux) ont pu être exploitées pour adapter les paramètres du protocole de communication et de l'application afin de limiter les coûts liés à la localisation.

2.1.2 Sur le parallélisme intrinsèque des programmes

À partir des applications typiques détaillées précédemment, on constate aisément qu'il est nécessaire d'être en mesure d'exprimer leur parallélisme intrinsèque lors de leur développement. On observe notamment que chaque nœud des réseaux est supposé assurer au moins deux tâches simultanées :

- Un travail de gestion du réseau de communication entre les nœuds, c'est-à-dire assurer la réception et la retransmission des messages circulant sur le médium radio afin que ceux-ci atteignent effectivement leur destination (*e.g.*, le puits ou tout autre nœud) ;
- Une tâche principale, ou *applicative*, de prélèvement d'informations depuis l'environnement du nœud au moyen de capteurs, et possiblement de pilotage d'actionneurs. Cette tâche transmet éventuellement à la couche de gestion du réseau des messages à communiquer à un ou plusieurs autres nœuds (*e.g.*, un puits, des « voisins », tous le réseau).

On remarquera d'une part que chacune de ces tâches est potentiellement décomposable en plusieurs travaux parallèles, au moins pour ce qui concerne le moment de leur conception. D'autre part, les applications elles-mêmes sont sujettes à un certain degré de parallélisme ; il n'est par exemple pas difficile, pour augmenter la réutilisation des plates-formes matérielles, d'imaginer qu'une application de surveillance de la qualité de l'air puisse fonctionner en parallèle et indépendamment de l'application de détection d'incendies ; il est aussi envisageable d'ajouter la possibilité d'interroger depuis la station de base de tout ou partie des capteurs de température d'une zone géographique donnée.

2.2 Plates-formes matérielles typiques

Un nœuds de réseaux de capteurs sans fil comprend au minimum un *microcontrôleur*, un émetteur-récepteur radio, divers capteurs et parfois quelques actionneurs. Ces composants matériels communiquent au moyen de bus, essentiellement de type série pour réduire la quantité des liens électriques requis sur le circuit imprimé, et donc la surface minimale de ce dernier. Si le nœud n'est pas branché sur le secteur (*i.e.*, le plus souvent), tous sont alimentés au moyen d'une batterie quelquefois rechargeable depuis l'environnement (*e.g.*, énergie solaire, vibrations).

2.2.1 Caractéristiques des microcontrôleurs

Contrairement à un microprocesseur classique, un microcontrôleur intègre sur une seule puce un calculateur (CPU) et un ensemble de périphériques. Puisque principalement destinés à assurer des tâches de contrôle embarqué, les microcontrôleurs ont leurs caractéristiques propres. Les contraintes d'intégration associées au contexte des systèmes embarqués, comme l'espace physiquement disponible, impliquent l'assemblage et le rapprochement d'un ensemble de fonctionnalités matérielles. C'est pourquoi on retrouve généralement au sein d'un microcontrôleur, en plus d'un CPU, un ou plusieurs générateurs d'horloges (oscillateurs à quartz ou résistance-condensateur) et compteurs associés (« *timers* » classiques et chiens de garde — « *watchdogs* »), ainsi que divers modules de mémoire persistante (Flash-ROM) et volatile (RAM). Certains microcontrôleur disposent d'un co-processeur destiné à assurer certaines tâches logicielles

spécifiques (*e.g.*, calculs en virgule flottante). Le fort couplage avec l'environnement physique induit aussi la présence d'un grand nombre de ports d'entrée-sortie série, de convertisseurs analogique-digital (ADC – « *Analog-to-Digital Converter* ») et digital-analogique (DAC – « *Digital-to-Analog Converter* »). Des entrées-sorties digitales génériques (GPIO – dont la valeur peut être logiquement lue ou écrite) sont aussi fournies.

Les modules d'interface série supportent plusieurs protocoles de communication sélectionnables dynamiquement par le logiciel s'exécutant sur le CPU. Notamment, les protocoles « *Inter Integrated Circuit* » I²C [125], RS232 [62], ou « *Serial Peripheral Interface* » SPI [120] sont très souvent supportés par au moins un de ces modules. En revanche, un même module ne peut piloter qu'un seul bus à la fois.

Requêtes d'interruptions matérielles. Une demande d'un traitement de requête d'interruption du CPU peut provenir d'un module périphérique inclus dans le microcontrôleur (*e.g.*, expiration d'un timer) ou extérieur à celui-ci (*e.g.*, évènement à prendre en compte depuis un émetteur-récepteur radio), voire du calculateur lui-même (*e.g.*, instruction invalide). En conséquence, le CPU effectue un déroutement, *a priori* temporaire, du flot d'exécution des instructions et sauvegarde d'un contexte minimal en mémoire de travail. Celui-ci permet ensuite au calculateur de continuer l'interprétation des instructions à partir de celle suivant l'interruption lorsque le traitement de la requête se termine.

Usuellement, les requêtes de déroutement peuvent être masquées individuellement (*i.e.*, chacune est ignorée tant que le masque correspondant est positionné) ou globalement (*i.e.*, toutes les interruptions non masquées individuellement sont alors autorisées à survenir, ou non). De plus, des priorités statiques sont associées à chaque source d'interruption pour spécifier lequel des traitements est exécuté lorsque plusieurs requêtes surviennent simultanément.

Modes de basse consommation. Les microcontrôleurs peuvent posséder plusieurs modes de fonctionnement ayant chacun des spécificités, au regard du comportement comme de la consommation énergétique. En effet, certains microcontrôleurs ne disposent pas uniquement des états de calcul normal et d'arrêt total dans lequel ils consomment très peu d'énergie¹ : ils peuvent parfois supporter des états de moindre consommation, où en plus du CPU, des horloges et oscillateurs internes sont également désactivés. Bien évidemment, à un instant donné, le nombre de modules matériels hors tension (*e.g.*, si l'horloge qui les cadence est elle-même désactivée) est corrélé avec une baisse de la consommation énergétique du microcontrôleur complet.

En outre, il est important de noter que comme pour un microprocesseur standard, d'un point de vue logiciel, un réveil du CPU est nécessairement la conséquence d'une requête d'interruption :

- Lors de la mise sous tension du microcontrôleur, le logiciel commence à s'exécuter en réponse à une interruption dédiée (souvent nommée « *system reset* »). Cette interruption *non masquable* peut également être câblée et configurée de manière à être générée lors d'une intervention extérieure (*e.g.*, pression sur un bouton) ou lors de l'expiration d'un chien de garde. Aussi, certains microcontrôleurs possèdent un module interne dit de « *brownout reset* », capable de déclencher cette même procédure de redémarrage lorsque la tension d'alimentation du circuit baisse sous un seuil critique ;
- Lorsque le calculateur est placé dans un mode de basse consommation à l'initiative du logiciel, l'exécution du programme est immédiatement stoppée, et ne peut continuer qu'après que le CPU a été réveillé par un signal externe. Un tel signal peut être généré par un module matériel intégré au microcontrôleur (*e.g.*, module d'entrée-sortie série, timer ou ADC), ou bien par un périphérique extérieur connecté à une entrée digitale générique configurée pour déclencher des interruptions.

1. Un microcontrôleur à l'arrêt mais sous tension consomme en principe uniquement l'énergie nécessaire au fonctionnement de son système de démarrage.

Absence de mécanismes de gestion de la mémoire. Les microcontrôleurs se caractérisent en outre par l'absence de mécanismes matériels avancés. Entre autres, ils ne comprennent généralement pas d'unité de gestion mémoire (MMU — « *Memory Management Unit* »), outil pourtant intégré aux microprocesseurs depuis 1985–1986, et permettant notamment d'assurer la virtualisation et l'isolation d'espaces en mémoire vive. Cette absence entraîne des complications lors de la programmation des microcontrôleurs.

En effet, la virtualisation de la mémoire est destinée à donner l'illusion au programmeur qu'il dispose d'un espace mémoire *directement adressable* plus grand qu'il ne l'est en réalité. Or, ce mécanisme requiert, en plus d'un support système conséquent, un matériel traducteur d'adresses virtuelles.

Aussi, sans protection matérielle d'espaces en mémoire, des erreurs logicielles que l'on rencontre pourtant encore souvent lorsque l'on programme dans un environnement classique deviennent encore plus difficilement détectables. Par exemple, des erreurs telles qu'un déréférencement de pointeur invalide ou l'accès à un espace mémoire incorrect (*i.e.*, n'appartenant pas conceptuellement à l'« unité de code s'exécutant », *e.g.*, une tâche) se manifestent à l'exécution, lorsqu'elles le font, par une corruption de données ou des instructions elles-mêmes. Les conséquences que l'on peut détecter n'apportent alors généralement que très peu d'indices sur la localisation et le moment de l'erreur initiale².

Remarque sur le fonctionnement des périphériques intégrés. L'ensemble des modules intégrés à un microcontrôleur est cadencé à partir de signaux d'horloges, en général générés par des oscillateurs (*e.g.*, quartz) internes ou externes. Les fréquences de ces horloges peuvent varier, mais l'une d'elles sert à cadencer le CPU. Aussi, le signal d'horloge cadencant chaque module est configurable logiciellement.

Cependant, selon le mode de fonctionnement du microcontrôleur, certains signaux d'horloge sont désactivés, et les modules cadencés par ceux-ci ne remplissent alors plus leur fonction jusqu'au retour en mode actif. Cette propriété a l'avantage de réduire la consommation énergétique totale du microcontrôleur lorsqu'il est dans un mode de basse consommation ; en revanche, le temps requis pour reprendre les calculs au réveil dépend aussi des horloges désactivées.

Au surplus, puisque certains modules intégrés sont inactifs, alors il importe que le microcontrôleur ait toujours un moyen de retourner en mode actif, c'est-à-dire qu'un périphérique non désactivé (interne ou externe) soit en mesure d'émettre une requête d'interruption.

2.2.2 Détails d'une plate-forme

Les nombreux exemples de plates-formes matérielles de nœuds de réseaux de capteurs sans fil incluent par exemple Telos [126], RatMote [20] et bien d'autres³. Tous ces nœuds partagent les propriétés décrites précédemment, avec parfois quelques spécificités liées aux applications pour lesquels ils sont conçus.

La figure 2.1 page suivante est une représentation de la plate-forme pour réseaux de capteurs sans fil matérielle Wsn430 [69]. Ce nœud est piloté par un modèle de microcontrôleur, réputé pour sa faible consommation énergétique et très répandu dans ce type de systèmes, le MSP430 de Texas Instruments [151]. Ce microcontrôleur est construit selon une architecture de von Neumann classique, avec un espace d'adressage partagé entre les différents périphériques intégrés. La largeur des adresses est globalement de 16 bits, autorisant un espace directement accessible de 64ko (une partie des périphériques intégrés ne sont cependant connectés que par des bus de 8 bits). Son calculateur intégré interprète un jeu d'instructions réduit, comprenant 27 opérations différentes, et 7 modes d'adressage ; ses registres sont larges de 16 bits. Ce

2. Quoique plus répandues dans les programmes directement écrits en langage d'assemblage, C ou C++, on en retrouve malheureusement encore trop souvent, bien que sous d'autres formes, dans les programmes implantés dans des langages d'un peu plus haut niveau comme Java. C'est dire si la probabilité d'occurrence d'une telle erreur reste non négligeable même sur microcontrôleur si l'on ne programme pas avec des outils et langages adaptés, ou au moins avec une certaine rigueur.

3. Le site collaboratif « *The Sensor Network Museum* » (<http://www.snm.ethz.ch/Main/HomePage>), hébergé à l'ETH Zürich, en recensait une trentaine fin 2011. Bien que construite à partir des recensement publiés et par divers contributeurs du domaine, cette liste est manifestement incomplète puisqu'il y manque encore à cette date RatMote et la plate-forme Wsn430 que nous détaillons ici. On remarque donc une grande diversité d'architectures existantes.

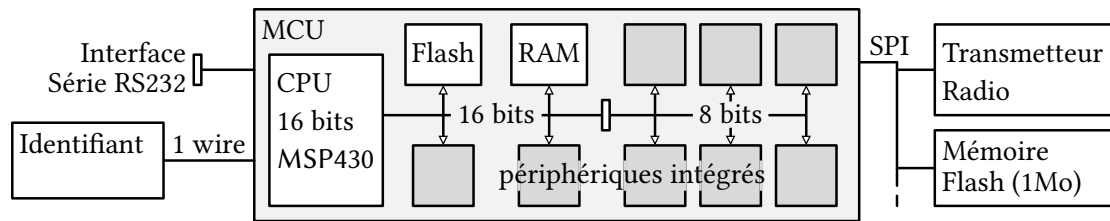


FIGURE 2.1 – Représentation de la plate-forme matérielle Wsn430 – directement inspirée d'un schéma de Fraboulet et al. [69].

microcontrôleur dispose de 10ko de mémoire vive et 48ko de mémoire de code Flash/ROM, reprogrammable. Entre autres périphériques, il intègre deux timers, des convertisseurs analogique-digital et digital-analogique de précisions variables (10 à 12 bits). Il comprend aussi deux modules d'entrée-sortie série, et supporte ainsi les trois protocoles listés dans la section 2.2.1 page 12.

Son horloge principale (qui cadence le CPU) supporte une fréquence configurable pouvant atteindre 8MHz.

Ce microcontrôleur possède six modes de fonctionnement, parmi lesquels :

- Le mode *actif* où le CPU et tous les modules configurés sont actifs. La puissance électrique consommée typique dans ce mode est de $250\mu\text{A}$ si l'horloge principale est cadencée à 1MHz, mais varie légèrement selon la tension d'alimentation et le nombre de périphériques actifs ;
- Le mode *LPM4* dans lequel le CPU et tous les modules sont inactifs. Le contenu de la mémoire de travail est effacé si le microcontrôleur entre dans ce mode ; il ne peut alors retourner en mode actif que par un changement d'état de sa broche de redémarrage, souvent reliée à un bouton poussoir. La puissance électrique consommée est réduite au strict minimum ($\approx 0, 1\mu\text{A}$) ;
- Le mode *LPM3* de consommation faible ($\approx 2\mu\text{A}$ à 3V), dans lequel une seule horloge interne reste active.

La plate-forme Wsn430 comprend, parmi ses périphériques externes, un émetteur-récepteur radio CC1100 [40], un module de 1Mo de mémoire Flash [146] et un capteur de température [45], tous reliés au microcontrôleur par l'intermédiaire d'un même bus SPI. Un capteur de luminosité [149] et un composant fournissant un numéro de série unique lisible par le logiciel s'exécutant sur le microcontrôleur [46], sont aussi inclus sur le circuit. (La disponibilité d'un composant matériel procurant un identifiant unique pour chaque nœud est très utile pour les différencier à moindre coût, notamment pour la génération d'adresses simplifiant l'infrastructure de gestion du réseau. Ainsi, une application peut être implantée dans la mémoire de programme de chaque nœud sans nécessiter de configuration particulière.)

Un nœud Wsn430 peut être alimenté en énergie de plusieurs manières, y compris directement par le secteur au moyen d'un adaptateur. Une batterie au Lithium rechargeable est fournie [153], mais une pile standard est également utilisable.

2.3 Contraintes intrinsèques

Des propriétés des applications des réseaux de capteurs sans fil et des plates-formes matérielles des nœuds qui les composent découlent un ensemble de contraintes qui doivent être prises en compte lors de leur développement. Ainsi, la combinaison du caractère distribué, des contextes de déploiement diversifiés, et de l'hétérogénéité des plates-formes matérielles, introduit des exigences pour l'implantation logicielle. On remarquera que certaines des contraintes de programmation logicielle des nœuds de ces réseaux sont partagées avec la classe des systèmes embarqués en général.

2.3.1 Réactivité

Parmi l'ensemble des systèmes interagissant avec leur environnement⁴, Harel et Pnueli [84] ont défini les *systèmes réactifs* par opposition aux *systèmes conversationnels* ou *interactifs* dans lesquels l'ordinateur contrôle lui-même sans contrainte toutes ses interactions avec son environnement. Les systèmes réactifs doivent réagir de façon continue aux stimuli provenant de celui-ci ; dans cette classe de systèmes les délais de réaction autorisés sont limités par la nature de l'environnement contrôlé.

Même si certaines applications des réseaux de capteurs sans fil semblent relever en première approche de systèmes conversationnels (e.g., interrogation de mesures physiques à l'initiative d'un opérateur), on peut considérer que les délais de réponse à respecter lors de l'implantation de systèmes communicants par radio placent les composants de ces réseaux dans la classe des systèmes réactifs. En effet, il est permis de définir l'environnement d'un nœud comme comprenant à la fois le système physique qu'il mesure, mais aussi l'ensemble des autres appareils avec lesquels il communique. Aussi, pour prendre un exemple applicatif, une contrainte sur le temps minimal de transmission d'une alarme à un puits introduit une astreinte forte, tant sur la fréquence d'échantillonnage d'un capteur que sur le temps de transfert d'un message radio.

2.3.2 Parcimonie d'usage des ressources

De par faible taille mémoire de travail et la vitesse de calcul limitée, ainsi que la quantité d'énergie disponible pour chaque nœud alimenté par batterie uniquement, une certaine parcimonie est de mise lors de la programmation logicielle des nœuds. Une implantation logicielle trop gourmande en énergie épuise la batterie trop vite et réduira la « durée de vie du réseau » (i.e., la mort d'un nœud peut être définie comme le moment où il ne remplit plus sa fonction et ne communique plus avec d'autres nœuds, mais la définition de la mort d'un réseau dépend trop de l'application pour être définie simplement). En outre, les situations de déploiement ne permettent pas toujours un remplacement de la batterie, ou le rendent difficile.

L'implantation du logiciel s'exécutant sur les nœuds doit donc se faire de manière à limiter l'usage de l'ensemble ces ressources.

2.3.3 Fiabilité des nœuds

Selon l'application, un haut niveau de fiabilité doit être assuré. Pour le monitoring scientifique ou la détection des incendies par exemple, il est important de pouvoir garantir le bon fonctionnement du réseau pendant un temps spécifié à l'avance pour ne pas compromettre l'expérience ou y attribuer une confiance injustifiée.

Si cette fiabilité n'est pas assurée par des redondances de nœuds pour tolérer un certain nombre d'erreurs ou de pannes, alors elle relève en premier lieu du bon comportement des nœuds individuels, donc du programme qui les gère. Dans tous les cas, de la fiabilité du logiciel s'exécutant sur les nœuds dépend la fiabilité du réseau complet.

Au surplus, sans support adéquat pour la reprogrammation dynamique, les bogues sont difficiles à corriger *a posteriori*, surtout si tous les éléments d'un réseau doivent être mis à jour.

Ainsi, la problématique de prise en compte de l'énergie et la sûreté de programmation sont primordiales dans l'implantation des nœuds de réseaux de capteurs sans fil. L'aspect rudimentaire et le peu de support matériel avancé disponible sur ces plates-formes en font un cadre privilégié pour identifier des méthodes génériques pour considérer ces problématiques.

4. Le reste des systèmes, c'est-à-dire ceux qui calculent uniquement une sortie en fonction d'une entrée, sont dits *transformationnels*. On retrouve dans cette dernière classe les compilateurs par exemple.

3 Prise en compte de l'énergie dans les réseaux de capteurs sans fil

Dans le contexte des réseaux de capteurs sans fil, le problème de la considération de la consommation énergétique se pose à plusieurs niveaux. L'objectif de cette section est de restreindre le champ de notre investigation à la gestion des nœuds indépendamment les uns des autres, en prenant surtout en compte l'impact de la gestion des périphériques qui les composent sur la durée de vie du nœud complet.

3.1 Éléments impactant la consommation énergétique

Consommation globale du réseau. En ce qui concerne l'infrastructure réseau, les différents protocoles de communication utilisés déterminent en partie les temps d'activité des quelques périphériques de transmission et, si ceux-ci n'incorporent pas en interne de mécanismes de retransmission automatique, du microcontrôleur également.

Ce domaine d'investigation relève de la conception de protocoles de gestion du réseau efficaces et supposent avant tout que les nœuds sont programmés efficacement. Il existe déjà beaucoup de propositions qui vont en ce sens, tant en ce qui concerne le routage des messages dans le réseau complet [85, 145] que pour le contrôle d'accès au médium radio [3, 91, 103, 161]. Des approches mixtes prenant en compte plusieurs de ces éléments sont également à l'étude, récemment par Heurtefeux *et al.* [86] par exemple.

Consommation locale des nœuds. Au niveau des plates-formes des nœuds du réseau, la qualité et les propriétés physiques des composants matériels qui les constituent impactent leur efficacité énergétique ; par exemple, la consommation liée aux différents modes de fonctionnement du microcontrôleur influence directement sur la durée de vie de chaque nœud.

La manière dont tous ces périphériques sont gérés par la couche logicielle dédiée à cette tâche a aussi un impact sur la consommation globale d'un nœud. Les différents états d'un émetteur-récepteur radio ont en effet des consommations énergétiques variables [40] ; les changements d'états de introduisent également des changements temporaires de la consommation du composant.

Au surplus, l'application déployée a un impact important sur la consommation énergétique du nœud : certaines contraintes, plus ou moins fortes, sont issues de sa spécification (ce que nous appelons la *logique applicative*), et jouent un rôle déterminant sur les fréquences de mise en fonctionnement des périphériques et du microcontrôleur, et, par là, sur la consommation énergétique du nœud complet. Une telle contrainte est par exemple issue des périodes d'échantillonnage de capteurs, qui dépend en général de la propriété physique mesurée et de la précision recherchée. Les protocoles employés pour la gestion du réseau imposent également des limites aux possibilités d'optimisation de la consommation énergétique de chaque nœud.

3.2 Restriction du problème à la gestion locale des nœuds

Dans cette thèse, nous nous intéressons principalement à la recherche d'une solution d'implantation tenant compte de l'énergie pour les systèmes embarqués en général. C'est pourquoi la composante relative aux protocoles de gestion du réseau, bien que primordiale dans ce domaine d'applications, sera mise de côté au profit d'une approche de gestion d'énergie dédiée à l'architecture logicielle des nœuds.

En outre, les choix des composants matériels sont un domaine pour lequel nous ne pouvons d'un point de vue logiciel, que définir des guides en termes fonctionnalités. La logique applicative est invariable (*i.e.*, définie avant le développement et imposée) et la seule contribution que nous pouvons envisager à ce niveau est également la définition de « bonnes pratiques » pour sa conception.

En revanche, il nous est possible d'imaginer des solutions en termes de techniques d'implantation et de gestion logicielle des divers composants des nœuds.

3.3 Vers une gestion avisée des périphériques consommateurs

Comme nous l'avons vu plus haut, les plates-formes matérielles que nous considérons sont en général alimentées au moyen de batteries, qui servent à faire fonctionner l'ensemble des périphériques. En un certain sens, la puissance fournie par la batterie à chaque instant peut être considérée comme constituant une ressource partagée au même titre que le temps d'activité du calculateur. Ainsi, prendre en compte l'énergie dans le développement des logiciels de gestion locale des nœuds de réseaux de capteurs sans fil implique de considérer simultanément l'impact de l'ensemble des composants sur cette source d'énergie commune.

Pour cette raison principalement, il est nécessaire d'avoir un minimum de connaissances relatives au comportement des batteries employées sur les nœuds pour identifier les propriétés (singulières, comme nous le verrons) de cette ressource. Nous devrions être ensuite en mesure de déduire quelques pistes vers une méthode de programmation tenant compte de ces dernières.

4 Caractérisation des batteries et besoins afférents

Afin d'identifier des pistes vers une conception des nœuds de réseaux de capteurs sans fil tenant compte de l'énergie, nous détaillons dans cette section quelques caractéristiques des batteries qui sont usuellement employées sur ces plates-formes. L'objectif est ici d'identifier ce que nous pourrions appeler de « bonnes pratiques » dans l'usage des batteries, ou encore de déterminer des cas d'utilisation potentiellement indésirables qui influeraient négativement sur sa durée de vie. Nous définirons les comportements réels des batteries par comparaison aux caractéristiques d'une batterie idéale.

Profil de consommation. Un *profil de consommation*, ou *profil de décharge* représente la variation au cours du temps de l'intensité du courant consommé. Un profil dit « constant » illustre trivialement une séquence de temps pendant laquelle l'intensité ne varie pas.

4.1 Batterie idéale

Une batterie idéale de capacité totale C a un comportement linéaire, c'est-à-dire qu'elle est considérée comme un réservoir d'énergie fournissant toujours une quantité prédéterminée de courant, quelle que soit le profil de consommation. Si l'on décharge une telle batterie avec un courant constant d'intensité I , alors la durée de vie L au bout de laquelle sa capacité effective devient nulle est calculable par :

$$L = \frac{C}{I}$$

Tout au long de sa durée de vie, la tension aux bornes d'une batterie idéale chute proportionnellement à sa capacité restante, et atteint un seuil lorsqu'elle est vide. Ainsi, elle fournit un moyen simple d'évaluer son niveau de décharge.

Avec une batterie idéale, nous conviendrons que le seul impact que peut avoir le logiciel s'exécutant sur un nœud de réseau de capteurs est de minimiser l'énergie consommée par les différents périphériques (c'est-à-dire pour chaque périphérique, l'intégrale du produit It , pour t le temps passé à consommer un courant d'intensité I).

Cependant, ce modèle ne reflète pas fidèlement les comportements effectivement observés dus aux propriétés physiques et chimiques des batteries. Par exemple, la vitesse des réactions chimiques qui ont lieu n'est pas infinie (*i.e.*, la résistance intrinsèque de la batterie n'est pas nulle), et l'intensité du courant instantanément consommable n'est pas directement calculable uniquement à partir d'une capacité totale C .

D'autres informations relatives aux comportements des batteries sont nécessaires pour définir de bonnes pratiques pour leur usage.

4.2 Première tentative d'investigation : examen de documentations constructeur

La première source à consulter lorsque l'on recherche des informations techniques sur un élément intégré à une plate-forme matérielle est bien évidemment la documentation fournie par un constructeur. Ces documents sont construits à partir de mesures effectuées par les constructeurs eux-mêmes sur les produits qu'ils vendent. Ainsi, les remarques suivantes peuvent être trouvées à la page 11 d'une documentation technique traitant de batteries « Alkaline Manganese Dioxide » [63], une technologie susceptible d'être utilisée pour alimenter des nœuds de réseaux de capteurs sans fil :

« In pulse applications, the duty cycle can impact battery capacity. A very light duty cycle will typically allow the battery time to recover and extend service versus a continuous drain. »

Outre quelques indications quantitatives comparant des énergies cumulées obtenues jusqu'à épuisement de la batterie, selon qu'elle est déchargée de manière continue ou non, peu d'informations précises peuvent être obtenues de ce document : on remarque principalement que le ratio des charges totales effectivement consommées varie approximativement de 20% à 250mA, à 30% à 1A, en faveur d'un profil de décharge périodique de 10s de consommation toutes les 100s (à consommation nulle le reste du temps), par rapport à un profil continu de même intensité.

Une première indication nous est donc donnée, relative à un comportement de ce type de batteries qui induirait un allongement substantiel de leur durée de vie lorsqu'elles sont utilisées par intermittence plutôt que de manière continue. Nous apprenons de plus que des phases de « repos » (*i.e.*, absence de consommation ou presque) permettraient de maximiser cet effet.

On découvre de plus dans ce document que la tension ne varie pas linéairement avec la capacité comme le ferait une batterie idéale, et ne reflète donc pas fidèlement sa charge restante (page 10) : en général, la tension commence par chuter rapidement vers un niveau comparativement plus stable, puis décroît de plus en plus vite lorsque la capacité restante dépasse un certain seuil. Ce comportement observé nous indique une fois encore qu'il n'est pas évident d'évaluer la durée de vie restante des batteries, même à partir de mesures directes sur celles-ci.

En suivant la même méthode d'investigation pour des batteries rechargeables « Nickel Metal Hydride » [64] ou encore « Lithium Ion » [75], nous observons l'absence de données significatives pour leur utilisation⁵. Au surplus, les fiches techniques dédiées aux performances des différents produits que nous avons trouvés, y compris concernant la batterie « Lithium Ion » fournie avec les nœuds Wsn430 décrits dans la section 2.2.2 page 14 [153], ne présentent que des mesures de durées de vie à partir de décharges à consommation constante ; il est en conséquence difficile d'évaluer l'impact des variations de consommation. Enfin, on remarque également la non linéarité de la mesure de tension directe par rapport à la charge restante.

À partir de ces quelques observations, nous remarquons qu'il est nécessaire de rechercher d'autres sources d'information pour identifier les comportements des batteries. Heureusement, il existe une communauté de chercheurs créant des modèles afin de prédire la durée de vie de ces sources d'énergie, et nous pouvons nous servir des caractéristiques de ces derniers pour tirer des conclusions plus générales concernant un usage conseillé de ces ressources.

4.3 Recours aux modèles existants

Pour obtenir des informations plus précises, nous pouvons en effet tirer partie de la vaste littérature existante concernant les modèles de batteries conçus pour estimer le plus précisément possible le comportement et la durée de vie de celles employées dans les réseaux de capteurs sans fil ou les systèmes autonomes

5. Nous remarquons cependant des comportements communs qui se manifestent à des degrés divers, mais qui impactent surtout le choix de la technologie de batteries suivant les applications et conditions de déploiement ; outre le prix, l'effet d'auto-décharge à long terme, la réaction aux pics de consommation (courant instantané maximum) ou encore l'impact de la température sur leur efficacité, varient selon les technologies.

plus généraux. Dans cette section, nous passons en revue quelques-uns de ces modèles en nous basant principalement sur le travail de Rao *et al.* [136], afin d'identifier certaines caractéristiques des batteries que nous pourrions exploiter, et en déduire ou repérer d'éventuels conseils d'usage.

Il existe différentes catégories de modèles de batteries, chacune variant en matière de précision et complexité calculatoire ou de mise en œuvre. Les modèles proposés que nous mentionnerons capturent plus ou moins bien les phénomènes physico-chimiques ayant lieu lors des différentes phases d'utilisation des batteries (y compris les périodes d'inactivité, et les successions de cycles recharge-décharge pour certains).

Un modèle empirique simple. Peukert a intégré dans son modèle la dépendance qu'il a observée entre l'intensité du courant consommé et la durée de vie de la batterie [109] — cet effet est usuellement appelé « *rate-dependency effect* » en anglais. Pour cela, il a proposé la loi suivante pour exprimer cette mesure de la durée de vie L :

$$L = \frac{C}{I^n}$$

De même que lors de la description de la batterie idéale (*cf.* § 4.1 page 18), C et I représentent respectivement la capacité théorique de la batterie et l'intensité du courant, considérée comme constante dans ce modèle. L'exposant n est le nombre de Peukert, qui dépend des propriétés de la batterie et de la température, et est à déterminer empiriquement. Il varie généralement entre 1,1 (pour une batterie de bonne qualité) et 1,4.

Ce modèle reflète bien le fait que la dépendance de la durée de vie par rapport à l'intensité du courant est souvent non négligeable : même pour $n = 1,1$, il devient parfois plus intéressant, à énergie totale cumulée constante, d'étaler dans le temps une consommation d'intensité plus faible. Cependant, la loi de Peukert ne décrit pas le comportement de récupération pendant les temps de repos évoqué dans la section 4.2 page précédente. Telle quelle, elle ne prend pas non plus en compte les effets possibles des variations d'intensité (*i.e.*, calculer une moyenne sur I ne suffit pas).

D'autres modèles ont été proposés pour remédier aux problèmes d'imprécision de cette loi ainsi que pour couvrir plus de cas. Parmi ceux-ci, les modèles cinétiques sont les plus précis, mais aussi les plus complexes.

Modèles cinétiques. Très précis, les modèles cinétiques de batteries prennent en compte les caractéristiques physico-chimiques des constituants des batteries et les propriétés de son environnement comme la température (un exemple de modèle cinétique est celui proposé par Doyle *et al.* [53], pour des cellules de batteries de type Lithium-Polymère). Les estimations de propriétés quantifiables de batteries sont obtenues par résolution numérique de systèmes d'équations aux dérivées partielles.

Ces modèles, développés par des spécialistes de ce domaine, requièrent généralement la spécification d'un très grand nombre de paramètres (parfois plus d'une cinquantaine), dépendant des propriétés physico-chimiques des composés et du procédé de construction de la batterie. Par conséquent, ils sont bien trop complexes pour que nous puissions en déduire quelque information utile pour un développement logiciel tenant compte de l'énergie. De par le grand nombre des paramètres impliqués et leur complexité, il est difficile d'en déduire des tendances générales sur le comportement des batteries.

Modèles abstraits et intermédiaires. Moins complexes, les modèles dits « abstraits », quant à eux, décrivent le comportement des batteries à partir d'une représentation analytique construite à partir de circuits électriques équivalents. Des modèles de ce type ont déjà été utilisés pour déterminer des politiques de gestion dynamique d'énergie ; comme Benini *et al.* [10] par exemple.

C'est également le cas de modèles empiriques plus précis que la loi de Peukert (*e.g.*, celui proposé par Pedram et Wu [124]), ou encore de modèles analytiques mixtes qui autorisent l'identification de comportements suffisamment abstraits en fonction d'un profil de décharge donné. Par exemple, Rakhmatov et Vrudhula [133] proposent l'usage de modèles mixtes pour guider l'ordonnancement dynamique de tâches dans le cadre de systèmes portables autonomes.

Rahmé *et al.* [131] s'inspirent de ce même modèle pour établir une méthode d'estimation en ligne de la quantité d'énergie restante, applicable selon eux aux nœuds de réseaux de capteurs sans fil. Nous notons que les expérimentations qu'ils ont menées leur ont également permis de déterminer que le modèle de la batterie idéale n'est pas réaliste dans le contexte des réseaux de capteurs sans fil, du fait de l'impact des effets qu'il ne prend pas en compte.

Rao *et al.* [135] utilisent encore ce type de modèles pour évaluer le compromis entre une politique d'optimisation basée sur la maximisation de la durée de vie de la batterie (*i.e.*, introduire des temps de pause longs pour bénéficier au mieux de ses effets intrinsèques), ou directement orientée vers la minimisation de l'énergie consommée (*i.e.*, étaler dans le temps des consommations d'intensité moindre). Les auteurs observent que, si le système fonctionne par intermittence avec de longs temps de pause, il est préférable de n'optimiser que la consommation énergétique sans tenir compte du comportement de la batterie. Nous ferons cependant remarquer que cette étude est réalisée dans le cadre de systèmes généraux alimentés par batteries (ordinateurs portables), et les temps de pause considérés sont de l'ordre de quelques dizaines de minutes.

De certains de ces modèles et de leurs applications dans le cadre de la conception de politiques de gestion dynamique de l'énergie, il ressort finalement que l'effet de récupération de capacité (« *recovery effect* ») évoqué dans la section 4.2 page 19 et pris en compte par les modèles ci-dessus, peut effectivement être exploité pour prolonger la durée de vie d'un système alimenté par batteries. Lorsque ce n'est pas le cas, alors une politique consistant à la réduction de l'énergie consommée en étalant les dépenses dans le temps est potentiellement efficace.

4.4 Bonnes pratiques et besoins

Dans cette section, nous déduisons de ce qui précède de bonnes pratiques d'implantation du logiciel qui gère les plates-formes matérielles, pour tenter d'exploiter au mieux les comportements singuliers des batteries, ou bien pour faciliter l'implantation de politiques de réduction de la consommation énergétique.

Fiabilisation matérielle. Des aménagements matériels sont usuellement incorporés aux plates-formes pour éviter autant que possible les chutes de tension lors d'une demande de courant dépassant l'intensité délivrable par la batterie (*e.g.*, condensateurs ou contrôleurs de charge). De plus, ces équipements servent également à s'accommoder de la réduction, graduelle mais non linéaire, de la tension aux bornes de la batterie, ou encore à évaluer son niveau de décharge.

Impact du logiciel. Cependant, de par la variabilité de la consommation des différents composants matériels présents sur une même plate-forme, surtout en fonction de leurs modes de fonctionnement respectifs, alors le comportement induit par le programme s'exécutant sur le microcontrôleur et la manière dont celui-ci gère ces divers éléments ont un impact non négligeable sur la durée de vie de la batterie.

Il nous paraît donc nécessaire d'identifier les actions du logiciel ayant des conséquences potentiellement néfastes sur les sources d'énergie pour être en mesure de les prévenir ou de limiter leurs conséquences. Pour cela, les comportements intrinsèques des batteries évoqués précédemment indiquent que plusieurs pistes sont envisageables suivant les importances relatives des effets de dépendance à la décharge ou de récupération en phase d'inactivité.

Même dans le cas de l'absence de considération du comportement intrinsèque des batteries, une connaissance de l'état de consommation de chaque composant matériel est nécessaire pour mettre en œuvre une politique de gestion directement axée sur la réduction de la consommation énergétique.

De la nécessité du contrôle global des systèmes. Pour être efficace dans cette prise en compte des comportements des batteries, ou encore pour l'implantation de politiques de réduction de la consommation

énergétique, nous avançons qu'il est nécessaire d'assurer un contrôle global de la consommation des divers composants matériels. En effet, ceux-ci sont le plus souvent tous alimentés par la même batterie.

En conséquence, l'action combinée de plusieurs périphériques commandés indépendamment peut provoquer un pic de consommation qu'il serait préférable d'éviter dans le cadre d'une politique de réduction de l'énergie consommée par exemple.

En outre, pour profiter momentanément de l'effet de récupération (ou de tout autre comportement non évoqué ici) lorsque la batterie est sujette à de tels effets, il est évidemment important de disposer d'une connaissance centralisée de l'état global de la plate-forme : il devient alors possible lorsque la logique applicative s'y prête, de décider dynamiquement au niveau logiciel d'étaler dans le temps les opérations énergétiquement coûteuses ou d'introduire des temps de pause ou de consommation réduite, selon les situations.

Pour étudier les possibilités de mise en œuvre des bonnes pratiques que nous avons identifiées, et notamment l'implantation de politiques de contrôle global des systèmes, nous avons dans un premier temps besoin d'examiner les pratiques actuelles dans le domaine de la gestion des périphériques. Cette étude est l'objet de la prochaine section.

5 Gestion des périphériques

Usuellement, la partie de la pile logicielle la plus proche du matériel et dédiée à sa gestion est composée de *pilotes de périphériques*.

5.1 Pilotes de périphériques

Un *pilote de périphérique* est une portion de logiciel dédiée à la gestion d'un composant matériel particulier. Son rôle est de fournir une abstraction de ce dernier, afin que les couches logicielles supérieures puissent manipuler le périphérique au moyen de primitives simplifiées fournies par le pilote. En général, les systèmes d'exploitation définissent un ensemble d'interfaces génériques pour différentes *classes de périphériques* (e.g., stockage persistant, communication réseau). Ces interfaces définissent les abstractions que doit fournir un pilote en fonction de la classe du composant qu'il gère.

Les pilotes sont directement utilisés par les *services de niveau système*, un « ordonnanceur » de tâches, un système de gestion de fichiers ou des communications avec d'autres systèmes informatisés, par exemple.

Un ensemble de pilotes de périphériques doit être développé pour qu'un système d'exploitation existant puisse s'exécuter sur une nouvelle plate-forme.

5.1.1 Code opérant

Le *code opérant* d'un pilote est la portion de code assurant le dialogue avec le périphérique. Son rôle est de manipuler les ports et registres d'entrées-sorties dont les lectures et écritures ont un impact sur le comportement du composant matériel piloté. Ce code opérant peut également envoyer des commandes à un pilote de bus (ou de DMA — « *Direct Memory Access* ») pour réaliser ces actions si le périphérique piloté n'est adressable qu'indirectement (ou dans le cas du DMA, pour réaliser des transferts volumineux sans monopoliser le CPU).

5.1.2 Code de contrôle

Le *code de contrôle* a pour objectif d'interpréter les commandes reçues depuis les autres parties du système. Ces commandes arrivent sous forme d'appels de routines, en provenance du périphérique lui-même (i.e., en cas de requêtes d'interruption), ou encore depuis les couches logicielles supérieures. Les premières notifient

un changement d'état du composant matériel, alors que les secondes sont potentiellement des demandes de changement de mode de fonctionnement de celui-ci.

La partie du pilote recevant ces requêtes assure le contrôle du comportement du périphérique, et réalise les opérations demandées au travers du code opérant.

5.2 Implantation des pilotes de périphériques

Depuis les premiers systèmes d'exploitation pour lesquels les pilotes étaient directement programmés en langage d'assemblage, puis le passage à la programmation en C, relativement peu d'évolutions ont été observées dans la définition d'abstractions et langages adaptés à une construction de pilotes de périphériques plus aisée et sûre, si on la compare avec l'augmentation de la complexité des systèmes. Quel que soit le domaine d'application, et même pour des contextes peu critiques, on note cependant un certain intérêt pour la recherche de méthodes et outils permettant d'obtenir une confiance relative en cette partie des piles logicielles dont dépend le bon fonctionnement des systèmes. Il importe en effet de s'assurer que les périphériques sont correctement gérés individuellement avant de les assembler, puis construire et tester le reste des services du système et les applications.

Dans la suite de cette section, nous cherchons à identifier les moyens à notre disposition pour la mise en œuvre de mécanismes de contrôle global des composants des plates-formes. Nous étudions pour cela les méthodes et outils existants pour la conception des pilotes.

Les recherches que nous citons portent principalement sur des systèmes qui semblent généralement d'architecture plus complexe que des nœuds de réseaux de capteurs sans fil. Cependant, les méthodes de développement des pilotes ne s'en éloignent que par des interactions plus complexes avec le reste du système logiciel (*e.g.*, concurrence de requêtes et réentrance, allocation de mémoire dynamique). Les caractéristiques des différentes classes de périphériques gérés sont également semblables.

Puisqu'un pilote de périphérique s'exécute conjointement avec au moins un composant matériel dont le comportement est souvent complexe, le test véritablement exhaustif est en général hors de portée pour ce type de programmes. À partir de ce constat, nous identifions deux familles de solutions possibles visant à rechercher une certaine sûreté parmi les méthodes de conception des pilotes de périphériques : l'usage d'analyses statiques pour la vérification de propriétés sur ces programmes, et la synthèse à partir de spécifications.

5.3 Vérification statique de propriétés sur les pilotes

Langages de gestion de ressources. Nous ne relevons dans cette catégorie que le langage Vault ; proposé par DeLine et Fähndrich [50], il est basé sur un système de types pour assurer que les accès aux ressources respectent effectivement les protocoles qui sont associés à chacune. Bien que ce langage ait été utilisé par ses auteurs pour la programmation de pilotes de périphériques desquels certaines erreurs d'utilisation de ressources étaient effectivement détectées, cette approche introduit des limitations fortes en matière d'expressivité et de coopération avec du code C classique.

Détection statique d'erreurs. La détection statique d'erreurs de programmation des pilotes est possible suivant les cas et les types d'erreurs à détecter, par exemple au moyen d'analyseurs statiques. Cependant, cette solution est limitée en ce qui concerne l'extensibilité et ne réduit pas le travail de développement, conséquent mais encore nécessaire, pour chaque nouveau périphérique non supporté.

Dans un contexte de systèmes critiques, Monniaux [119] utilise l'outil d'interprétation abstraite ASTRÉE [21] sur un pilote de contrôleur de bus USB, et identifie du même coup quelques limitations de l'approche d'analyse statique sur ce type de programmes très lié au comportement d'un composant matériel. Notamment, la considération conjointe du pilote de périphérique et du périphérique piloté complexifie la modélisation de ces deux éléments si l'objectif est de vérifier l'absence d'erreurs à l'exécution du premier.

Pour les systèmes de bureau, Ball *et al.* [7] proposent l’outil « *Static Driver Verifier* » (SDV) pour appliquer une méthode de *model-checking* sur des pilotes de périphériques. Des propriétés de l’interface des modules utilisés par le pilote à vérifier sont exprimées dans le langage de spécification d’interfaces SLIC dont la syntaxe est inspirée du C (pour correspondre aux habitudes des programmeurs des pilotes). Ces modules sont les portions de code de gestion de structures fournies par le noyau du système d’exploitation (*e.g.*, listes chaînées, verrous, gestionnaire de mémoire dynamique, données de requêtes de changement d’état du périphérique), plus les autres pilotes coopérant avec celui à vérifier. SLIC est construit pour exprimer les protocoles d’utilisation des interfaces sous la forme de machines d’états dont les transitions sont déclenchées selon les différents appels de routines des modules. Contrairement au cas précédent, le comportement du périphérique lui-même n’est pas considéré dans cette approche, et seules des propriétés relatives aux interactions du pilote avec les autres modules logiciels sont vérifiées.

Application au contexte des réseaux de capteurs sans fil. Dans le domaine des réseaux de capteurs sans fil, il n’existe à notre connaissance aucune proposition effective pour la vérification statique des pilotes de périphériques associés aux composants qu’ils gèrent.

En outre, l’architecture en *exo-noyaux* des principaux systèmes dilue la frontière entre ces pilotes et les applications. Cette caractéristique des systèmes a en revanche un avantage : les éventuelles techniques de vérification statique appliquées le sont en général sur les piles logicielles complètes, et certaines erreurs de programmation des pilotes peuvent être détectées par ce moyen.

5.4 Synthèse de pilotes

L’approche de synthèse, contrairement à la vérification statique, offre des garanties de correction par construction⁶ et semble plus extensible. Dans cette section, nous passons en revue quelques propositions et résultats dans ce domaine.

5.4.1 Synthèse du code opérant

Le langage Devil, proposé par Mérillon *et al.* [118], permet d’exprimer une spécification de haut niveau des primitives de communication à exécuter pour piloter un périphérique. En effet, les auteurs ont identifié que l’écriture manuelle de ces portions de code est sujette à l’apparition d’erreurs difficiles à détecter, principalement en raison de l’usage intensif d’opérations au niveau bit (*e.g.*, masques, décalages) et des besoins d’indépendance au « *boutisme* ».

À partir d’une spécification d’interface écrite en Devil et spécifique à un périphérique, un outil génère des portions de code sous la forme de *macros* C, qui permettent de manipuler les ports et registres d’entrées-sorties en faisant abstraction des manipulations délicates exposées plus haut. Ainsi, la correction du pilote dépend majoritairement de la spécification en Devil⁷ et du bon usage des primitives obtenues.

Devil a été utilisé avec succès pour la génération de code opérant de pilotes pour des périphériques matériels complexes (*e.g.*, contrôleurs « *Integrated Drive Electronics* » — IDE, Ethernet). Les mesures de performance obtenues en comparant les résultats avec des pilotes entièrement écrits en C ont montré la viabilité de cette approche, tant par les temps d’exécution obtenus que par les quelques possibilités de détection d’erreurs d’usage des primitives offertes par Devil.

Abstraction des bus. La première version de Devil supporte difficilement certaines constructions pourtant courantes dans les systèmes embarqués. Par exemple, ceux-ci font souvent usage de bus de type série pour communiquer avec les périphériques des plates-formes. Aussi, la grande variabilité des jeux

6. La correction de la spécification mise à part, problème déjà présent dans le cas de la vérification statique.

7. Bogues des compilateurs de Devil ou C mis à part, bien entendu.

d'instructions et des architectures des bus implique d'écrire différentes versions des spécifications des interfaces pour chaque périphérique, voire de modifier le compilateur Devil.

C'est pourquoi Mérillon et Muller [117] proposent l'usage de Trident, une extension de Devil destinée à permettre l'expression d'accès aux périphériques à travers des bus, notamment ceux de type série particulièrement répandus dans le contexte des systèmes embarqués.

Vérification d'invariants. Plus récemment, Sun *et al.* [147], s'inspirèrent de Devil pour définir le langage HAIL pour la synthèse de code opérant selon eux plus adapté aux systèmes embarqués. HAIL diffère de Devil sur plusieurs aspects, dont la possibilité de spécifier des espaces d'adresses et leurs propriétés (à accès direct, indirect par bus série, etc.). De plus, HAIL permet de définir un certain nombre d'invariants qui sont facultativement vérifiés à l'exécution :

- Les contraintes logiques servent à détecter les erreurs du programmeur, ou encore les éventuels bogues du matériel ;
- Les contraintes séquentielles décrivent des restrictions sur l'ordre d'opérations de lecture ou écriture des différents (bits des) ports et registres d'entrées-sorties. Exprimées dans un formalisme inspiré de la logique temporelle [138], ces contraintes permettent notamment d'écrire des machines à états implicites décrivant le protocole accepté par le périphérique. Leur vérification dynamique autorise la détection d'usages incorrects de l'interface par la portion du périphérique écrite manuellement (*i.e.*, le code de contrôle).

5.4.2 Synthèse de pilotes complets

Également inspirés de Devil, Conway et Edwards [42] proposent NDL, centré sur la description du comportement du périphérique piloté à l'aide de machines d'états explicites. Les auteurs argumentent en faveur de la description des comportements du pilote sous la forme d'automates en avançant qu'elle est similaire à celle retrouvée dans la plupart des documentations constructeurs des périphériques. Wang *et al.* [155] utilisent une approche très similaire à NDL pour la synthèse du code de contrôle.

Ryzhyk *et al.* [140] suggèrent également une approche de synthèse du code de pilotes à partir de spécifications d'interfaces et de comportements. Pour cela, ils proposent l'outil Termite, qui génère un pilote à partir :

- d'une spécification du comportement commun à la classe à laquelle appartient le périphérique. Elle définit un ensemble d'évènements correspondants à des actions réalisables par tout périphérique de cette classe. Pour un contrôleur Ethernet par exemple, les évènements comprennent la fin de transmission ou réception d'une trame, ou encore le changement de statut du lien physique (*e.g.*, transition connecté-déconnecté) ;
- d'une spécification de l'interface à fournir par le pilote à synthétiser ;
- d'une spécification de l'interface du périphérique du point de vue logiciel.

Les interfaces des deux derniers points détaillent des comportements en termes d'automates dont les transitions sont déclenchées par des évènements de classe ou des messages. Ces derniers sont par exemple l'écriture d'une valeur dans un registre par le pilote, l'appel d'une fonction par le système d'exploitation, ou l'envoi d'une requête d'interruption par le périphérique.

À partir de ces spécifications, Termite génère l'ensemble du code du pilote en C. (Nous ferons remarquer que cette proposition se base sur une variante de la synthèse de contrôleur exprimée en termes de théorie des jeux. Nous aborderons les principes de la synthèse de contrôleur à la section 2 page 56 du chapitre 4.)

5.5 Quelques observations

Ce rapide examen des méthodes d'implantation des pilotes de périphériques nous permet de relever quelques conséquences impactant notre objectif de contrôle global.

D'une part, les pilotes sont développés de manière indépendante les uns des autres. De par leur rôle d'abstraction des périphériques, il est difficile d'extraire une information claire sur l'état de celui-ci sans en modifier l'interface. Aussi, l'absence, à notre connaissance, de propositions d'analyses statiques d'un pilote prenant en compte le comportement du périphérique qu'il gère limite les possibilités d'extraction *a posteriori* de ces informations d'états.

D'autre part, la majorité des approches de synthèse du code de contrôle des pilotes s'attellent à la génération de code à partir de descriptions sous la forme d'automates dont certains états reflètent le mode de fonctionnement du périphérique (et donnent donc une idée de la puissance électrique qu'il consomme). En revanche, ces propositions génèrent toujours chaque pilote sans prendre en considération les comportements d'autres périphériques.

6 Conclusion du chapitre : résumé du problème

La complexité de programmation des nœuds de réseaux de capteurs sans fil est due à plusieurs facteurs, dont les principaux sont les restrictions liées aux faibles quantités d'énergie fournies par les batteries qu'ils embarquent, mais aussi les contraintes induites par la taille de la mémoire de travail et la faible puissance de calcul de leur microcontrôleur. La grande variabilité et l'hétérogénéité des architectures matérielles des nœuds rend également le développement de logiciel pour ces plates-formes plus complexe.

L'expression du parallélisme est au surplus nécessaire à la conception des applications usuelles de ces systèmes. Du reste, la gestion des transmissions radio impose des contraintes supplémentaires de réactivité, et les microcontrôleurs employés limitent fortement les ressources disponibles pour les programmes. Les contraintes de déploiement sont aussi non négligeables, notamment en ce qui concerne la difficulté de corriger d'éventuels « bogues » *a posteriori*.

En cela, cette classe d'applications et les plates-formes matérielles employées représentent une cible idéale dans le cadre de la recherche de solutions aux problèmes classiques d'implantation des systèmes embarqués.

Pour cette classe de systèmes, distribués par essence, il existe de fait plusieurs niveaux où des solutions peuvent être envisagées afin d'influencer l'énergie consommée par chaque élément du réseau. Par exemple, les caractéristiques des protocoles de gestion de l'infrastructure de communication sans fil impactent fortement le temps d'inactivité du calculateur et de l'émetteur-récepteur radio.

Au niveau des nœuds individuels, nous avons vu en outre que les batteries peuvent avoir des comportements singuliers qu'il est possible d'exploiter pour utiliser au mieux l'énergie disponible initialement, et augmenter par là la durée de vie du système. Ainsi, l'objectif de réduction de la consommation énergétique d'une plate-forme ne peut se résumer à une gestion locale de chacun de ses périphériques.

La connaissance d'un état global est donc essentielle à la portion du logiciel embarqué dont le rôle est de gérer les composants matériels. Cependant, de par la manière dont sont usuellement programmés les pilotes de ces périphériques, nous avons pu constater qu'il est la plupart du temps difficile d'introduire des mécanismes génériques de maintien d'une connaissance de cet état global.

Dans la suite de cette thèse, nous passerons en revue l'état de l'art de la programmation des réseaux de capteurs sans fil, en insistant notamment sur la manière dont sont gérés les périphériques. Nous aborderons aussi les quelques solutions proposées pour introduire une connaissance centralisée de l'état d'un ensemble de ressources. Nous pourrions alors proposer en connaissance de cause et en s'inspirant des suggestions existantes, une méthode de gestion globale qui permet de répondre aux problématiques d'économies d'énergie et de gestion de ressources partagées exposées précédemment.

DEUXIÈME PARTIE

ÉTAT DE L'ART

GESTION DE RESSOURCES ET IMPLANTATION DES NŒUDS DE RÉSEAUX DE CAPTEURS SANS FIL

Contenu du chapitre

| | | |
|-------|--|----|
| 1 | Principes généraux de gestion du microcontrôleur | 30 |
| 1.1 | Programmation du calculateur | 30 |
| 1.2 | Structuration des programmes : fil et contexte d'exécution | 31 |
| 1.2.1 | Définitions informelles | 31 |
| 1.2.2 | Implantation monofil | 31 |
| 1.2.3 | Implantation multifils | 32 |
| 1.3 | Traitement des requêtes d'interruption matérielles | 32 |
| 2 | Modèles de programmation | 33 |
| 2.1 | Déclenchement de réactions | 33 |
| 2.2 | Boucle de scrutation explicite | 33 |
| 2.3 | Réalisation de la concurrence | 34 |
| 2.3.1 | Programmation événementielle | 34 |
| 2.3.2 | Programmation multifils | 35 |
| 3 | Langages dédiés à l'implantation | 36 |
| 3.1 | Approches langages à composants | 36 |
| 3.2 | Protothreads. | 37 |
| 3.3 | Utilisation d'automates hiérarchiques | 37 |
| 4 | Approches d'implantation des applications | 38 |
| 4.1 | Implantation sur machine nue | 38 |
| 4.2 | Support système pour l'implantation | 38 |
| 4.2.1 | Systèmes multifils | 39 |
| 4.2.2 | Systèmes événementiels | 39 |
| 4.2.3 | Une comparaison | 42 |
| 4.2.4 | Hybrides | 42 |
| 4.3 | Machines virtuelles de haut niveau | 43 |
| 4.3.1 | Vue d'ensemble | 43 |
| 4.3.2 | Machines virtuelles construites à partir d'applications | 44 |
| 4.4 | Discussion | 45 |

| | | |
|-----|--|----|
| 5 | La gestion des périphériques dans les réseaux de capteurs sans fil | 45 |
| 5.1 | Gestion de ressources décentralisée | 45 |
| 5.2 | Une approche centralisée multifils | 46 |
| 6 | Conclusion du chapitre | 47 |

Au chapitre précédent, nous avons passé en revue les applications et plates-formes typiques des réseaux de capteurs sans fil, ainsi que les problématiques d’implantation afférentes. Les différents défis relatifs à l’usage de batteries pour fournir l’énergie nécessaire au fonctionnement de ces systèmes autonomes y ont également été abordés. Nous en avons conclu qu’une attention particulière doit être adressée à cet aspect dans le cadre de leur programmation, spécialement en ce qui concerne le développement du logiciel de gestion des composants matériels. Réduire autant que possible l’énergie consommée par chacun d’eux constitue un objectif important lors du développement pour ces plates-formes. Ainsi, des outils d’assistance à l’implantation s’avèrent nécessaires afin de satisfaire l’ensemble des contraintes posées.

Il existe déjà un grand nombre d’approches destinées à gérer le microcontrôleur, toutes se basant sur divers *modèles de programmation concurrente*. Ces derniers représentent chacun une solution viable à un ensemble de problèmes posés, et se révèlent tous plus ou moins adaptés en fonction de la classe d’application concernée.

Nous nous proposons dans ce chapitre d’ébaucher une taxonomie de ces solutions d’implantation, c’est-à-dire d’identifier pour chacune les caractéristiques importantes et les apparentements, afin de les comparer et d’en extraire les points essentiels en vue de la proposition d’une solution au problème du contrôle global des ressources présenté au chapitre 2.

1 Principes généraux de gestion du microcontrôleur

Étant données les contraintes et spécificités des microcontrôleurs exposées dans la section 2.2.1 page 12, l’objectif des implantations pour réseaux de capteurs sans fil est principalement d’exploiter les capacités calculatoires de ce matériel au mieux, tout en le maintenant dans un état de basse consommation d’énergie le plus souvent possible, lorsqu’aucun calcul n’est plus à effectuer.

1.1 Programmation du calculateur

Nous avons vu à la section 2.2.2 du chapitre 2 que les microcontrôleurs intègrent des calculateurs d’architecture relativement simples, avec des jeux d’instruction réduits et une largeur de mot ne dépassant pas 16 bits. L’absence de mécanismes matériels de protection ou de virtualisation des ressources telles que la mémoire de travail, en fait des plates-formes à modèle mémoire nécessairement plat (« *flat memory* »). En conséquence toute partie du programme peut accéder en lecture et écriture à toute partie de la mémoire, y compris les instructions du programme lui-même.

De plus, ces plates-formes sont encore majoritairement utilisées dans un contexte de contrôle embarqué où, lorsque le domaine d’application n’est pas critique, les programmes sont le plus souvent écrits directement en langage d’assemblage, C ou « au mieux » en C++. En conséquence de l’utilisation de ces langages, peu de protection est supposée de la part du système d’exécution, et les erreurs éventuelles sont parfois difficiles à détecter.

Dans la suite de ce chapitre, nous détaillerons des mécanismes d’expression et d’implantation de programmes concurrents sur les calculateurs compris dans les microcontrôleurs que nous considérons. Pour cela, il est nécessaire d’introduire succinctement la notion de fil d’exécution et de contexte.

1.2 Structuration des programmes : fil et contexte d'exécution

1.2.1 Définitions informelles

Un *flot*, *flux* ou *fil d'exécution*, désigne une séquence d'instructions exécutée par le calculateur.

Ce que nous appelons *contexte*, *pile d'exécution* ou encore *pile d'appels*, matérialise à l'exécution la trace des routines du programme successivement appelées qui ne se sont pas encore terminées, pour mener à l'instruction courante d'un flot d'exécution. Ainsi, le contexte comprend, entre autres informations, la séquence des adresses de retour des différents appels de routines.

En pratique, le *pointeur de pile* est un registre du processeur dédié à la mémorisation de l'adresse du sommet de la pile associée au flot d'exécution courant.

On notera en outre que la plupart des implantations du langage C¹ et des conventions de programmation des calculateurs, incluent également dans le contexte au moins l'environnement local : la valeur des variables définies localement et des paramètres des appels de routines.

L'implantation traditionnelle des programmes directement en langage d'assemblage, C ou tout dérivé, peut se faire en utilisant un ou plusieurs contextes d'exécution en fonction de l'expressivité recherchée et des contraintes en matière de ressources. Nous ne considérons pas dans la suite de cette section les autres techniques de programmation sans pile existantes, pour lesquelles il n'existe manifestement pas d'outils d'implantation d'applications de taille raisonnables pour les architectures de calculateurs que nous considérons. Nous ne regardons pas non plus dès ici les techniques *ad hoc* d'usage du pré-processeur C pour contourner une partie des contraintes d'expressivité imposées par l'usage de flots uniques.

1.2.2 Implantation monofil

L'usage d'un flot d'exécution unique limite notablement la quantité de mémoire de travail occupée par des piles d'exécution, mais réduit les possibilités d'expression du parallélisme.

Avantages. Premièrement, cette solution est simple d'utilisation et relativement portable, puisqu'elle n'implique pas le codage et l'usage d'un support de manipulation des contextes, par essence très dépendant de l'architecture du calculateur. Deuxièmement, l'usage d'un contexte unique réduit *a priori* la fragmentation interne imposée par cette pile. Même en supposant que l'on puisse calculer la profondeur maximale de l'enchaînement des appels à partir d'une routine donnée, il est toujours nécessaire d'ajouter à cette quantité l'espace requis pour l'exécution des traitements d'interruption (dont la profondeur de contexte peut également être très variable — cf. § 1.3 plus loin). La quantité de mémoire en pile dédiée à l'exécution des traitements est donc à multiplier par le nombre de contextes, et réduire ce dernier introduit mécaniquement une baisse de ces espaces.

En n'utilisant qu'un unique contexte, la taille à réserver en mémoire de travail pour la pile d'exécution est extensible *en théorie*. En effet, en supposant la pile grandissant vers le tas (et les éléments de celui-ci déplaçables à volonté), faire coïncider le bas de l'unique pile d'exécution à une extrémité de l'espace adressable permet de l'étendre autant que nécessaire (au détriment de la taille du tas toutefois).

Contraintes d'expressivité induites. En revanche, et toujours dans un cadre d'implantation séquentielle bas niveau en C ou dans un langage similaire, les contraintes imposées par l'usage d'un contexte unique ne sont pas négligeables. En effet, un tel choix interdit le codage de routines bloquantes du point de vue d'un fil d'exécution en déroutant explicitement le CPU vers un autre (pour lesquelles il faut bien évidemment au moins deux flots d'exécution).

Par exemple, il est trivialement impossible d'implanter dans un cadre monofil une routine `sleep(t)`, se terminant après qu'un temps t s'est écoulé, sans effectuer d'attente active (en scrutant une valeur de

1. Pour ne pas dire « toutes »...

timer par exemple) ou passive (en armant un timer, puis en plaçant le calculateur dans un mode d'inactivité jusqu'à l'occurrence de l'interruption associée).

S'affranchir de ces contraintes d'expressivité, lors d'une programmation directement en C des applications, nécessite d'utiliser plusieurs fils d'exécution.

1.2.3 Implantation multifils

Au prix d'un surcoût occasionné par la nécessaire présence en mémoire et la gestion simultanée de plusieurs piles d'exécution, une implantation multifils autorise le codage et l'usage de routines qui ne bloquent que le flot d'exécution qui les appelle. En effet, celles-ci peuvent alors explicitement dérouter le flot d'exécution du calculateur vers un contexte en attente d'une libération de celui-ci, au moyen d'une opération de changement de contexte :

Changement de contexte. On appelle *changement de contexte* une opération consistant à agir sur l'état et les registres du calculateur pour interrompre temporairement l'exécution du fil d'exécution courant, sauvegarder son contexte, et reprendre une exécution précédemment interrompue en restaurant son contexte depuis la mémoire de travail.

Dans la pratique, un changement de contexte est généralement réalisé au moyen d'une procédure dédiée dont le rôle est d'être appelée explicitement par un fil d'exécution lorsqu'il doit céder sa possession du calculateur à un autre fil.

Considérations sur le coût des changements de contexte. Dans le cas des microprocesseurs pour applications peu contraintes comme les ordinateurs de bureau, un changement de contexte implique également des opérations coûteuses au niveau des caches et des matériels de gestion de la mémoire virtuelle du processeur (e.g., purge de caches et autres tables de tous types dont le contenu détermine chaque contexte, ou ne doit être visible que du contexte rendant la main). Ces aspects alimentent l'idée que le changement de contexte est une opération à effectuer le moins souvent possible.

Cependant, sur un microcontrôleur, un contexte est presque exclusivement constitué des quelques registres du calculateur, et l'absence de mécanismes matériels d'isolation à gérer à ces moments réduit leur coût. En outre, l'absence de caches de la mémoire sur les architectures que nous considérons annihile les conséquences des changements de contexte sur les performances des accès à celle-ci.

1.3 Traitement des requêtes d'interruption matérielles

Les mécanismes matériels de traitement des requêtes d'interruption tels que présentés dans la section 2.2.1 page 13 sauvegardent automatiquement des données nécessaires à la reprise du calcul interrompu dans la pile d'exécution dont le sommet est désigné par le pointeur de pile. De plus, le processeur commence alors à exécuter une routine associée statiquement à la source de l'interruption, appelée *traitant*. En outre, les interruptions sont le plus souvent masquées globalement par le processeur au commencement de l'exécution du traitant, et elles sont restaurées à sa terminaison (lors de la restauration du registre d'état depuis la zone de mémoire désignée par le pointeur de pile). En conséquence, et à moins de ne jamais autoriser le traitement d'interruptions par ce mécanisme, il convient de toujours conserver un espace libre en haut de pile, dans lequel ces données et le contexte complet des éventuelles routines appelées par le traitant peuvent être placés.

Par exemple, nous avons vu que dans le cas du calculateur intégré aux microcontrôleurs de la famille MSP430 [151], les données sauvegardées lorsque survient une requête d'interruption sont constituées de l'adresse de la prochaine instruction à exécuter à la terminaison du traitant, ainsi que du contenu du registre d'état du CPU précédant le déroutement.

La conservation des données du processeur dans la pile d'exécution lorsqu'une interruption advient permet de considérer le traitant comme une routine quelconque appelée de manière *asynchrone*, *i.e.*, sans que le programme en ait pris l'initiative. Par conséquent, il est tout à fait possible d'effectuer un changement de contexte alors que l'exécution d'un traitant d'interruption ne s'est pas encore terminée² : un retour dans cette pile initiale suffira à terminer celui-ci, puis à revenir aux instructions préalablement interrompues.

Imbrication des interruptions. Enfin, nous ferons remarquer qu'il est également possible, moyennant quelques manipulations techniques communément employées dans le cadre de la programmation de noyaux de systèmes, d'imbriquer les traitants d'interruptions ; c'est-à-dire d'autoriser de nouvelles interruptions à survenir lorsqu'un traitant est déjà en cours d'exécution. Ce moyen permet la réduction du délai requis pour traiter d'éventuelles interruptions survenant pendant le traitement de requêtes moins prioritaires.

2 Modèles de programmation

Comme expliqué dans la section 2.1.2 page 12, l'expression du parallélisme nécessaire à la description d'une application pour réseaux de capteurs sans fil implique l'usage de méthodes et modèles de programmation adéquats. De plus, nous avons également vu section 2.3.1 page 16 que les nœuds de ces réseaux entrent dans la catégorie des systèmes réactifs, c'est-à-dire qu'il doivent être programmés en tenant compte de contraintes temporelles pour assurer un fonctionnement normal du réseau, ou tout simplement le respect de la logique applicative (*e.g.*, si une période d'échantillonnage de capteurs minimale est spécifiées). Dans cette section, nous nous proposons de passer en revue les principaux modèles de programmation applicables à ces systèmes monoprocesseurs.

Avant d'aborder les aspects liés à l'expression de la concurrence, nous décrivons un principe fondamental d'implantation des systèmes réactifs.

2.1 Déclenchement de réactions

Tout système informatisé *interactif* ou *réactif* (*cf.* § 2.3.1 page 16), s'il est construit à partir de circuits *synchrones* (ce qui est presque majoritairement le cas des systèmes actuels), se doit de *réagir* aux événements qui lui parviennent en provenance de son environnement. Pour cela, il doit scruter l'ensemble de ses entrées périodiquement et, si l'une d'elles change d'état, alors il exécute les calculs qui lui sont associés et continue la scrutation lorsque ce traitement se termine. Un tel système se base donc toujours sur un comportement périodique théoriquement infini.

La définition d'un changement d'état d'une entrée varie selon sa nature. Pour un signal Booléen par exemple, l'opération de scrutation du changement d'état peut consister à détecter tout changement de valeur, ou un front montant uniquement. Dans le cas d'un signal numérique, le changement d'état peut aussi représenter le dépassement d'un seuil fixé. Dans tous les cas, les entrées du programme sont *échantillonnées*, et la détection ou non d'un changement d'état dépend du rythme d'exécution de la boucle implicite.

2.2 Boucle de scrutation explicite

Hérité des systèmes de contrôle embarqués temps-réel usuellement implantés sur des microcontrôleurs, le modèle de programmation par boucle de scrutation (ou « boucle de contrôle ») rend explicite le comportement périodique évoqué dans la section précédente. La scrutation se fait par lecture d'un registre ou d'un port configuré en entrée ; ces opérations ne sont jamais bloquantes, et une seule pile d'exécution est nécessaire pour implanter ce type de programmes.

2. Une attention particulière doit alors être portée sur la gestion des acquittements d'interruptions.

Aussi, le rythme d'exécution de la boucle de scrutation dépend du nombre d'instructions impliquées dans les calculs à réaliser à chaque exécution de son corps. Si le temps d'exécution du corps de la boucle est variable, alors le rythme d'échantillonnage des entrées n'est pas constant. Cependant, l'usage d'un mode de fonctionnement du CPU pendant lequel il ne calcule pas, combiné avec un timer, permet d'introduire un temps de pause, et de rendre constant la période de scrutation. Ces temps de pause sont les seuls moments pendant lesquels le programme ne calcule pas, et la programmation par boucle de scrutation explicite entraîne nécessairement des réveils périodiques, même si le système n'a aucune action à réaliser pendant plusieurs tours de boucle.

Programmer directement en C une boucle de scrutation dans laquelle des calculs complexes, ou impliquant des mémorisations de valeurs d'une itération à l'autre, peut être difficile et peu sûr. Les langages synchrones ont été conçus pour pallier ce problème, et les systèmes de contrôle critiques sont généralement programmés par ce moyen. L'implantation des programmes synchrones en boucle de scrutation est fondée sur l'ordonnancement statique des comportements concurrents. Nous verrons les principes de base de ces langages au chapitre 4.

2.3 Réalisation de la concurrence

L'implantation de programmes concurrents pour lesquels les différents calculs ne sont pas ordonnancés statiquement requièrent un système d'exécution se chargeant de réaliser cette tâche dynamiquement.

2.3.1 Programmation événementielle

La *programmation événementielle* (« *event-driven programming* », dite aussi *programmation par événements*), comme son nom l'indique, est fondée sur un contrôle du flot d'exécution en fonction d'*événements* : une *action* peut être associée à chacun, similairement au principe d'association d'un traitant à une requête d'interruption matérielle. Lors de l'*émission* d'un événement par l'une quelconque des parties du logiciel, l'action correspondante, s'il y en a une, est placée dans une file d'attente globale afin d'être exécutée ultérieurement. Il est rare d'autoriser l'association de plusieurs actions à un seul événement.

Dans un modèle d'exécution événementiel, la boucle de scrutation présentée à la section 2.1 page précédente est implicitement simulée par le processeur, qui scrute l'état des drapeaux d'interruption entre chaque exécution d'une instruction, et déclenche potentiellement un appel de traitant. Ainsi, le principe de la programmation par événements délègue le travail de scrutation des entrées du programme au système d'exécution sous-jacent. Si ce dernier est le processeur, alors le rythme de scrutation est très rapide, et les traitements d'interruptions sont presque immédiats.

Exécution des actions « jusqu'au-boutiste ». Le cœur de tout programme événementiel est le *répartiteur* (« *dispatcher* ») qui puise les événements dans la file d'attente et déclenche l'exécution des actions associées. Ces actions sont exécutées de manière atomique : pendant ce temps, toute nouvelle occurrence d'un événement, par un traitant d'interruption matérielle ou émission par une instruction du langage, ne fait qu'insérer celui-ci dans la file d'attente globale ; il sera traité par le répartiteur dès que positionné en tête de file.

Tout comme pour la boucle de scrutation explicite, une unique pile d'exécution suffit à l'implantation de systèmes d'exécution basés sur des événements. Toutes les actions s'exécutent alors dans la même pile, et une procédure à la racine de toute chaîne d'appels joue le rôle de *répartiteur* : elle est en principe constituée d'une boucle scrutant perpétuellement la file des événements en attente. Si un événement est présent, alors l'action associée est exécutée jusqu'à sa terminaison. Sinon, aucun calcul n'est plus nécessaire (jusqu'à la prochaine interruption matérielle) et le processeur peut alors être mis dans un état d'attente.

Par rapport au modèle en boucle de contrôle, le calculateur ne calcule pas nécessairement en permanence et peut rester en mode de basse consommation tant qu'aucune requête d'interruption n'est à traiter.

Problème de « stack ripping ». En général, un programme implanté selon un modèle dirigé par des événements souffre du syndrome de « *stack ripping* » (que je traduirai par « déchirement de contexte »). Ce problème se manifeste lorsque des données doivent être transmises d'un traitant d'évènement à l'autre. Comme le programme ne peut traiter d'autres événements qu'une fois que le traitant courant se termine, alors aucune donnée ne peut être conservée dans la pile d'exécution. Ainsi, des manipulations dédiées sont souvent mises en œuvre pour assurer la transmission de données locales entre les traitants. L'usage de ces solutions *ad hoc* compromettent fortement la lisibilité des programmes.

Notons au passage que Chandrasekaran *et al.* [35] proposent un langage et une analyse statique dédiée pour contourner ce problème ; cependant, cette technique n'a, à notre connaissance, été mise en œuvre que dans le cadre de la programmation de pilotes.

Expression de tâches longues. Enfin, l'expression de tâches longues dans un style basé sur des événements est incompatible avec la politique d'exécution « jusqu'au-boutiste » si des actions concurrentes nécessitent un traitement dans un délai court après l'occurrence de leur événement. Une solution à ce problème est la division du calcul en plusieurs étapes, ensuite réparties en traitants successifs (avec, bien entendu, l'apparition potentielle du syndrome de déchirement de contexte évoqué ci-dessus).

Enfin, on notera que la plupart des systèmes pour réseaux de capteurs sans fil basés sur un modèle d'exécution événementiel, intègrent également des mécanismes pour implanter quelques tâches concurrentes classiques (qui ont leur propre pile d'exécution).

2.3.2 Programmation multifils

Le modèle de programmation *multifils* (ou *multithreading*) classique repose sur l'utilisation simultanée de plusieurs contextes d'exécution. À tout instant de l'exécution d'un système multifils, à tout contexte correspond une tâche. Dans un CPU sans support avancé pour le parallélisme, une seule tâche est en cours d'exécution, ou bien le CPU ne calcule pas.

multifils coopératif. Le modèle de concurrence dit « multifils coopératif » consiste à exécuter un fil d'exécution par contexte, et à en changer lorsque la tâche courante se bloque : elle cède alors explicitement le processeur à une autre tâche au moyen d'instructions dédiées (*e.g.*, `yield()`). L'exécution de ces instructions est le plus souvent la conséquence d'un blocage de la tâche courante lors de l'appel d'une primitive de synchronisation, ou encore lors d'un appel vers un autre service du système requérant ce blocage.

L'exécution de ces instructions provoque un appel au gestionnaire des tâches. Si une tâche est éligible, alors un changement de contexte est exécuté et la tâche sélectionnée continue son exécution depuis l'instruction où elle avait été interrompue. Dans le cas contraire, le processeur est mis en pause.

Ce modèle de concurrence réduit *a priori* le surcoût lié aux changements de contexte, puisque les tâches gardent un contrôle explicite de leur utilisation du processeur [37].

De même que pour le modèle événementiel, mais dans une moins grande mesure cependant, l'usage d'un système coopératif limite l'implantation de tâches longues si on ne peut ajouter simplement des instructions explicites pour céder le CPU aux tâches concurrentes.

multifils préemptif. Le modèle de concurrence dit « multifils préemptif » ou « *temps-partagé* », initialement proposé par Bemer [9] et Everett *et al.* [68]³, consiste à virtualiser la disponibilité du processeur en donnant à chaque tâche l'illusion qu'elle s'exécute sans interruption lorsqu'elle calcule. En pratique, la différence notable par rapport au modèle coopératif et l'usage d'interruptions du processeur pour déclencher des changements de contexte.

3. Ce papier est peut-être une des bases de la cybernétique...

Ce principe est plus simple pour le programmeur d'une seule tâche, qui peut utiliser les constructions habituelles en usage sur les systèmes classiques. Dans ce cas en revanche, l'appréhension et le « débogage » d'une application complète comprenant plusieurs tâches communicantes et partageant des ressources devient plus difficile, notamment en raison du fort non-déterminisme du comportement des programmes introduit par le mécanisme de préemption.

3 Langages dédiés à l'implantation

3.1 Approches langages à composants

nesC. Le langage nesC, de Gay *et al.* [73], est probablement le langage dédié le plus utilisé pour implanter ces systèmes. Il est employé à la fois pour programmer du code bas niveau pour la gestion du matériel, et pour l'implantation des applications. nesC est très populaire dans la communauté des développeurs d'applications pour réseaux de capteurs sans fil, et l'approche à composants permet la réutilisation d'une grande base de code lors du développement d'une nouvelle application.

Les programmes nesC sont séparés en *configurations*, *modules* et *implantations*, et toutes les connections de communications entre les *ports* des modules sont statiques. Une application nesC complète est compilée en un seul bloc de code C autonome (*i.e.*, comprenant le moteur du système d'exécution). Cette compilation non modulaire, viable pour les petits systèmes programmés en nesC, autorise l'*inlining* des nombreuses petites fonctions afin de réduire la taille du code.

Les modules comprennent des *tâches* capables d'appeler des portions de code dites *synchrones* (des appels bloquants du point de vue de la tâche). Les tâches sont des traitants du modèle de programmation événementiel, c'est-à-dire qu'une seule tâche peut s'exécuter à la fois. Leurs exécutions sont déclenchées en *postant* un message. Tout code synchrone peut être temporairement interrompu par du code dit *asynchrone*. De cette manière, nesC met en œuvre un mécanisme d'ordonnancement à deux niveaux :

- L'exécution du code synchrone est déclenchée par une boucle de scrutation explicite d'une file d'attente des tâches postées ;
- L'exécution du code asynchrone est déclenchée par la boucle de scrutation des interruptions du processeur.

Les problèmes fréquemment rencontrés dans les programmes nesC sont les situations de compétition (*race conditions*), dues à l'usage des deux niveaux d'ordonnancement. En effet, les interactions implicites entre ces deux parties (l'une peut interrompre l'autre à tout moment), pose problème en cas d'utilisation non protégée d'une variable modifiable une portion de code asynchrone. Les situations d'interblocages entre tâches sont également récurrentes.

De plus, une application simple devient rapidement difficilement intelligible, de par la grande diversité et le nombre de composants à utiliser, ainsi que par les fortes coopérations implicites entre modules. Ainsi, plus d'une cinquantaine de modules, même très simples, sont parfois nécessaires pour implanter une application simple. Ceci constitue selon nous un bruit non négligeable complexifiant l'appréhension du comportement global des programmes.

nesC n'intègre aucun support pour la gestion de mémoire dynamique, ni de passage de données dans les messages ; des variables locales statiques doivent être utilisés pour cela, ou des mécanismes *ad hoc* faisant intervenir du code C externe. En conséquence, les implantations de composants souffrent très souvent du syndrome de déchirement de contexte (*cf.* § 2.3.1 page 34). Ces aspects rendent également les codes peu lisibles.

glasC et TINYGALS. Par ailleurs, Cheong et Liu [39] proposent d'étendre le langage nesC afin de rendre explicite les tâches concurrentes, et ainsi détecter par avance les éventuels problèmes de concurrence.

Cheong *et al.* [38] font un usage du terme « GALS » désignant ici un système où les *communications* sont globalement asynchrones et localement synchrones. Cette approche clarifie cependant les notions et problèmes de nesC, en les formalisant, et en imposant des contraintes structurelles sur les programmes. Notamment, il y a une distinction entre événements et interruptions matérielles, avec des contraintes de construction différentes suivant les cas.

Trois types d'évènements déclenchent des calculs :

1. une interruption matérielle ;
2. un appel de *méthode* ;
3. un passage de *jeton*.

Dans les *modules* ou *acteurs*, toute communication est nécessairement synchrone (appels de méthode). Ces derniers sont construits à l'aide de *composants*, et communiquent au moyen de files d'attente dans lesquelles les données sont encapsulées dans des jetons. Au sein d'un module, les composants et leurs liens (appels de méthodes) forment nécessairement un graphe orienté acyclique, pour éviter certains problèmes de réentrance.

Ainsi, un programme construit selon le modèle TINYGALS doit respecter certaines contraintes structurelles pour éviter les problèmes de réentrance des méthodes :

- Les composants *sources* (*i.e.*, qui traitent des interruptions) peuvent déclencher des exécutions (en encapsulant un traitant d'interruption matérielle), mais ne peuvent être des composants *déclenchés* ou *appelés* également (et *vice versa*) ;
- Au sein d'un même module, un composant source ne peut appeler (même indirectement) une méthode d'un autre composant si ce dernier peut être lui-même être déclenché hors d'un traitement d'interruptions ;
- D'autres contraintes s'appliquent selon que l'on autorise les interruptions à être imbriquées ou non.

Des tampons d'écriture permettent de gérer implicitement les situations de concurrence d'accès aux variables globales. Les valeurs lues sont actualisées de manière sûre entre l'exécution de deux modules.

3.2 Protothreads.

Les *proto-threads* sont une construction proposée par Dunkels *et al.* [59] pour la programmation de systèmes pour réseaux de capteurs sans fil. Les auteurs exploitent le pré-processeur C pour l'implantation de « *microthreads* » (que l'on pourrait traduire par « *microtâches* » ou « *tâches légères* »). Cette approche permet l'écriture de programmes événementiels dans un style multi-flux, sur le principe des coroutines [102], mais ne nécessite qu'une pile d'exécution pour s'exécuter.

3.3 Utilisation d'automates hiérarchiques

Remarquant les problèmes liés à l'usage de la programmation événementielle pour encoder des applications s'exprimant bien sous forme d'automates Kasten *et al.* [99, 98] proposent un langage d'expression de ces automates avec *variables d'état*. Ce concept est utilisé, entre autres objectifs, pour réduire l'occupation mémoire des programmes. Ces derniers sont construits selon le modèle « Object State Model » – OSM, inspiré des STATECHARTS [83] et SYNCCHARTS [129].

Les programmes OSM sont exprimés comme une composition parallèle et hiérarchique d'automates finis. L'aspect hiérarchique permet d'associer un sous-automate à un état ; cet automate n'est alors actif que lorsque son parent est dans l'état raffiné. Les états définissent également la portée de variables utilisables uniquement par des sous-automates. Les transitions des automates sont déclenchées sur *émission* d'un événement, et peuvent également être gardées à l'aide de prédicats sur les valeurs des paramètres de l'évènement déclenchant la transition et des variables locales de l'état source et des états parents de l'automate. Le principe de hiérarchie est utilisé pour l'optimisation de l'usage de la mémoire de travail : la

mémoire occupée par des variables attachées à deux états distincts d'un même automate peuvent en effet occuper des zones se chevauchant.

Les auteurs proposent des outils transformant un programme OSM vers C, avec une étape de compilation intermédiaire par ESTEREL [14]. D'après les auteurs, les programmes exprimés en OSM sont généralement plus compacts qu'en ESTEREL.

Discussion. OSM autorise un développement modulaire, mais le langage proposé est exclusivement dédié à l'implantation des applications et suppose la présence d'un support système minimal consistant en un mécanisme de gestion de file d'ensemble d'événements, ainsi que de services et pilotes de périphériques capables de les émettre.

En outre, les situations de concurrence d'accès en écriture des variables d'un état par deux automates fils doivent être gérées manuellement (et aucune indication n'est donnée sur une méthode pour gérer ce problème).

Comme tout programme exprimé à l'aide d'automates, l'expression de longues séquences d'actions est fastidieuse.

4 Approches d'implantation des applications

Dans le contexte des réseaux de capteurs sans fil, l'implantation d'applications se fait usuellement en choisissant dans un premier temps une option parmi les deux suivantes :

- la programmation sur machine nue ;
- l'emploi d'un support système, de plus ou moins haut niveau.

4.1 Implantation sur machine nue

La programmation sur machine nue consiste à concevoir la pile logicielle complète sans aucun support système. Le développeur emploie une bibliothèque C simple ne fournissant aucune abstraction permettant la programmation parallèle : un unique contexte d'exécution est disponible initialement, dans lequel la fonction `main()` commence à s'exécuter — typiquement, au (re-)démarrage du calculateur, c'est-à-dire lorsque l'interruption « *system reset* » est levée (cf. § 2.2.1 page 13).

De plus, cette bibliothèque peut intégrer du code facilitant l'emploi de fonctionnalités spécifiques au micro-contrôleur, telles que la sélection parmi ses divers modes de fonctionnement, la gestion des interruptions (masques, priorités, etc.), ou encore l'accès aux registres d'entrée-sortie ;

Les sources et pilotes de périphériques fournis parmi les outils SensLAB [25] sont un exemple de développements sur machine nue.

Dans la suite de cette section, nous énumérons quelques solutions existantes pour l'implantation des applications à l'aide d'un support système.

4.2 Support système pour l'implantation

L'emploi d'un support système étend la solution de la section précédente avec l'inclusion de *services de niveau système*. Dans ce cas, une abstraction est fournie afin de faciliter l'expression de comportements concurrents, accompagnés de mécanismes de communication et synchronisation associés. Un ensemble d'autres services de plus haut niveau sont généralement disponibles, tels que la gestion des transmissions réseau ou de mémoire persistante ; parfois, un ou plusieurs systèmes de fichiers sont également proposés.

Les différents systèmes sont choisis en fonction des besoins des applications et des composants des matériels. Pour une plate-forme donnée, la disponibilité de pilotes de périphériques, ou la complexité de programmation de ces derniers, sont des paramètres importants à prendre en considération pour le choix

d'un système. Les possibilités de reconfigurations et redéploiements de code par le réseau sont par exemple très recherchées lorsqu'elles autorisent la correction de bogues ou la mise à jour des programmes sans intervention directe sur chaque nœud.

4.2.1 Systèmes multifils

De nombreux systèmes multifils ont été proposés pour la programmation des applications pour réseaux de capteurs sans fil.

RETOS [32, 33] et MANTIS [19] sont tous deux des représentants de cette famille de systèmes. Le premier inclut des mécanismes de chargement dynamique de code, et d'isolation entre code utilisateur et code noyau par transformation du code lors de son chargement. Il comprend également un gestionnaire de tâches supportant des priorités statiques ou dynamiques, et compatible temps-réel. MANTIS intègre des mécanismes rudimentaires de gestion de l'énergie à l'initiative de l'application [55].

FREERTOS est un système d'exploitation temps-réel classique [137] également utilisé dans ce contexte ; Il propose aussi l'usage de *tâches coopératives* (sous forme de co-routines) similaires aux proto-threads.

Toutes ces solutions de systèmes multifils diffèrent surtout par les plates-formes supportées.

NANO-RK. Suggéré par Eswaran *et al.* [67], NANO-RK est un système d'exploitation intégrant des capacités temps-réel pour ordonnancer avec une connaissance *a priori* des délais d'exécution. NANO-RK utilise un paradigme de réservation de ressources inspiré du concept de « *resource kernel* » Rajkumar *et al.* [132], où les applications doivent explicitement demander du temps d'exécution et requérir les accès aux ressources ; ces requêtes peuvent être refusées par le système.

Des analyses statiques sont employées pour vérifier que l'ensemble des tâches, avec leurs usages des différentes ressources, peut être ordonnancé. En outre, l'approche utilisée ne permet pas la création dynamique de tâches. Il reste toutefois possible de changer dynamiquement des priorités, périodes et autres configurations (telles que les tailles de tampons ou de pile d'exécution), au détriment de la précision des analyses statiques.

t-kernel. Dans le t-kernel, Gu et Stankovic [77] implantent logiciellement des mécanismes de protection traditionnellement fournis à l'aide de support matériel : le code binaire des programmes est traduit en-ligne afin de mimer une isolation entre le code du noyau et le reste de l'application. De plus, des points de préemption forcée (*i.e.*, pour « rendre la main » au noyau) sont insérés à chaque instruction de branchement, empêchant alors qu'une partie de l'application ne monopolise le processeur. (Il est à noter que, pour des raisons évidentes de performance, des filtres à compteurs sont utilisés dans l'instrumentation des branchements *en arrière*. Ainsi les opérations itératives ne sont pas interrompues en permanence.)

Gu et Stankovic proposent également de contourner la limitation imposée par l'absence de mécanisme matériel de virtualisation de l'espace adressable : des *barrières mémoires* sont ajoutées dans le code traduit pour chaque accès vers le tas (*i.e.*, hors pile d'exécution et adresses constantes). Un mécanisme de *swap* pour le t-kernel, destiné à l'usage des modules mémoires à accès coûteux (*e.g.*, de type flash), est aussi avancé par les mêmes auteurs [76].

Du point de vue des applications, le t-kernel fournit une interface de communication avec le noyau qui ressemble fortement à celle d'un noyau classique : des *appels système* permettent aux différents fils d'exécution de requérir des services de la part du noyau, et un ensemble d'événements est également émis par divers composants du noyau, auxquels l'application peut attacher des traitants.

4.2.2 Systèmes événementiels

La popularité du paradigme de programmation par événements dans le domaine des réseaux de capteurs sans fil n'est plus à démontrer. En conséquence, un grand nombre de supports système fournissent au

programmeur d'application, entre autres outils, une abstraction du calculateur permettant une implantation dans un style événementiel. Nous décrivons quelques systèmes existants dans cette section.

SOS. Le système SOS est conçu de manière à permettre la reconfiguration dynamique des applications [82]. Ce système est construit à l'aide de *modules* communicants par envois de messages asynchrones à l'aide d'outils fournis par le noyau. Des communications par appels bloquants sont toutefois possibles.

Les modules sont désignés par leur nom, et l'implantation associée peut changer dynamiquement. Le code réalisant la traduction des données locales d'un module depuis une ancienne version est à fournir par le programmeur. Pour les besoins de la reconfiguration dynamique, SOS fournit un service d'enregistrement-souscription des fonctions bloquantes exportée par les modules ; leurs appels se font alors indirectement.

De plus, des priorités sont associées aux messages transmis afin d'extraire un maximum de calculs des traitants d'interruptions et, par là, augmenter la réactivité du système. Chaque message contient d'éventuels paramètres associés, ainsi que des informations de type pour détecter certaines erreurs de programmation et incompatibilités lors des reconfigurations.

En outre, le noyau SOS propose un mécanisme de pistage du possesseur d'une donnée allouée dynamiquement, afin de réduire les risques de fuite de mémoire. Il implante également un ramasse-miettes simpliste sur ses structures internes pour pallier certaines défaillances dans le remplacement dynamique de modules.

La reconfiguration dynamique autorisée par SOS introduit des complications significatives pour sa programmation. En effet, un module peut ne plus exister au moment de la délivrance d'un message qui lui était destiné, ou encore une exportée peut être implantée plusieurs fois dans le système. Lorsque le système est en cours de reconfiguration, un mécanisme du noyau gère les dépendances inter-modules (à spécifier manuellement lors de sa programmation) et détecte ces erreurs. En cas de problème, l'insertion d'un module peut être réessayée ultérieurement, ou annulée ; une réduction de fonctionnalité du système est alors possible.

Les quelques expérimentations réalisées avec SOS laissent penser qu'il est possible de fournir, même dans le cadre très contraint des réseaux de capteurs sans fil, quelques abstractions classiquement fournies par des systèmes d'exploitation plus génériques, et ce sans forcément sacrifier les performances et l'efficacité énergétique. Néanmoins, aucun support n'est fourni par SOS pour l'écriture des pilotes de périphériques, et la gestion de l'énergie est entièrement laissée à l'initiative de l'application.

Sur le modèle de SOS, Lorien [127] est un système d'exploitation pour réseaux de capteurs sans fil, construit à base de composants entièrement dynamique et axé sur les possibilités reprogrammation des applications. Cependant, aucun support de niveau système n'est détaillé pour la gestion des ressources.

TINYOS. Levis *et al.* [107], Hill *et al.* [88] ont proposé en 2000 la première version de TINYOS, basée sur le langage nesC décrit dans la section 3.1 page 36. La popularité de nesC est due à son utilisation par ce système. TINYOS hérite des particularités de ce langage, et supporte un grand nombre de plates-formes matérielles.

Un support pour la reprogrammation dynamique d'applications TINYOS est fourni par DELUGE [90, 60]. De même que dans SOS, la gestion de l'énergie est laissée à l'initiative de l'application et aucune décision n'est prise par le système.

Un avantage intéressant de ce système est lié à l'outil TOSSIM [106], qui permet la simulation de réseaux de capteurs sans fil complets à partir d'applications utilisant des composants fournis par TINYOS (*cf.* § 4.2.2). TOSSIM simule le comportement de chaque nœud en exécutant une version compilée du code de l'application, dans laquelle les composants des pilotes de composants matériels sont remplacés par des versions reliées au moteur de simulation.

Nous aborderons dans la section 5.1 page 45 les caractéristiques d'une version plus récente de TINYOS, notamment en ce qui concerne la gestion des ressources.

Pixie OS. Lorincz *et al.* [111] proposent une architecture de système d'exploitation incorporant d'autres abstractions, et implanté en nesC afin d'autoriser la réutilisation des pilotes de périphériques employés dans TINYOS.

Inspiré d'ECOSystem [162] et Eon [143], Pixie OS fournit un support pour l'implantation d'applications tenant compte de la gestion de ressources. Ces dernières peuvent être la bande passante du médium radio, l'espace de stockage, ou encore l'énergie. Pour cela, les modules sont organisés selon un modèle *flot de données*, accompagné de tickets symbolisant la propriété d'une certaine quantité de ressource physique ; Les tickets pour la réservation d'une certaine quantité de ressource sont obtenus par l'intermédiaire de *médiateurs* (« *brokers* ») dédiés à chacune, qui prennent des décisions d'attribution (ou non) individuellement, sans connaissance concernant l'état des autres ressources.

L'utilisation des médiateurs pourrait autoriser l'implantation d'une politique de gestion globale des ressources, mais cela nécessiterait la mise en œuvre d'un dialogue entre chacun d'eux, complexe à réaliser dans le cadre d'applications émettant des demandes de ressources concurrentes. La programmation d'un médiateur global serait complexe à réaliser pour la même raison.

Enfin, ce système suppose la possibilité d'estimer un quantité de ressource physique consommée par une portion de code, ce qui n'est pas une tâche triviale.

CONTIKI. CONTIKI a été proposé par Dunkels *et al.* [57]. C'est un des systèmes les plus utilisés de nos jours dans le contexte des réseaux de capteurs sans fil. Il est essentiellement basé sur les proto-threads (voir § 3.2 page 37) ; il exécute donc principalement des programmes écrits sous forme événementielle.

Un objectif de CONTIKI est de réduire la taille et la complexité du code finalement déployé à un strict minimum, tout en étant suffisamment flexible pour permettre l'implantation d'applications complexes. Pour cela, ce système est architecturé sous forme d'exo-noyau, pour lesquels une réduction au maximum des abstractions fournies par le noyau est un objectif central de conception [65]. Par ailleurs, CONTIKI propose un ensemble de bibliothèques regroupant des fonctions d'accès au matériel ; les programmeurs d'applications incluent celles dont ils ont besoin lors de la phase de construction du système complet.

Une conséquence notable de l'approche de construction par exo-noyau est l'intrication résultante entre le code des applications, celui des services du système, et les pilotes de périphériques. Ceci peut entraîner dans les faits une difficulté d'appréhension du comportement global du système complet, et une modification comportementale au niveau des couches basses de celui-ci peut avoir des conséquences non négligeables.

Par défaut, deux types de processus s'exécutent dans une unique pile d'exécution commune dans CONTIKI :

- Les *traitants d'événements* sont les actions du modèle événementiel classique. Ils s'exécutent dès occurrence d'un événement auquel il sont abonnés ;
- Les *fonctions de scrutation d'événements* sont exécutées périodiquement par le répartiteur.

De plus, les communication inter-processus doivent exclusivement être effectuée au moyen d'événements postés dans la file d'attente globale, ou directement par appel de fonction.

Les interruptions ne sont jamais masquées dans CONTIKI afin de ne pas entraîner de latences dans leurs traitements. En conséquence, les traitants d'interruption ne sont pas autorisés à poster d'événements dans la file d'attente globale, puisque cela induirait des situations de concurrence potentielles ; ils doivent plutôt utiliser un mécanisme à base de drapeaux. Pour détecter l'occurrence d'une interruption, les processus doivent alors employer une fonction de scrutation, qui est exécutée entre chaque traitement d'évènement classique si un drapeau a été positionné (*i.e.*, assigné par un traitant d'interruption).

Redéploiement de code. Le noyau CONTIKI intègre une bibliothèque spécifique de chargement pour le redéploiement de code binaire, possiblement à travers le réseau [56]. Cependant, tout remplacement d'un service utilisant un état interne implique que celui-ci possède une fonction de sauvegarde de cet état : une fois installé, l'implantation de remplacement doit être en mesure de réutiliser cet état. Contrairement à SOS, CONTIKI n'impose pas d'utilisation de code *déplaçable* (PIC – « *Position Independent Code* »), *i.e.*, relogeable

en l'état, simplement par déplacement en mémoire et sans traduction : cette structure de code, bien que limitant la taille maximale des binaires, facilite la reprogrammation dynamique (il n'est plus nécessaire d'embarquer un dispositif de transfert (« *relocation* ») encombrant).

Hybridation avec un modèle multifils. Optionnellement, une bibliothèque dédiée est disponible dans CONTIKI pour implanter des processus préemptibles. Ces derniers possèdent une pile propre, et peuvent donc exécuter des fonctions bloquantes, ou encore effectuer des calculs coûteux sans perturber les latences des autres traitements de manière trop conséquente.

Aspects relatifs à la consommation énergétique. De même que la première version de TINYOS, CONTIKI n'intègre pas non plus de dispositif avancé de gestion de l'énergie. En revanche, Dunkels *et al.* [58] proposent Energest, un mécanisme destiné à estimer la consommation d'énergie pendant l'exécution en se basant sur des annotations dans le code source du système. Un suivi est alors possible via un ordinateur classique connecté au nœud à surveiller ; les différents modes de consommation de chaque périphérique, y compris le microcontrôleur, sont alors disponibles, permettant finalement d'identifier les phases de fonctionnement les plus consommatrices.

Enfin, l'unique information exposée par le noyau dans le but d'aider les applications à décider s'il est judicieux de placer le microcontrôleur dans un mode de basse consommation est la taille de la file d'événements en attente. Cette donnée est certes suffisante pour des applications simples, mais des piles logicielles plus complexes peuvent rendre la décision très délicate. Le choix du mode de basse consommation adéquat est aussi rendu très complexe par cette décision, et comme pour TINYOS, des solutions *ad hoc* doivent souvent être employées pour pallier à ce manque d'informations.

Autres propositions. En 2010, Chen *et al.* [37] proposèrent Enix, qui utilise un support du compilateur pour autoriser la virtualisation de la mémoire sans support matériel. Ce mécanisme est implanté par insertion de code à la compilation, et l'ajout d'un mécanisme de *swap* assuré à l'aide de mémoires lentes (type flash). LiteOS [27, 28]⁴ propose des abstractions de type UNIX [139] pour programmer des applications sur réseaux de capteurs sans fil.

4.2.3 Une comparaison

Mozumdar *et al.* [121] et Duffy *et al.* [54] ont comparé plusieurs approches pour différencier les modèles de programmation purement événementiels et multifils. Ils en ont déduit qu'elles se valent dans la plupart des cas, mais relèvent, bien entendu une variation selon le type d'applications et le degré de parallélisme requis, ainsi que les motifs de communication entre les tâches.

Comme toujours, ses compromis sont à rechercher entre l'aisance d'expression des tâches parallèles et des communications et l'efficacité des implantations.

4.2.4 Hybrides

Alors qu'un système multifils peut simplement être utilisé pour implanter un système d'exécution purement basé sur des événements, l'opération inverse requiert des adaptations non triviales du système d'exécution. Pour cela, des approches hybrides laissent le choix du modèle de concurrence au programmeur de l'application, autorisant ainsi l'intégration de tâches longues dans un système événementiel.

Par exemple, Welsh et Mainland [157] proposent l'usage de deux fils d'exécution, un seul étant autorisé à appeler des fonctions bloquantes : une unique pile d'exécution est requise dans ce cas.

4. LiteOS a été développé à partir d'une chaîne d'outils dédiés à la programmation des applications pour réseaux de capteurs sans fil en C++.

TOSThreads [101] est aussi une extension de nesC destinée au support de tâches longues ayant leur propre pile d'exécution. Dans les faits, il s'agit d'une bibliothèque de fonctions *ad hoc* écrites directement en C et langage d'assemblage.

Nitta *et al.* [122] proposent Y-Threads, un modèle de programmation multifiels où toutes les tâches, en plus de leurs piles respectives d'usage classique, partagent une même pile d'exécution dans laquelle les appels de fonctions non bloquantes peuvent être réalisés. Les auteurs avancent que cette technique permet une réduction substantielle de la mémoire occupée, tout en permettant une expression plus simple des applications.

TinyThread [116] reprend également l'idée d'une bibliothèque de fonctions bloquantes et de pile partagée afin d'implanter un modèle multifiels coopératif. Ils fournissent en plus un outil d'estimation de la taille minimale des différentes piles d'une application donnée.

Enfin, SensPire OS [52] est conçu à partir de techniques en provenance des applications temps-réel pour améliorer la *prédictibilité* de leur système. SensPire OS est programmé avec le langage associé CSpire, dont le processus de compilation permet le partage d'une partie des piles d'exécution des tâches. Il traite en outre les requêtes d'interruption matérielles en deux phases pour réduire certaines latences de traitement.

4.3 Machines virtuelles de haut niveau

Dans le contexte de la programmation des nœuds de réseaux de capteurs sans fil, les machines virtuelles de haut niveau sont proposées dans l'optique de réduire de la taille du code, et permettre un déploiement plus rapide des applications.

4.3.1 Vue d'ensemble

Par opposition à une machine virtuelle *système* (ou *plate-forme*) qui *émule* fidèlement une plate-forme existante⁵, une machine de haut niveau (parfois également appelée « machine langage ») simule une machine abstraite. Les exemples de telles machines abstraites incluent la machine virtuelle Java [110], ou encore la « *Categorical Abstract Machine* » [44], initialement cible d'implantation du langage CAML [160].

Similairement aux machines classiques, une machine virtuelle de haut niveau interprète un jeu d'instructions ou code abstrait ; si ces instructions sont codées sur huit bits, on parle généralement de *code-octet* (*bytecode*). Une machine abstraite fournit généralement une abstraction suffisante pour rendre les programmes qu'elle interprète indépendants de la plate-forme réelle sur laquelle elle s'exécute. Les avantages d'utilisation d'une telle machine abstraite incluent principalement :

- La *sûreté* de l'environnement d'exécution, au sens où les erreurs d'une application peuvent en principe être détectées en amont par l'interprète des instructions du langage (principe du *bac à sable*) ;
- La *compacité* du code produit : tout calcul arbitrairement complexe fréquemment utilisé dans les applications peut être codé par une seule instruction, puis implanté nativement et de manière optimisée dans l'interprète. Le code des applications, lorsqu'elles sont compilées, est donc *a priori* à la fois moins gourmand en espace mémoire, et moins coûteux à déployer sur un réseau qu'un code binaire équivalent. Le coût calculatoire de l'interprétation peut aussi être modéré pour cette même raison, d'autant plus que le domaine des applications est restreint.
- La *portabilité* : un programme ciblant une machine abstraite peut théoriquement s'exécuter sur toute plate-forme où cette dernière a été implantée⁶.

De l'ensemble de ces caractéristiques, il ressort que l'utilisation de machines virtuelles de haut niveau dans le contexte des réseaux de capteurs sans fil peut avoir un impact significatif en ce qui concerne l'occupation

5. en règle générale...

6. Cet argument est en vérité à mitiger si la machine virtuelle exploite fortement des propriétés d'un matériel particulier. Par exemple, le modèle de cohérence mémoire de la spécification Java [110] n'est pas adaptable sans difficulté à toute plate-forme (matérielle + système d'exploitation) [130].

mémoire des applications ainsi que les possibilités de reprogrammation après déploiement. Nous aborderons dans la suite de cette section quelques exemples d’approches utilisant ce type d’outils afin d’en identifier les principaux bénéfices et défauts dans ce contexte.

4.3.2 Machines virtuelles construites à partir d’applications

L’usage de machines virtuelles réglées en fonction des applications a initialement été proposé par Levis *et al.* [105]. Un nombre croissant d’approches se basent sur ce concept pour réduire la taille des codes ainsi que les coûts de leurs déploiements. Costa *et al.* [43] en ont déjà répertoriées en 2007, en incluant les machines abstrayant le réseau complet ; nous aborderons les principales solutions consacrées à la programmation des nœuds individuellement dans la suite de cette section, en mettant l’accent sur leurs propriétés en matière de consommation énergétique et d’implantation.

MATÉ. Levis et Culler [104] proposent l’usage de MATÉ, un générateur de machine virtuelle à pile de haut niveau optimisée. Les applications sont déployées et interprétées sur les nœuds sous forme d’un code-octet configurable, incluant des instructions abstraites de haut niveau afin d’obtenir des programmes efficaces en matière de performance et d’occupation de la mémoire.

Les données à fournir dans le cadre de la solution proposée sont les suivantes :

- Une spécification du langage de programmation, qui permet à MATÉ de générer un code-octet dédié (*e.g.*, de type opération arithmétique, adressage) ;
- Les événements d’entrée auxquels les programmes peuvent réagir (*e.g.*, expiration de compteur de temps ou réception de paquet) ;
- Les primitives du langage, opérations abstraites qui seront implantées nativement et représentées par une instruction unique (*e.g.*, calcul de racine carrée, moyenne, tri ou filtre ; lecture d’une mesure depuis un capteur, ou envoi d’un message).

Ces éléments permettent de générer une machine virtuelle sous la forme d’un programme à intégrer dans TINYOS (*cf.* § 4.2.2 page 40), ainsi qu’un compilateur du langage d’entrée précédemment spécifié vers le code-octet interprété par celle-ci. MATÉ intègre un système de propagation du code dans le réseau [108], ainsi qu’un mécanisme *ad hoc* de sécurisation des transmissions de code.

Dans MATÉ, des sections critiques implicites sont ajoutés automatiquement autour des accès à chaque variable qui est également écrite par un traitant d’évènement. Ce mécanisme est introduit pour permettre une gestion transparente des situations de concurrence, et simplifier la programmation par rapport à nesC.

Dunkels *et al.* [56] ont implanté CVM (« *Contiki Virtual Machine* ») dans CONTIKI, une machine virtuelle spécifique similaire à MATÉ. CVM exploite les possibilités de chargement dynamique de code binaire proposées dans CONTIKI. Il ressort de leurs expérimentations que l’approche combinant une implantation à la fois en code natif et en code interprété peut s’avérer énergétiquement efficace, sans toutefois empêcher le redéploiement de code binaire.

TINYVM. Plus récemment, Hong *et al.* [89] proposèrent d’utiliser VMgen [66] afin de générer une machine virtuelle optimisée pour une application particulière. nesC est utilisé comme langage d’entrée, et des annotations doivent être ajoutées manuellement afin de spécifier quelles portions de la source sont à compiler en code-octet ou nativement. Les fonctions qui constituent des instructions de la machine virtuelle à générer sont aussi à préciser. L’interpréteur résultant est finalement synthétisé en fonction de l’application, et seuls les code-octets utilisés sont effectivement implantés pour en réduire la taille.

En outre, les mêmes auteurs proposent de tirer partie de la connaissance du code de l’ensemble de l’application pour compresser le code-octet à l’aide d’un algorithme de Huffman classique, dont la décompression reste relativement peu coûteuse et peut donc être réalisée en-ligne.

Puisque l’architecture de TINYVM est entièrement basée sur nesC, le modèle de concurrence utilisable dans ce cadre est exclusivement basé sur des évènements.

Autres propositions. SCYLLA [144] proposent une approche de compilation en-ligne de code compact au moment de sa réception. Cette technique permet de réduire la taille du code circulant dans le réseau en limitant les pénalités sur les performances.

NUCLEOS, conçu par Hahn et Chou [79], interprète des scripts compacts exprimant des programmes *synchrones flots de données* (voir le chapitre suivant à ce sujet) que les même auteurs avaient avancé précédemment [78]. Ces scripts sont le produit d'une compilation effectuant un ordonnancement statique des tâches concurrentes du programme.

Enfin, Balani *et al.* [6] avancent l'usage de DVM (« *Dynamically extensible Virtual Machine* ») une machine virtuelle extensible se basant sur les modules dynamiques de SOS (cf. § 4.2.2 page 40).

Ces machines virtuelles de haut niveau facilitent les reconfigurations et reprogrammations d'une partie du logiciel applicatif par le réseau, avec la transmission de scripts et codes de tailles réduites. De plus, leurs performances sont généralement acceptables, par exemple dans MATÉ grâce aux codes-octets optimisés pour les opérations les plus fréquentes, et les primitives calculatoires et d'entrées-sorties implantées en code natif.

En revanche, peu de considérations sont apportées à la gestion des ressources matérielles, qui restent implantées dans le moteur d'exécution des machines virtuelles.

Nous pouvons noter qu'aucun mécanisme n'est jamais prévu parmi les approches que nous avons énumérées, pour faciliter la programmation des pilotes de périphériques ainsi que leurs interactions. Aussi, la facilitation de la gestion des ressources est rarement prise en compte. Les avancées de ces systèmes se focalisent souvent sur la réduction de leur empreinte mémoire et la mise en place de mécanismes pour la reconfiguration dynamique. Enfin, ils supposent généralement que la couche logicielle de gestion des périphériques est immuable.

4.4 Discussion

Il ressort de cette énumération des techniques et méthodes d'implantation d'applications pour les réseaux de capteurs sans fil une forte focalisation des projets de recherche sur les méthodes et modèles de programmation qui permettent d'implanter efficacement des applications sur ces plates-formes. Les propriétés de reprogrammation dynamique et de conception par composants ont notamment été très étudiées. Les approches visant la réduction de l'empreinte mémoire des systèmes sont également nombreuses.

5 La gestion des périphériques dans les réseaux de capteurs sans fil

Nous notons l'existence de quelques travaux destinés à la gestion de ressources au moyen d'une organisation spécifique des pilotes de périphériques. La première que nous évoquerons ici est la méthode utilisée à partir de la seconde version de TINYOS. La seconde est une approche dédiée aux systèmes multifiils.

5.1 Gestion de ressources décentralisée

ICEM (« *Integrating Concurrency Control and Energy Management (in Device Drivers)* ») est la solution utilisée pour le contrôle de ressources dans TINYOS (cf. § 4.2.2 page 40). Elle a été proposée par Klues *et al.* [100], et consiste, comme son nom l'indique, en une méthode intégrant la gestion de la concurrence et de l'énergie dans les pilotes de périphériques des plates-formes. (Dans cette approche et contrairement à la terminologie employée dans cette thèse, les services de niveau systèmes (différentes couches de gestion du réseau, systèmes de gestion de fichiers), sont également nommés « pilotes de périphériques ».)

L'idée principale est d'exploiter l'information offerte par les différentes requêtes d'accès aux périphériques par l'application, de manière potentiellement concurrentes, pour contrôler le mode de fonctionnement du périphérique.

Pour cela, ICEM offre au programmeur d'applications trois manières d'exprimer le niveau de concurrence supporté par un périphérique :

Pilotes virtualisés. Un pilote *virtualisé* autorise une concurrence implicite entre plusieurs *clients* (composants utilisateurs), du point de vue desquels le pilote est toujours disponible. Les requêtes sont placées dans des files d'attente et servies selon une politique désirée (e.g., équité) à l'aide d'informations sur l'état des clients. Cette même information sert également à gérer le mode de fonctionnement du périphérique ; à l'éteindre lorsque la file d'attente devient vide par exemple.

Bien que la plus simple d'utilisation pour les clients, cette classe de pilotes ne doit être utilisées que lorsqu'une certaine latence est autorisée.

Pilotes dédiés. Un pilote *dédié* autorise un contrôle total de la part de leur unique client. Ceux-ci doivent utiliser une interface de contrôle spécifique pour allumer ou éteindre le périphérique.

Un pilote dédié est utilisé pour gérer les ressources de très bas niveau (e.g., un timer) ou encore des services de niveau système qui ne sont ni partagés entre plusieurs clients, ni liés à une ressource physique (et qui n'ont donc pas besoin d'être « éteintes »).

Pilotes partagés. Un pilote *partagé* gère plusieurs clients, mais leur présente une interface de gestion supplémentaire : un verrou de contrôle (« *power lock* »). Avant d'accéder à un pilote partagé, chaque client doit acquérir le verrou associé. Ce même verrou doit être libéré par le client s'il n'accède plus au pilote. Contrairement à un pilote virtualisé qui place ses requêtes fonctionnelles dans une file d'attente et les exécute directement lorsqu'elle en sortent, un pilote partagé place les requêtes d'acquisition du verrou dans une file d'attente. Des *arbitres* servent à déterminer la politique de service des verrous de contrôle (e.g., dans l'ordre d'arrivée, tourniquet).

Les situations dans lesquelles un pilote partagé est utilisé, sont typiquement lorsque ses clients doivent exécuter des séquences d'opérations atomiques. Un pilote partagé est aussi requis lorsqu'une opération requiert la possession de plusieurs ressources en même temps.

Un composant additionnel par pilote partagé (le « *power manager* ») est en outre chargé de la politique de gestion de l'état de la ressource sous-jacente.

Discussion. Cette architecture mène clairement vers l'implantation d'une politique de gestion de ressources *décentralisée* entre les divers pilotes de périphériques. Il par ailleurs difficile de l'utiliser pour implanter un composant additionnel nécessitant une connaissance globale de l'état de la plate-forme.

Ainsi, le calcul du meilleur état de basse consommation du processeur proposé par les auteurs requiert l'écriture d'une fonction *ad hoc* coûteuse nécessitant l'interrogation de l'état de chaque module intégré au microcontrôleur. L'état des périphériques externes n'est pas pris en compte, alors que nous avons vu qu'ils peuvent impacter le meilleur état (cf. § 2.2.1 page 14, chap. 2).

Enfin, l'usage de verrous pour gérer la concurrence des accès aux ressources, combinée à la mise en file d'attente des requêtes d'acquisition par les pilotes partagés, mène très vite à des situations d'interblocage très difficiles à détecter. Les auteurs reconnaissent que l'augmentation de la complexité de l'infrastructure de gestion de la plate-forme et de la structure interne du système d'exploitation peut rendre difficile la détection et l'élimination des situations d'interblocages potentiels.

5.2 Une approche centralisée multifils

Choi *et al.* [41] suggèrent une architecture logicielle pour la programmation des pilotes de périphériques dans les systèmes multifils pour réseaux de capteurs sans fil. Cette organisation est bâtie autour d'un gestionnaire centralisant une information sur l'état de chaque périphérique.

Avant d'accéder à un composant matériel, la couche logicielle utilisant le pilote correspondant doit émettre une requête au gestionnaire centralisé grâce à un ensemble de routines dédiées. Ces routines reçoivent en argument un identifiant de périphérique. Selon leurs spécifications :

- Elles se terminent immédiatement en retournant une valeur signifiant que l'accès au périphérique est autorisé ou interdit ;
- Elles bloquent le flot d'exécution courant tant que le gestionnaire n'autorise pas l'accès au périphérique, ou qu'un certain temps ne s'est écoulé.

Finalement, le gestionnaire doit être notifié de la disponibilité d'un périphérique lorsque celui-ci n'est plus utilisé.

Grâce à la réception des requêtes d'accès et de libération des périphériques, le gestionnaire centralisé est en mesure d'interdire les accès concurrents aux ressources. À l'aide de ces informations également, il peut décider d'éteindre un périphérique qui n'est plus utilisé au moyen d'une interface dédiée qui doit être exposée par chaque pilote. Ces données l'autorisent aussi à déterminer le meilleur état de basse consommation du microcontrôleur.

Cette proposition d'architecture est la seule, à notre connaissance, à constituer une solution dédiée à la gestion globale de l'état d'une plate-forme matérielle. Cependant, ses auteurs ne donnent pas d'indications sur la manière de programmer le gestionnaire centralisé, qui doit pourtant supporter des requêtes concurrentes et administrer un certain nombre de primitives et structures de synchronisation entre les tâches. Aussi, des situations d'interblocages et de famines difficiles à détecter peuvent éventuellement survenir. Enfin, certains périphériques ne sont pas contrôlés alors qu'ils peuvent potentiellement impacter le meilleur mode de basse consommation du microcontrôleur.

6 Conclusion du chapitre

Nous avons abordé dans ce chapitre les méthodes de gestion de ressources et d'implantation des nœuds de réseaux de capteurs sans fil.

Le constat principal que nous faisons est qu'il existe déjà un grand nombre d'approches destinées à améliorer diverses propriétés de ces programmes. Notamment, les problématiques liées au redéploiement et à la reconfiguration dynamique des applications semblent nécessaires lors du développement dans ce contexte. La diminution de l'occupation de l'espace mémoire et de la taille des codes déployés par le réseau sont aussi des objectifs de recherche.

En revanche, la plupart des solutions d'implantation délèguent à l'application le travail de gestion des ressources. Or, la connaissance de l'état effectif de chaque composant n'est pas clairement disponible à ce niveau d'abstraction. Cependant, la seule proposition de méthode pour le développement des couches logicielles de gestion des périphériques qui permet le maintien d'une information centralisée ne supporte pas tous les modèles d'exécution concurrente.

Nous jugeons donc intéressant de proposer une solution visant à conserver la possibilité d'exécuter la majorité des applications déjà écrites, tout en assurant un contrôle global au niveau des pilotes de périphériques. Une approche basée sur un principe de (para-)virtualisation [159] nous paraît appropriée, puisque laissant la possibilité d'implantation des systèmes existants avec un minimum de modifications.

Dans le prochain chapitre, nous abordons les concepts qui sont nécessaires à la compréhension de la contribution, qui est l'objet de la dernière partie de cette thèse.

PROGRAMMATION SYNCHRONE ET SYNTHÈSE DE CONTRÔLEUR

Contenu du chapitre

| | | |
|-------|---|----|
| 1 | Programmation synchrone | 50 |
| 1.1 | Langages synchrones | 50 |
| 1.1.1 | Principe de compilation | 50 |
| 1.1.2 | Langages existants | 51 |
| 1.2 | Fragment de LUSTRE | 51 |
| 1.2.1 | Exemple de programme LUSTRE | 51 |
| 1.2.2 | Outils afférents | 52 |
| 1.2.3 | Compilation en C | 52 |
| 1.3 | Machines de Mealy Booléennes | 54 |
| 1.3.1 | Définition informelle | 54 |
| 1.3.2 | Exemple | 54 |
| 1.3.3 | Compositions | 55 |
| 2 | Synthèse de contrôleur | 56 |
| 2.1 | Principe | 56 |
| 2.1.1 | Contrôle automatique | 57 |
| 2.1.2 | Application aux système à évènements discrets | 57 |
| 2.2 | Outils existants | 58 |
| 2.3 | Exemples d'applications | 58 |
| 2.3.1 | Robotique | 58 |
| 2.3.2 | Gestion de tâches et tolérance aux pannes | 59 |
| 2.3.3 | Conception par contrats avec BZR | 59 |

Dans ce chapitre, nous présentons l'ensemble des concepts nécessaires à la compréhension de la contribution de cette thèse.

Le/La lect-eur/riche déjà familiarisé-e avec la programmation synchrone est invité-e à se reporter directement à la section 2 page 56. Cette dernière section du chapitre porte sur les principes de base de la synthèse de contrôleur, et peut également ne pas être parcourue par le/la lect-eur/riche connaissant déjà ces concepts.

1 Programmation synchrone

Dans un premier temps, nous décrirons succinctement les principes généraux de la programmation synchrone, puis nous prendrons l'exemple du langage LUSTRE comme représentant des langages d'implantation existants. Nous détaillerons ensuite une représentation de ces programmes sous forme de machine de Mealy Booléennes communicantes ; ces notations nous seront utiles pour la présentation de la contribution de cette thèse, dans les chapitres suivants.

1.1 Langages synchrones

Les différents langages synchrones existants [12] permettent l'expression de comportements et calculs complexes de manière compositionnelle. Ils partagent un certain nombre de principes de base, dont l'hypothèse de synchronisme et la construction modulaire :

Hypothèse de synchronisme. Ces langages permettent de décrire les programmes synchrones d'une manière similaire aux circuits séquentiels de même nom. Dans ces derniers, tous les calculs des composants sont cadencés à l'aide d'un signal d'horloge commun à tous les éléments du circuit. Idéalement, lors de sa conception fonctionnelle, les calculs et communications sont supposés instantanés, et les temps de propagation des signaux électriques dans le circuit résultant sont considérés dans un second temps.

De manière similaire, un programme synchrone est conçu en faisant l'hypothèse que l'ensemble des calculs et communications se font en temps nul lors d'*instants discrets* successifs ou *réactions* : il existe donc une notion de *temps logique discret*.

Les composants des programmes synchrones communiquent à l'aide de *signaux*. Les valeurs de ces derniers sont propagées selon le principe de *diffusion synchrone asymétrique* : les divers composants des programmes peuvent les émettre mais ne peuvent forcer leur absence (*i.e.*, pendant un instant, un signal est présent s'il est émis par au moins un composant). Par corollaire, il devient possible d'écrire un test d'*absence* d'un signal, « vrai » si celui-ci n'est pas émis dans l'instant. De plus, la définition d'instants successifs donne un cadre sémantique clair à la mémorisation des valeurs des signaux d'un pas d'exécution à l'autre (*i.e.*, ces éléments de mémorisation correspondent aux bascules et registres dans les circuits) ; la définition de comportements à partir des valeurs de signaux aux instants précédents est donc autorisée.

(Les différents signaux des programmes synchrones que nous considérons dans cette thèse sont des Booléens, mais des types plus complexes sont en général fournis par les langages synchrones existants.)

Modularité. Les programmes complexes sont exprimables à partir de compositions de sous-programmes. Les opérateurs de composition fournis par les langages autorisent l'expression de parallélisme (*i.e.*, cadencement de calculs avec la même horloge de base) ou de hiérarchie (*i.e.*, cadencement avec une horloge plus lente que l'horloge de base).

1.1.1 Principe de compilation

Tout programme synchrone peut s'exprimer sous la forme d'un *système de transitions symbolique*, c'est-à-dire :

- une fonction de transition f calculant à chaque instant discret, l'état suivant s' à partir de l'état courant s et des entrées I : $s' = f(s, I)$;
- une fonction de sortie g calculant les sorties O à partir des mêmes données : $O = g(s, I)$.

La compilation des langages synchrones a été beaucoup étudiée depuis leur création, y compris dans le cadre d'implantations sur plates-formes distribuées [30]. Dans cette thèse, nous nous intéressons à la compilation des programmes synchrones destinés à s'exécuter sur une plate-forme ne fournissant pas de support matériel pour le parallélisme effectif.

Ainsi, contrairement au domaine des circuits où ce parallélisme est conservé jusqu'à l'implantation, dans ce cadre le parallélisme de description des comportements concurrents disparaît le plus souvent lors de la phase d'implantation. Ce parallélisme de conception est *compilé*, et les différents calculs du programme sont ordonnancés statiquement pour obtenir du code purement séquentiel de comportement fonctionnellement équivalent.

Dans ce contexte, l'objectif du procédé de traduction d'un programme exprimé dans un langage synchrone est d'obtenir un *noyau réactif*, c'est-à-dire une représentation exécutable du système de transitions symbolique que le programme décrit. Le noyau réactif comprend au minimum :

- de la mémoire interne qui constitue l'état du programme, persistant d'un instant à l'autre ;
- une procédure d'initialisation de cet état ;
- une procédure $\text{step}(i)$, dont le rôle est de calculer les sorties et le nouvel état des variables internes en fonction des entrées i (il s'agit en général pour cette procédure de calculer f et g simultanément).

1.1.2 Langages existants

Il existe plusieurs langages synchrones de styles variés. SIGNAL [13, 4] et LUSTRE [29] sont des langages déclaratifs flots de données. SYNCCHARTS [5] et ARGOS [114] permettent de construire des programmes par composition d'automates (nous décrivons le formalisme à la base du langage ARGOS dans la section 1.3 page 54). ESTEREL [14] adopte un style impératif.

Dans la section suivante, nous nous focalisons sur la présentation du fragment du langage LUSTRE nécessaire à la compréhension de la contribution de cette thèse ; le/la lecteur/ice intéressé-e est invité-e à se reporter à la littérature citée pour des détails sur les constructions du langage LUSTRE non abordés ici. De plus, quelques aspects spécifiques à d'autres langages synchrones existants seront évoqués au chapitre 6.

1.2 Fragment de LUSTRE

Les principes fondamentaux du langage LUSTRE ont été initialement proposés par Caspi *et al.* [29] en 1987, et sont directement inspirés des réseaux de Kahn [97] sans tampons (*i.e.*, dont les processus communiquent à travers des files d'attente de taille bornée).

LUSTRE est un langage déclaratif *flot de données* : les composants, ou *nœuds*, qu'il permet de construire communiquent à l'aide de *signaux* (flux de données typés).

Les interfaces des nœuds possèdent des signaux d'entrées et des signaux de sortie. Leur implantation est réalisée de manière déclarative à l'aide d'un ensemble d'équations exprimant les sorties en fonction des entrées, et peut faire intervenir des opérateurs de *mémorisation* et d'*initialisation* sur les flux (ces derniers servent à préciser les valeurs initiales des données mémorisées, indéfinies sinon). Pour simplifier l'écriture des formules définissant les sorties des nœuds, des variables *internes* (ou *locales*) peuvent être utilisées.

1.2.1 Exemple de programme LUSTRE

Nous présentons un extrait de la syntaxe du langage à partir d'exemples simples.

Le listing 4.1 page suivante présente la définition du nœud `un_sur_deux`, dont la sortie `b` est vraie si son entrée `a` a été vraie un nombre pair d'instants depuis le début de l'exécution du programme.

À partir de ce premier programme, nous construisons dans le listing 4.2 page suivante un nœud `un_sur_quatre` par composition parallèle de deux instances du nœud `un_sur_deux`. Ces deux instances communiquent par le signal `b` « vrai » toutes les deux occurrences de `a` ; similairement, cest « vrai » tous les deux instants où `b` l'est. Ainsi, on obtient finalement par application de deux opérations simples à partir de `a`, un nœud dont la sortie Booléenne n'est « vraie » que tous les quatre instants où son entrée est « vraie ».

```

1 node un_sur_deux (a: bool) returns (b: bool);
2 var pair: bool; -- Signal interne, pour simplifier l'expression des équations.
3 let
4   -- Nous ne mémorisons que la parité du nombre d'instants où 'a' est vrai
5   -- depuis le premier instant (initialement considéré « pair » pour simplifier,
6   -- d'où l'initialisation de 'pair' à « vrai » si 'a' n'est pas positionné au
7   -- premier instant --- 'not a ->') ; 'pre pair' représente la valeur de 'pair'
8   -- à l'instant précédent.
9   pair = not a -> if a then not pre pair else pre pair;
10  -- 'b' est « vrai » si 'a' est positionné un nombre pair de fois depuis le
11  -- premier instant.
12  b = a and pair;
13 tel

```

LISTING 4.1 – Un nœud LUSTRE simple, dont la sortie `b` est positionnée (i.e., de valeur « vraie ») tous les deux instants pour lesquels l'entrée `a` est « vraie ».

```

1 node un_sur_quatre (a: bool) returns (c: bool);
2 var b: bool; -- Signal interne permettant aux instances
3 let -- de 'un_sur_deux' de communiquer.
4   b = un_sur_deux (a); -- Première instance.
5   c = un_sur_deux (b); -- Seconde instance.
6 tel

```

LISTING 4.2 – Composition en LUSTRE de deux instances du nœud `un_sur_deux` du listing 4.1. Ces deux instances communiquent par l'intermédiaire du signal local `b`. La sortie `c` du nœud `un_sur_quatre` est positionnée tous les quatre instants pour lesquels l'entrée `a` est « vraie ».

1.2.2 Outils afférents

Un compilateur LUSTRE académique a été développé¹. La chaîne d'outils autour du langage LUSTRE permet également la vérification de propriétés par *model-checking* [80] ou *interprétation abstraite* [94], le débogage de programmes réactifs [72] ou encore leur test [93].

Sur le plan industriel, SCADE a été construit à partir de LUSTRE, et est employé pour la conception de systèmes critiques de domaines comme l'aéronautique, le nucléaire ou le ferroviaire, pour ne citer que quelques-uns.

LUSTRE autorise une intégration simple des programmes synchrones dans des modules C, et permet également d'exprimer l'appel de fonctions externes directement dans le code LUSTRE des nœuds. Dans ce dernier cas, il est cependant nécessaire de prendre garde aux différents effets de bords induits².

1.2.3 Compilation en C

Il existe deux méthodes pour obtenir un noyau réactif à partir d'un programme en LUSTRE. Elles ont pour point commun de traiter un programme dans lequel les nœuds ont été *expansés*, c'est-à-dire que le programme à compiler est préalablement transformé en un seul système d'équations par un procédé similaire à l'*inlining* de code impératif.

1. Ce compilateur est disponible à l'adresse <http://www-verimag.imag.fr/The-Lustre-Toolbox.html>.

2. Le processus de compilation, jusqu'à la sixième version du langage, ne garantit pas qu'une fonction externe est appelée autant de fois qu'elle n'est écrite dans le programme. Il n'y a pas non plus de garantie sur l'ordre des appels de code externe.

```

/* Le noyau réactif produit par le compilateur Lustre: */

int M1, M2, M3; /* Variables d'état. */
void init () { M3 = 1; } /* Initialisation. */
void run_step (int a) { /* Exécute un instant et produit les */
    int L1, L2, L3, L4, L5, L6, L7; /* sorties en fonction de l'entrée 'a'. */
    L3 = M3 | M1; L2 = ~L3; L6 = M3 | M2; L5 = ~L6 & a; L1 = L2 & L5;
    main_0_c(L1); /* Émet la sortie 'c'. */
    L4 = L3 & ~L5; L7 = L6 & ~a; M1 = L4 | L1; M2 = L7 | L5; M3 = 0;
}

/* Exemple de code à ajouter pour obtenir un programme complet: */

void main_0_c (int x) {
    printf ("%d\n", x); /* Affichage de la sortie 'c'. */
}

int main () {
    init (); /* Initialisation. */
    while (1) { /* Infiniment. */
        int a;
        printf ("Donnez 'a' (0/1):"); /* Récupération de l'entrée 'a'. */
        scanf ("%d", &a);
        run_step (a); /* Calcul de l'état suivant, et production des sorties. */
    }
}

```

LISTING 4.3 – Noyau réactif obtenu par compilation en boucle simple du programme LUSTRE du listing 4.2, et exemple de programme principal le pilotant.

Compilation en automate. La première méthode consiste à calculer explicitement les fonctions de transition et de sortie de l'automate associé au comportement du programme. Le procédé consiste à construire explicitement l'ensemble de ses états (les états étant distingués par les valeurs des signaux Booléens mémorisés), et à associer à chacun une séquence d'instructions calculant l'état suivant (les nouvelles valeurs des signaux Booléens mémorisés) et les sorties en fonction des entrées. L'état du programme en mémoire est alors constitué de l'adresse de la séquence d'instructions correspondante, plus les signaux mémorisés non Booléens.

Cette méthode de compilation n'est envisageable que pour des systèmes n'incorporant que peu de mémoires Booléennes, en raison de l'impact de l'explosion combinatoire du nombre d'états sur la taille du code produit.

Compilation en boucle simple. Le principe de la seconde méthode consiste à décomposer et ordonner les équations à partir des dépendances entre les signaux. L'évaluation séquentielle de celles-ci permet de calculer les sorties et les nouvelles valeurs des variables internes à partir des entrées et des valeurs précédentes des signaux mémorisés.

Bien que produisant parfois une procédure `step()` calculatoirement plus coûteuse que la méthode précédente, la taille de ce code est linéaire en la taille du programme LUSTRE initial.

Le listing 4.3 présente le résultat de l'application de la seconde méthode de compilation du programme LUSTRE donné au listing 4.2 page précédente. La première partie est le noyau réactif proprement dit : la déclaration des variables d'état et des fonctions d'initialisation `init()` et d'exécution d'un instant `run_step()`. La seconde partie illustre une manière d'utiliser ce noyau réactif en fournissant une procédure pour la sortie du programme, et l'appel de la routine `run_step(i)` dans une boucle infinie. L'exécution de

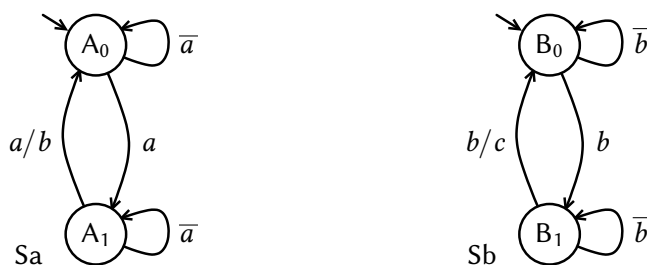


FIGURE 4.1 – Une représentation (Sa) du comportement du nœud `un_sur_deux` du listing 4.1 page 52, ainsi qu’une seconde version dont les signaux ont été renommés pour correspondre à l’instance de la ligne 5 du listing 4.2. Comme dans le nœud `un_sur_quatre`, ces deux comportements sont synchronisés par le signal local `b`.

ce programme C affiche « 1 » toutes les quatre occurrences de la valeur 1 pour l’entrée `a`³.

1.3 Machines de Mealy Booléennes

Comme on l’a vu, tout composant d’un programme synchrone peut s’exprimer sous la forme d’une machine de Mealy Booléenne finie, dont une transition est prise à chaque réaction. Le formalisme décrit ici est repris du travail de Maraninchi et Rémond [112, 114]. Nous évoquerons au chapitre 6 quelques outils permettant l’implantation de programmes directement exprimés sous forme de machines de Mealy Booléennes.

1.3.1 Définition informelle

Comme toute machine de Mealy au sens usuel, celles que nous considérons ici sont des automates possédant un nombre fini d’états et des transitions entre ceux-ci. Elles font intervenir un ensemble de signaux, dont une partie est reçue en entrée et l’autre est potentiellement émise ; notons cependant que ces deux parties ne sont pas nécessairement disjointes. Les transitions sont pourvues de deux attributs :

- Une *condition de déclenchement*, formule Booléenne quelconque portant sur les signaux d’entrée de l’automate. Lors d’une réaction, la transition est prise si l’automate est dans son état source et la condition est « vraie » ;
- Un ensemble de signaux émis lorsque la transition est prise.

En outre, puisque fondées sur les mêmes constructions que les automates finis classiques, les mêmes définitions s’appliquent également sur les machines de Mealy Booléennes que nous considérons. Ainsi, nous pouvons réemployer les notions de machine déterministe (*i.e.*, quel que soit l’état source et les valeurs des entrées, il existe au plus une transition dont la condition de déclenchement est « vraie ») ; de même pour la notion de réactivité (*i.e.*, quel que soit l’état source et les valeurs des entrées, il existe toujours une transition dont la condition de déclenchement est « vraie »), ou encore la minimisation par exemple.

1.3.2 Exemple

La partie gauche de la figure 4.1 est une représentation d’un exemple de machine de Mealy Booléenne Sa. Elle possède deux états `A0` et `A1`, dont le premier est l’état initial. La transition de `A1` à `A0` (étiquetée « `a/b` ») est déclenchée lorsque le signal `a` est « vrai » ; si elle est prise, elle émet également le signal `b`. Si l’ensemble de signaux émis par une transition est vide, alors nous l’omettrons dans les représentations des automates.

3. En fait, la sortie est impaire toutes les quatre occurrences d’une valeur impaire en entrée.

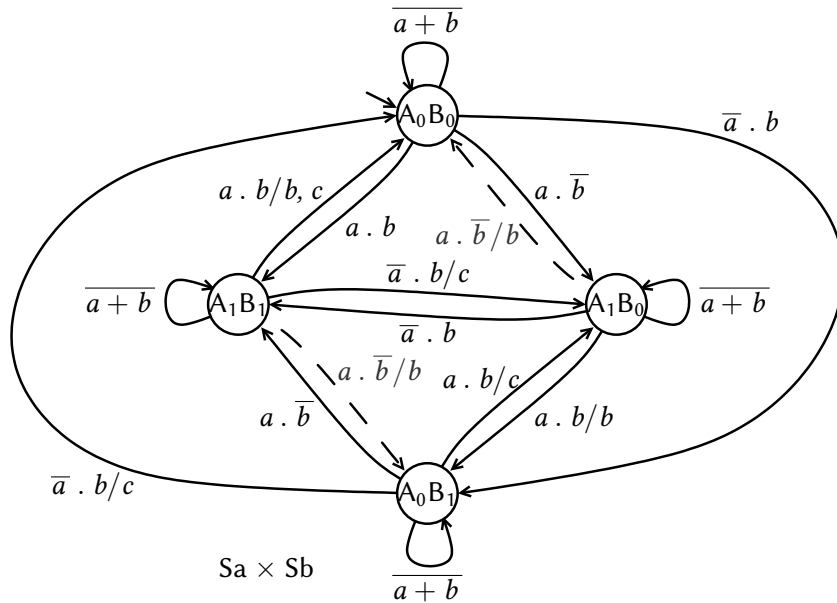


FIGURE 4.2 – Produit synchrone brut des machines S_a et S_b de la figure 4.1 page précédente. Les transitions dont la condition de déclenchement est intrinsèquement contradictoire (i.e., n'étant déclenchées qu'en l'absence d'un signal qu'elles émettent pourtant) sont indiquées en pointillés.

Par ailleurs, S_a est l'automate minimal décrivant directement le comportement intrinsèque du nœud `un_sur_deux` du listing 4.1 page 52. Par exemple, l'état A_0 correspond aux instants où l'entrée a a été « vraie » un nombre pair d'instants, c'est-à-dire que l'expression « **pre pair** » est « vraie ».

1.3.3 Compositions

L'expression de comportements complexes est possible directement au moyen d'une seule machine de Mealy Booléenne. Cependant, cette manière de les exprimer n'est pas modulaire et manque clairement de flexibilité. C'est pourquoi des opérateurs de composition des machines permettent la définition de ces mêmes comportements d'une manière modulaire.

Maraninchi [112] introduit plusieurs opérateurs de composition de machines de Mealy Booléennes communicantes. Parmi celles-ci, nous présentons la composition parallèle et l'encapsulation, qui nous serviront dans les chapitres suivants pour détailler la contribution de cette thèse.

Parallélisme. Dans le cadre synchrone, la composition parallèle appliquée aux machines de Mealy Booléennes se traduit dans ce formalisme par la synchronisation des changements d'états des comportements concurrents. Ainsi, le comportement global de la composition parallèle de deux machines s'obtient en calculant le produit synchrone des deux automates. De plus, les attributs d'une nouvelle transition $t_{i,j}$ résultant de la mise en parallèle de deux transitions t_i et t_j est calculée comme suit :

- La nouvelle condition de déclenchement $c_{i,j}$ est la conjonction des conditions originales : $c_{i,j} = c_i \cdot c_j$;
- Le nouvel ensemble de signaux émis $e_{i,j}$ est l'union : $e_{i,j} = e_i \cup e_j$.

Le résultat du produit synchrone des machines S_a et S_b de la figure 4.1 page ci-contre est donné dans la figure 4.2. Nous notons ce produit « $S_a \times S_b$ ». La transition d'état source A_1B_1 , destination A_1B_0 et d'étiquette « $\bar{a} \cdot b/c$ », est le résultat de la mise en parallèle des transitions de A_1 à A_1 (étiquetée « \bar{a} ») de la machine S_a , et de B_1 à B_0 (étiquetée « b/c ») de S_b .

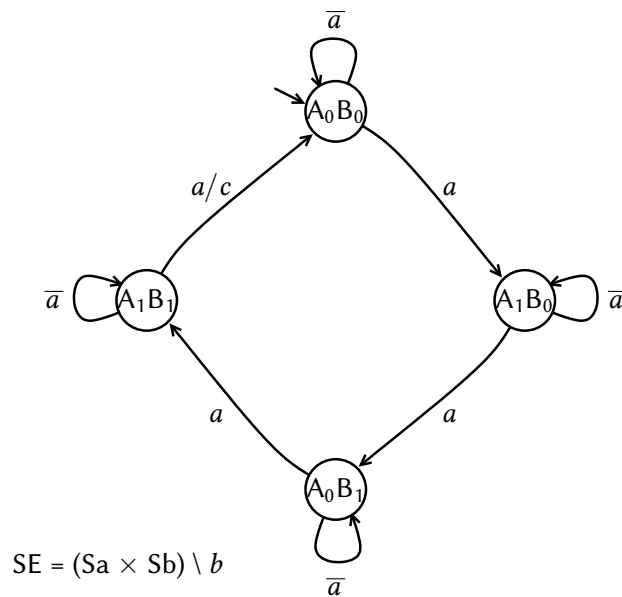


FIGURE 4.3 – L’automate produit SE , après encapsulation par le signal b . Il représente le comportement du nœud `LUSTRE un_sur_quatre` du listing 4.2 page 52.

Encapsulation. L’opérateur d’*encapsulation* sert à exprimer une limitation de la portée d’un signal donné, et permet la construction d’une nouvelle machine de Mealy dont le comportement n’est plus influencé par une éventuelle émission de ce signal par une autre partie du système qu’elle-même.

Appliquer un encapsulation par le signal b sur le produit $Sa \times Sb$ de la figure 4.2 page précédente, permet de s’assurer que le signal b , s’il est émis pendant un instant, ne l’est que par la machine $Sa \times Sb$. Pour représenter l’automate résultant, il suffit d’éliminer les transitions rendues insatisfaisables par la connaissance de l’absence b : c’est-à-dire les transitions dont la condition n’est valide que si b est émis, et n’émettant pas b elles-mêmes. Ce résultat est représenté dans la figure 4.3, qui décrit finalement le comportement attendu du nœud `LUSTRE un_sur_quatre` du listing 4.2 page 52 en émettant c tous les quatre instants où le signal a est présent.

2 Synthèse de contrôleur

2.1 Principe

Le principe de la synthèse de contrôleur a été proposé par Ramadge et Wonham [134] dans un cadre formel de théorie des langages. Ce principe de base a été dérivé en une méthode constructive de conception de systèmes informatisés pour le contrôle de phénomènes continu (*e.g.*, domaine de l’automatique), ou à évènements discrets. Les algorithmes de synthèse de contrôleur sont de la même famille et de même complexité que ceux du *model-checking*.

L’idée est de concevoir un *contrôleur* à partir d’un *modèle* d’entités à contrôler et d’une définition de *propriétés à garantir* portant sur ces modèles, tel que la composition de ce contrôleur avec le modèle vérifie les propriétés données. Pour que le contrôleur puisse avoir un impact sur le comportement du système afin d’interdire des comportements spécifiés incorrects par les propriétés à imposer, il faut préalablement déterminer certaines entrées dites *contrôlables* des modèles, sur lesquelles peut alors agir le contrôleur.

Les propriétés sont des contraintes imposées sur le système. Elle sont potentiellement quantitatives (*e.g.*, spécification d’une borne sur la valeur d’une sortie), peuvent exprimer des relations entre d’éventuels états

des modèles (e.g., exclusions mutuelles), ou encore incorporent des aspects temporels (e.g., ordonnancement d'évènements).

Les données de la synthèse comprennent possiblement des *objectifs de contrôle*, qui spécifient des critères à optimiser. Ces objectifs portent par exemple sur la minimisation d'une somme de grandeurs liées aux états du système (e.g., température, énergie).

Ainsi, cette méthode de conception autorise l'implantation correcte par construction de systèmes vérifiant des propriétés complexes, et ne nécessite pas de modification du comportement intrinsèque des modèles.

2.1.1 Contrôle automatique

En pratique, la synthèse de contrôleur est surtout appliquée dans le cadre du contrôle automatique de systèmes physiques. Dans ce contexte, les modèles représentent des phénomènes physiques à contrôler, et sont construits à partir de lois régissant ces comportements. Les propriétés globales à garantir portent sur l'environnement physique dont des informations sont récupérées via des capteurs, et l'objectif est de construire un système informatisé dans le but de le contrôler au moyen d'actionneurs.

Un exemple classique d'application de cette technique est le contrôle de température, où la propriété globale spécifie que cette grandeur doit rester dans un intervalle donné [8].

Dans cette thèse, nous nous intéressons plutôt à des systèmes à évènements discrets, c'est-à-dire dont les entités à contrôler ont un comportement descriptible principalement par des automates finis.

2.1.2 Application aux système à évènements discrets

Dans le domaine des systèmes à évènements discrets, la synthèse de contrôleur est applicable pour garantir des propriétés portant sur l'état des systèmes. Ainsi, des algorithmes de synthèse de contrôleur ont été proposés pour ce domaine.

Illustration du principe. Nous prenons un cas simple pour illustrer le principe de la synthèse dans le domaine des systèmes à évènements discrets. Pour cela, nous décrivons le résultat de l'algorithme de base de la synthèse de contrôleur directement appliqué sur des machines de Mealy Booléennes.

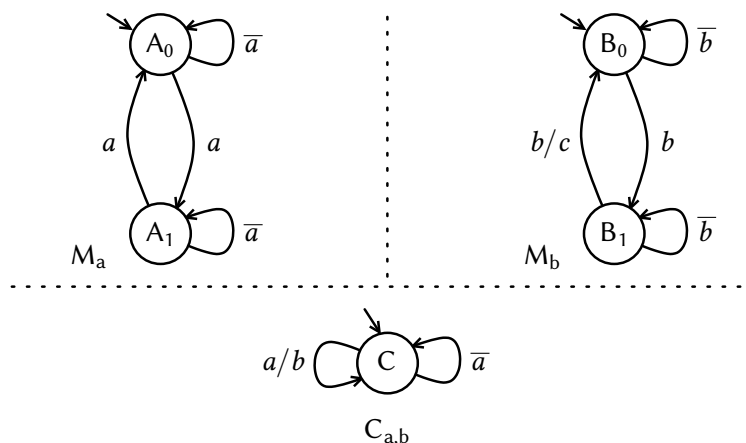


FIGURE 4.4 – Deux machines de Mealy isolées (i.e., qui ne communiquent pas), et un contrôleur $C_{a,b}$ qui garantit la propriété voulue après encapsulation par le signal contrôlable b .

Considérons donc les modèles de comportement M_a et M_b de la figure 4.4. À partir de ces machines et sans les modifier, nous proposons de synthétiser un contrôleur garantissant que si M_a est dans l'état A_0 (respectivement A_1), alors M_b est dans l'état B_0 (respectivement B_1). Pour cela bien entendu, nous devons

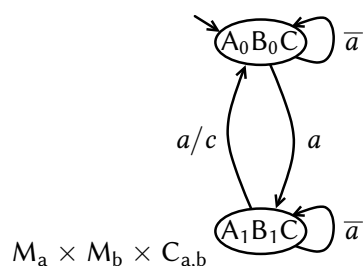


FIGURE 4.5 – Représentation du comportement du système contrôlé.

disposer d'un moyen d'intervention sur les signaux pilotant au moins un de ces automates ; dans le cas contraire, il est impossible de définir un comportement interdisant M_a et M_b d'atteindre simultanément un état interdit. Nous supposons donc que le signal b est contrôlable, c'est-à-dire qu'il soit une sortie du contrôleur à construire, et que a reste une entrée du système.

À partir de M_a , M_b , l'ensemble $\{b\}$ des entrées contrôlables, et la propriété à assurer, un algorithme de synthèse de contrôleur peut être utilisé pour extraire la machine de Mealy $C_{a,b}$ donnée dans la figure 4.4 page précédente (en l'occurrence ici, il s'agit de la fonction identité $f(a) = a$).

Après composition parallèle de M_a , M_b et $C_{a,b}$, puis encapsulation de b , on obtient la machine de Mealy donnée dans la figure 4.5. Elle décrit le comportement souhaité du système, où la propriété donnée est vérifiée.

2.2 Outils existants

Le travail relatif à la synthèse de contrôleur demeure essentiellement théorique, et peu d'outils permettent de générer du code à partir du contrôleur produit ; la plupart se focalisent sur l'évaluation des nouveaux algorithmes de synthèse ou la simulation.

Parmi les outils permettant de générer du code, UPPAAL-TIGA [31] permet la synthèse à partir d'automates temporisés. Cependant, cet objectif n'est pas réalisable directement est implique l'usage de transformations *ad hoc* du contrôleur et des modèles (dont un passage par Matlab-Simulink) — [47].

Dans la communauté synchrone, nous ne connaissons que SIGALI [11, 115], qui permet d'appliquer des algorithmes de synthèse à partir de modèles et propriétés exprimés en SIGNAL [13]. Cependant, SIGALI impose également des transformations non triviales du contrôleur si l'objectif est la génération de code exécutable.

2.3 Exemples d'applications

Plusieurs travaux proposent d'utiliser une approche de synthèse de contrôleur dans le cadre des systèmes à événements discrets.

2.3.1 Robotique

Altisen *et al.* [1] proposent d'appliquer ce principe pour la construction d'une couche logicielle de garantie de propriétés globales d'un système robotique. Conceptuellement, cette couche logicielle est intercalée entre l'application et les périphériques matériels qu'elle dirige (*e.g.*, bras mécanique). Ces propriétés portent directement sur l'état des différentes pièces mécaniques pilotées par le système informatique, dont certaines configurations sont interdites (car pouvant physiquement endommager le robot). Ils utilisent SIGALI pour générer un contrôleur et simuler son exécution conjointe avec un modèle du système.

Chandra *et al.* [34] appliquent une approche similaire pour le contrôle d'une chaîne d'assemblage automatisée, mais se placent dans le cadre de modèles asynchrones, et ne précisent pas si leur technique est automatisée.

2.3.2 Gestion de tâches et tolérance aux pannes

Dans le contexte des systèmes distribués temps-réel, Girault et Rutten [74] proposent une méthode de conception d'un contrôleur destiné à gérer la migration des tâches afin d'assurer un certain niveau de tolérance aux pannes, en accord avec une politique de gestion donnée. Cette proposition se base sur l'application de la synthèse de contrôleur à partir d'une modélisation des tâches et de l'environnement du système (modèle des fautes potentielles). Le contrôleur qu'ils synthétisent peut également remplir des objectifs d'optimisation, pour le placement des tâches par exemple.

Delaval et Rutten [49] étendent cette approche en définissant le langage NEMO dont le processus de compilation intègre une phase de synthèse de contrôleur. NEMO s'inscrit dans le contexte du développement sûr de systèmes temps-réel multi-tâches. Il sert à générer un gestionnaire de tâches correct à partir d'une spécification synthétique de ressources (périphérique partagé, temps CPU), de tâches consommant ces ressources en fonction de leurs différents modes de fonctionnement, et d'applications décrivant la fonctionnalité du système. Des contraintes temporelles sur l'usage des ressources peuvent aussi être imposées.

2.3.3 Conception par contrats avec BZR

Delaval *et al.* [48] proposent BZR⁴, un langage synchrone issu d'heptagon⁵ et une chaîne d'outils associée, destinés à autoriser l'implantation correcte par construction de composants spécifiés à l'aide de *contrats*. Leur approche s'appuie sur une application modulaire de la synthèse de contrôleur afin de garantir les propriétés imposées par les contrats de chaque nœud du programme ; SIGALI est incorporé dans BZR dans ce but. L'approche modulaire, bien que réduisant la quantité de contrôleurs potentiellement synthétisables, autorise toutefois une conception plus flexible des systèmes et a l'avantage considérable de réduire les coûts induits par la synthèse. Les contrôleurs obtenus sont finalement traduits en code C, puis intégrés aux code généré depuis les nœuds par un compilateur de langage synchrone.

BZR a été exploité efficacement dans divers contextes où le contrôle automatisé trouve une place de choix dans la génération de code de niveau système. Dans le contexte des systèmes adaptatifs et dynamiquement reconfigurables par exemple, Bouhadiba *et al.* [22] utilisèrent cette chaîne d'outils avec succès pour assurer un contrôle de ressources efficace à partir de modèles et de mesures sur les activités des composants d'un système de type serveur de données Internet.

4. Disponible à l'adresse : <http://bzs.inria.fr/>.

5. Le langage heptagon est de conception relativement proche de LUSTRE, mais destiné à l'investigation de la compilation modulaire des langages synchrones (<http://synchronics.inria.fr/>).

TROISIÈME PARTIE

CONTRIBUTION

PROPOSITION D'ARCHITECTURE LOGICIELLE : PROGRAMMATION SYNCHRONE ET CONTRÔLE GLOBAL

Contenu du chapitre

| | | |
|-------|--|----|
| 1 | Introduction du chapitre | 64 |
| 2 | Description générale de l'architecture proposée | 64 |
| 2.1 | Constitution de l'invité | 65 |
| 2.2 | Rôle de la couche de contrôle | 65 |
| 3 | La couche d'adaptation | 65 |
| 3.1 | Gestion de la plate-forme matérielle par l'invité | 66 |
| 3.2 | Exécution de l'invité | 66 |
| 4 | Description générale de la couche de contrôle et définitions | 66 |
| 4.1 | Structure de la couche de contrôle : la membrane et le noyau réactif | 67 |
| 4.1.1 | Le noyau réactif | 67 |
| 4.1.2 | La membrane | 68 |
| 4.2 | Chemins d'exécution dans la couche de contrôle | 68 |
| 4.2.1 | Premier exemple : cas d'une requête à l'initiative de l'invité | 68 |
| 4.2.2 | Deuxième exemple : cas d'une requête d'origine matérielle | 69 |
| 5 | Détails de la membrane | 70 |
| 5.1 | Gestion globale du microcontrôleur | 70 |
| 5.1.1 | Exécution de l'invité | 70 |
| 5.1.2 | Gestion du mode de basse consommation | 71 |
| 5.2 | Traitement des requêtes matérielles | 71 |
| 5.3 | Traitement des requêtes logicielles | 72 |
| 5.4 | Exécution du noyau réactif | 72 |
| 5.5 | Exécution des traitants d'interruptions virtuelles | 73 |
| 6 | Détails du noyau réactif | 74 |
| 6.1 | Automates de pilotes | 74 |
| 6.2 | Automate de contrôle | 75 |
| 6.3 | Construction du noyau réactif | 75 |

| | | |
|---|--|----|
| 7 | Un scénario d'exécution simple | 76 |
| 8 | Conclusion du chapitre | 79 |

1 Introduction du chapitre

Nous avons abordé dans les chapitres précédents les différentes techniques d'implantation de nœuds de réseaux de capteurs sans fil. De cette étude, nous avons pu déduire qu'une attention particulière a surtout été portée sur les modèles de programmation concurrente et la conception des pilotes de périphériques. En revanche, les solutions d'implantation des logiciels de gestion des plates-formes proposées ne permettent pas d'envisager une gestion globale de celles-ci, et s'orientent plutôt vers une prise de décision décentralisée, parfois même *ad hoc*. Au chapitre , nous avons pourtant identifié qu'un contrôle global s'avère nécessaire afin d'assurer un ensemble de propriétés exprimées sur l'ensemble des composants matériels. L'objectif de la proposition détaillée dans ce chapitre est d'apporter une solution réaliste à ce problème.

Nous présenterons la solution proposée par rapport à la pile logicielle complète dans un premier temps. Nous aborderons les détails de sa structure dans un second temps.

Enfin, on notera que pour la commodité de lecture des portions techniques, les listings de code et quelques éléments de structure de la solution seront retranscrits en anglais.

Sur la « para- »virtualisation. Une couche de *virtualisation* classique joue un rôle d'intermédiaire exclusif entre logiciel et plate-forme matérielle ; elle a pour but d'intercepter les accès aux ressources et d'assurer leur partage et éventuellement leur contrôle (même si ce dernier aspect est à notre connaissance assez peu répandu — le but du jeu étant en général de sérialiser les accès concurrents à une même ressource physique). La solution présentée dans cette thèse se base essentiellement sur la possibilité de *contrôle* offerte dans une telle architecture : elle permet de donner un cadre générique à la centralisation des décisions relatives à la plate-forme complète ¹.

En pratique, on ne peut se baser sans difficulté technique supplémentaire sur une solution de virtualisation au sens classique du terme en raison de l'absence de mécanismes matériels adéquats sur les microcontrôleurs usuellement employés pour la construction de nœuds de réseaux de capteurs sans fil (cf. § 2.2.1 page 12). C'est pourquoi nous proposons une constructions logicielle détournée : la *paravirtualisation* [159] consiste principalement à s'affranchir de la contrainte de non modification et d'isolation de l'invité en exposant une interface spécifique de manipulation d'une plate-forme abstraite gérée au sein d'une couche logicielle dédiée. On notera cependant que nous n'écartons pas l'idée d'adapter l'approche détaillée dans la suite de cette thèse en une plate-forme de virtualisation totale (de type système ou langage, indifféremment).

2 Description générale de l'architecture proposée

De manière similaire à l'approche de paravirtualisation classique, l'idée est d'insérer une *couche de contrôle* au travers de laquelle le reste de la pile logicielle interagit avec l'ensemble des composants matériels de la plate-forme.

La figure 5.1 page suivante représente le principe de l'approche que nous proposons. Empruntant le vocabulaire associé aux techniques de virtualisation logicielle, nous dénommerons *invité* (ou *système invité*) le haut de la pile logicielle. Cette partie peut comprendre plusieurs *tâches*, au sens de *processus* concurrents : une tâche peut par exemple être un fil d'exécution classique, ou encore un proto-thread [59] (cf. § 3.2 page 37).

1. On notera que le principe de virtualisation ne se limite en général pas à cette vision simpliste centrée sur la gestion de ressources partagées ; mais nous jugeons suffisante la présentation de l'idée générale qui en est faite ici dans le but d'appréhender la solution que nous proposons.

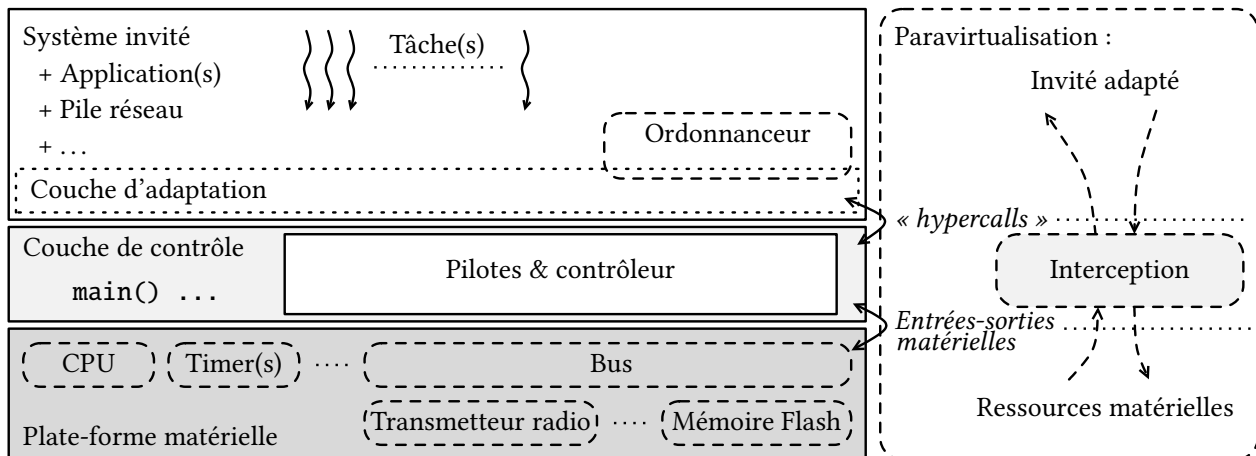


FIGURE 5.1 – Représentation globale de l'approche.

2.1 Constitution de l'invité

L'invité comprend :

- Le code de la ou les applications, possiblement constituées de plusieurs *tâches* (au sens de flots d'exécution concurrent) ;
- L'ensemble des services systèmes utilisés par les applications, tels que pile réseau ou système de gestion de fichiers ;
- Éventuellement, un *noyau* logiciel assurant au minimum la gestion des tâches des deux éléments précédents ;
- Une *couche d'adaptation*, dont le rôle est de gérer les interactions avec la couche de contrôle.

Afin de communiquer avec la plate-forme matérielle, l'invité utilise des fonctions² dédiées en lieu et place des manipulations de registres d'entrée-sortie usuelles. La couche d'adaptation représente la portion de l'invité modifiée afin de gérer la plate-forme à l'aide d'appels à ces fonctions.

2.2 Rôle de la couche de contrôle

Le rôle de la couche de contrôle est d'assurer une *gestion holistique, globale*, des ressources de la plate-forme. Pour cela, elle intercepte l'ensemble des requêtes matérielles (les interruptions) et logicielles (les appels de l'invité au travers de la couche d'adaptation). Elle maintient une *vue fidèle* de l'état de chaque ressource à gérer, c'est-à-dire principalement (mais pas exclusivement) les composants matériels tels que les périphériques et liens de communication.

L'interface qu'elle expose à l'invité présente une abstraction calquée sur les ressources effectivement pilotées, et est constituée de routines à appeler par la couche d'adaptation. La couche de contrôle émet également des requêtes d'interruption virtuelles (« *Virtual IRQ* » – VIRQ) que la couche d'adaptation peut traiter similairement au mécanisme classique de traitement des requêtes d'interruption matérielles.

3 La couche d'adaptation

La couche d'adaptation correspond à la partie de la pile logicielle invitée qui doit être modifiée afin d'utiliser la couche de contrôle pour accéder aux ressources gérées par cette dernière. Elle comprend des

2. Remarque sur le vocabulaire : le terme de « *fonction* » sera employé par la suite dans un sens généralisé « à la C » ; c'est-à-dire qu'elles seront désignées comme telles même si elles provoquent des effets de bord.

```

1 turn_adc_on ()          /* provided by the adaptation layer to the upper layers. */
2   R = on_sw (adc_on);   /* submit request 'adc_on' to the control layer. */
3   if (acka ∈ R) return success;
4   return error;        /* return an error code if request has been refused. */
5
6 turn_adc_on ()          /* (ibid), however blocking until ADC is on. */
7   R = on_sw (adc_on);   /* submit request 'adc_on' to the control layer. */
8   if (acka ∈ R) return success;
9   timer_wait (some time); /* if request refused, block for some time... */
10  turn_adc_on ();       /* ... and then retry. */

```

LISTING 5.1 – Deux versions d’une fonction de pilote de convertisseur analogique-digital, à inclure dans la couche d’adaptation. *adc_on* est une entrée de la couche de contrôle pouvant être refusée, auquel cas *ack_a* n’appartient pas à l’ensemble de comptes-rendus retourné par *on_sw(adc_on)*. Cette dernière fonction est exportée par la couche de contrôle (cf. § 4.1 page suivante et § 5.3 page 72 pour des détails la concernant).

routines abstrayant la gestion du matériel du point de vue de l’invité, ainsi qu’au moins une procédure de déclenchement de l’exécution de ce dernier.

3.1 Gestion de la plate-forme matérielle par l’invité

Au niveau du logiciel invité, la gestion de la plate-forme matérielle se fait au moyen d’un ensemble de routines exposées à celui-ci par la couche d’adaptation. Le rôle de ces routines est d’émettre les *requêtes logicielles* à la couche de contrôle. Le listing 5.1 illustre deux exemples de telles fonctions de gestion d’un convertisseur analogique-digital, destinées à faire partie de la couche d’adaptation d’un système invité.

Ces routines peuvent également abonner des *traitants* aux requêtes d’interruption virtuelles émises par la couche de contrôle, qui sont alors exécutés lors de l’occurrence de ces dernières.

Ainsi, cette partie du logiciel invité représente la traduction des accès au matériel par les pilotes de périphériques de l’invité en émissions de requêtes logicielles vers la couche de contrôle. Les traitants de requêtes d’interruption matérielles habituels deviennent des traitants de requêtes d’interruption virtuelles émises par la couche de contrôle.

3.2 Exécution de l’invité : première racine du graphe d’appel de l’invité

Le deuxième rôle de la couche d’adaptation consiste à fournir une procédure *run_guest()* destinée à être exécutée par la couche de contrôle. Cette routine représente le point d’exécution de l’invité qui ordonne et lance éventuellement l’exécution des tâches. Elle se termine lorsque toutes celles-ci sont bloquées et qu’il ne reste plus aucun calcul à réaliser par le CPU, signifiant alors que la couche de contrôle peut placer le microcontrôleur dans un mode de basse consommation jusqu’à la prochaine occurrence d’une requête d’interruption virtuelle³.

4 Description générale de la couche de contrôle et définitions

Nous donnerons dans la présente section une description de la structure et du comportement de la couche de contrôle nécessaires à la compréhension de son fonctionnement. Chaque partie de sa structure fera l’objet d’une description détaillée dans les sections 5 et 6.

3. On notera que si *run_guest()* est construite pour ne jamais se terminer, alors il est également possible d’employer une routine dédiée de la couche de contrôle pour réaliser cette mise en mode de basse consommation à l’initiative de l’invité

4.1 Structure de la couche de contrôle : la membrane et le noyau réactif

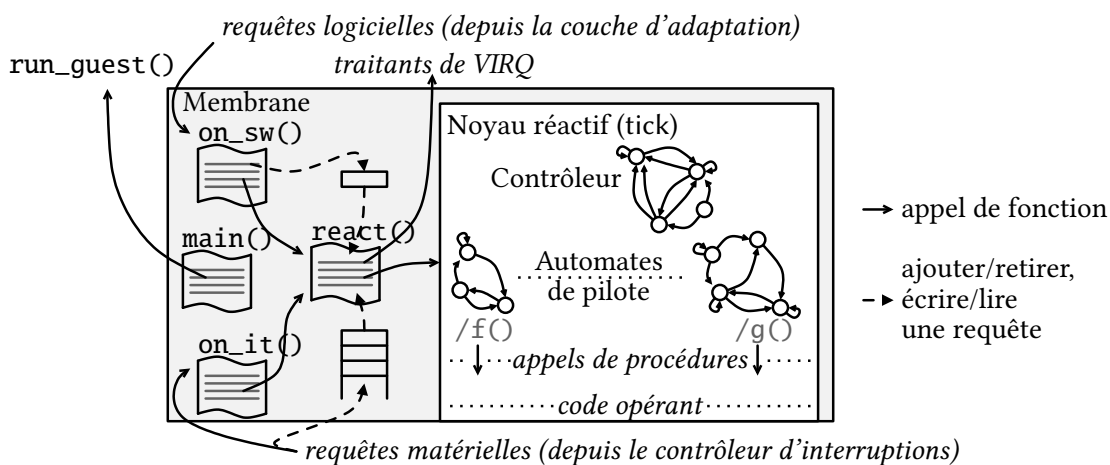


FIGURE 5.2 – Architecture de la couche de contrôle.

La couche de contrôle est composée de deux parties : le *noyau réactif* et la *membrane* ; la figure 5.2 représente cette structure interne.

4.1.1 Le noyau réactif

Le noyau réactif (parfois aussi dénommé tick dans la suite) provient de l'ensemble des *pilotes de périphériques*, ainsi que d'un *contrôleur*. Les pilotes de périphériques sont composés de deux parties selon de modèle « partie commande/partie opérative » (PC/PO) des architectures de calculateurs et automatismes classiques :

- Une portion de *code opérant* ou *bas-niveau* est la partie opérative du pilote. Il s'agit d'un ensemble de procédures séquentielles accédant principalement des registres d'entrée-sortie du matériel, plus éventuellement quelques données internes (e.g., caches, tampons) ;
- Un *automate*⁴ de pilote en représente la partie contrôle. Certaines de ses sorties déclenchent l'exécution de code opérant du pilote. Cet automate génère également des *comptes-rendus*⁵ afin de communiquer des informations concernant l'état du périphérique qu'il gère. Ses entrées se répartissent en deux catégories :
 - des *signaux d'entrée* représentant l'émission de requêtes matérielles ou logicielles ;
 - des *signaux d'approbation* émis par le contrôleur.

Le rôle du contrôleur est de restreindre le comportement du système global, au moyen des signaux d'approbation.

Interface du noyau réactif. Le tick est un objet, au sens où il « encapsule » des données et comportements ; il fournit les méthodes `init()` et `run_step(input_event)`. Cet objet est à considérer comme entièrement *passif*, en ce sens qu'il ne calcule que pendant un appel de l'une de ses méthodes. Il est le résultat de l'assemblage des codes opérants des pilotes ainsi que de la composition parallèle du contrôleur et des automates de pilote compilée en code exécutable. Ainsi, un appel de la méthode `run_step(input_event)`

4. Remarque sur le vocabulaire : les automates décrits ici sont plus précisément des « transducteurs », mais nous garderons le terme « automate » par commodité.

5. Attention toutefois, contrairement au cas de l'architecture matérielle selon le modèle PC/PO dont nous réutilisons la terminologie, ces comptes-rendus ne sont pas émis par le code opérant et reçus par les automates des pilotes. Ils sont plutôt issus de ceux-ci, puis éventuellement transmis à l'invité par la membrane.

exécute exactement une transition de l'automate compilé (*i.e.*, une transition du produit synchrone des automates), appelle possiblement quelques routines de code opérant des pilotes, et produit des comptes-rendus. `input_event` représente la *valuation* de chaque signal d'entrée des automates de pilote.

4.1.2 La membrane

Le rôle de la membrane est de déclencher l'exécution de `tick.run_step(input_event)` lorsque nécessaire. Pour cela, elle gère une file d'attente de requêtes matérielles, traite les requêtes en provenance de l'invité, et construit l'entrée `input_event` appropriée afin d'exécuter le noyau réactif. Après chaque appel au tick, elle récupère l'ensemble des comptes-rendus émis par les automates des pilotes, et les transmet à l'invité. Notamment, elle traduit les comptes-rendus en requêtes d'interruption virtuelles.

Constitution de la membrane. Du point de vue structurel, la membrane est constituée de quatre routines principales qui définissent son comportement :

- `main()` est la première routine exécutée lors du démarrage du microcontrôleur⁶. Cette routine gère l'initialisation du noyau réactif et de quelques composants matériels, ainsi que l'exécution de l'invité et la mise en mode de basse consommation du microcontrôleur ;
- `on_it()` est le seul et unique point d'entrée des requêtes matérielles : elle doit être appelée pendant l'exécution d'un traitant d'interruption après que ce dernier a placé au moins une requête dans la file d'attente correspondante. Cette procédure appelle `react()` ;
- `on_sw(r)` est l'unique routine qui doit être appelée par la couche d'adaptation pour émettre la requête logique `r`. Elle appelle `react()`, et se termine en retournant les comptes-rendus sur le traitement de la requête donnée ;
- `react()` est principalement constituée d'une boucle. Elle construit `input_event`, c'est-à-dire la valuation de chaque signal d'entrée des automates des pilotes, à partir des éléments de la file d'attente des requêtes matérielles, ainsi que d'une requête logique éventuelle. Elle appelle alors `tick.run_step(input_event)` et répète ce processus tant qu'il existe des requêtes matérielles dans la file d'attente associée. Si une requête logique `r` est prise en compte lors de la construction d'un `input_event` du tick, alors les signaux de sortie correspondant sont mémorisés afin d'être subséquentement utilisés comme résultats de l'appel `on_sw(r)`. Lorsqu'il n'existe plus de requête matérielle en attente, l'ensemble des comptes-rendus résultants de toutes les exécutions du tick depuis le dernier appel à `react()` son récupérés, et les traitants de requêtes d'interruption virtuelles qui leur sont éventuellement associés sont finalement exécutés.

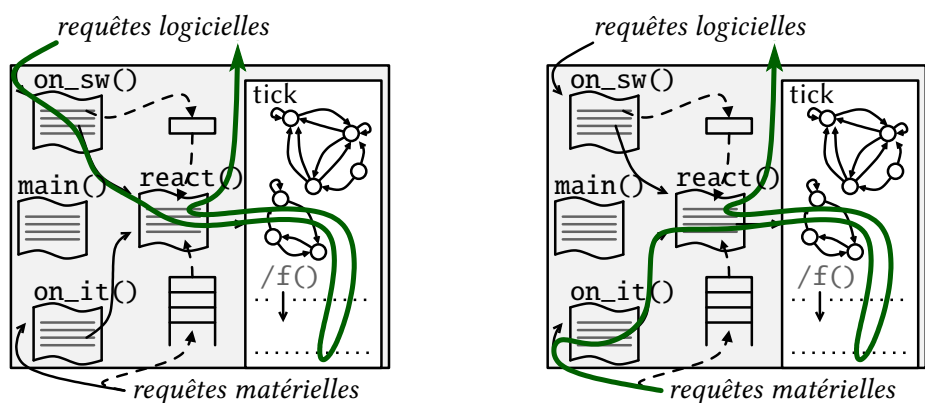
4.2 Chemins d'exécution dans la couche de contrôle

La figure 5.3 page suivante décrit deux chemins d'exécution typiques, un premier lors de l'émission d'une requête logique seule, l'autre dans le cas d'une interruption matérielle. Pour un exemple de scénario d'exécution plus général où des requêtes logicielles et matérielles sont émises concurremment, nous invitons le/la lecteur/rice à se reporter à la section 7 page 76.

4.2.1 Premier exemple : cas d'une requête à l'initiative de l'invité

Considérons dans un premier temps la figure 5.3-(a), et supposons qu'aucune interruption matérielle ne survient. La couche logique invitée, désirant effectuer une opération sur un périphérique matériel (*e.g.*, allumer le convertisseur analogique-digital), appelle la routine adéquate implantée dans la couche d'adaptation (par exemple une version de `turn_adc_on()` exposée listing 5.1 page 66). Cette dernière émet

6. quelques initialisations mises à part, celle de la mémoire de travail notamment.



(a) traitement de requête logique

(b) traitement de requête matérielle

FIGURE 5.3 – Illustration de deux chemins d'exécution différents dans la couche de contrôle. (a) Sur émission de la requête logique par la couche d'adaptation au moyen de `on_sw()`, la procédure `react()` est appelée. À son tour, cette dernière déclenche une exécution de `tick.run_step()`, et éventuellement des routines de code opérant (e.g., `f()`). Lorsque `tick.run_step()` se termine, quelques traitements préalablement abonnés aux interruptions virtuelles associées aux comptes-rendus du tick sont exécutés. (b) Similairement, l'émission d'une requête d'interruption matérielle déclenche l'exécution du tick et de traitements d'interruptions virtuelles au travers de `on_it()`.

alors une requête logique r en appelant la routine `on_sw(r)` de la couche de contrôle (toujours dans notre exemple, `on_sw(adc_on)`).

`on_sw(r)` mémorise r dans une cellule dédiée en mémoire et appelle `react()`, qui récupère à son tour la requête et appelle `tick.run_step(input_event)` après avoir construit `input_event`. Dans l'exemple, cet évènement est de la forme $adc_on \cdot \bar{x} \cdot \bar{y} \dots$ où x, y, \dots représentent l'ensemble des signaux d'entrée du tick excepté adc_on , c'est-à-dire toutes les requêtes matérielles et logicielles non émises dans ce cas explicatif. L'exécution du noyau réactif avec cet évènement déclenche la transition appropriée depuis son état courant, exécute éventuellement du code opérant, et se termine en retournant l'ensemble des comptes-rendus émis par la transition. `react()` récupère cette sortie pour l'associer à la requête r : ce résultat sera ensuite utilisé comme résultat de l'appel `on_sw(r)`. Avant de se terminer, `react()` appelle potentiellement quelques traitements d'interruptions virtuelles de la couche d'adaptation. Dans cet exemple, l'exécution n'est jamais interrompue pendant toute la durée du traitement de la requête ; en conséquence, la boucle de `react()` n'itère qu'une seule fois.

4.2.2 Deuxième exemple : cas d'une requête d'origine matérielle

Considérons maintenant la figure 5.3-(b), et supposons également qu'aucune autre interruption matérielle que celle traitée ici n'advient. Après l'émission d'une requête d'interruption matérielle, un traitement insère une requête irq_a dans la file d'attente appropriée et appelle ensuite `on_it()`. Ceci peut arriver n'importe quand, et en particulier lorsque le calculateur est en train d'exécuter des instructions du logiciel, y compris un appel à `react()`. Dans ce dernier cas, `on_it()` laisse la requête dans la file d'attente et se termine immédiatement. Sinon, les instructions qui s'exécutaient sont préemptées, et `on_it()` appelle `react()` pour traiter la requête. `react()` récupère cette dernière depuis la file d'attente et, comme dans l'exemple précédent, exécute `tick.run_step(input_event)` après avoir construit `input_event` (dans ce cas précis, ce dernier est de la forme $irq_a \cdot \bar{x} \cdot \bar{y} \dots$ où x, y, \dots représentent tous les signaux d'entrée du tick, moins irq_a). La suite de l'exécution est similaire au cas précédent.

```

1  /* usual software entry point, always executed with interrupts disabled.      */
2  main ()
3      tick.init ();                               /* initialize the reactive part.    */
4      while (true) {
5
6          enable_interrupts ();                   /* enable interrupts before entering guest. */
7          run_guest (); /* run guest until it requests entering low-power mode (LPM)
8                          by returning from this function.                        */
9          disable_interrupts (); /* disable interrupts, because 'enter_lpm()' may
10                                 need to perform some computation.                */
11
12         /* here, we are guaranteed that no emission of software request is
13            possible until the next occurrence of an interrupt request.          */
14
15         enter_lpm (); /* enter LPM, and enable interrupts so that the MCU can be
16                        awakened: this function will return upon the next hardware
17                        event occurrence (cf. function 'on_it()' of Listing 5.4);
18                        also disables interrupts on return.                      */
19     }

```

LISTING 5.2 – Une procédure `main()`, symbolisant les instructions exécutées au (re-)démarrage du microcontrôleur. Les détails destinés à empêcher l'appel de `enter_lpm()` dans le cas où une requête d'interruption advient pendant l'exécution des instruction entre les lignes 7 et 9 ont été omis par soucis de clarté.

5 Détails de la membrane

Décrivons maintenant la partie gauche de la figure 5.2 page 67, c'est-à-dire les portions de code dont le rôle est d'entrelacer efficacement les exécutions du noyau réactif et de l'invité. Elle récupère toutes les requêtes en entrée, les traduit en événements pour le tick et transmet ses sorties à l'invité, soit directement au moyen d'interruptions virtuelles, soit comme résultat d'émission d'une requête logique. Les événements qui déclenchent une exécution du noyau réactif sont donc des interruptions matérielles ou des requêtes logicielles.

5.1 Gestion globale du microcontrôleur

Le listing 5.2 présente la procédure `main()`, exécutée au démarrage du microcontrôleur (c'est-à-dire, lorsque l'interruption « *system reset* » survient – cf. § 2.2.1 page 12)⁷.

5.1.1 Exécution de l'invité

Le rôle de la procédure `main()`, hormis l'initialisation du calculateur et des divers modules du microcontrôleur, est de s'assurer que ce dernier est dans un état de basse consommation le plus souvent possible. Dans cette optique, elle s'articule autour d'une boucle infinie activant en permanence l'exécution de l'invité au moyen de la routine `run_guest()` implantée dans la couche d'adaptation ; cette dernière n'est en général qu'un appel vers le gestionnaire des tâches de l'invité, qui s'exécute jusqu'à ce que toutes soient bloquées (cf. § 3 page 65). Si `run_guest()` se termine, alors aucune tâche n'est plus éligible et aucun calcul ne reste à faire par l'invité dans l'immédiat. Depuis le démasquage des interruptions (ligne 6 du listing 5.2) jusqu'à leur désactivation après le retour de `run_guest()`, `on_sw()` et `on_it()` sont possiblement exécutées.

7. Il est à noter que cette manière de gérer l'exécution de l'invité est aisément adaptable à une exécution multifiels et à l'usage d'une procédure dédiée à la mise en mode de basse consommation à l'initiative de l'invité : elle est présentée ici sous cette forme à des fins d'exemple simple, sans découpage du `main()`.

```

1 example_timer_irq_handler () /* note interrupts are automatically disabled by
2                               hardware when this handler is executed.      */
3   acknowledge_irq ();          /* clear pending IRQ flag.                  */
4   hw_queue.push (irq_expired); /* push hardware event corresponding to the
5                               current interrupt request.                  */
6   on_it ();                    /* next, call 'on_it()'.            */

```

LISTING 5.3 – Exemple de traitant d'interruption pour la gestion de l'expiration d'un timer. `hw_queue` est la file d'attente des requêtes matérielles, consommées par la procédure `react()` (elle-même possiblement appelée par `on_it()` – cf. listing 5.4 page suivante). Les détails requis pour autoriser l'imbrication des traitements d'interruption ont été omis, cependant un tel comportement est envisageable avec l'architecture de la couche de contrôle exposée dans ce chapitre.

5.1.2 Gestion du mode de basse consommation

Lorsque l'invité n'effectue plus aucun calcul, la procédure `main()` utilise `enter_lpm()` (ligne 15 du listing 5.2) pour placer le microcontrôleur en mode de basse consommation. Sur occurrence de requêtes d'interruption matérielles (une seule ou plusieurs survenant simultanément), le calculateur retourne en mode de calcul et commence à exécuter un traitant d'interruption.

Exemple de traitant de requête d'interruption matérielle. Le listing 5.3 exemplifie un tel traitant de requête d'interruption matérielle pour la couche de contrôle. L'un de ses rôles est d'insérer un nouvel élément dans la file d'attente des requêtes matérielles, puis d'appeler `on_it()` (lignes 4 et 6). Si cette dernière procédure déclenche l'exécution d'une réaction (ou plusieurs dans le cas où de nouvelles requêtes matérielles ont été reçues et prises en compte pendant l'exécution du tick ou de traitants d'interruptions virtuelles), alors elle configure également le calculateur pour qu'il reste actif après que le traitant d'interruption matérielle courant se termine. Dans le cas contraire, le microcontrôleur retrouve automatiquement son mode d'opération précédent l'interruption matérielle.

Exécution de l'invité après un « réveil » du microcontrôleur. Reconsidérons maintenant le code de la procédure `main()` exposée dans le listing 5.2 page précédente. `enter_lpm()` se termine si le calculateur est configuré par `on_it()` pour rester actif après que tous les traitements de requêtes d'interruption matérielles se finissent. Finalement, des tâches de l'invité, si elles sont devenues éligibles lors de l'exécution éventuelle de traitants d'interruptions virtuelles après la ou les réactions déclenchées par `on_it()`, peuvent être de nouveau élues et exécutées pendant l'appel à `run_guest()` qui suit. Sinon, `run_guest()` se termine immédiatement et le calculateur est de nouveau placé en mode de basse consommation par `enter_lpm()`.

Après avoir détaillé le mécanisme de gestion de l'état du microcontrôleur, nous pouvons maintenant aborder les portions spécifiques aux traitements des requêtes matérielles et logicielles.

5.2 Traitement des requêtes matérielles

Comme vu précédemment, les traitants d'interruptions matérielles doivent appeler `on_it()` après avoir inséré une requête matérielle dans la file d'attente correspondante. Cette routine, décrite dans le listing 5.4 page suivante, appelle la procédure `react()` si une exécution de celle-ci n'était pas déjà en cours lorsque l'interruption survient (avec l'aide du drapeau `already_in_reaction`, dont on notera par ailleurs qu'il est toujours accédé lorsque les interruptions sont masquées). Si elle appelle `react()`, `on_it()` configure le microcontrôleur pour qu'il reste actif après que le traitant d'interruption courant se termine, ce afin que toute tâche nouvellement éligible au niveau de l'invité puisse être exécutée (cf. § 5.1 page précédente).

```

1 on_it () /* note that the event has already been pushed in 'hw_queue', and
2           interrupts are disabled. */
3 if (! already_in_reaction) {
4     react (); /* trigger reaction. */
5     stay_aware (); /* ensure the MCU will stay in active mode upon return from
6                   interrupt, if it was in low-power mode hitherto. */
7 }

```

LISTING 5.4 – La routine `on_it()`, appelée dès l'insertion d'une requête matérielle dans la file d'attente `hw_queue`.

```

1 on_sw (software_input)
2
3 sw_req.create (software_input); /* create a new software request. */
4
5 disable_interrupts (); /* disable interrupts to protect request management. */
6 sw_cell.set (sw_req); /* assign the software request cell. */
7 react (); /* trigger reaction. */
8 enable_interrupts (); /* re-enable interrupts. */
9
10 return sw_req.result (); /* return the result that has been recorded in the
11                          software request object by 'react()'. */

```

LISTING 5.5 – La fonction `on_sw()`. L'objet `sw_req` mémorise à la fois le signal d'entrée donné en paramètre et les comptes-rendus résultant de l'exécution du tick correspondante. Cette dernière donnée est affectée dans `react()` (cf. listing 5.6 page suivante). `sw_cell` est une référence désignant une requête logicielle non encore traitée par `react()`, ou `nil` sinon.

5.3 Traitement des requêtes logicielles

Le rôle de la fonction `on_sw()`, décrite dans le listing 5.5, est de déclencher les réactions nécessaires au traitement des requêtes logicielles dès leur émission pas la couche d'adaptation. Elle se termine en retournant l'ensemble des comptes-rendus associés à la requête par la procédure `react()`.

5.4 Exécution du noyau réactif

La routine `react()`, décrite listing 5.6 page suivante, a pour objectif la consommation des requêtes logicielles et matérielles.

Lorsqu'elle est appelée (soit par `on_it()`, soit par `on_sw()`), le drapeau `already_in_reaction` est levé afin d'éviter que de nouvelles requêtes d'interruption matérielles ne déclenchent une réaction concurrente (cf. `on_it()`, listing 5.4). Cette dernière situation est possible si une requête matérielle est émise par un traitant d'interruption pendant l'exécution de `tick.run_step()` (instructions entre les lignes 12 et 14 du listing de `react()`). Dans un tel cas, cette requête matérielle est prise en compte (*i.e.*, retirée de la file `hw_queue` et utilisée pour exécuter le noyau réactif) quand l'appel de `tick.run_step()` courant se termine. Le drapeau `already_in_reaction` est abaissé si la file d'attente des requêtes matérielles est toujours vide à la fin d'une réaction (ligne 25 du listing).

La boucle des lignes 6 à 22 extrait et traite toute requête matérielle en attente dans `hw_queue` tant que cette file n'est pas vide. Elle récupère également toute requête logicielle désignée par `sw_cell`, s'il en existe une. À chaque itération, toutes les requêtes sont récupérées depuis ces deux structures, et un événement d'entrée est construit puis utilisé pour exécuter le noyau réactif (ligne 13).

```

1 react ()
2
3 already_in_reaction = true;    /* start the reaction (needed in 'on_it()'). */
4 all_outputs = empty_set ();
5
6 do {
7
8     /* build an input event by extracting the software request (if any) and
9        all the hardware events from the queue: */
10    input_event = build_input_event (hw_queue, sw_cell);
11
12    enable_interrupts ();        /* enabling interrupts allows new */
13    outputs = tick.run_step (input_event); /* hardware events to be pushed */
14    disable_interrupts ();      /* into 'hw_queue' during the tick. */
15
16    all_outputs.merge (outputs); /* union (bit-array operation actually). */
17
18    /* if there was a software request, then setup its result: */
19    if (sw_cell != nil) sw_cell.set_result (outputs);
20    sw_cell = nil;             /* there is no more pending software request. */
21
22 } while (! hw_queue.is_empty ());
23
24 /* here, hardware queue is empty, and no software request is pending. */
25 already_in_reaction = false; /* notify we leave the reaction. */
26
27 /* eventually, execute VIRQ handlers associated with all emitted outputs: */
28 execute_virq_handlers (all_outputs);

```

LISTING 5.6 – La routine `react()`, toujours appelée avec les interruptions désactivées.

Le résultat de cet appel du tick (*i.e.*, l'ensemble des comptes-rendus émis) est alors enregistré comme résultat de l'éventuelle requête logicielle prise en compte (ligne 19) ; il sera ensuite utilisé comme résultat de la routine `on_sw()`.

Finalement, ce même résultat est fusionné (dans `all_outputs`, initialisé à l'ensemble vide au début du corps de `react()`), par union d'ensemble classique avec les sorties déjà regroupées pendant les itérations précédentes de la boucle. Cet ensemble sert à la ligne 28, où les requêtes d'interruption virtuelles éventuellement associées aux sorties émises pendant la réaction sont levées et possiblement exécutées. Notons enfin que de nouvelles exécutions de `react()` sont possibles durant cette dernière étape à la fois :

- sur requête matérielle puisque les traitants d'interruptions virtuelles peuvent être interrompus ainsi (voir § 5.5) ;
- sur requête logicielle qui peut être émise directement par ces mêmes traitants.

5.5 Exécution des traitants d'interruptions virtuelles

Les requêtes d'interruption virtuelles sont émises à la fin de l'exécution de la procédure `react()`. Le traitant unique exécuté lors de l'émission d'un compte-rendu donné par le tick peut être changé ou désabonné dynamiquement par l'invité. Ces dernières peuvent (facultativement, au prix d'un surcoût à l'implantation) refléter les propriétés habituelles des interruptions matérielles classiques : elles peuvent être masquées, scrutées, imbriquées et statiquement associées à des priorités ; elles peuvent également requérir un acquittement de la part de leur traitant. Le choix de satisfaction de ces différentes propriétés par une implantation de couche d'adaptation ne dépend pas de l'architecture de la couche de contrôle, mais plutôt des propriétés du matériel rendues nécessaires par le modèle d'exécution du logiciel invité.

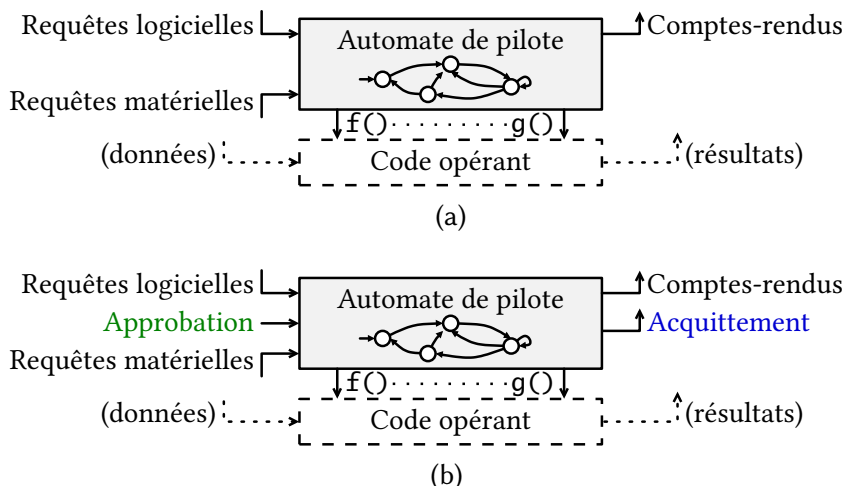


FIGURE 5.4 – (a) Représentation schématique de la structure d'un pilote de périphérique simple. Certaines routines du code opérant acceptent potentiellement des données de travail à fournir par la couche d'adaptation, et produit des résultats à récupérer par cette dernière ;
 (b) Le même pilote en version contrôlable : au moins une *entrée d'approbation*, ainsi qu'au moins une *sortie d'acquittement*, ont été ajoutées.

On notera enfin à propos des traitants d'interruptions virtuelles qu'ils s'exécutent alors que les interruptions matérielles sont autorisées : ils sont donc toujours préemptibles par les interruptions du calculateur.

6 Détails du noyau réactif

Dans cette section, nous détaillerons la structure et les éléments à la base de la construction du tick, c'est-à-dire la partie droite de la figure 5.2 page 67.

6.1 Automates de pilotes

Concevoir un modèle de pilote de périphérique comme un automate explicite est relativement aisé (cf. § 5.1 page 22). Les états de cet automate reflètent les modes d'opération du périphérique, ne serait-ce que deux états marche et arrêt. Les transitions de cet automate sont déclenchées par des entrées qui peuvent être séparées en deux catégories :

- les requêtes logicielles de changement de mode ou d'opérations simples (qui seront alors des transitions d'un état vers lui-même) ;
- et les notifications en provenance du périphérique.

Le point principal de la construction de l'automate est alors de garantir que l'état reflète fidèlement le mode de fonctionnement du périphérique.

Transformer ce modèle en un automate de pilote revient à ajouter des sorties sur les transitions. Ces sorties représentent alors les portions de *code opérant* à exécuter pour entraîner le changement effectif de mode d'opération du périphérique. La figure 5.4-(a) est une représentation schématique d'un pilote complet, comprenant également une portion de code opérant.

Notion de « contrôlabilité » des pilotes. Une notion centrale de la conception des pilotes que nous proposons est la *contrôlabilité*. En effet, afin de pouvoir assurer un contrôle global sur les composants de la plate-forme, les requêtes logicielles (plus éventuellement certaines requêtes matérielles) ne doivent pas être systématiquement mises en œuvre par les pilotes de périphériques. En termes de synthèse de

contrôleur (cf. § 2 page 56), cela signifie que les automates à contrôler doivent recevoir des *entrées de contrôle* supplémentaires ; sans celles-ci, les objectifs de contrôle peuvent ne pas être réalisables. Ces entrées supplémentaires seront appelées des signaux d'*approbation*.

Afin de notifier l'émetteur d'une requête logique (la couche d'adaptation de l'invité en l'occurrence), des sorties additionnelles d'*acquiescement* sont aussi utilisées (en principe une par pilote, mais plusieurs sont envisageables puisqu'elles ne sont interprétées que par l'invité). Elles sont émises lorsque des transitions contrôlables sont prises.

La figure 5.4-(b) illustre les ajouts de signaux nécessaires pour rendre contrôlable le pilote de la figure 5.4-(a).

Complément sur les notations. En nous inspirant de la syntaxe graphique usuelle des machines de Mealy Booléennes (cf. § 1.3 page 54), nous utiliserons la notation suivante pour les étiquettes des transitions des automates : « $i_1 . \bar{i}_2 / o_1, o_2, f()$ » où $i_1 . \bar{i}_2$ est un exemple de formule booléenne construite à partir de l'ensemble des signaux d'entrée, o_1 et o_2 sont des sorties, et $f()$ dénote un appel vers une portion de code opérant exécutée lorsque la transition est prise. Dans un souci de lisibilité des figures, les boucles d'un état vers lui-même seront omises si elles ne comportent pas de sorties ni d'appel de code opérant.

Dans la suite de cette section, nous ne considérerons plus que des automates de pilote contrôlables ; pour l'un des plus simples, nous expliquerons comment ajouter ces entrées de contrôle et acquiescements.

Exemple de construction d'un automate de pilote contrôlable. La figure 5.5-(a) est une représentation d'un pilote de timer.

Dans la figure 5.5-(b), le pilote de la figure 5.5-(a) a été rendu contrôlable en ajoutant l'entrée d'approbation ok_t : une *transition contrôlable* (c'est-à-dire dont la condition booléenne peut être écrite sous la forme d'une conjonction d'un signal d'approbation avec une formule quelconque sur les autres entrées de l'automate) n'est déclenchée que lorsque le signal représentant la requête d'entrée est « vrai » et le contrôleur émet ok_t . Dans cet exemple, ack_t est l'unique signal d'acquiescement additionnel.

On notera enfin que, quand l'automate du pilote est dans l'état Disabled, la condition $start . \overline{ok_t}$ est vraie (c'est-à-dire que le contrôleur refuse la requête logique *start*), il existe une boucle revenant vers Disabled qui n'émet aucun signal (et en particulier ack_t) ; cette boucle n'est pas représentée dans la figure 5.5-(b).

Sur le même modèle que précédemment, les figures 5.6 et 5.7 décrivent des automates de pilote contrôlables, l'un pour un convertisseur analogique-digital, l'autre pour un pilote de transmetteur radio (sans gestion d'erreur ni d'« écoute passive », toutes deux implantables similairement).

6.2 Automate de contrôle

La figure 5.8 représente un exemple de contrôleur conçu pour gérer les pilotes de convertisseur analogique-digital et transmetteur radio des figures 5.6 page 77 et 5.7 page 77. Il assure une propriété d'exclusivité entre trois états fortement consommateurs du transmetteur radio (Tx, Rx ainsi que RxPacket) et le mode de fonctionnement (On) du convertisseur. Par exemple, lorsque le contrôleur est dans l'état Radio, signifiant ici que ce périphérique est alors en train de consommer, le convertisseur est nécessairement dans son état Off et peut atteindre On (i.e., que ok_a vaut vrai) si et seulement si le transmetteur entre dans ses états Idle ou Sleep.

6.3 Construction du noyau réactif

Finalement, les automates des pilotes et le contrôleur sont compilés en un bloc de code formant le tick et se comportant comme le produit des automates. Pour le même exemple que précédemment, l'automate du produit est dans l'état $Free \times Off \times Idle$ quand le contrôleur, le convertisseur et le transmetteur radio sont respectivement dans leurs états Free, Off et Idle.

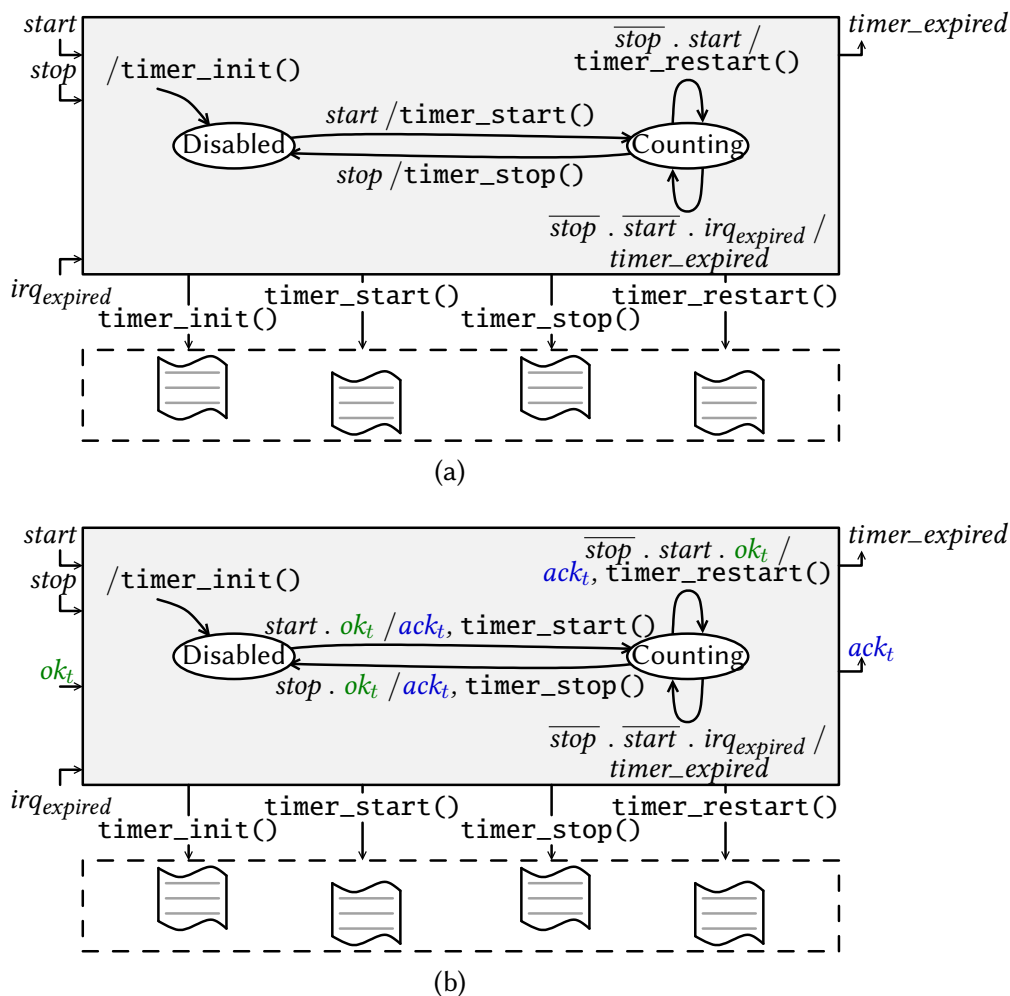


FIGURE 5.5 – (a) Un pilote de timer original. $\text{timer_init}()$, $\text{timer_start}()$, $\text{timer_restart}()$ et $\text{timer_stop}()$ dénotent des portions de code opérant, chacune encapsulant les accès aux registres d'entrée-sortie de ce périphérique nécessaires à la réalisation effective de l'opération décrite. Le signal d'entrée irq_expired symbolise l'arrivée d'une requête matérielle, dont le sens est l'expiration du timer (i.e., un compteur matériel a atteint une certaine valeur). Finalement, start et stop sont des signaux d'entrée qui représentent des requêtes logicielles issues de l'invité pour agir sur ce périphérique ; timer_expired est un compte-rendu reflétant l'expiration du timer.

(b) Pilote de timer en version contrôlable. L'entrée d'approbation ok_t et la sortie d'acquiescement ack_t sont les seuls signaux additionnels.

7 Un scénario d'exécution simple

Nous illustrons dans la figure 5.9 page 78 un entrelacement possible des exécutions entre la couche de contrôle et des tâches de l'invité pour une application typique ; le noyau réactif est ici supposé ne comprendre que les automates de pilote des figures 5.5-(b), 5.6 et 5.7, ainsi que du contrôleur de la figure 5.8 page suivante. Nous y représentons une trace des états du contrôleur, du convertisseur et du transmetteur radio entre chaque réaction. Nous décrivons également le rôle de la couche d'adaptation tout au long du processus, en inscrivant les requêtes logicielles qu'elle émet à la couche de contrôle. L'invité comprend deux tâches, chacune déclenchée par deux timers distincts :

- T_{sensing} gère un capteur au travers du convertisseur analogique-digital ;

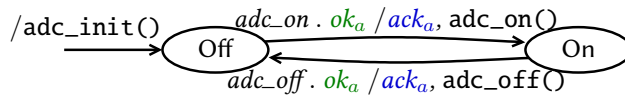
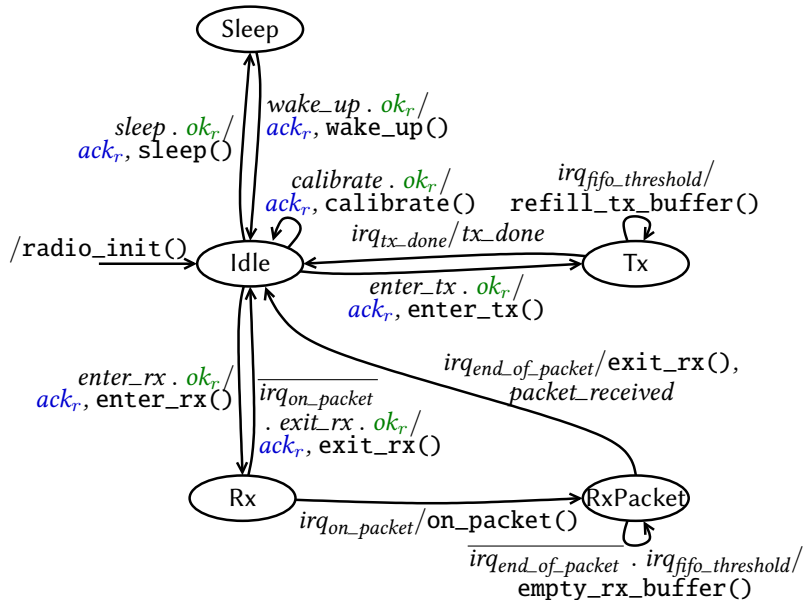
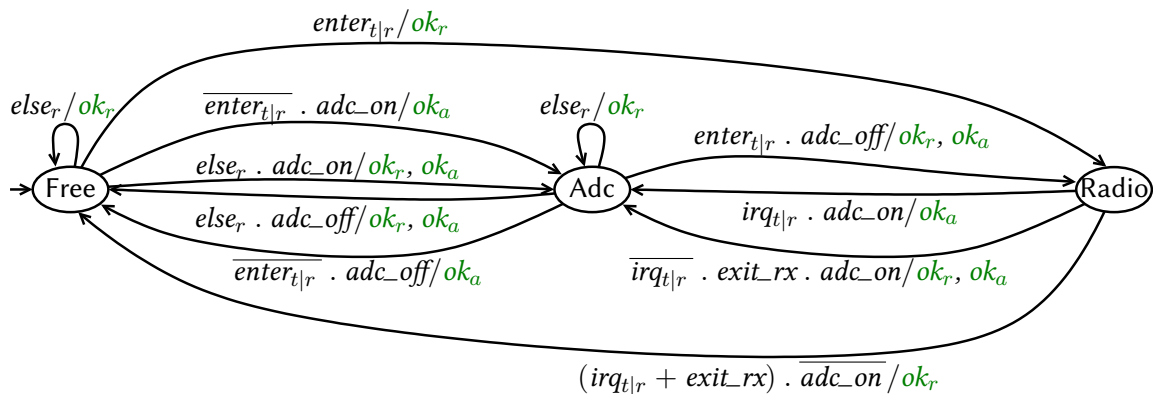


FIGURE 5.6 – Automate de pilote contrôlable pour un convertisseur analogique-digital (ADC).

FIGURE 5.7 – Automate de pilote contrôlable pour un émetteur-récepteur radio (de type CC1100 [40]). ok_r et ack_r sont respectivement des signaux d'approbation et d'acquiescement. Ce pilote est dans l'état Tx lorsque le périphérique transmet un paquet et dans l'état RxPacket lorsqu'il en reçoit un ; il est dans Rx s'il écoute le canal radio, mais qu'aucun paquet n'est détecté.

$$\left(\begin{array}{ll} enter_{t|r} = enter_tx + enter_rx & \text{– sens : tentative de placer la radio dans Tx ou Rx} \\ irq_{t|r} = irq_{end_of_packet} + irq_{tx_done} & \text{– sens : la radio entre dans l'état Idle} \\ else_r = \overline{enter_{t|r}} . (calibrate + wake_up + sleep) & \text{– sens : la radio reste dans Idle ou Sleep} \end{array} \right)$$

FIGURE 5.8 – Automate de contrôle simple pour les pilotes des figures 5.6 et 5.7.

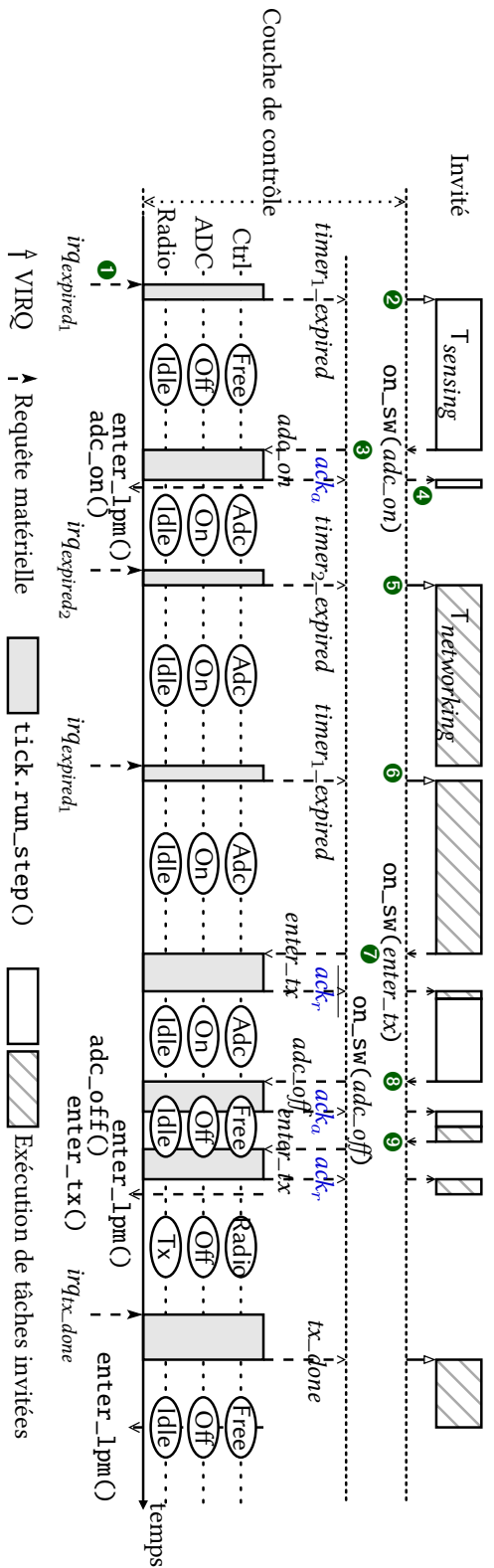


FIGURE 5.9 – Un exemple d'exécution de la pile logicielle complète. Les entrées et sorties du noyau réactif ne sont représentées que lorsqu'elles sont significatives. Sur le modèle des chronogrammes, le temps s'écoule de la gauche vers la droite.

Dans un premier temps, un traitant d'interruption insère la requête matérielle $irq_{expired_1}$, dans la file d'attente, puis appelle $on_it()$ ①. Une exécution du noyau réactif prenant en compte la requête insérée auparavant est alors déclenchée. Le compte-rendu $timer_expired$ appartient au résultat du tick (ce signal est émis par l'automate du pilote d'un timer), et la procédure $react()$ lève l'interruption virtuelle associée : le traitant qui y est abonné s'exécute, débloque la tâche $T_sensing$, puis se termine. Après que la routine $1pm_enter()$ appelée par $main()$ se termine, l'invité peut continuer à exécuter cette dernière tâche ②.

Dès l'émission d'une requête logicielle par $T_sensing$ (par un appel $on_sw(adc_on)$), une nouvelle réaction est provoquée ③. Puisqu'alors l'automate du tick est dans l'état $Free \times Off \times Idle$, cette nouvelle entrée déclenche la transition déjà prise comme exemple dans la section 6.3 : la requête est approuvée et le code opérant désigné par $adc_on()$ est exécuté pour mettre le convertisseur en état de marche. L'appel $on_sw(adc_on)$ a donc pour résultat ack_a , et la tâche arrêtée plus tard sont exécutées : la procédure $main()$ rappelle alors $enter_1pm()$ ④.

De manière similaire au cas précédent, $irq_{expired_2}$ déclenche une réaction, et la tâche $T_networking$ devient éligible et s'exécute ⑤. Pendant ces calculs, elle est préemptée par une nouvelle interruption émettant la requête matérielle $irq_{expired_1}$, une réaction, et un appel du traitant de l'interruption virtuelle associée au compte-rendu $timer_expired$ ⑥.

À son tour, ce traitant libère la tâche de gestion du capteur $T_sensing$. Cependant elle ne s'exécute pas immédiatement : lorsque le traitant de l'interruption matérielle du timer se termine, la tâche gérant le réseau continue à s'exécuter. Cette dernière essaie alors de placer le transmetteur dans un mode d'émission au moyen d'un appel $on_sw(enter_tx)$ ⑦. En revanche, puisque le contrôleur est dans l'état Adc , il n'autorise pas cette requête (sa sortie est ack_r) : la tâche réseau peut alors choisir de rendre la main et réessayer plus tard.

Par la suite, la tâche de gestion du capteur s'exécute et arrête le convertisseur ($on_sw(adc_off)$) ⑧ ; la réaction associée exécutée $adc_off()$, et met à jour l'état du contrôleur. Enfin, la prochaine émission de $enter_tx$ est acceptée par le tick, et la transmission peut s'effectuer ⑨.

- $T_{networking}$ gère le réseau sans fil.

En outre, on considère que le calculateur est initialement dans l'un de ses modes de basse consommation.

Remarque sur le réalisme de l'exemple. Nous ferons remarquer que cet exemple est principalement destiné à illustrer les interactions entre la couche de contrôle et l'invité, et le comportement de ce dernier n'est pas nécessairement souhaitable dans le cadre d'un déploiement réaliste. En effet, dans une telle situation, l'extinction du convertisseur devrait être plus prioritaire que la poursuite de la gestion du réseau lors de la seconde occurrence de $irq_{expired_1}$, et ce pour réduire l'ensemble des délais de latence :

- Si les tâches sont ordonnancées à l'aide de priorités, alors à cet instant $T_{sensing}$ devrait être plus prioritaire que $T_{networking}$;
- Sinon, la requête d'extinction du convertisseur analogique-digital devrait être envoyée directement lors du traitement de l'interruption virtuelle associée à $timer_1_expired$.

8 Conclusion du chapitre

Nous avons détaillé dans ce chapitre l'architecture logicielle que nous proposons pour permettre un contrôle global d'une plate-forme matérielle. Nous pouvons maintenant évaluer cette solution à partir de l'implantation d'un prototype de couche de contrôle ; nous aborderons également les principes que nous avons identifiés pour l'adaptation d'un système existant, ainsi que quelques usages et extensions possibles non évoqués jusqu'ici. Dans le chapitre suivant, nous expliquerons les choix d'outils que nous avons effectués dans ce but. L'implantation et les évaluations seront présentées au chapitre 7.

OUTILLAGE ET CHOIX

Contenu du chapitre

| | | |
|-------|---|----|
| 1 | Introduction du chapitre | 81 |
| 2 | Choix d'une plate-forme cible | 81 |
| 3 | Implantation de machines de Mealy communicantes | 82 |
| 3.1 | Synchronous C | 82 |
| 3.1.1 | Objectifs et principes | 82 |
| 3.1.2 | Quelques propriétés du langage Synchronous C | 85 |
| 3.1.3 | Contraintes et limitations | 85 |
| 3.1.4 | Évolutions récentes | 85 |
| 3.2 | ARGOS | 86 |
| 3.2.1 | Objectifs et principes | 86 |
| 3.2.2 | Le compilateur ARGOS existant | 86 |
| 3.3 | Remarques à propos de la causalité | 88 |
| 3.3.1 | Exemple de dialogue instantané : le rendez-vous | 88 |
| 3.3.2 | Implantation séquentielle du rendez-vous | 88 |
| 3.3.3 | Traitement des cycles de causalité | 89 |
| 4 | Conclusion du chapitre | 90 |

1 Introduction du chapitre

Nous effectuerons au chapitre 7 une évaluation de l'approche d'implantation proposée précédemment. Cependant, la réalisation effective d'un prototype de couche de contrôle nécessite le choix d'une plate-forme d'implantation, ainsi que l'emploi d'outils adéquats pour le codage des pilotes afin de pouvoir créer un noyau réactif à partir du contrôleur et des automates représentant le comportement des ressources à gérer.

Dans le présent chapitre, nous évaluerons les quelques outils disponibles pour réaliser cette tâche, puis nous présenterons succinctement la chaîne d'outils que nous avons choisie pour procéder à l'évaluation.

2 Choix d'une plate-forme cible

Nous avons choisi la plate-forme cible Wsn430 (présentée en exemple dans la section 2.2.2 page 14) pour réaliser cette expérimentation pour deux raisons principales. Premièrement, la suite d'outils SensTools [92] autorise un développement entièrement hors-ligne du logiciel à déployer sur les nœuds.

Dans cette suite d'outils, Fraboulet *et al.* [36, 69] proposent des outils pour l'émulation de réseaux complets, qui permettent donc un développement et une validation partielle des protocoles de gestion du réseau à implanter dans un système invité.

Émulation d'une plate-forme. Parmi ces outils, WSim simule un nœud de réseaux de capteurs sans fil isolément, en émulant précisément le CPU intégré au microcontrôleur. WSim permet le test et le débogage des programmes déployés qui s'y exécutent comme sur la plate-forme réelle. Les sources d'imprécision de cet outil proviennent de la simulation des différents périphériques, notamment en ce qui concerne leur comportement temporel et les erreurs matérielles éventuelles.

Simulation d'un réseau de capteurs sans fil. L'outil WSNet, permet la connexion de plusieurs instances du simulateur WSim, dans le but de simuler un réseau complet en spécifiant les caractéristiques du canal de transmission radio. Cette ressource est intéressante pour l'implantation des protocoles de gestion du réseau.

Enfin, une expertise à propos de l'usage de cette architecture de nœud de réseaux de capteurs sans fil était disponible au laboratoire puisque plusieurs de ses membres étaient impliqués dans des projets en collaboration avec le laboratoire où la plate-forme Wsn430 a été conçue — il s'agit du projet ARESA [51] et de son successeur ARESA 2¹.

3 Implantation de machines de Mealy communicantes

Dans cette section, nous argumenterons nos choix concernant un langage d'implantation des automates du tick dans le prototype de couche de contrôle.

Caractéristiques requises. Outre l'expression de comportements à l'aide de machines de Mealy Booléennes communicantes à composition parallèle synchrone (*cf.* § 1.3 page 54), les critères suivants figurent parmi les caractéristiques déterminantes pour le choix d'une méthode d'implantation :

- L'interaction avec du code écrit en C : il doit être possible de spécifier l'exécution de code opérant lors du déclenchement de transitions ;
- Le code résultant doit être compact et efficace pour être compatible avec les contraintes mémoires imposées par le contexte des réseaux de capteurs sans fil.

À notre connaissance, il existe très peu d'outils satisfaisant ces propriétés ensemble. Nous détaillerons ici les langages Synchronous C et ARGOS, et quelques-unes de leurs caractéristiques respectives.

La première section ne décrit pas une contribution de cette thèse, mais présente un langage qui a été originellement utilisé pour l'évaluation de la couche de contrôle.

3.1 Synchronous C

3.1.1 Objectifs et principes

En 2009, von Hanxleden [154] proposa « SyncCharts in C » (plus tard renommé en « Synchronous C »)², une extension du langage C basée sur un usage astucieux du préprocesseur. Elle permet une expression relativement aisée de *flots de contrôle réactifs* concurrents (*cf.* § 1 page 50), en fournissant des mécanismes pour la spécification de comportements et de communications synchrones associées.

Synchronous C est conçu pour offrir une représentation exécutable et lisible des SYNCCHARTS proposés par André [5], à leur tour initialement destinés à exprimer sous forme d'une composition d'automates,

1. Des informations à propos de ce projet sont disponibles à l'adresse : <http://aresa2.minalogic.net/>.

2. Une distribution de Synchronous C est disponible à l'adresse <http://www.informatik.uni-kiel.de/rtsys/sc/>.

des programmes ayant la sémantique du langage impératif synchrone ESTEREL [14]. De manière similaire aux proto-threads (cf. § 3.2 page 37), Synchronous C exploite des constructions très concrètes du langage C (e.g., « goto » et étiquettes — peu recommandées lors d'un développement manuel) pour implanter un parallélisme de coroutines [102].

Le principe de programmation d'un système réactif en Synchronous C est de construire une procédure (nommée `tick()` dans la suite de cette section), dont chaque appel exécute une réaction du système complet. Cette routine se termine à la fin de chaque instant.

Description informelle d'un fragment des SYNCCHARTS. En SYNCCHARTS, chaque *tâche*, ou comportement, est un seul automate ou une composition parallèle d'automates. (Ceux-ci sont hiérarchiques, au sens où chaque état peut être raffiné et définir un comportement par une tâche.) Chaque automate peut en outre posséder un état terminal, et les comportements composés possèdent éventuellement des signaux locaux propres, c'est-à-dire qui ne sont émis et reçus que par leurs constituants. Les transitions des automates possèdent plusieurs propriétés :

- Elles sont déclenchées sur réception d'un événement (on dit aussi que le signal est « présent » dans ce cas, tout signal non émis dans l'instant ou en entrée étant « absent » par défaut) ;
- Elles peuvent émettre un ensemble de signaux, qui sont alors transmis à tout comportement en attente de l'un d'eux selon un modèle de diffusion (i.e., tout automate dans un état dont une transition est déclenchée par la réception de ce signal change d'état dans l'instant selon cette transition) ;
- Des priorités additionnelles, parfois encodées dans les conditions des transitions, peuvent être nécessaires pour définir quelle transition est à suivre parmi plusieurs dont le signal déclencheur est présent simultanément ;
- Une propriété relative à la préemption faible ou forte de leur état source (non détaillée dans cette thèse puisque non requise pour la lecture de la suite — nous invitons le/la lect-eur/rice intéressé-e à se reporter à la littérature relative aux langages réactifs pour plus de détails).

Les caractéristiques des SYNCCHARTS en font un bon candidat pour l'expression de systèmes réactifs exprimés à partir de machines de Mealy communicantes, et Synchronous C en est par conséquent un langage d'implantation envisageable.

Aspects sémantiques pour l'implantation. Pour l'évaluation d'une réaction, la sémantique des SYNCCHARTS suppose l'existence d'un moteur d'exécution ordonnant les opérations des différents comportements isolément en fonction des présences et émissions des signaux et des états des automates, et ce jusqu'à ce qu'aucune transition ne soit plus déclenchable. La traduction en ESTEREL se base également sur le moteur d'exécution de ce langage.

Synchronous C s'affranchit d'un tel mécanisme qui peut s'avérer coûteux à l'implantation, et propose d'assigner manuellement des priorités aux différentes tâches pour déterminer l'ordre d'évaluation des transitions. Ces priorités sont modifiables dynamiquement à l'aide d'instructions du langage, y compris lors de l'évaluation des conditions de déclenchement des transitions d'automates.

Exemple de programme. Le listing 6.1 page suivante présente la syntaxe d'un fragment de Synchronous C en exemplifiant le codage de la mise en parallèle des deux machines de Mealy Sa et Sb introduites dans la figure 4.1 page 54³. Après un premier appel d'initialisation, chaque exécution de la routine `tick()` exécute une transition de l'automate SE de la même figure.

3. On notera qu'une telle traduction est moins directe dans le cas où les conditions de déclenchement des transitions sont des formules Booléennes structurées, mais reste possible toutefois.

```

1  typedef enum {a, b, c} sig_t;                /* déclaration des signaux. */
2
3  typedef enum {                               /* déclaration des identifiants des tâches. */
4    TickEnd,                                  /* (= 0) inutilisé.          */
5    MainTask,                                 /* tâche parente.          */
6    SbTask, SaTask,                           /* tâches des automates.   */
7  } id_t;
8
9  extern int tick (void) { /* routine représentant le système réactif complet. */
10     TICKSTART (MainTask); /* spécification de la tâche principale (parente). */
11
12     FORK (Sa, SaTask); /* déclaration du comportement de la tâche fille SaTask. */
13     FORK (Sb, SbTask); /* idem SbTask. */
14     FORKE (Parent);
15
16     Sa:                                     /* machine Sa. */
17     A0:                                     /* état initial. */
18     AWAIT (a); /* attend la présence du signal a à partir du prochain insant. */
19     TRANS (A1); /* transite immédiatement dans l'état A1. */
20     A1:
21     AWAIT (a); /* attend a. */
22     EMIT (b); /* émet b dans l'instant. */
23     TRANS (A0); /* transite immédiatement dans l'état A0. */
24
25     Sb:                                     /* machine Sb (similaire à Sa). */
26     B0:
27     AWAIT (b);
28     TRANS (B1);
29     B1:
30     AWAIT (b);
31     EMIT (c);
32     TRANS (B0);
33
34     Parent:
35     JOIN; /* jonction: ne termine jamais si l'une des tâches SaTask ou SbTask */
36     TERM; /* ne se termine pas. */
37
38     TICKEND; /* fin du tick. */
39 }

```

LISTING 6.1 – Un codage possible en Synchronous C de la composition parallèle des deux machines de Mealy Sa et Sb de la figure 4.1 page 54. Le signal d'entrée a est déterminé comme présent pendant un instant (l'instruction **AWAIT** (a) se termine) s'il a été émis explicitement (par l'instruction **EMIT** (a)) depuis le retour du précédent appel de tick(). On remarquera que SbTask < SaTask, c'est-à-dire que les priorités intrinsèques de ces deux tâches font que, à chaque réaction, la machine Sa teste la présence de a et émet éventuellement b avant que Sb ne teste la présence de ce dernier signal.

3.1.2 Quelques propriétés du langage Synchronous C

Puisque exclusivement construit sous la forme de *macros* C, cet outil est portable sur toute machine pour laquelle il existe un compilateur pour ce langage, autrement dit presque toute machine de type Harvard ou de von Neumann. Comme il ne requiert pas d'autre outil spécifique ou bibliothèque de fonctions additionnelles (à l'exception notable d'une fonction de manipulation de champs de bits, par ailleurs disponible sous la forme d'une unique instruction sur certaines architectures matérielles), il est adapté au développement pour plates-formes fournissant peu de mémoire de travail. Ce langage reste simple d'utilisation tant que le nombre de comportements parallèles reste raisonnable, et que ceux-ci ne mettent pas en jeu de motifs de communication rendant les spécifications des priorités des tâches trop complexes (*cf.* section suivante). Par essence, il permet une interaction aisée avec du code C.

On notera de plus que la spécification de plusieurs sous-systèmes réactifs s'exécutant isolément et ne communiquant pas directement (c'est-à-dire que les communications entre ces sous-systèmes ne sont pas supposées instantanées et utilisent des tampons d'au moins une cellule) est autorisée par l'exploitation des propriétés des symboles du langage C.

3.1.3 Contraintes et limitations

En revanche, un certain nombre de contraintes rendent son usage peu adéquat lorsque le système à spécifier implique beaucoup de tâches ou des motifs de communication complexes.

Limitation liée au nombre de comportements et de signaux. En utilisant l'implantation disponible et pour une architecture donnée, le nombre maximum de comportements parallèles d'un système réactif et de signaux, qui peuvent être spécifiés dépend de la largeur du mot mémoire de cette plate-forme : il correspond directement au nombre de bits de ces mots.

Cependant, on peut mitiger ce problème puisqu'il est envisageable de modifier l'implantation originale de Synchronous C pour permettre la spécification d'un nombre supérieur de tâches distinctes. Un examen rapide permet d'affirmer que l'impact sur le coût calculatoire d'une telle modification serait assez limité ; une telle assertion reste néanmoins à vérifier.

Limitation de l'assignation manuelle des priorités aux tâches. Comme nous l'avons évoqué dans l'exemple du listing 6.1 page ci-contre, l'implantation de comportements impliquant des communications intertâches devient difficile à réaliser manuellement lorsque le nombre de comportements impliqués augmente, ou que les conditions de déclenchement des transitions se complexifient avec des tests d'absence de signaux.

3.1.4 Évolutions récentes

On notera à propos de Synchronous C l'intégration en 2010 d'une extension au sein du projet KEILER [71]⁴ qui permet la spécification graphique de systèmes réactifs sous forme de SYNCCHARTS. Cet outil de modélisation intègre également un générateur de code automatisé transformant des SYNCCHARTS vers un programme en Synchronous C. Une syntaxe textuelle est aussi proposée, assez inspirée du langage présenté dans la section suivante, et permettant ainsi une meilleure identification des erreurs de programmation par retour à la source.

Depuis 2011 et la publication par Traulsen *et al.* [152] d'un procédé de compilation permettant l'automatisation de l'assignation des priorités dynamiques des comportements, cette compilation se fait à partir de SYNCCHARTS sans avoir à définir les priorités des tâches manuellement.

4. Disponible à l'adresse <http://www.informatik.uni-kiel.de/rtsys/kieler/>.

3.2 ARGOS

Une seconde méthode d’implantation disponible consiste en l’utilisation du langage ARGOS [114] pour la spécification de programmes synchrones sous forme de compositions de machines de Mealy Booléennes. Le langage que nous considérons dans cette section est un fragment d’ARGOS : par souci de concision, nous ne détaillons pas certains opérateurs (*e.g.*, hiérarchie, inhibition) ou encore les programmes avec variables (*i.e.*, mémoire de type quelconque non encodée dans un ou plusieurs automates), dont nous ne ferons pas usage par la suite.

3.2.1 Objectifs et principes

De même que SYNCCHARTS, ARGOS permet la spécification explicite de compositions d’automates communiquants à l’aide des signaux Booléens. Pour la définition de la sémantique des constructions parallèles d’ARGOS, Maraninchi et Rémond [114] prennent une approche différente de celle des concepteurs des langages synchrones dans le style des SYNCCHARTS, en insistant sur la détection et le rejet des comportements invalides, c’est-à-dire non réactifs ; dans un cadre d’implantation, les programmes non déterministes sont également considérés invalides.

Maraninchi [112] fournit une définition de la sémantique des constructions d’ARGOS par une traduction en systèmes d’équations Booléennes plutôt que par l’intervention d’une mécanique d’exécution ordonnant les changements d’états au sein de chaque instant. Cette traduction permet à Maraninchi et Halbwachs [113] de proposer une méthode de compilation vers un langage s’apparentant à LUSTRE [26]. Ce langage cible a l’avantage d’autoriser l’usage du vaste éventail d’outils qui lui sont associés, notamment la compilation vers un programme C, le test ou la vérification formelle de propriétés sur les programmes (*cf.* § 1.2 page 51). De plus, cette méthode de compilation n’introduit pas d’accroissement de la taille des programmes : la taille du système d’équations résultant d’une composition parallèle varie linéairement par rapport à la taille des descriptions des automates.

Exemple de programme. Le listing 6.2 page ci-contre présente un extrait de la syntaxe textuelle d’ARGOS au moyen d’un codage de la composition parallèle des machines de Mealy de la section 1 page 50. La compilation de ce programme en LUSTRE produit le code du listing 6.3 page ci-contre. Traduit en C par le compilateur LUSTRE académique, ce programme devient une procédure dont le comportement est similaire à la routine `tick()` implantée en Synchronous C à partir des mêmes machines de Mealy, et présentée dans le listing 6.1 page 84.

3.2.2 Le compilateur ARGOS existant

Au début de la réalisation du travail présenté dans cette thèse, il n’existait manifestement pas de compilateur d’ARGOS produisant du code LUSTRE sans effectuer préalablement le produit des automates, et donc non sujet à l’explosion du nombre d’états.

Le seul outil disponible au laboratoire avait été implanté dans le cadre d’études sur les aspects dans le cadre des systèmes réactifs par Altisen *et al.* [2]⁵. Cependant, ce dernier outil n’effectue la traduction en LUSTRE qu’après avoir calculé le produit des automates ; par conséquent, il est très sensible au syndrome d’explosion du nombre d’états et fournit une sortie, lorsqu’il y en a une, de taille déraisonnable à une implantation réaliste.

5. Ce compilateur et une documentation de la syntaxe textuelle d’ARGOS sont disponibles à l’adresse <http://www-verimag.imag.fr/~altisen/DSTAUCH/ArgosCompiler/>.

```

1 process SE (a) (c)           // déclaration du comportement SE.
2 {
3     internal b              // encapsulation (et déclaration du
4     {                       // signal local b).
5         controller          // machine Sa.
6         {
7             init A0         // déclaration de l'état initial de Sa.
8             A0 { }         // depuis l'état A0...
9             -> A1 with a;   // ... aller en A1 si a est « vrai ».
10            +> A0;         // ... rester en A0 sinon.
11
12            A1 { }         // depuis l'état A1...
13            -> A0 with a / b; // ... aller en A0 en positionnant b
14                       // si a est « vrai ».
15            +> A1;         // ... rester en A1 sinon.
16        }
17    ||                       // composition parallèle.
18    controller              // machine Sb (similaire à Sa).
19    {
20        init B0
21        B0 { }
22        -> B1 with b;
23        +> B0;
24
25        B1 { }
26        -> B0 with b / c;
27        +> B1;
28    }
29 }
30 }

```

LISTING 6.2 – Implantation en ARGOS de la composition parallèle des deux machines de Mealy S_a et S_b de la figure 4.1 page 54, synchronisées avec le signal b par un opérateur d'encapsulation. Une transition « $+> E$ with c ; » n'est prise que si la condition c est « vrai » et que la disjonction des conditions des transitions précédentes (dans l'ordre de déclaration) est « fausse » (sucre syntaxique facilitant l'expression de comportements déterministes); une transition notée « $+> E$; » est une version abrégée de « $+> E$ with true; ».

```

1 node SE (a: bool) returns (c: bool);
2 var inA0, inA1, mA, inB0, inB1, mB, b: bool;           -- variables internes.
3 let
4     inA1 = not mA;    inA0 = mA;    b = a and inA1;   -- codage de la machine Sa.
5     mA = true -> pre (not a and inA0 or a and inA1);
6     inB1 = not mB;   inB0 = mB;    c = b and inB1;   -- codage de la machine Sb.
7     mB = true -> pre (not b and inB0 or b and inB1);
8 tel

```

LISTING 6.3 – Traduction directe en LUSTRE du programme du listing 6.2. mA et mB sont les variables codant les états des automates S_a et S_b respectivement : elles sont persistantes puisque leur valeur est utilisée pour le calcul de l'instant suivant ; e.g., mA vaut vrai si S_a est dans l'état A_0 (elle vaut vrai au premier instant, puisque A_0 est l'état initial de la machine S_a). b est l'unique variable partagée parmi les deux ensembles d'équations correspondants au deux automates. Le reste des variables sert au nommage de formules pour la lisibilité.

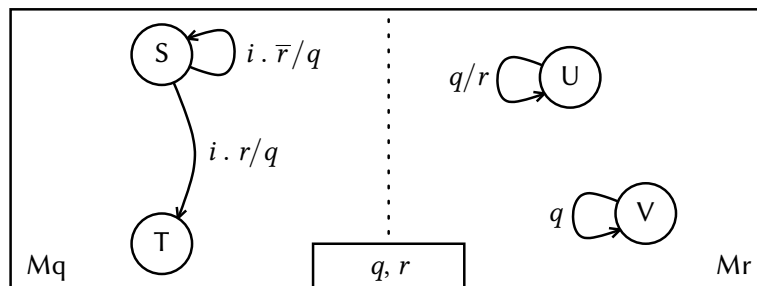


FIGURE 6.1 – Illustration du dialogue instantané sous la forme d’une composition parallèle de machines de Mealy Booléennes, dont seuls les états et transitions significatifs ont été représentés. q et r sont des signaux « encapsulés », c’est-à-dire qu’ils ne peuvent être émis que par les deux machines représentées. Cet exemple est repris de Maraninchi et Rémond [114].

3.3 Remarques à propos de la causalité

Comme on l’a vu dans la section 1 page 50, le paradigme synchrone se base sur l’instantanéité des communications entre les éléments exprimables des programmes. De plus, pour autoriser une spécification (voire une compilation) modulaire de ceux-ci, les concepteurs des langages synchrones y incorporent des opérateurs de composition permettant la définition et l’implantation isolée des comportements.

Dans un cadre d’implantation de programmes synchrones, le problème dit de *causalité* survient lorsqu’une composition introduit un *cycle de causalité*, c’est-à-dire que la réponse d’un sous-programme est calculée à partir d’elle-même. Traduit sous forme de circuit synchrone, un cycle de causalité introduit une *boucle combinatoire*.

3.3.1 Exemple de dialogue instantané : le rendez-vous

La figure 6.1 présente un motif de dialogue instantané sous la forme d’un extrait de programme dans la syntaxe graphique d’ARGOS. Dans cet exemple, si la machine M_q est dans l’état S et que l’entrée i est « vraie », alors la transition vers l’état T n’est prise que si M_r est dans l’état U ; il s’agit donc d’un *rendez-vous* entre ces deux comportements. Le signal local q représente la question émise par M_q , et r est la réponse émise par M_r depuis l’état U uniquement si la requête q l’est aussi.

3.3.2 Implantation séquentielle du rendez-vous

Directement traduit en un programme séquentiel, l’exemple précédent requiert en quelque sorte un aller-retour entre les machines M_q et M_r pour implanter le rendez-vous spécifié.

Si chacune d’elles est associée à une tâche d’un système distribué ou simplement multitâches, alors des primitives de synchronisation autorisant l’implantation de rendez-vous sont nécessaires (*i.e.*, au minimum des « *mutex* » plus variables de condition, sémaphores, moniteurs, ou des variantes distribuées). Il est bien connu que l’usage correct de ces dernières est difficile, d’autant plus que le nombre de tâches impliquées augmente. De plus, les motifs de communication incorrects tels que ceux introduisant un interblocage potentiel sont difficilement détectables à partir de programmes exprimés sous la forme de tâches séquentielles concurrentes employant ces mécanismes pour se synchroniser.

Dans le monde synchrone en revanche, l’expression du parallélisme sert surtout à la spécification et est compilée lors de la traduction en code séquentiel. De tels motifs de communication apparaissent alors clairement sous la forme de cycles de causalité et diverses techniques couramment utilisées existent pour repérer les programmes incorrects (*e.g.*, qui auraient potentiellement introduit des interblocages s’ils avaient

```

1 node id_incorrect (i: bool) returns (o: bool);
2 var inT, inS, mT, inV, inU, mV, r, q: bool;
3 let
4   inT = mT; inS = not mT; q = inS and i and r; -- codage de Mq.
5   mT = false -> pre (inT or inS and i and r);
6   inV = mV; inU = not mV; r = inU and q;      -- codage de Mr.
7   mV = false -> pre inV;
8   o = false;                                -- jamais émis dans le programme.
9 tel

```

LISTING 6.4 – Traduction directe en LUSTRE du programme de la figure 6.1 page précédente. Pour cette implantation, S et U sont les états initiaux de leurs automates respectifs. Dans ce programme, il existe une boucle combinatoire impliquant les variables q et r : r est exprimée en fonction de q , elle-même définie à partir de r .

```

1 node id_correct (i: bool) returns (o: bool);
2 var inT, inS, mT, inV, inU, mV, r, q: bool;
3 let
4   inT = mT; inS = not mT; q = inS and i and inU; -- codage de Mq.
5   mT = false -> pre (inT or inS and i and r);
6   inV = mV; inU = not mV; r = q;                -- codage de Mr.
7   mV = false -> pre inV;
8   o = false;                                -- jamais émis dans le programme.
9 tel

```

LISTING 6.5 – Traduction en LUSTRE du programme de la figure 6.1, en appliquant la transformation des systèmes d'équations proposée par Halbwachs et Maraninchi [81]. Pour cette implantation, S et U sont les états initiaux de leurs automates respectifs. Ce programme ne comprend plus de boucle combinatoire ; seules les définitions de q et r diffèrent par rapport au listing 6.4.

été implantés avec des primitives de synchronisation classiques), et éliminer les cycles éventuels lors de la compilation des programmes corrects :

3.3.3 Traitement des cycles de causalité

Différentes approches ont été adoptées selon les caractéristiques et les besoins d'expressivité des langages.

Des langages déclaratifs proches des circuits synchrones dans le style de LUSTRE rejettent les programmes sur un critère syntaxique : tout programme comprenant un cycle de causalité est considéré comme incorrect. Par exemple, la traduction directe en LUSTRE du programme de la figure 6.1 page ci-contre en utilisant la méthode évoquée dans la section 3.2 page 86, est donné dans le listing 6.4. Ce programme est incorrect puisqu'il comprend un cycle de causalité entre q et r .

Des langages plus proches d'un style impératif comme ESTEREL ou SYNCCHARTS permettent l'expression de programmes comprenant des cycles de causalité lorsque celles-ci n'introduisent pas d'indéterminisme. Cependant, la détection des programmes incorrects est un problème difficile, et les analyses et algorithmes de compilation employés font encore l'objet de travaux de recherche [61, 128, 142, 150].

Enfin, dans le cas d'ARGOS, l'expression des dialogues instantanés est souvent utile (entre autres, pour l'expression des rendez-vous comme dans notre exemple). C'est pourquoi Halbwachs et Maraninchi [81] proposent une méthode d'analyse symbolique des systèmes d'équations Booléennes permettant d'assurer une propriété de *consistance* de ceux-ci, c'est-à-dire de vérifier s'ils ont une solution unique. De plus, si le système a une solution unique, cette analyse produit un nouvel ensemble d'équations équivalent sans boucle

combinatoire ; ce dernier est donc utilisable dans le cadre de la transformation en LUSTRE de programmes ARGOS dont le résultat d'encodage sans ce traitement postérieur serait autrement refusé.

Considérons le système d'équations obtenu à partir du programme de la figure 6.1 page 88. Les équations comprenant un cycle de causalité forment le système suivant :

$$\begin{cases} q = r . i . inS \\ r = q . inU \end{cases}$$

où inS et inU représentent la valeur des variables d'état et ne sont vraies que si les machines M_q et M_r sont respectivement dans leur état S et U . L'application de la méthode de transformation proposée par Halbwachs et Maraninchi sur ce système d'équations révèle qu'il n'a qu'une seule solution (pour $q = r = inS . inU . i$), et le nouvel ensemble d'équations fourni permet de définir les valeurs de q et r sans cycle de causalité :

$$\begin{cases} q = inS . inU . i \\ r = q \end{cases}$$

Le programme LUSTRE résultant de ces transformations est exposé dans le listing 6.5 page précédente.

Bien entendu, il existe des situations pour lesquelles le système n'admet pas de solution unique. Elles signifient que le programme spécifié n'est pas déterministe et ne peut donc pas être implanté en l'état.

4 Conclusion du chapitre

Après avoir réalisé une première implantation d'un tick comportant quelques pilotes de périphériques pour la couche de contrôle à l'aide de Synchronous C, l'assignation manuelle des priorités et les contraintes relatives au nombre de signaux et comportements révéla rapidement les limites de cette solution de mise en œuvre.

Pour cette raison, et puisque la traduction manuelle d'ARGOS en LUSTRE ou le codage direct des automates dans ce langage s'avère fastidieuse et sujette à erreur, une implantation d'un compilateur transformant un programme spécifié en ARGOS vers un programme LUSTRE correspondant en utilisant la traduction évoquée dans la section 3.2 page 86 a été réalisée. En outre, la méthode d'analyse servant à la détection et la résolution de certains cycles de causalité a été implantée pour permettre un usage plus libre des opérateurs de composition d'ARGOS.

Maintenant que nous avons détaillé les choix effectués pour l'implantation du prototype de couche de contrôle, nous pouvons dans le chapitre suivant effectuer quelques évaluations de celle-ci afin d'en apprécier les caractéristiques et le réalisme.

IMPLANTATION ET ÉVALUATIONS

Contenu du chapitre

| | | |
|-------|---|-----|
| 1 | Introduction du chapitre | 92 |
| 2 | Implantation de la couche de contrôle | 92 |
| 2.1 | Construction du noyau réactif | 92 |
| 2.2 | Adaptation de systèmes invités | 93 |
| 2.2.1 | Adaptation de CONTIKI | 93 |
| 2.2.2 | Mtk : un noyau multifils d'architecture classique | 94 |
| 3 | Étude de cas et principes d'adaptation | 94 |
| 3.1 | Présentation de l'étude de cas | 94 |
| 3.2 | Détails de l'implantation originale | 95 |
| 3.2.1 | Modèle de programmation et exigences implicites | 95 |
| 3.2.2 | Exemple de fonction | 97 |
| 3.3 | Principes de l'adaptation | 97 |
| 3.3.1 | Identification des états du périphérique | 97 |
| 3.3.2 | Traitement des requêtes refusées | 98 |
| 3.4 | Adaptation effective | 98 |
| 3.4.1 | Une adaptation possible | 98 |
| 3.4.2 | Une autre adaptation | 100 |
| 3.4.3 | Quantification des modifications requises pour l'étude de cas | 100 |
| 3.5 | Discussion sur l'adaptation | 100 |
| 4 | Évaluation quantitative | 101 |
| 4.1 | Enpreinte mémoire | 101 |
| 4.2 | Coût calculatoire | 102 |
| 4.3 | Comparaison avec des approches existantes | 102 |
| 5 | Évaluation qualitative et extensibilité | 102 |
| 5.1 | Sur le développement des pilotes | 103 |
| 5.2 | Extensibilité | 103 |
| 5.2.1 | Accès directs au matériel | 103 |
| 5.2.2 | Choix « éclairé » du mode de basse consommation | 104 |
| 5.2.3 | Émission simultanée de plusieurs requêtes logicielles | 105 |
| 6 | Considérations sur les interblocages | 105 |
| 6.1 | Exemples de situations d'interblocage potentiel | 105 |
| 6.1.1 | Blocage au niveau du gestionnaire des tâches de l'invité | 106 |
| 6.1.2 | Blocage au niveau des ressources gérées par la couche de contrôle | 106 |

| | | |
|-------|--|-----|
| 6.2 | Possibilités de détection hors ligne | 106 |
| 6.2.1 | Sur les interblocages au niveau de l'invité | 107 |
| 6.2.2 | Traitement du problème de progression au niveau des ressources | 107 |
| 6.3 | Conclusion sur les interblocages | 108 |
| 7 | Synthèse de contrôleur automatisée | 108 |
| 7.1 | Sur le non déterminisme des contrôleurs produits | 109 |
| 7.1.1 | Origine du non déterminisme | 109 |
| 7.1.2 | Traitements possibles | 109 |
| 7.2 | Le cas de BZR/SIGALI | 110 |
| 7.2.1 | Le traitement du non déterminisme dans BZR/SIGALI | 110 |
| 7.2.2 | Expérience d'implantation | 111 |
| 7.3 | Vers une chaîne d'outils idéale | 111 |
| 8 | Conclusion du chapitre | 112 |

1 Introduction du chapitre

Dans ce chapitre, nous présenterons dans un premier temps quelques travaux d'implantation qui ont été réalisés en vue de la validation du concept présenté au chapitre 5. Une étude de cas servira ensuite à exemplifier les changements à effectuer lors de l'adaptation d'un service système existant afin d'utiliser la couche de contrôle pour gérer le matériel : elle permettra alors de procéder à une première évaluation qualitative de l'approche, ainsi que de définir quelques limites de son usage. Puis quelques extensions possibles seront proposées afin d'examiner la généralité et la souplesse de la solution. Des limites relatives à l'apparition d'interblocages potentiels dus au contrôle de ressources contraignant seront exposées, mises en perspectives avec les contraintes imposées par d'autres approches autorisant un minimum de contrôle. Enfin, des pistes vers l'utilisation d'outils permettant l'automatisation de la synthèse du contrôleur seront présentées, pour finalement nous amener à identifier les caractéristiques nécessaires à une chaîne d'outils que nous jugerions idéale pour réaliser cette tâche.

De même que dans le chapitre 5, les listings de code seront retranscrits en anglais.

2 Implantation de la couche de contrôle

Nous avons implanté la couche de contrôle spécifiée dans le chapitre 5 afin de contrôler avec cet outil la plate-forme matérielle Wsn430 prise en exemple dans la section 2.2.2 page 14. Cette plate-forme s'imposa comme cible d'implantation en raison de la disponibilité d'une chaîne d'outils permettant l'émulation des nœuds pris isolément, ainsi que la simulation d'un canal radio complet (*cf.* § 2 page 81).

2.1 Construction du noyau réactif

En nous basant sur le modèle présenté dans la section 6.1 page 74, nous avons implanté un ensemble complet de pilotes de périphériques afin de construire une couche de contrôle pour la plate-forme Wsn430. Le noyau réactif a été spécifié à l'aide d'automates de pilotes entièrement spécifiés en ARGOS, le tout traduit en LUSTRE, puis en C à l'aide de la chaîne de compilation académique associée (*cf.* § 1.2 page 51).

Un contrôleur a été conçu manuellement, afin d'assurer quelques propriétés à garantir sur les ressources de la plate-forme matérielle. Ces dernières se rangent dans la catégorie des *propriétés de sûreté*, telles que l'exclusion mutuelle pour la gestion de ressources partagées, ou encore la réduction des pics de consommation en empêchant l'atteignabilité d'états globaux où au moins deux périphériques (comme un ADC ou le transmetteur radio) sont dans leurs modes de fonctionnement les plus gourmands en énergie.

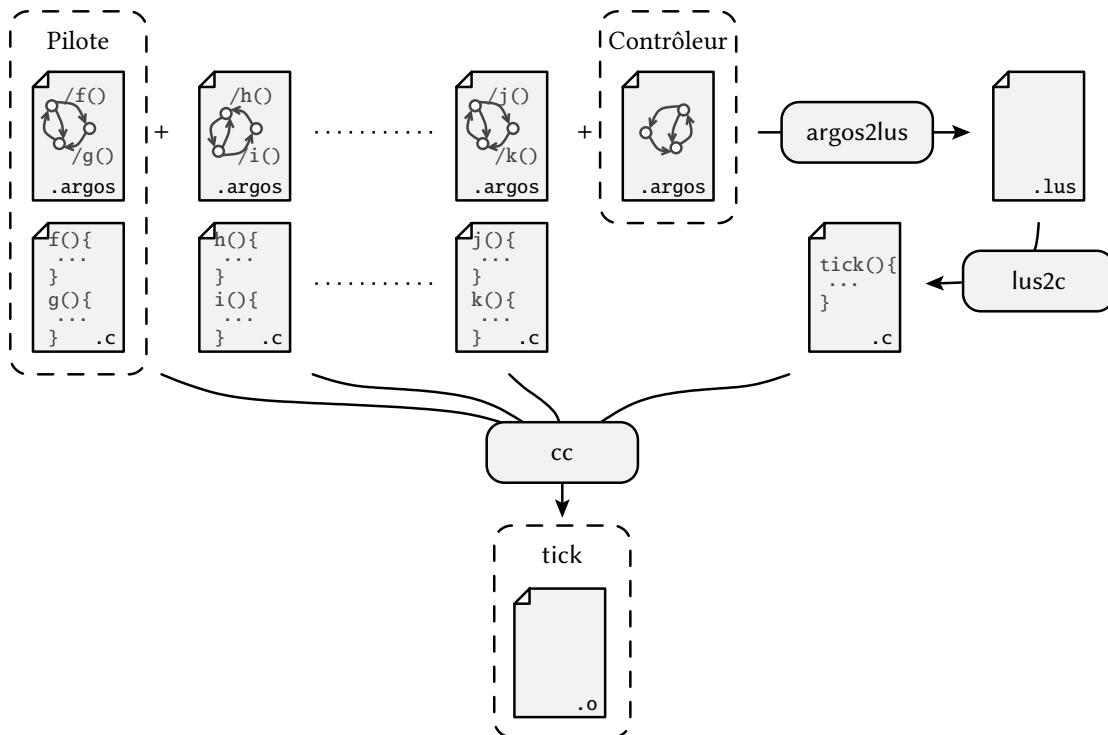


FIGURE 7.1 – Outils utilisés pour la construction du noyau réactif du prototype de couche de contrôle. *argos2lus* est le compilateur d'ARGOS utilisé pour traduire la composition parallèle des automates des pilotes vers un programme LUSTRE ; *lus2c* transforme ce dernier en un code C, et *cc* compile le tout en code objet pour la plate-forme cible.

La figure 7.1 résume la chaîne d'outils d'implantation que nous avons employé pour construire le noyau réactif de la couche de contrôle.

2.2 Adaptation de systèmes invités

En ce qui concerne les couches logicielles de l'invité, nous avons adapté deux systèmes d'exploitation conçus pour s'exécuter nativement sur des nœuds de réseaux de capteurs sans fil, afin qu'ils utilisent la couche de contrôle décrite dans la section ci-dessus pour gérer la plate-forme matérielle.

2.2.1 Adaptation de CONTIKI

L'adaptation de CONTIKI (cf. § 4.2.2 page 41) requiert l'écriture d'un ensemble de pilotes de périphériques compatibles avec les constructions de ce système, et destinés à faire remonter les abstractions présentées par la couche de contrôle au niveau des services implantés dans CONTIKI.

L'écriture de ces pilotes de périphériques adaptés pour dialoguer avec la couche de contrôle ne présente pas de difficulté particulière autre que la contrainte d'employer les moyens de notification d'erreur spécifiques à ce système lors de refus de requêtes. De plus, la plupart des abstraction génériques utilisées par les services systèmes implantés dans CONTIKI doivent présenter une interface de manipulation « standardisée » qui ne comporte souvent que des appels bloquants qui, lorsqu'ils se terminent, ont effectué l'action demandée ou retournent un code d'erreur ; pour cela, les pilotes sont implantés en utilisant de l'attente active si besoin. Si nécessaire, cette scrutation du matériel peut se traduire lors de l'adaptation par une scrutation des comptes-rendus de la couche de contrôle.

2.2.2 Mtk : un noyau multifils d'architecture classique

Afin de s'assurer que l'architecture de la couche de contrôle est compatible avec des manipulations plus classiques des contextes d'exécution que celles utilisées pour implanter les proto-threads de CONTIKI, nous avons construit et adapté les pilotes de périphériques d'un noyau de système multifils préemptif. (Nous avons initialement construit ce dernier dans l'optique de l'architecturer autour d'une structure assurant un contrôle global des pilotes de la plate-forme ; l'isolation ultérieure de la couche de contrôle à partir de ce système permet d'identifier le caractère « virtualisant » de celle-ci.) Ce noyau ordonnance ses tâches à l'aide de priorités dynamiques, et fait usage de primitives de synchronisations classiques.

L'écriture des pilotes adaptés pour cet invité fut similaire au développement de ceux de CONTIKI, excepté pour ce qui est de l'usage des primitives de communications bloquantes autorisées par son modèle de programmation concurrente plus « libéral » dans ce domaine.

3 Étude de cas et principes d'adaptation

Comme expliqué au chapitre 5 et dans la section précédente, une étape préliminaire d'adaptation est requise en vue d'utiliser une pile logicielle invitée dans le cadre d'une couche de contrôle, y compris si celle-ci ne requiert pas de support système. Pour évaluer les modifications à réaliser et mettre en avant les principes généraux d'une telle adaptation, nous présentons dans cette section une étude de cas.

3.1 Présentation de l'étude de cas

Nous avons choisi d'illustrer ce processus au travers du « portage » d'une implantation existante de protocole de contrôle d'accès au support physique (« *Medium Access Control* » — MAC) ; dans le modèle OSI [164], cette partie de la pile réseau est responsable de la gestion des liens de données au dessus du niveau physique (dans notre cas, ce dernier est le canal de transmission radio) : elle manipule des *trames* encapsulant des *paquets*, eux-mêmes reçus de ou transmis à la couche de routage.

Base de travail. Plus spécifiquement, il s'agit d'une implantation du protocole X-MAC [24], un MAC conçu dans l'optique de réduire la consommation énergétique globale dans les réseaux de capteurs sans fil en maîtrisant les communications radio. Elle est extraite de la boîte à outils SensTools [92], et a été conçue pour s'exécuter sans support système particulier. Elle utilise directement des pilotes de périphériques inclus dans la même chaîne d'outils, qui gèrent la plate-forme matérielle Wsn430.

Concernant le pilote de l'émetteur-récepteur radio CC1100 [40] fourni dans cet ensemble d'outils, on remarquera qu'il est relativement simpliste ; il ne spécifie aucun comportement ni ne maintient en interne l'état du périphérique, et est simplement constitué d'une couche d'encapsulation des commandes et informations envoyées et reçues sur le bus et les liens digitaux reliant le microcontrôleur au transmetteur (il s'agit surtout d'une couche de nommage des commandes et données).

Objectif. L'objectif du travail d'adaptation est de modifier l'implantation originale du protocole afin que celui-ci accède aux composants matériels au travers de la couche de contrôle : les pilotes initialement employés seront retirés de la pile logicielle complète, et les périphériques seront gérés par la couche de contrôle, elle-même actionnée par le X-MAC via la couche d'adaptation. Nous montrons dans la section suivante que cette transformation ne requiert que peu de changements dans la manière dont le X-MAC agit sur le matériel.

Remarques sur les caractéristiques du transmetteur radio. Le protocole X-MAC requiert quelques capacités spécifiques de la part du transmetteur radio, notamment en matière de transitions entre les différents modes de fonctionnement du périphérique. Afin d'être en mesure d'exécuter ce protocole sur

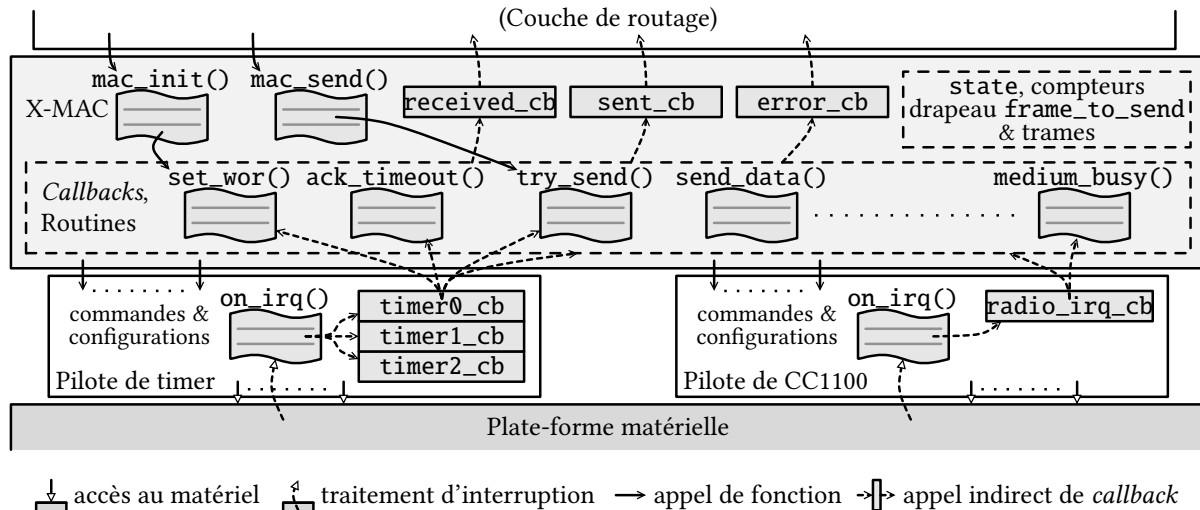


FIGURE 7.2 – Représentation de la structure de l'implantation originale du protocole X-MAC et des pilotes de périphériques qu'elle utilise. Les « appels indirects de callbacks » sont des appels de routines par référence modifiable dynamiquement.

la plate-forme d'exemple, le pilote de transmetteur radio intégré dans la couche de contrôle doit être une version augmentée de celui représenté dans la figure 5.7 page 77, lors de la description de la solution.

En pratique, il a plus d'états et de transitions contrôlables. Il comprend notamment un état additionnel qui représente un mode d'émission depuis lequel le périphérique entre automatiquement en mode d'écoute lorsque la transmission d'un paquet s'achève, et ce sans notification préalable du microcontrôleur. De plus, il supporte un mode d'écoute passive reflétant un état de basse consommation du composant dans lequel ce dernier entre automatiquement en mode de réception lorsqu'il détecte un signal radio.

3.2 Détails de l'implantation originale

La figure 7.2 décrit la structure de l'implantation originale du protocole X-MAC à adapter, ainsi que les interactions avec les composants des couches logicielles voisines.

3.2.1 Modèle de programmation et exigences implicites

Après un premier appel à la procédure d'initialisation `mac_init()`, tout nouveau calcul effectué par cet X-MAC est déclenché soit :

- lors du traitement d'une requête d'interruption matérielle ;
- à l'initiative de la partie supérieure de la pile réseau — typiquement, une couche de routage.

En d'autres termes, cette implantation utilise un style de programmation événementiel, où les déclencheurs sont soit des interruptions, soit des appels directs de routines depuis d'autres parties de la pile logicielle (cf. § 2.3.1 page 34).

Cet X-MAC modifie dynamiquement les références des routines que les pilotes exécutent lors de la réception de requêtes d'interruption matérielles. Par exemple, `radio_irq_cb` peut désigner une fonction, qui sera alors exécutée sur requête d'interruption envoyée par le transmetteur radio. Enfin, trois timers distincts peuvent être utilisés simultanément, chacun étant capable de déclencher l'exécution d'une routine différente.

Interface exposée à la couche de routage. L'envoi d'un paquet se fait au moyen de la fonction `mac_send()`, laquelle se termine immédiatement en retournant un entier non nul en cas d'erreur dans son

```

1  /* note that 'mac_send()' copies the buffer iff it succeeds...          */
2  if (mac_send (buffer, buffer_length, destination_address) != 0)
3  { ... } /* usage error (either the given packet is too long or there is one
4           to send already).                                           */
5
6  ... /* do some more work, without detecting if the packet has effectively
7         been sent, or that an error occurred (both detectable by executions of
8         callbacks pointed to by 'sent_cb' or 'error_cb').             */
9
10 /* potential failure: the MAC may still be sending the packet given on line 2.*/
11 if (mac_send (buffer, buffer_length, destination_address) != 0)
12 { ... } /* usage error again...                                       */

```

LISTING 7.1 – Exemple d’usage incorrect de l’implantation de l’X-MAC par la couche de routage : le second appel à `mac_send()` à la ligne 11 échouera si les calculs à effectuer depuis le premier appel à la ligne 2 ne laissent pas assez de temps au MAC pour finir l’envoi du premier paquet.

utilisation, ou zéro si le processus d’envoi a démarré avec succès. Subséquemment, la couche de routage est notifiée du statut final de cette opération par l’exécution de fonctions de rappel (« *callbacks* ») :

- La fonction désignée par la référence `sent_cb`, s’il en existe une, est exécutée lorsque la transmission d’une trame a réussi ;
- Sinon, celle désignée par `error_cb` est appelée (lorsque, e.g., le canal radio est surchargé).

En outre, `received_cb` peut aussi désigner une fonction qui sera alors exécutée lors de la réception d’une trame.

La valeur de retour de ces fonctions de rappel sert à transmettre des informations sur le prochain état du microcontrôleur, requis par les couches logicielles supérieures. Par exemple, si le routage est un composant actif comprenant des tâches qui peuvent être débloquentes lors d’un appel à l’une de ses procédures de rappel, alors il lui est nécessaire de signifier au niveau des traitants d’interruptions que le microcontrôleur doit continuer à calculer après qu’ils se terminent (si l’appel de cette procédure de rappel est une conséquence d’une telle interruption).

Structure interne (et contraintes d’usage additionnelles). Dans les faits, cette implantation est l’encodage dans un style événementiel de l’automate décrivant le protocole X-MAC, du point de vue des nœuds. En effet, l’état est encodé à chaque instant par l’association des fonctions de rappel aux différentes références fournies par les pilotes de périphériques sous-jacents, plus quelques mots de mémoire de travail qui persiste entre les appels aux diverses routines de l’X-MAC. Pour prendre un exemple, considérons la routine `set_wor()` qui place le transmetteur radio dans un mode d’écoute passive : elle enregistre également `read_frame()` comme étant la fonction à appeler dès la prochaine réception d’une interruption depuis la radio (`radio_irq_cb` – indiquant qu’une trame a été reçue lorsque le périphérique est dans un tel mode de fonctionnement).

Au reste, une variable `state` code trois états de l’X-MAC de manière très abstraite (*écoute passive*, *transmission* ou *réception*), ainsi qu’un drapeau additionnel `frame_to_send` et deux compteurs, constituent l’ensemble de la mémoire interne servant à la gestion de l’état. Ces dernières données ont deux objectifs :

- détecter une utilisation erronée de l’X-MAC par la couche de routage ;
- décider quel est l’état du protocole lors de l’occurrence de certains événements.

En outre, cette implantation n’est pas ré-entrante. D’une part, aucune de ses routines ne doit être appelée si les requêtes d’interruption matérielles sont autorisées. (Plus précisément, seules les interruptions en provenance du transmetteur radio et du timer utilisé doivent être désactivées : d’autres interruptions peuvent rester démasquées. Dans ce cas, aucun appel depuis la couche de routage ne doit être effectué si l’exécution

```

1  /* function for sending a frame contained in txframe.                */
2  static uint16_t send_data(void) {
3
4      timerB_unset_alarm(ALARM_PREAMBLE);      /* unset alarm.          */
5
6      cc1100_cmd_idle();                        /* goto idle operating mode. */
7      cc1100_cmd_flush_rx();                   /* flush FIFOs.              */
8      cc1100_cmd_flush_tx();
9
10     /* dictate the device to enter idle mode upon end of transmission: */
11     cc1100_cfg_txoff_mode(CC1100_TXOFF_MODE_IDLE);
12
13     cc1100_cmd_tx();                          /* goto transmit mode.       */
14     cc1100_fifo_put((uint8_t*)&txframe.length, txframe.length+1);
15
16     /* setup function to be executed upon end of transmission:         */
17     cc1100_gdo0_register_callback(send_done);
18
19     return 0;    /* indicate we can stay in low-power mode, if we were already. */
20 }

```

LISTING 7.2 – Une routine extraite de l'implantation originale de l'X-MAC.

d'une routine de l'X-MAC est en attente de la fin du traitement d'une interruption en provenance d'un autre périphérique.) D'autre part, elle ne supporte pas non plus de requêtes concurrentes en provenance de la couche de routage : comme exemplifié dans le listing 7.1 page précédente, tout appel à `mac_send()` doit nécessairement être suivi de l'exécution d'une des fonctions de rappel désignées par `sent_cb` ou `error_cb`, avant que `mac_send()` puisse être appelée à nouveau.

3.2.2 Exemple de fonction

Le listing 7.2 présente une fonction interne de l'implantation du MAC. Elle amorce la transmission du contenu du tampon `txframe`, et est toujours appelée explicitement par d'autres routines du MAC, c'est-à-dire que son exécution n'est jamais déclenchée par un pilote de périphérique au travers d'une référence de fonction de rappel.

Dans un premier temps, elle désactive un timer¹ qui peut encore être actif à ce point de l'exécution (ligne 4) et prépare le périphérique pour la transmission (lignes 6 à 14). Entre-temps, la radio est configurée pour entrer automatiquement en mode *idle* dès que la trame est transmise (à la ligne 11). Les valeurs de `state` et `frame_to_send` restent inchangées : une analyse relativement simple du graphe d'appel des routines du MAC nous instruit qu'elles indiquent toujours que le MAC est en train de transmettre un paquet lorsque `send_data()` est appelée. Enfin, la fonction de rappel associée à l'interruption matérielle issue du transmetteur radio (à ce point d'exécution toujours configurée pour être émis à la fin de la transmission par le périphérique), est remplacée par la fonction `send_done()` à la ligne 17.

3.3 Principes de l'adaptation

3.3.1 Identification des états du périphérique

Afin d'adapter l'implantation du protocole X-MAC que nous avons décrite ci-dessus, les transitions et commandes de configuration impactant le mode de fonctionnement des composants matériels doivent être

1. Ce type de protocole MAC est basé sur un principe d'échantillonnage de préambule qui requiert, au moment de la transmission d'une trame, d'envoyer des préambules de manière périodique.

identifiées, puis traduites en émissions de requêtes logicielles correspondantes vers la couche de contrôle. Cette tâche est réalisable par rétro-ingénierie, en découvrant les états effectifs des périphériques à partir des commandes qui sont envoyées à leurs pilotes respectifs. Lorsque l'un d'eux ne maintient aucune information explicite liée à cet état (comme le transmetteur radio dans notre étude de cas), ces informations requises doivent être déduites du flot de contrôle et du graphe d'appel. Dans le cas de l'encodage du code à porter en un style événementiel (comme dans le cas d'exemple), nous ferons remarquer que ce graphe n'est pas suffisant, et qu'une analyse plus détaillée incluant les affectations des différentes références des fonctions de rappel s'avère nécessaire.

Une fois cette information extraite, les commandes originales sont remplacées par des appels à des fonctions de la couche d'adaptation émettant les requêtes logicielles de changement de mode d'opération.

3.3.2 Traitement des requêtes refusées

Puisque la couche de contrôle peut refuser les requêtes logicielles susceptibles de faire entrer la plateforme dans un état violant des propriétés globales, un soin particulier est nécessaire pour traiter les émissions potentiellement refusables.

Le plus souvent, plusieurs choix sont envisageables lors de l'adaptation de code existant. Par exemple, l'adaptation d'une routine supposée ne jamais faillir (c'est-à-dire qu'aucun mécanisme n'est disponible pour signifier une erreur aux niveaux supérieurs de la pile logicielle) peut se faire très simplement par l'usage d'une fonction de la couche d'adaptation émettant la requête de manière répétée jusqu'à ce qu'elle soit acceptée ; ainsi, l'appelant d'une telle routine peut alors ne pas être modifié. D'autres possibilités résident dans la réutilisation des mécanismes de rapport d'erreur fournis, par exemple en exécutant une fonction de rappel dédiée après plusieurs essais d'émission infructueux.

3.4 Adaptation effective

Dans cette section, nous détaillerons les résultats de l'emploi des principes détaillés § 3.3 page précédente dans le cas de notre étude de cas.

3.4.1 Une adaptation possible

Le listing 7.3 page ci-contre présente le code résultant d'une adaptation du code original du listing 7.2 page précédente.

Assez clairement, quelques-unes des commandes envoyées au transmetteur radio peuvent être directement traduites en émissions des requêtes logicielles correspondantes vers la couche de contrôle ; c'est par exemple le cas de la ligne 6 du listing 7.2 page précédente, devenant la ligne 30 du listing 7.3. Un mécanisme de répercussion d'erreur est utilisable simplement puisque l'implantation originale de l'X-MAC expose la référence de routine de rappel `error_cb` aux niveaux logiciels supérieurs. En employant cette solution dans ce cas d'exemple, une requête refusée à chaque essai (dans les procédures `radio_idle()` ou `radio_send()`) serait visible depuis la couche de routage comme un échec d'envoi. Des informations détaillées pourraient également être transmises comme argument de la routine de rappel, afin d'indiquer la cause de l'erreur pour que le niveau responsable du routage puisse prendre une décision éclairée.

De plus, les instructions des lignes 11 et 13 dans le code original ont entraîné l'usage de la transition vers un état d'émission retournant automatiquement dans le mode *idle* à la fin de la transmission, sans requérir une action de la part du microcontrôleur. Ainsi, dans le cas de notre transmetteur radio l'appel de `radio_send()` à la ligne 41 émet le signal `enter_tx` à la couche de contrôle. Cette fonction est construite sur un modèle similaire à `radio_idle()`, mais gère les données d'entrées (le tampon d'émission) associées au signal émis.

Finalement, les abonnements des traitants des requêtes d'interruption matérielles furent traduits en associations des mêmes routines de rappel aux interruptions virtuelles correspondantes, émises par la

```

1  /* example wrapper function for radio management through the control layer.
2     this function is part of the adaptation layer.                                     */
3  extern status_t radio_idle (void) {
4
5     /* some arbitrary value defining the maximum number of retries; we could
6        also make it an argument, retry infinitely, etc.:                             */
7     const uint8_t max_tries = 8;
8     cl_outputs_v received; /* this set will be assigned by 'on_sw()'. */
9     uint8_t tries = 0; /* tries counter. */
10
11    on_sw (cc1100_idle, & received); /* first try. */
12
13    while (! cl_outputs_test (received, cc1100_ack) && /* while we... */
14           tries++ != max_tries) { /* ... do not receive the acknowledgment: */
15        ... /* some code to be executed on forbidden request... note that it can
16            be a call to a blocking function (e.g., for waiting some time); */
17        on_sw (cc1100_idle, & received); /* next try. */
18    }
19
20    /* return an error code on constant refusal: */
21    return cl_outputs_test (received, cc1100_ack) ? SUCCESS : -EFAIL;
22 }
23
24 /* adapted function for sending data, with advanced refusal handling. */
25 static uint16_t send_data (void) {
26
27    timerB_unset_alarm (ALARM_PREAMBLE); /* unset alarm. */
28
29    /* goto idle operating mode (this also flushes the buffers): */
30    if (radio_idle () != SUCCESS) {
31        /* the request has been refused, we can choose to notify the error;
32           one could also try to enter in low-power listening here. (*) */
33        if (error_cb) /* notify, if we have an error handler. */
34            return error_cb (); /* we could also pass a detailed error code. */
35        set_wor (); /* otherwise, enter low-power listening mode. */
36    }
37
38    /* setup function to be executed upon end of transmission (VIRQ handler): */
39    radio_register_transmission_done_cb (send_done);
40
41    if (radio_send ((uint8_t*)&txframe.length, txframe.length+1) != SUCCESS) {
42        if (error_cb) return error_cb (); /* ibid (*) */
43        set_wor ();
44    }
45
46    return 0; /* this value is ignored by the control layer actually. */
47 }

```

LISTING 7.3 – Adaptation possible du code original du listing 7.2, incluant un mécanisme avancé de gestion de des requêtes refusées. On notera que les fonctions `radio_idle()` et `radio_send()` font partie de la couche d'adaptation : elles émettent des requêtes logicielles à la couche de contrôle, gèrent les données utiles, et interprètent les sorties (en utilisant notamment le prédicat `cl_outputs_test()`, vrai si un compte-rendu donné appartient à l'ensemble résultant d'une émission de requête). Nous donnons `radio_idle()` en exemple : elle essaie d'émettre la requête `cc1100_idle` plusieurs fois jusqu'à réception de l'acquittement correspondant `cc1100_ack`, ou se termine en retournant un code d'erreur sinon.

```

1  /* adapted function for sending data (debug version).          */
2  static uint16_t send_data (void) {
3
4      timerB_unset_alarm (ALARM_PREAMBLE);    /* unset alarm.      */
5
6      /* goto idle operating mode (this also flushes the buffers): */
7      assert (radio_idle () == SUCCESS);
8
9      /* setup function to be executed upon end of transmission (VIRQ handler): */
10     radio_register_transmission_done_cb (send_done);
11
12     /* actual transmission:                                     */
13     assert (radio_send ((uint8_t*)&txframe.length), txframe.length+1) == SUCCESS);
14
15     return 0;          /* this value is ignored by the control layer actually. */
16 }

```

LISTING 7.4 – Une autre adaptation possible du code original du listing 7.2 page 97, utile dans l’optique de tester l’usage du transmetteur radio par l’implantation de l’X-MAC. La procédure `assert()` est l’outil de débogage classique qui « imprime » (si possible) généralement un message permettant de localiser l’erreur a posteriori, puis bloque l’exécution, si son argument s’évalue à « faux ».

couche de contrôle. Dans le listing 7.3, la ligne 39 enregistre `send_done()` comme traitant à exécuter à la fin de la transmission ; après l’exécution de cette instruction, et si l’interruption virtuelle n’est pas masquée, la fonction spécifiée sera appelée lors de l’émission du compte-rendu `tx_done` par la couche de contrôle.

3.4.2 Une autre adaptation

Le listing 7.4 présente une possibilité d’adaptation supplémentaire du code originale du listing 7.2 page 97. Cette seconde solution de portage, combinée avec une version non contrôlable d’un pilote de transmetteur radio intégrée dans la couche de contrôle, peut s’avérer très efficace dans la découverte d’usages incorrects de ce périphérique par une implantation du protocole X-MAC ; par exemple, l’usage de transitions entre modes d’opérations inexistantes peut être aisément détecté par ce moyen.

3.4.3 Quantification des modifications requises pour l’étude de cas

Dans les faits, l’adaptation de l’implantation originale du protocole X-MAC considérée dans cette étude de cas requiert l’identification de 28 commandes émises vers le transmetteur radio et leur traduction en appels vers 9 fonctions de la couche d’adaptation (toutes similaires ou plus simples que celle présentée en exemple dans le listing 7.3 page précédente). De plus, le traitement des requêtes refusées fut facilité par l’existence préalable de la référence vers une fonction de rappel `error_cb`, qui permet de faire remonter le traitement de certains refus aux niveaux supérieurs de la pile réseau : les refus de requêtes s’y manifestent alors de la même manière que des situations de surcharge du canal radio. Enfin, plus de 85% des lignes du code original (approximativement 700 lignes de code C) restèrent inchangées.

3.5 Discussion sur l’adaptation

Le processus d’adaptation décrit dans cette étude de cas nous permet de mettre en avant quelques conseils d’usage de notre solution.

D’une part, l’adaptation de pilotes existants requiert d’identifier et de clarifier les différents modes de fonctionnement du périphérique géré, ainsi que les transitions entre eux. Pour l’implantation originale

considérée dans étude, et *a fortiori* tout encodage d'automate sous forme événementielle avec changement dynamique des abonnement des fonctions de rappel, une analyse du graphe d'appel ne suffit pas pour déterminer l'ensemble de ces transitions ; la connaissance des différentes routines potentiellement exécutées à chaque instant est nécessaire. Une méthode d'adaptation illustrée dans la section 3.4.2 page ci-contre permet d'effectuer des tests simples de l'usage correct des modes et transitions du périphérique.

D'autre part, suivant le contexte, les refus de requête définitifs (*i.e.*, tous les essais d'émission à la couche de contrôle ont été refusés) peuvent s'apparenter à des défaillances du matériel ou de l'environnement qui sont parfois déjà prises en compte par les couches supérieures du logiciel ; dans notre exemple, la couche de routage peut observer de telles situations comme une surcharge du canal radio. Bien évidemment, cette agrégation des cas d'erreur peut ne pas être envisageable dans bon nombre de situations, mais nous pensons qu'une telle vision permet de rester compatible avec un grand nombre d'implantations de couches de routage ne supportant pas d'autres retours d'erreur que ceux concernant la surcharge du canal radio.

En revanche, une gestion plus fine des refus de requêtes implique des choix qui, s'ils impactent peu le comportement du pilote, nécessitent parfois d'en modifier l'interface (ne serait-ce qu'en ajoutant des codes d'erreur à utiliser par les couches logicielles supérieures).

4 Évaluation quantitative

Nous avons décrit à la section 2 page 92 l'implantation d'un prototype correspondant à l'architecture logicielle permettant la mise en œuvre centralisée de politiques de contrôle global que nous avons décrite au chapitre 5. Cette implantation a été testée avec succès à l'aide du simulateur cycle-précis décrit section 2 page 81, ce qui est une garantie raisonnable de sa capacité à s'exécuter sur une plate-forme réelle : les éventuelles différences de comportement avec la plate-forme réelle résident dans les aspects temporels des communications sur les liens digitaux et les erreurs matérielles pouvant survenir dans les périphériques ; seule une émulation fidèle des instructions du CPU et des modes différents périphériques est nécessaire à l'évaluation de l'intégration de la couche de contrôle avec l'invité.

Cependant, une technique permettant d'assurer des propriétés globales introduit presque assurément des surcoûts en matière d'occupation mémoire ou de la charge du processeur. Dans le but d'apprécier le réalisme de notre solution, nous estimons ces coûts dans la présente section, et les comparons à des données disponibles concernant des systèmes existants.

4.1 Empreinte mémoire

L'empreinte mémoire de la partie contrôle du tick (*i.e.*, le code et les données résultant de la compilation des automates des pilotes et du contrôleur global simple) implanté dans le prototype est environ de 1,5 à 2 kilo-octets.

L'espace de la pile d'exécution requis pour ce calcul est environ 100 octets. En revanche, nous ferons remarquer qu'il ne peut y avoir qu'un appel de `react()` en cours à la fois ; en conséquence cet espace peut aisément être partagé par les différentes tâches de l'invité par de simples sauts de pile à l'appel et au retour du tick. Les autres parties de la couche de contrôle comprennent principalement le code opérant des pilotes de périphériques qui seraient inclus dans l'invité s'il s'exécutait sans couche de paravirtualisation. En outre, l'adaptation de l'invité introduit une portion de code supplémentaire de très petite taille mais difficilement quantifiable, puisque les quelques fonctions d'adaptation (se chargeant des essais d'émission des requêtes) sont partageables afin que le code des boucles d'essai d'émissions des requêtes ne soit présent qu'une fois en mémoire de travail. L'implantation actuelle du code et des structures de données statiques liées aux interactions entre le code opérant, l'invité et le tick, occupe environ 1,2 kilo-octet.

| | TINYOS-1.1 (noyau) | TINYOS-1.1 | RETOS | MtK/Couche de contrôle |
|-----|--------------------|------------|---------|------------------------|
| ROM | 11,2 ko | 21 ko | 23,1 ko | 24,5 ko |
| RAM | 311 o | 798 o | 824 o | 806 o |

TABLE 7.1 – *Empreinte mémoire typique de quelques systèmes pour nœuds de réseaux de capteurs sans fil existants, comprenant divers pilotes de périphériques et services systèmes. La colonne « MtK/Couche de contrôle » représente la combinaison des implantations de la couche de contrôle et du système présenté à la section 2.2.2 page 94. (Les données concernant TINYOS et RETOS ont été reprises de Cha et al. [32].)*

$$\frac{\text{ICEM avec } n \text{ arbitres \& } m \text{ « power managers »}}{\lesssim 350n + 400m} \quad \Bigg| \quad \begin{array}{l} \text{tick} \\ \lesssim 1\,600 \end{array}$$

TABLE 7.2 – *Surcoût en matière de cycle de calcul du CPU impliqué par la solution décentralisée « ICEM » (emploi d’arbitres et de gestionnaires d’énergie locaux spécifiques), comparé à celui introduit par notre implantation.*

4.2 Coût calculatoire

Le coût temporel d’une exécution du tick dépend des signaux d’entrées : ce temps correspond à au plus 1 600 cycles (200 μ s sur un microcontrôleur MSP430 dont le CPU est cadencé à 8MHz) dans notre implantation actuelle².

4.3 Comparaison avec des approches existantes

En l’absence de chiffres de référence significatifs concernant les surcoûts d’occupation mémoire et calculatoire impliquées par d’éventuelles solutions existantes d’implantation de contrôle global dans les systèmes d’exploitation pour nœuds de réseaux de capteurs sans fil, nous évaluons ces surcoûts pour notre solution en la comparant à quelques systèmes proposant des mécanismes de contrôle local. Nous résumons dans le tableau 7.1 les empreintes mémoires typiques de quelques-uns de ces systèmes. Comparés à ces chiffres, notre implantation permettant d’assurer un contrôle global introduit une augmentation de moins de 10% de l’occupation mémoire. Nous estimons encourageant et réaliste le rapport entre cette augmentation et le bénéfice apporté par l’implantation d’une politiques de contrôle global possible avec notre proposition.

Le tableau 7.2 nous permet de comparer les ordres de grandeur des surcoûts en matière de cycles de calcul du CPU introduits par la solution proposée par ICEM (cf. § 5.1 page 45) et notre proposition. Même si le tick est exécuté à chaque requête de changement de mode ou opération requérant des accès à des ressources gérées par la couche de contrôle, et de même que pour l’occupation mémoire, nous considérons raisonnables le surcoût introduit par la couche de contrôle par rapport au bénéfice apporté par la possibilité de contrôle global.

5 Évaluation qualitative et extensibilité

Après l’évaluation quantitative donnée dans la section précédente destinée à montrer le caractère réaliste de l’approche que nous proposons, nous pouvons aborder les différents aspect originaux qui nous permettrons d’évaluer les points forts de cette solution en matière d’implantation et d’extensibilité.

2. On notera que 1 600 cycles est, sur les plates-formes matérielles que nous considérons, du même ordre de grandeur que le coût d’un appel à `printf` pour émettre sur un lien série classique un entier raisonnablement grand mis sous la forme d’une chaîne de caractères.

5.1 Sur le développement des pilotes

Nous estimons que le développement manuel de pilotes de périphériques destinés à être intégrés à la couche de contrôle peut s'avérer plus simple que dans le cadre de systèmes classiques. La principale raison en est que dans notre approche de construction et d'intégration des pilotes, une distinction claire est requise entre, d'une part la spécification du comportement relatif au mode d'opération du périphérique et d'autre part le code opérant prenant en compte les données utiles.

Concernant les pilotes adaptés (*i.e.*, qui font partie du système invité), nous avons exemplifié lors de la description du processus d'adaptation de la section 3 page 94 des modifications nécessaires au traitement des requêtes refusées. Cependant, nous pensons que ce besoin de modification met en avant les erreurs potentielles en les rendant explicites, et oblige à les traiter de manière claire. Cela aide donc le programmeur du logiciel à réaliser une conception simple et élégante du système, y compris pour les pilotes de périphériques et services systèmes.

5.2 Extensibilité

Dans le but d'évaluer l'extensibilité de notre approche, nous énumérons dans cette section quelques exemples de cas complexes à traiter ou requérant parfois des solutions *ad hoc* selon d'autres méthodes d'implantation.

5.2.1 Accès directs au matériel

Dans quelques cas, il peut être intéressant d'autoriser un accès direct aux ressources matérielles par des tâches de l'invité.

Exemple d'accès direct. Prenons l'exemple de l'usage d'un système invité multitâches, comportant une tâche dont le rôle est d'émettre périodiquement des rapports au moyen d'un bus RS232. Admettons également que cette dernière envoie des caractères sur ce bus en accédant directement au module de communication série (UART) pour réduire les délais de transmission entre les octets transmis.

Supposons maintenant qu'un périphérique soit accédé sporadiquement à l'aide d'un bus connecté au même UART (un module de mémoire Flash par exemple). Puisqu'il y a dans ce cas des possibilités d'accès concurrents à l'UART à la fois par les émissions RS232 et le pilote du module mémoire, alors une propriété d'exclusion mutuelle doit être assurée au sein de la couche de contrôle. Ainsi, tous les accès au bus doivent être interceptés et contrôlés.

Précision du problème. Avec notre solution, et en première approche, envoyer une séquence de caractères nécessiterait au minimum une exécution du tick, pour laquelle le temps d'exécution dépendrait fortement de la quantité d'octets à transmettre. Outre la nécessité d'utiliser des tampons parfois coûteux en espace mémoire, cette technique induit donc des délais de latence potentiellement conséquents pour la répercussion vers l'invité des requêtes matérielles survenant entre-temps.

Cependant, nous allons voir qu'il est en tout à fait envisageable d'autoriser un accès direct à une ressource matérielle par une tâche de l'invité sans nécessairement empêcher son contrôle par la couche de contrôle.

Principe de traitement « coercitif ». En effet, de tels accès directs sont possibles en assimilant la tâche concernée à un périphérique additionnel contrôlé au niveau de la couche de contrôle. Il est alors possible de modéliser son comportement à l'aide d'un automate à au moins deux états (utilisant le bus, ou non) et des transitions contrôlables dont certaines sorties sont transmises au gestionnaire des tâches de l'invité. Cet automate est ensuite ajouté à l'ensemble des automates pour lequel un contrôleur est conçu.

Ce principe de traitement des accès directs est schématisé dans la figure 7.3 page suivante. Par exemple, lorsque le contrôleur force la prise d'une transition entre les états « T n'accède pas au bus » et « T utilise

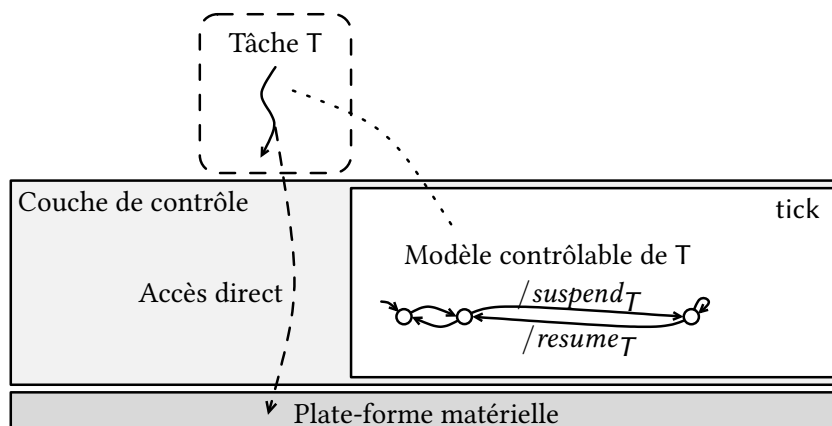


FIGURE 7.3 – Principe de contrôle des accès directs aux ressources par une tâche de l’invité. Les comptes-rendus $resume_T$ et $suspend_T$ représentent des « ordres » transmis à l’invité au moyen d’interruptions virtuelles pour que son gestionnaire de tâches modifie l’état effectif de T .

potentiellement le bus », le compte-rendu $resume_T$ est émis par le tick puis interprété par l’invité au moyen d’un traitant d’interruption virtuelle associé, qui replace alors T dans la file des tâches éligibles (ou tout autre état précédant sa suspension par la couche de contrôle).

Ce mécanisme permet le maintien de l’invariant suivant : tant que la tâche est potentiellement éligible au niveau du gestionnaire des tâches de l’invité, son modèle dans la couche de contrôle est dans l’état « T utilise potentiellement le bus » et les autres utilisateurs du bus n’y ont pas accès.

Principe de traitement « permissif ». La modélisation des tâches de l’invité peut être également beaucoup plus fine et prendre en compte les différentes émissions de requêtes logicielles, voire des émissions dédiées par la tâche concernée : il est concevable de construire cette tâche de manière à encadrer les séries d’accès directs par des demandes d’autorisation au moyen de requêtes dédiées. Le modèle de la tâche ne serait plus directement dirigé par le contrôleur, mais plutôt par la tâche elle-même ; ses transitions seraient explicitement *contraintes* plutôt que *forcées*.

Nous ferons cependant remarquer que l’ensemble de ces modélisations se font au détriment de la complexité du contrôleur.

5.2.2 Choix « éclairé » du mode de basse consommation

Notre approche peut également être employée pour déterminer un mode de basse consommation « optimal » à partir de la connaissance de l’état de chaque périphérique de la plate-forme, tout en imposant l’existence d’une possibilité de réveil. En effet, il est important de s’assurer que le microcontrôleur n’entre pas dans un mode de basse consommation duquel il ne peut sortir que par des requêtes d’interruption dont les sources sont des périphériques qui ne sont pas dans un mode de fonctionnement susceptible de les émettre. En outre, une contrainte concernant le temps pris par cette phase de réveil, qui varie selon le mode choisi, peut aussi être à prendre en compte lorsqu’un périphérique attend une réponse rapide de la part du microcontrôleur.

Assurer que le microcontrôleur est toujours dans le meilleur mode de basse consommation lorsqu’il n’est pas actif est un problème qui n’est solutionné qu’avec des mécanismes *ad hoc* dans les approches évoquées au chapitre 3, mais correspond à un problème usuel de synthèse de contrôleur.

Formulation sous forme d’objectif de contrôle. En effet, déterminer le mode de fonctionnement du microcontrôleur le moins gourmand en énergie tout en tenant compte des possibilités de réveil du CPU ou

des contraintes de temps de réveil est une instance d'un problème de contrôle global. De plus, l'information relative aux interruptions potentiellement émises par chaque périphérique peut être retrouvée à partir de son mode de fonctionnement, c'est-à-dire de l'état de l'automate le modélisant.

Ainsi, en intégrant un modèle du microcontrôleur au sein de la couche de contrôle sous la forme d'un automate dans lequel les changements de mode sont représentés à l'aide de transitions contrôlables, il est possible de formuler ce problème sous la forme d'une interdiction d'états globaux (pour éviter les réveils impossibles) combinée à un objectif qualitatif (*i.e.*, minimisation de la consommation des états du microcontrôleur). Une sortie supplémentaire du tick indique alors le meilleur mode de basse consommation atteignable à chaque instant.

5.2.3 Émission simultanée de plusieurs requêtes logicielles

Une dernière extension de la couche de contrôle que nous évoquerons, relative à la construction de la membrane celle-ci, consiste en la possibilité d'émission par l'invité d'un ensemble de requêtes logicielles de manière simultanée. Leur prise en compte retardée sera également envisagée.

La première transformation est relativement aisée à réaliser au niveau de la couche de contrôle, puisqu'il suffit de transformer la référence vers une requête logicielle (`sw_cell`) en un ensemble de requêtes (ou plutôt, en une unique requête codant l'émission simultanée de plusieurs signaux d'entrée du noyau réactif). L'implantation d'un module de gestion d'ensembles de signaux est en outre déjà nécessaire pour traiter les comptes-rendus résultant de l'exécution du tick.

Cette modification permet par exemple la délégation à la couche de contrôle du choix de l'ordre de réalisation d'actions considérées alors comme concurrentes par l'invité. Des requêtes logicielles ne nécessitant pas une prise en compte immédiate pourraient aussi être accumulées afin d'être effectivement utilisées par la couche de contrôle lors de la réception d'une requête matérielle ou d'une requête logicielle immédiate.

Après avoir présenté des évaluations quantitatives de la proposition détaillée au chapitre 5, ainsi que quelques usages et extensions, nous allons maintenant nous attarder sur les solutions à deux problèmes importants qui restent à aborder pour être complet sur les aspects qualitatifs : il s'agit d'une part du traitement des interblocages potentiels survenant lors de l'utilisation de politiques de gestion de ressources contraignantes, d'autre part des possibilités et limites actuelles de l'usage des outils de synthèse de contrôleur automatisés.

6 Considérations sur les interblocages

Un problème important qui peut survenir lors de l'usage d'une politique de contrôle global est relatif aux interblocages. En effet, l'imposition d'une politique de contrôle empêchant volontairement la progression dans l'usage de ressources peut faire émerger des situations d'interblocage potentiel.

6.1 Exemples de situations d'interblocage potentiel

Nous prenons dans cette section deux cas d'exemples pour illustrer les deux origines de ce problème. Dans chacun des cas, nous considérons les automates incorporés à la couche de contrôle décrits dans la figure 7.4 page suivante, ainsi que les deux extraits de code des tâches T_a et T_b suivants :



FIGURE 7.4 – Deux extraits d’automates de pilotes incorporables à la couche de contrôle.

```

Ta:
...
R = ∅;
while (acka ∉ R) {
...
R = on_sw (a);
...
}
►...

Tb:
...
R = ∅;
while (ackb ∉ R) {
...
R = on_sw (b);
...
}
►...
    
```

On suppose ici que seule la tâche T_a de l’invité émet les requêtes concernant directement U_a (c’est-à-dire a), de même que T_b pour U_b . Si l’automate U_a est dans l’état A , alors T_a ne continue son exécution avec les instructions qui suivent la boucle d’émission de la requête a qu’une fois que cette dernière a bien été prise en compte par la couche de contrôle, c’est-à-dire que la transition vers l’état A' a bien été prise; *idem* concernant T_b avec la ressource U_b . Enfin, nous admettons que U_a et U_b sont respectivement dans les états A et B initialement.

6.1.1 Blocage au niveau du gestionnaire des tâches de l’invité

Une première situation d’interblocage peut survenir s’il existe une propriété d’exclusion mutuelle imposée entre les états A' et B' , pour éviter des pics de consommation d’énergie par exemple.

L’interblocage advient lorsque T_a et T_b effectuent un rendez-vous à l’aide de primitives de synchronisation classiques après leur boucle d’émission de requête logicielle (points notés « ► » dans les extraits de code des tâches). En effet, chacune ne peut atteindre son point de rendez-vous qu’une fois que les deux automates ont atteint un état global pourtant interdit par la propriété d’exclusion mutuelle sur A' et B' .

6.1.2 Blocage au niveau des ressources gérées par la couche de contrôle

Une seconde situation d’interblocage, qui n’est cette fois pas liée aux synchronisations entre les tâches de l’invité, est possible si les états globaux $A \times B'$ et $A' \times B$ sont rendus inaccessibles par le contrôleur (par exemple, il pourrait s’agir pour le premier état global interdit d’éviter des pics de consommation, pour le second d’une exclusion mutuelle pour l’accès à un lien de communication). En outre, l’unique possibilité de sortir de l’état $A \times B$ est dans ce cas que l’invité émette les requêtes a et b simultanément — ce qui est rendu impossible par construction de la couche de contrôle et la répartition des rôles des tâches de l’exemple.

6.2 Possibilités de détection hors ligne

Les exemples exposés précédemment illustrent deux catégories d’interblocages qui peuvent apparaître par l’usage d’une couche de contrôle telle que nous l’avons décrite. Les causes de chacun d’eux sont fondamentalement différentes : le premier exemple exhibe une incompatibilité de l’invité par rapport aux

propriétés globales garanties par la couche de contrôle, alors que le second révèle un problème éventuel dans la construction de cette dernière.

Les remèdes et possibilités de détection diffèrent suivant les cas.

6.2.1 Sur les interblocages au niveau de l'invité

Interblocages intrinsèques. Pour une partie des interblocages potentiels au niveau de l'invité, nous affirmons qu'ils ne sont pas liés à l'usage de la couche de contrôle, mais plutôt à une incompatibilité entre la *logique applicative* et les propriétés globales imposées.

Nous avançons en effet que l'imposition d'une politique de contrôle global similaire à celles que la couche de contrôle peut assurer, si elle est réalisée à l'aide de primitives de synchronisation comme des verrous ou des moniteurs, peut également mener à l'apparition d'interblocages si les propriétés imposées ne sont pas compatibles avec la manière dont le logiciel utilise la plate-forme ; c'est-à-dire qu'elles restreignent trop l'ensemble des comportements possibles. Ces interblocages potentiels sont *intrinsèques* à la combinaison de la logique applicative et des propriétés que l'on cherche à assurer : toute architecture logicielle de fonctionnalité équivalente et sans interblocage potentiel violerait certaines de ces propriétés, quelle qu'elle soit.

Par exemple, l'usage de mécanismes d'arbitrage et de synchronisation bloquants comme proposés par Klues *et al.* [100] dans le cadre de la méthode de conception des pilotes de périphériques ICEM (*cf.* § 5.1 page 45) sont aussi concernés par ces limitations. En effet, et pour reprendre la situation de l'exemple de la section 6.1.1 page ci-contre, l'exclusivité entre les modes de fonctionnement représentés par les états A' et B' des automates devrait alors être assurée à l'aide de primitives de synchronisation empêchant la progression de l'une des tâches de gestion des ressources modélisées par U_a ou U_b , et le rendez-vous suivant ne serait pas réalisé.

Autres interblocages. L'autre partie des interblocages potentiels provient des erreurs usuellement commises lors du développement avec des primitives de synchronisation bloquantes. Il est de plus possible qu'ils soient similaires à la situation prise en exemple dans la section 6.1.1, en faisant intervenir des contraintes d'usage des ressources gérées par la couche de contrôle.

(Bien que Zheng *et al.* [163] aient récemment proposé des méthodes pour leur détection statique, l'élimination des situations d'interblocage potentiel dues à un usage incorrect de primitives de synchronisation reste un problème difficile et implique souvent un travail conséquent de réorganisation des différentes tâches du système.)

Détection statique. Dans tous les cas, nous pouvons considérer que l'extension présentée dans la section 5.2.1 page 103, où les comportements des tâches de l'invité sont modélisés comme des ressources gérées par la couche de contrôle, mène vers une méthode de détection de ces interblocages. En effet, Wang *et al.* [156] montrent que ces situations, comme toute propriété qui ne peut être imposée sur un ensemble de comportements trop peu contrôlables, se révèlent par une synthèse de contrôleur infructueuse. Ainsi, un échec de la synthèse dans ce cadre pourrait avoir deux causes :

- les propriétés à assurer sont trop restrictives par rapport aux automates des pilotes ;
- il existe une situation d'interblocage potentiel.

6.2.2 Traitement du problème de progression au niveau des ressources

Considérons maintenant le second exemple, pour lequel aucune action sur l'invité autre que l'implantation de l'extension évoquée dans la section 5.2.3 page 105 n'apporte de solution. En effet, l'émission simultanée de plusieurs requêtes logicielles est possiblement un moyen de résoudre ce type d'interblocage ; cependant

dans ce cas, il est nécessaire de transformer l'architecture des tâches de l'invité pour pouvoir émettre les requêtes simultanément.

Connaissance de l'exclusivité des requêtes logicielles. En conservant l'exclusivité des émissions de requêtes logicielles, il est possible d'éliminer une partie des états globaux problématiques en spécifiant cette contrainte imposée par la construction de la couche de contrôle lors de la conception du contrôleur.

Pour notre exemple, l'état global $A \times B$ devient un puits : dans le produit des automates U_a et U_b moins les états interdits par le contrôleur pour cause de violation des propriétés d'exclusions mutuelles, l'état $A' \times B'$ n'est plus atteignable si l'on admet que a et b sont exclusifs (*i.e.*, l'assertion $\overline{a \cdot b}$ est toujours vérifiée).

Objectif de progression. Assurer ensuite une propriété de *progress* exprimant, pour simplifier, qu'il doit toujours exister une exécution possible menant à certains états de chaque ressource, rend alors inaccessible tout état global ne menant pas vers des états spécifiés comme toujours atteignables.

Appliquée à l'exemple, une contrainte supplémentaire de progression rend l'état global $A \times B$ inaccessible puisque les seules transitions sortantes de cet état bouclent vers lui-même. Si A et B sont les états initiaux des automates U_a et U_b , alors la synthèse échoue.

Garantir une telle propriété de progrès complexifie fortement la conception du contrôleur, mais nous verrons dans la section 7 qu'il existe des méthodes d'automatisation de cette tâche.

6.3 Conclusion sur les interblocages

Pour conclure à propos des interblocages, nous mettrons en avant que, dans le cadre de notre approche, il est possible d'en détecter une partie en intégrant des modèles des tâches lors de la synthèse du contrôleur. De plus, nous avançons qu'une partie des interblocages potentiels surviennent à cause d'une incompatibilité entre les propriétés assurées par la couche de contrôle et la logique applicative de l'invité ; toute architecture compatible avec la logique applicative doit être nécessairement plus « libérale » en matière de propriétés à imposer. Enfin, il est nécessaire de bien spécifier le cadre d'usage des ressources du tick afin d'éviter tout problème de blocage à ce niveau.

7 Synthèse de contrôleur automatisée

Dans la présentation de l'approche et de son évaluation, et jusqu'à la section précédente, nous proposons de concevoir les contrôleurs manuellement sans réellement aborder l'automatisation de cette construction. Nous ne prenons de plus en exemple que des propriétés de sûreté comme des exclusions mutuelles d'accès à des ressources.

Cependant, cette tâche s'avère souvent complexe à réaliser et devient de moins en moins envisageable au fur et à mesure que le nombre de ressources et de propriétés augmente. La prise en compte d'un nombre même relativement restreint de contraintes peut également s'avérer hors de portée, et l'assistance d'outils de vérification de propriétés permettant de s'assurer que le contrôleur remplit bien sa tâche (entre autres) n'est pas d'un grand secours dans ces cas ; par exemple Pace *et al.* [123] proposent une méthode de génération de contre-exemples qui permettent alors d'identifier la portion du contrôleur (un état) qui doit être modifiée, mais ce type de processus est fastidieux³. Dans de tels cas, une solution réside dans l'usage d'un outil permettant l'automatisation de la synthèse du contrôleur global, ou d'une partie de celui-ci.

Nous avons déjà présenté le principe général de la synthèse de contrôleur dans la section 2 page 56 du chapitre 4.

3. Et on remarquera qu'il s'agit en vérité de l'application manuelle du principe de base des algorithmes de synthèse de contrôleur évoqués dans la section 2 page 56.

7.1 Sur le non déterminisme des contrôleurs produits

7.1.1 Origine du non déterminisme

Un outil de synthèse de contrôleur automatisée produit le contrôleur *le plus permissif*, c'est-à-dire le contrôleur qui ne fait qu'*interdire* les comportements spécifiés comme invalides. En conséquence, celui-ci est (le plus souvent) non déterministe.

Par exemple, garantir une propriété d'exclusion mutuelle sur l'usage d'une ressource implique que le contrôleur *choisisse* d'accorder son usage à l'un des utilisateurs potentiels lorsque plusieurs la requièrent simultanément, auquel cas le contrôleur est non déterministe si ce choix n'est pas explicitement exprimé par une propriété d'*équité* à assurer.

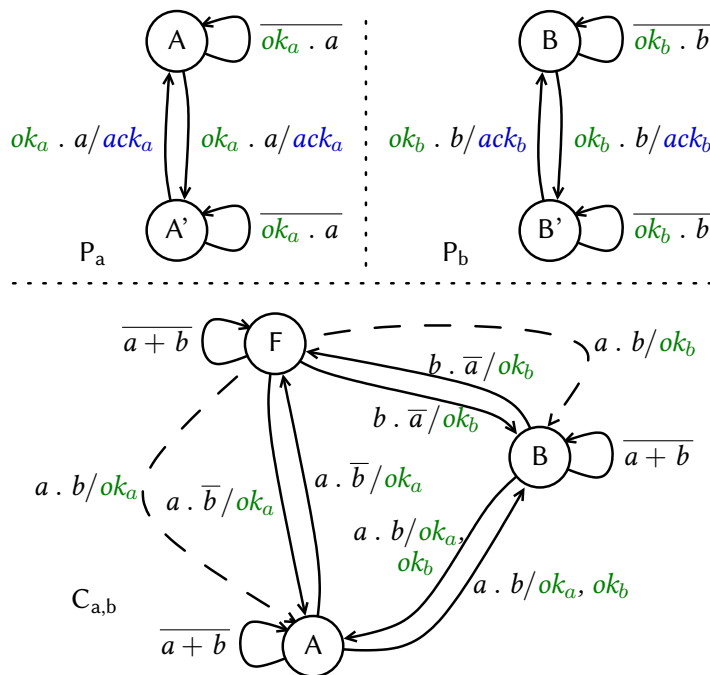


FIGURE 7.5 – Deux exemples d'automates de pilotes P_a et P_b , ainsi que le contrôleur le plus permissif $C_{a,b}$ interdisant l'état global $A' \times B'$. Ce contrôleur est non déterministe en son état F si $a \cdot b$ est « vrai ».

La figure 7.5 illustre une situation d'exemple où le contrôleur le plus permissif $C_{a,b}$ produit pour assurer une propriété d'exclusion mutuelle entre les états A' et B' de leurs automates respectifs. Le non déterminisme de l'état F du contrôleur représente en vérité le choix d'un utilisateur prioritaire parmi P_a et P_b , lorsque les deux requêtes a et b surviennent au même instant.

Une conséquence de l'usage d'un outil de synthèse de contrôleur dans un cadre d'implantation est donc la nécessité d'identifier et d'éliminer le non déterminisme éventuel du contrôleur obtenu.

7.1.2 Traitements possibles

Parmi les solutions à ce problème, une première consiste à ne garder statiquement qu'un seul choix lorsque plusieurs sont autorisés par le contrôleur. Cependant, cela génère un système potentiellement non équitable. Dans l'exemple précédent, cette méthode entraînerait le choix permanent du même demandeur par le contrôleur résultant lors de l'occurrence de requêtes conflictuelles.

(On remarquera par ailleurs que le contrôleur conçu manuellement et représenté dans la figure 5.8 page 77 – § 6.2, chap. 5, est déterministe à cause d'une sur-spécification volontaire de la propriété assurée, qui est initialement présentée comme une exclusion mutuelle simple. La sur-spécification implicite réside dans une

priorité donnée en permanence à l'un des utilisateurs de la ressource. Ces priorités y sont encodées dans les formules Booléennes des transitions ; e.g., dans l'état Free, les requêtes destinées au transmetteur radio ont toujours priorité sur celles du convertisseur analogique-digital. Dans ce cas, le choix inverse était également envisageable, tout comme le codage d'une priorité tournante à l'aide d'états supplémentaires — auquel cas un autre choix implicite du contrôleur résiderait dans la détermination du premier à avoir la priorité.)

Une autre possibilité consiste à tirer au sort à l'exécution, parmi l'ensemble des choix autorisés.

Utilisation d'oracles. Une troisième méthode que nous détaillerons ici consiste en l'extraction du non déterminisme par l'introduction d'*oracles* au contrôleur : un choix non déterminé entre deux transitions t_1 et t_2 est alors encodé par l'ajout de i sur t_1 et $\neg i$ sur t_2 , i étant une entrée dite « *oracle* » permettant d'extraire (une partie de) l'indéterminisme de l'état source de t_1 et t_2 en le rendant explicite. La figure 7.6 représente le contrôleur $C_{a,b}$ de la figure 7.5 page précédente dans lequel le non déterminisme a été éliminé par ce moyen.

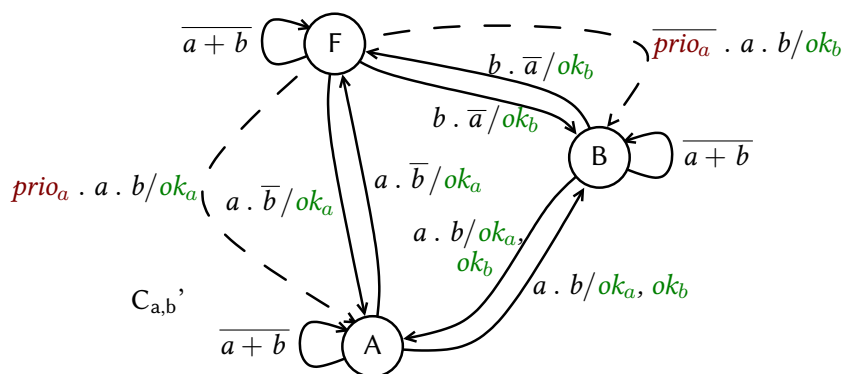


FIGURE 7.6 – Le contrôleur $C_{a,b}$ de la figure 7.5 page précédente, rendu déterministe par l'introduction de l'oracle $prio_a$; $prio_a$ « vrai » signifie que l'utilisateur P_a a priorité sur P_b dans le cas où les deux requêtes arrivent concurremment.

Une fois l'introduction des oracles réalisée, leurs valeurs respectives peut être générée par un ordonnanceur additionnel, possiblement construit de manière à respecter des contraintes d'équité. Ce dernier peut être un ensemble d'automates placés en parallèle avec le contrôleur et les automates des pilotes, ou tout autre mécanisme conçu d'une manière plus appropriée à la gestion de files d'attentes.

Nous évoquerons une dernière méthode d'élimination du non déterminisme du contrôleur assez similaire à l'ajout d'oracles dans la section 7.2.1.

7.2 Le cas de BZR/SIGALI

Nous avons décrit succinctement la chaîne d'outils BZR/SIGALI dans la section 2.3.3 page 59 du chapitre 4. Nous avons également déjà évoqué dans cette même section le travail de Bouhadiba *et al.* [22], qui utilisèrent BZR pour générer du code de niveau système.

Nous considérons ici BZR en tant qu'unique outil de synthèse à notre connaissance, dans la communauté travaillant sur les langages synchrones et ses usages, qui permet la génération de code exécutable à partir des contrôleurs.

7.2.1 Le traitement du non déterminisme dans BZR/SIGALI

L'approche proposée par les concepteurs de BZR pour l'élimination du non déterminisme est relativement proche de l'introduction d'oracles. Elle consiste en l'ajout de *variables fantômes*, qui sont des entrées

supplémentaires du système que l'on peut choisir de fixer ou de faire varier à l'implantation. À chaque entrée contrôlable du système considéré (*i.e.*, sortie du contrôleur) est associée une telle variable dont la valeur à chaque instant est celle *souhaitée* pour l'entrée correspondante. Des priorités statiques distinctes sont également associées aux entrées contrôlables à partir de critères syntaxiques. Lorsqu'un choix sur la valeur d'un ensemble de ces entrées est indéterminé, le contrôleur est généré pour assigner à chacune la valeur de sa variable fantôme, en commençant par la plus prioritaire.

Une critique que nous faisons concernant cette technique d'élimination du non déterminisme du contrôleur est son aspect « boîte noire » : il est difficile d'évaluer l'influence d'un choix des priorités des entrées contrôlables, ou encore de l'impact effectif des différentes valeurs des variables fantômes.

7.2.2 Expérience d'implantation

Au moment de la réalisation de ce travail, BZR produit le contrôleur sous la forme d'une fonction C qui est appelée par la routine résultant de la compilation en C du système contrôlé (pour nous, il s'agirait de la méthode `run_step()` du noyau réactif). Pour un nombre raisonnable d'automates à contrôler et en spécifiant une propriété d'équité explicitement dans les contrats afin de réduire l'impact du choix des valeurs des variables fantômes, la fonction produite est souvent de taille assez conséquente et son implantation sur un nœud de réseaux de capteurs sans fil n'est pas raisonnablement envisageable.

Nous ferons remarquer ici que la relation entre la taille des spécifications des composants et contrats, et celle du code produit (de même que le coût calculatoire de la synthèse), n'est pas une fonction monotone en raison des manipulations complexes de diagrammes de décision [23] sous-jacentes. Cependant, la relative opacité des algorithmes implantés dans SIGALI ne nous permettent pas de déterminer les moyens d'action à notre disposition, s'il y en a, pour réduire la taille du code potentiellement atteignable.

Les résultats de ces quelques expérimentations d'usage de BZR nous permettent en revanche de mettre en relief des caractéristiques que nous jugeons essentielles dans le cadre de l'automatisation (même partielle) de la synthèse du contrôleur de la couche de contrôle.

7.3 Vers une chaîne d'outils idéale

La figure 7.7 page suivante schématise la chaîne d'outils idéale que nous envisageons pour notre cas d'utilisation de la synthèse de contrôleur automatisée. Elle consisterait à produire un contrôleur sous la forme d'un programme synchrone avec oracles, afin qu'il soit ensuite intégré au système complet avant que ce dernier ne soit transformé en code C.

Simplification des équations. Le passage par un format intermédiaire en équations autorise, lors de la mise en parallèle du contrôleur avec les automates des pilotes, à profiter d'éventuelles simplifications possibles lors du calcul de la fonction de transition. En effet, quelques expériences préliminaires consistant en l'implantation du contrôleur produit par SIGALI sous la forme d'un programme LUSTRE intégré au système complet tel quel, indiquent qu'il n'est pas déraisonnable de songer à profiter des simplifications des équations réalisables après l'expansion du programme en un unique ensemble d'équations Booléennes (comprenant donc les automates des pilotes plus le contrôleur).

Signification et assignation des valeurs des oracles : aspect semi-automatique du processus de synthèse. Il semblerait cependant qu'une solution entièrement automatique pour l'attribution des valeurs des oracles relève pour le moment plus de science fiction que d'une proposition réaliste. C'est pourquoi, si elles ne sont pas assignées dynamiquement à l'aide d'un générateur pseudo-aléatoire, il est important d'identifier la signification de chacune en matière de transitions entre les états globaux. Pour cela, il paraît nécessaire de permettre la représentation du contrôleur sous la forme d'un automate, ou encore d'obtenir

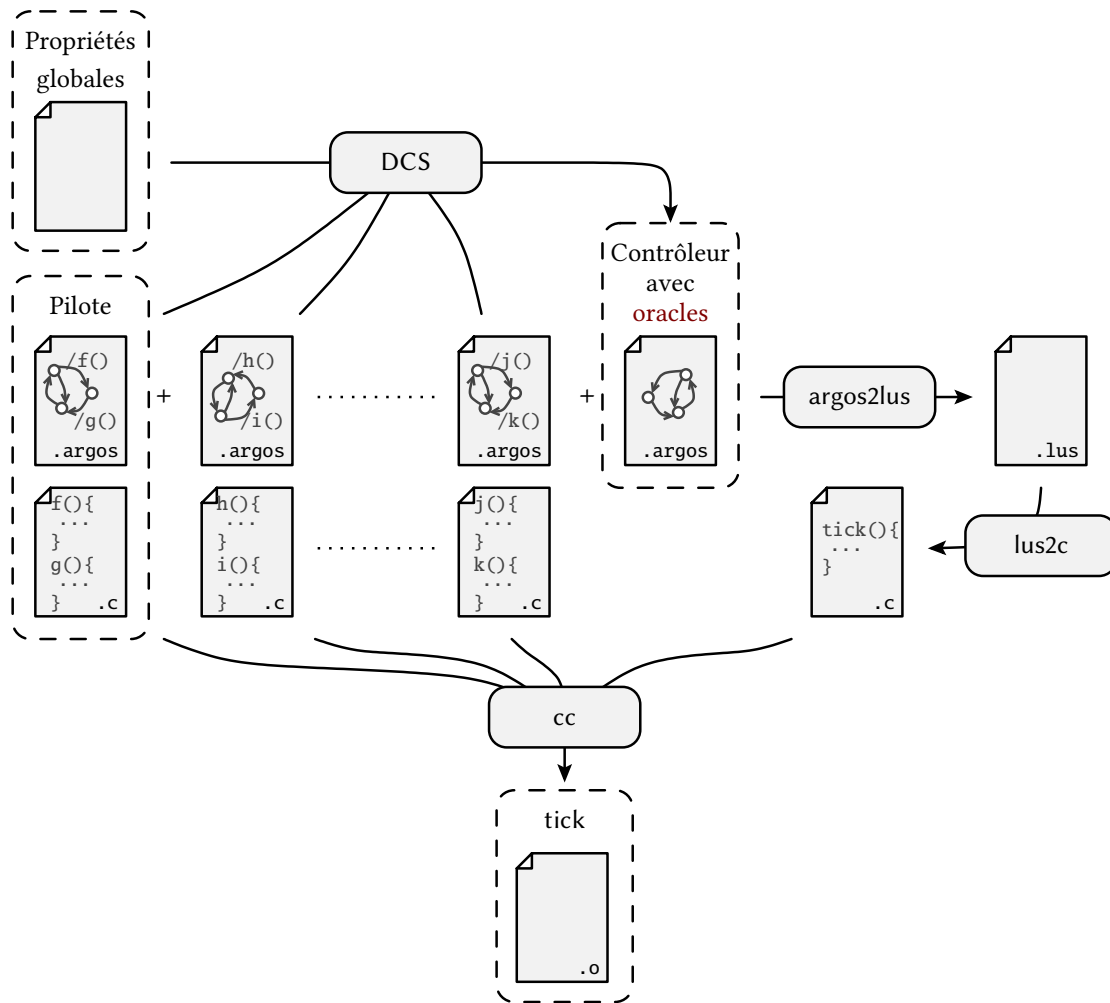


FIGURE 7.7 – Chaîne d’outils idéale incorporant un outil de synthèse de contrôleur discret (DCS). Cet outil est utilisé pour générer (semi-)automatiquement un contrôleur global à partir des automates des pilotes et autres ressources à gérer, et d’une représentation des propriétés à assurer. Ces dernières devraient pouvoir être exprimées directement à partir des états et éventuellement des transitions des automates à contrôler.

des informations relatives aux états globaux dont les transitions font intervenir des oracles. Dans ce cadre, la problématique du retour à la source paraît primordiale.

Dans notre contexte enfin, et si une signification claire est donnée à chaque oracle, il est même envisageable d’en conserver quelques-uns en les exposant à l’invité via une interface de configuration pour que ce dernier puisse choisir, dynamiquement ou non, un comportement particulier parmi plusieurs autorisés.

8 Conclusion du chapitre

Nous avons détaillé dans ce chapitre les travaux d’implantation d’une preuve du concept présenté au chapitre 5. Ce prototype a été réalisé dans l’optique de fournir quelques évaluations quantitatives de la couche de contrôle, pour en évaluer les caractéristiques et la faisabilité.

Nous avons également exposé une étude de cas afin d’estimer l’impact de l’adaptation nécessaire pour exécuter un système invité conjointement avec la couche de contrôle. Elle nous a permis d’avancer quelques

conseils pour réaliser cette tâche, tout en en révélant quelques limites, pour les choix à effectuer lors des traitements des requêtes refusées notamment.

Plus tard, une évaluation qualitative du concept a été présentée. Nous avons identifié que le cadre synchrone et la distinction entre les parties contrôle et gestion de données utiles des pilotes de périphériques de la couche de contrôle rend leur développement plus rigoureux et permet d'extraire un modèle explicite des comportements possibles du périphérique. En outre, des extensions de l'approche destinées à traiter des cas de contrôle complexes ou requérant parfois des solutions *ad hoc* selon d'autres méthodes d'implantation ont été présentées.

Suivit alors une discussion à propos des interblocages potentiels où nous avons identifié deux aspects. D'une part, une partie de ceux-ci est liée à une incompatibilité entre la logique applicative et les propriétés assurées par la couche de contrôle ; une méthode d'implantation sans interblocages mènerait nécessairement à une violation d'au moins une de ces propriétés globales. D'autre part, l'intégration dans le noyau réactif de modèles des tâches de l'invité, mise en perspective avec les résultats de Wang *et al.* [156], ouvrent la voie vers une méthode de détection hors ligne de ces interblocages potentiels.

Enfin, nous avons détaillé des pistes vers une chaîne d'outils que nous jugeons idéale, permettant la synthèse semi-automatisée du contrôleur de la couche de contrôle à partir des propriétés et des automates des pilotes. Nous y avons plus particulièrement discuté de l'élimination du non-déterminisme du contrôleur produit, ainsi que de quelques problèmes relatifs à la compilation modulaire que nous identifiâmes à partir d'expérimentations avec l'outil de synthèse BZR.

À partir du chapitre suivant, nous tirerons des conclusions plus génériques des expériences réalisées dans le cadre des réseaux de capteurs sans fil, notamment en considérant les impacts éventuels de notre proposition sur la classes des systèmes embarqués en général.

CONCLUSION

Contenu du chapitre

| | | |
|---|------------------------|-----|
| 1 | Bilan | 115 |
| 2 | Perspectives | 115 |

1 Bilan

Nous avons proposé dans cette thèse une architecture logicielle visant à permettre le contrôle globale de ressources au niveau de plates-formes matérielles. Les avantages de cette approches sont :

- Une distinction claire des objectifs de contrôle, notamment en séparant l’implantation des parties contrôles des pilotes et en utilisant une solution basée sur la synthèse de contrôleur autorisant leur développement de manière modulaire ;
- L’utilisation d’un langage synchrone au cœur de la couche de contrôle, qui permet l’implantation efficace des pilotes et du contrôleur ;
- La diversité des propriétés globales qui peuvent être imposées au systèmes invité, ainsi que la variété des extensions envisageables.

Notre méthode requiert une adaptation des codes d’applications existantes, mais ne modifie pas la manière dont elles sont conçues et programmée. Nous avons également illustré et quantifié les modifications requises pour adapter une portion représentative de code de niveau système, et montré par cette expérience que la méthode utilisée est suffisamment générale pour être applicable à un grand nombre de modules logiciels existants.

Nous avons évoqué dans ce même chapitre une solution pour une meilleure intégration des éventuelles tâches de l’application avec la couche de contrôle, spécialement dans une optique de détection hors ligne des situations d’interblocages. Cependant, la mise en œuvre d’une telle méthode complexifie considérablement la création manuelle du contrôleur global : il est notablement fastidieux de déterminer l’absence de solution d’un problème de synthèse.

Enfin, nous avons abordé quelques pistes vers l’automatisation de la synthèse du contrôleur global.

2 Perspectives

Réservation de ressources. La possibilité de réservation de ressources entre également dans le cadre de la synthèse de contrôleurs, mais demanderait d’écrire les automates des pilotes d’une manière adaptée. L’expression des propriétés globales en serait également plus complexe. Par surcroît, l’impact sur la programmation des applications serait relativement important.

Synthèse automatisée. La perspective principale que nous identifions consiste en l'investigation des limites des outils disponibles pour l'automatisation de la synthèse du contrôleur global. Les derniers développements dans le domaine de la synthèse de contrôleur dans le cadre synchrone nous paraissent encourageant.

Synthèse de pilotes. Nous pensons également qu'il serait intéressant d'étendre les techniques de synthèse de pilotes de périphériques évoqués dans la section 5.4 page 24 du chapitre 2, pour générer les pilotes contrôlables dont nous proposons de faire usage.

Programmation synchrone des applications. Une autre perspective qui nous semble intéressante est l'emploi d'une infrastructure similaire à la couche de contrôle pour la programmation synchrone des parties contrôles des tâches de l'application complète, comprenant également les services de niveau système. Notamment, l'écriture sous forme événementielle des automates, d'usage dans beaucoup de systèmes existants, pourrait être aisément remplacée par l'expression des mêmes automates dans un langage proche de celui que nous avons utilisé dans cette thèse. Par ce moyen, le parallélisme de description serait entièrement compilé, tout en représentant un modèle formel de l'application complète. Ce dernier autoriserait alors l'usage de l'ensemble des techniques d'analyses statiques disponibles pour les langages synchrones.

Des possibilités sont également offertes pour la simulation efficace de réseaux complets, à partir des modèles simples mais fidèles que sont les automates des pilotes.

Extension au contrôle de systèmes à micro-noyau. Dans un papier court présenté lors d'une session « *Fun Ideas and Thoughts* » d'une conférence internationale, nous avons avancé quelques idées vers une généralisation de l'approche que nous proposons à des systèmes de plus grande taille [15].

Nous y faisons le constat que les architectures monolithiques des noyaux de systèmes apportent certes une plus grande flexibilité dans leur programmation, mais présentent cependant des risques significatifs en matière de sûreté et de sécurité. Par ailleurs, nous faisons remarquer que, même dans ces cas les pilotes sont toujours programmés de manière individuelle et qu'il est difficile de retirer une connaissance globale de l'état de la plate-forme sous-jacente.

Nous y relevons enfin que le principe de contrôle de notre approche peut être étendu à certaines architectures de systèmes d'exploitation, notamment les systèmes à base de micro-noyau.

BIBLIOGRAPHIE

- [1] Karine ALTISEN, Aurélie CLODIC, Florence MARANINCHI et Éric RUTTEN : « Using controller-synthesis techniques to build property-enforcing layers ». *In Proceedings of the 12th European Conference on Programming, ESOP'03*, pages 174–188, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-00886-1. (cité page 58)
- [2] Karine ALTISEN, Florence MARANINCHI et David STAUCH : « Aspect-oriented programming for reactive systems: Larissa, a proposal in the synchronous framework ». *Sci. Comput. Program.*, 63:297–320, décembre 2006. ISSN 0167-6423. (cité page 86)
- [3] Ibrahim AMADOU, Guillaume CHELIUS et Fabrice VALOIS : « Energy-Efficient Beacon-less Protocol for WSN ». *In Proceedings of the 22nd IEEE Symposium on Personal, Indoor, Mobile and Radio Communications, PIMRC '11*, Washington, DC, USA, novembre 2011. IEEE Computer Society. (cité page 17)
- [4] Pascalin AMAGBÉGNON, Loïc BESNARD et Paul LE GUERNIC : « Implementation of the data-flow synchronous language SIGNAL ». *In Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, PLDI '95*, pages 163–173, New York, NY, USA, 1995. ACM. ISBN 0-89791-697-2. (cité page 51)
- [5] Charles ANDRÉ : « Representation and Analysis of Reactive Behaviors: A Synchronous Approach ». *In Computational Engineering in Systems Applications, CESA '96*, pages 19–29. IEEE-SMC, juillet 1996. (cité pages 51 et 82)
- [6] Rahul BALANI, Chih-Chieh HAN, Ram Kumar RENGASWAMY, Ilias TSIGKOIANNIS et Mani SRIVASTAVA : « Multi-level software reconfiguration for sensor networks ». *In Proceedings of the 6th ACM & IEEE International Conference on Embedded Software, EMSOFT '06*, pages 112–121, New York, NY, USA, 2006. ACM. ISBN 1-59593-542-8. (cité page 45)
- [7] Thomas BALL, Ella BOUNIMOVA, Byron COOK, Vladimir LEVIN, Jakob LICHTENBERG, Con MCGARVEY, Bohus ONDRUSEK, Sriram K. RAJAMANI et Abdullah USTUNER : « Thorough static analysis of device drivers ». *In Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006, EuroSys '06*, pages 73–85, New York, NY, USA, 2006. ACM. ISBN 1-59593-322-0. (cité page 24)
- [8] Andrea BALLUCHI, Luca BENVENUTI, Howard WONG-TOI, Tiziano VILLA et Alberto L. SANGIOVANNI-VINCENTELLI : « A Case Study of Hybrid Controller Synthesis of a Heating System ». *In Proceedings of the 5th European Control Conference, ECC '99*, septembre 1999. (cité page 57)
- [9] Robert W. BEMER : « How to consider a computer ». *Automatic Control Magazine*, pages 66–69, 1957. (cité page 35)

- [10] Luca BENINI, Giuliano CASTELLI, Alberto MACII et Riccardo SCARSI : « Battery-Driven Dynamic Power Management ». *IEEE Des. Test*, 18:53–60, mars 2001. ISSN 0740-7475. (cité page 20)
- [11] Albert BENVENISTE, Patricia BOURNAI, Thierry GAUTIER, Michel LE BORGNE, Paul LE GUERNIC et Hervé MARCHAND : « The SIGNAL declarative synchronous language : controller synthesis and systems/architecture design ». In *Proceedings of the 40th IEEE Conference on Decision and Control*, CDC '01, pages 3284–3289, Washington, DC, USA, 2001. IEEE Computer Society. (cité page 58)
- [12] Albert BENVENISTE, Paul CASPI, Stephen A. EDWARDS, Nicolas HALBWACHS, Paul LE GUERNIC et Robert DE SIMONE : « The synchronous languages 12 years later ». *Proceedings of the IEEE*, 91(1):64–83, janvier 2003. ISSN 0018-9219. (cité page 50)
- [13] Albert BENVENISTE, Paul LE GUERNIC et Christian JACQUEMOT : « Synchronous programming with events and relations: the SIGNAL language and its semantics ». *Sci. Comput. Program.*, 16:103–149, septembre 1991. ISSN 0167-6423. (cité pages 51 et 58)
- [14] Gérard BERRY et Georges GONTHIER : « The ESTEREL synchronous programming language: design, semantics, implementation ». *Sci. Comput. Program.*, 19:87–152, novembre 1992. ISSN 0167-6423. (cité pages 38, 51 et 83)
- [15] Nicolas BERTHIER, Florence MARANINCHI et Laurent MOUNIER : « Global Platform Management by Using Synchronous Device Drivers in μ -Kernel-based Systems ». *Fun Ideas and Thoughts Session, Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '11, avril 2011. (cité page 116)
- [16] Nicolas BERTHIER, Florence MARANINCHI et Laurent MOUNIER : « Synchronous programming of device drivers for global resource control in embedded operating systems ». In *Proceedings of the 2011 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '11, pages 81–90, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0555-6. (cité page 6)
- [17] Nicolas BERTHIER, Florence MARANINCHI et Laurent MOUNIER : « Synchronous programming of device drivers for global resource control in embedded operating systems ». *ACM Trans. Embed. Comput. Syst.*, 2012. (à paraître). (cité page 6)
- [18] Jan BEUTEL, Stephan GRUBER, Andreas HASLER, Roman LIM, Andreas MEIER, Christian PLESSL, Igor TALZI, Lothar THIELE, Christian TSCHUDIN, Matthias WOEHRLER et Mustafa YUECEL : « PermaDAQ: A scientific instrument for precision sensing and data recovery in environmental extremes ». In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, IPSN '09, pages 265–276, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-5108-1. (cité page 11)
- [19] Shah BHATTI, James CARLSON, Hui DAI, Jing DENG, Jeff ROSE, Anmol SHETH, Brian SHUCKER, Charles GRUENWALD, Adam TORGERSON et Richard HAN : « MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms ». *Mob. Netw. Appl.*, 10:563–579, août 2005. ISSN 1383-469X. (cité page 39)
- [20] Jó Ágila BITSCH LINK, Thomas BRETGELD, André GOLIATH et Klaus WEHRLE : « RatMote: a sensor platform for animal habitat monitoring ». In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '10, pages 432–433, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-988-6. (cité page 14)
- [21] Bruno BLANCHET, Patrick COUSOT, Radhia COUSOT, Jérôme FERET, Laurent MAUBORGNE, Antoine MINÉ, David MONNIAUX et Xavier RIVAL : « A static analyzer for large safety-critical software ».

- In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 196–207, New York, NY, USA, 2003. ACM. ISBN 1-58113-662-5. (cité page 23)
- [22] Tayeb BOUHADIBA, Quentin SABAH, Gwenaël DELAVAL et Éric RUTTEN : « Synchronous control of reconfiguration in fractal component-based systems: a case study ». *In Proceedings of the 9th ACM International Conference on Embedded Software*, EMSOFT '11, pages 309–318, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0714-7. (cité pages 59 et 110)
- [23] Randal E. BRYANT : « Graph-Based Algorithms for Boolean Function Manipulation ». *IEEE Trans. Comput.*, 35:677–691, août 1986. ISSN 0018-9340. (cité page 111)
- [24] Michael BUETTNER, Gary V. YEE, Eric ANDERSON et Richard HAN : « X-MAC: a short preamble MAC protocol for duty-cycled wireless sensor networks ». *In Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, pages 307–320, New York, NY, USA, 2006. ACM. ISBN 1-59593-343-3. (cité page 94)
- [25] Clément BURIN DES ROSIERS, Guillaume CHELIUS, Eric FLEURY, Antoine FRABOULET, Antoine GALLAIS, Nathalie MITTON et Thomas NOËL : « SensLAB Very Large Scale Open Wireless Sensor Network Testbed ». *In Proceedings of the 7th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCOM)*, Shanghai, China, avril 2011. (cité page 38)
- [26] C2A-SYNCHRON : « The common format of synchronous languages — The declarative code DC version 1.0 ». Rapport technique, SYNCHRON project, octobre 1995. (cité page 86)
- [27] Qing CAO et Tarek ABDELZAHER : « liteOS: a lightweight operating system for C++ software development in sensor networks ». *In Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, pages 361–362, New York, NY, USA, 2006. ACM. ISBN 1-59593-343-3. (cité page 42)
- [28] Qing CAO, Tarek ABDELZAHER, John STANKOVIC et Tian HE : « The LiteOS Operating System: Towards Unix-Like Abstractions for Wireless Sensor Networks ». *In Proceedings of the 7th International Conference on Information Processing in Sensor Networks*, IPSN '08, pages 233–244, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3157-1. (cité page 42)
- [29] Paul CASPI, Daniel PILAUD, Nicolas HALBWACHS et John A. PLAICE : « LUSTRE: a declarative language for real-time programming ». *In Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 178–188, New York, NY, USA, 1987. ACM. ISBN 0-89791-215-2. (cité page 51)
- [30] Paul CASPI, Norman SCAIFE, Christos SOFRONIS et Stavros TRIPAKIS : « Semantics-preserving multitask implementation of synchronous programs ». *ACM Trans. Embed. Comput. Syst.*, 7:15:1–15:40, janvier 2008. ISSN 1539-9087. (cité pages 4 et 50)
- [31] Franck CASSEZ : « Efficient on-the-fly algorithms for partially observable timed games ». *In Proceedings of the 5th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS'07, pages 5–24, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-75453-9, 978-3-540-75453-4. (cité page 58)
- [32] Hojung CHA, Sukwon CHOI, Inuk JUNG, Hyoseung KIM, Hyojeong SHIN, Jaehyun YOO et Chanmin YOON : « RETOS: resilient, expandable, and threaded operating system for wireless sensor networks ». *In Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, IPSN '07, pages 148–157, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-638-7. (cité pages 39 et 102)

- [33] Hojung CHA, Sukwon CHOI, Inuk JUNG, Hyoseung KIM, Hyojeong SHIN, Jaehyun YOO et Chanmin YOON : « The RETOS operating system: kernel, tools and applications ». In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, IPSN '07, pages 559–560, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-638-7. (cité page 39)
- [34] Vigyan CHANDRA, Zhongdong HUANG et Ratnesh KUMAR : « Automated control synthesis for an assembly line using discrete event system control theory ». *IEEE Trans. Syst. Man and Cybernetics*, 33 (2):284–289, mai 2003. ISSN 1094-6977. (cité page 59)
- [35] Prakash CHANDRASEKARAN, Christopher L. CONWAY, Joseph M. JOY et Sriram K. RAJAMANI : « Programming asynchronous layers with CLARITY ». In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 65–74, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-811-4. (cité page 35)
- [36] Guillaume CHELIUS, Antoine FRABOULET et Éric FLEURY : « Worldsens: a fast and accurate development framework for sensor network applications ». In *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC '07, pages 222–226, New York, NY, USA, 2007. ACM. ISBN 1-59593-480-4. (cité page 82)
- [37] Yu-Ting CHEN, Ting-Chou CHIEN et Pai H. CHOU : « Enix: a lightweight dynamic operating system for tightly constrained wireless sensor platforms ». In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 183–196, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0344-6. (cité pages 35 et 42)
- [38] Elaine CHEONG, Judy LIEBMAN, Jie LIU et Feng ZHAO : « TinyGALS: a programming model for event-driven embedded systems ». In *Proceedings of the 2003 ACM Symposium on Applied Computing*, SAC '03, pages 698–704, New York, NY, USA, 2003. ACM. ISBN 1-58113-624-2. (cité page 37)
- [39] Elaine CHEONG et Jie LIU : « galsC: A Language for Event-Driven Embedded Systems ». In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*, DATE '05, pages 1050–1055, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2288-2. (cité page 36)
- [40] *CC1100 Low-Power Sub-1GHz RF Transceiver*. Chipcon Products, 2006. <http://focus.ti.com/lit/ds/symlink/cc1100.pdf>. (cité pages 15, 17, 77 et 94)
- [41] Haksoo CHOI, Chanmin YOON et Hojung CHA : « Device driver abstraction for multithreaded sensor network operating systems ». In *Proceedings of the 5th European Conference on Wireless Sensor Networks*, EWSN '08, pages 354–368, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-77689-5. (cité page 46)
- [42] Christopher L. CONWAY et Stephen A. EDWARDS : « NDL: a domain-specific language for device drivers ». In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '04, pages 30–36, New York, NY, USA, 2004. ACM. ISBN 1-58113-806-7. (cité page 25)
- [43] Nuno COSTA, Antonio PEREIRA et Carlos SERODIO : « Virtual Machines Applied to WSN's: The state-of-the-art and classification ». In *Proceedings of the 2nd International Conference on Systems and Networks Communications*, ICSNC '07, pages 50–, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2938-0. (cité page 44)
- [44] Guy COUSINEAU et Pierre-Louis CURIEN : « The categorical abstract machine ». *Sci. Comput. Program.*, 8:173–202, avril 1987. ISSN 0167-6423. (cité page 43)

- [45] *DS1722 Digital Thermometer with SPI/3-Wire Interface*. Dallas Semiconductor, 2001. <http://www.maxim-ic.com/datasheet/index.mvp/id/2766>. (cité page 15)
- [46] *DS2411 Silicon Serial Number*. Dallas Semiconductor, 2006. <http://www.maxim-ic.com/datasheet/index.mvp/id/3711>. (cité page 15)
- [47] Alexandre DAVID, Jacob GRUNNET, Jan JESSEN, Kim LARSEN et Jacob RASMUSSEN : « Application of Model-Checking Technology to Controller Synthesis ». In Bernhard AICHERNIG, Frank de BOER et Marcello BONSANGUE, éditeurs : *Formal Methods for Components and Objects*, volume 6957 de *Lecture Notes in Computer Science*, pages 336–351. Springer Berlin / Heidelberg, 2012. ISBN 978-3-642-25270-9. (cité page 58)
- [48] Gwenaël DELAVAL, Hervé MARCHAND et Éric RUTTEN : « Contracts for modular discrete controller synthesis ». In *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES '10*, pages 57–66, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-953-4. (cité page 59)
- [49] Gwenaël DELAVAL et Éric RUTTEN : « A domain-specific language for task handlers generation, applying discrete controller synthesis ». In *Proceedings of the 2006 ACM Symposium on Applied Computing, SAC '06*, pages 901–905, New York, NY, USA, 2006. ACM. ISBN 1-59593-108-2. (cité page 59)
- [50] Robert DELINE et Manuel FÄHNDRICH : « Enforcing high-level protocols in low-level software ». In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, PLDI '01*, pages 59–69, New York, NY, USA, 2001. ACM. ISBN 1-58113-414-2. (cité page 23)
- [51] Mischa DOHLER, Dominique BARTHEL, Florence MARANINCHI, Laurent MOUNIER, Stéphane AUBERT, Christophe DUGAS, Aurélien BUHRIG, Franck PAUGNAT, Marc RENAUDIN, Andrzej DUDA, Martin HEUSSE et Fabrice VALOIS : « The ARESA Project: Facilitating Research, Development and Commercialization of WSNs ». In *Proceedings of 4th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks, SECON '07*, pages 590–599, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 1-4244-1268-4. (cité page 82)
- [52] Wei DONG, Chun CHEN, Xue LIU, Yunhao LIU, Jiajun BU et Kougen ZHENG : « SenSpire OS: A Predictable, Flexible, and Efficient Operating System for Wireless Sensor Networks ». *IEEE Trans. Comput.*, PP:1–14, mars 2011. ISSN 0018-9340. (cité page 43)
- [53] Marc DOYLE, Thomas F. FULLER et John NEWMAN : « Modeling of Galvanostatic Charge and Discharge of the Lithium/Polymer/Insertion Cell ». *J. Electrochem. Soc.*, 140(6):1526–1533, juin 1993. ISSN 0013-4651. (cité page 20)
- [54] Cormac DUFFY, Utz ROEDIG, John HERBERT et Cormac J. SREENAN : « An Experimental Comparison of Event Driven and Multi-Threaded Sensor Node Operating Systems ». In *Proceedings of the 5th IEEE International Conference on Pervasive Computing and Communications Workshops, PERCOMW '07*, pages 267–271, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2788-4. (cité page 42)
- [55] Cormac DUFFY, Utz ROEDIG, John HERBERT et Cormac J. SREENAN : « Improving the energy efficiency of the MANTIS kernel ». In *Proceedings of the 4th European Conference on Wireless Sensor Networks, EWSN '07*, pages 261–276, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-69829-6. (cité page 39)

- [56] Adam DUNKELS, Niclas FINNE, Joakim ERIKSSON et Thiemo VOIGT : « Run-time dynamic linking for reprogramming wireless sensor networks ». In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, pages 15–28, New York, NY, USA, 2006. ACM. ISBN 1-59593-343-3. (cité pages 41 et 44)
- [57] Adam DUNKELS, Bjorn GRONVALL et Thiemo VOIGT : « Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors ». In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, LCN '04, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2260-2. (cité page 41)
- [58] Adam DUNKELS, Fredrik ÖSTERLIND, Nicolas TSIFTES et Zhitao HE : « Software-based sensor node energy estimation ». In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, pages 409–410, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-763-6. (cité page 42)
- [59] Adam DUNKELS, Oliver SCHMIDT, Thiemo VOIGT et Muneeb ALI : « Protothreads: simplifying event-driven programming of memory-constrained embedded systems ». In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, pages 29–42, New York, NY, USA, 2006. ACM. ISBN 1-59593-343-3. (cité pages 37 et 64)
- [60] Prabal K. DUTTA, Jonathan W. HUI, David C. CHU et David E. CULLER : « Securing the deluge Network programming system ». In *Proceedings of the 5th International Conference on Information Processing in Sensor Networks*, IPSN '06, pages 326–333, New York, NY, USA, 2006. ACM. ISBN 1-59593-334-4. (cité page 40)
- [61] Stephen A. EDWARDS : « Compiling Esterel into sequential code ». In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, pages 322–327, New York, NY, USA, 2000. ACM. ISBN 1-58113-187-9. (cité page 89)
- [62] *EIA Standard RS-232-C*. Electronic Industries Association, 2001 Eye Street N.W., Washington DC 20006, 1969, 1981. (cité page 13)
- [63] *Alkaline Handbook — version Alk1.3*. Energizer Battery Manufacturing Inc., 2008. http://data.energizer.com/PDFs/alkaline_appman.pdf. (cité page 19)
- [64] *Nickel Metal Hydride — version NiMH01.11*. Energizer Battery Manufacturing Inc., 2010. http://data.energizer.com/PDFs/nickelmetalhydride_appman.pdf. (cité page 19)
- [65] Dawson R. ENGLER, M. Frans KAASHOEK et James O'TOOLE, Jr : « Exokernel: an operating system architecture for application-level resource management ». *SIGOPS Oper. Syst. Rev.*, 29:251–266, décembre 1995. ISSN 0163-5980. (cité page 41)
- [66] M. Anton ERTL, David GREGG, Andreas KRALL et Bernd PAYSAN : « Vmgen: a generator of efficient virtual machine interpreters ». *Softw. Pract. Exper.*, 32:265–294, mars 2002. ISSN 0038-0644. (cité page 44)
- [67] Anand ESWARAN, Anthony ROWE et Raj RAJKUMAR : « Nano-RK: An Energy-Aware Resource-Centric RTOS for Sensor Networks ». In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 256–265, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2490-7. (cité page 39)
- [68] Robert R. EVERETT, Charles A. ZRAKET et Herbert D. BENINGTON : « SAGE - A Data-Processing System for Air Defense ». In *Proceedings of the Eastern Joint Computer Conference*, pages 148–155. IRE, 1957. (cité page 35)

- [69] Antoine FRABOULET, Guillaume CHELIUS et Éric FLEURY : « Worldsens: development and prototyping tools for application specific wireless sensors networks ». *In Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, IPSN '07, pages 176–185, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-638-7. (cité pages 14, 15 et 82)
- [70] Adrien FRIGGERI, Guillaume CHELIUS, Éric FLEURY, Antoine FRABOULET, France MENTRÉ et Jean-Christophe LUCET : « Reconstructing social interactions using an unreliable wireless sensor network ». *Comput. Commun.*, 34:609–618, avril 2011. ISSN 0140-3664. (cité page 11)
- [71] Hauke FUHRMANN et Reinhard von HANXLEDEN : « Taming graphical modeling ». *In Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I*, MODELS'10, pages 196–210, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-16144-8, 978-3-642-16144-5. (cité page 85)
- [72] Fabien GAUCHER : *Étude du débogage de systèmes réactifs et application au langage synchrone Lustre*. Thèse de doctorat, Institut National Polytechnique de Grenoble, novembre 2003. (cité page 52)
- [73] David GAY, Philip LEVIS, Robert von BEHREN, Matt WELSH, Eric BREWER et David E. CULLER : « The nesC language: A holistic approach to networked embedded systems ». *In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 1–11, New York, NY, USA, 2003. ACM. ISBN 1-58113-662-5. (cité page 36)
- [74] Alain GIRAULT et Éric RUTTEN : « Automating the addition of fault tolerance with discrete controller synthesis ». *Form. Methods Syst. Des.*, 35:190–225, octobre 2009. ISSN 0925-9856. (cité page 59)
- [75] *Lithium Ion technical handbook*. Gold Peak Industries (Taiwan), Ltd., décembre 2000. http://www.gpina.com/pdf/Li-ion_Handbook.pdf. (cité page 19)
- [76] Lin GU et John A. STANKOVIC : « *t-kernel*: A translative OS kernel for sensor networks ». Rapport technique UVA/CS-2005-09, Department of Computer Science, University of Virginia, 2005. (cité page 39)
- [77] Lin GU et John A. STANKOVIC : « *t-kernel*: providing reliable OS support to wireless sensor networks ». *In Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, pages 1–14, New York, NY, USA, 2006. ACM. ISBN 1-59593-343-3. (cité page 39)
- [78] Jiwon HAHN et Pai H. CHOU : « Buffer optimization and dispatching scheme for embedded systems with behavioral transparency ». *In Proceedings of the 7th ACM & IEEE International Conference on Conference on Embedded Software*, EMSOFT '07, pages 94–103, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-825-1. (cité page 45)
- [79] Jiwon HAHN et Pai H. CHOU : « Nucleos: a runtime system for ultra-compact wireless sensor nodes ». *In Proceedings of the 10th ACM International Conference on Conference on Embedded Software*, EMSOFT '10, pages 149–158, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-904-6. (cité page 45)
- [80] Nicolas HALBWACHS, Fabienne LAGNIER et Christophe RATEL : « Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language LUSTRE ». *IEEE Trans. Softw. Eng.*, 18:785–793, septembre 1992. ISSN 0098-5589. (cité page 52)
- [81] Nicolas HALBWACHS et Florence MARANINCHI : « On the symbolic analysis of combinational loops in circuits and synchronous programs ». *In Proceedings of the 21th Euromicro Conference on Design of Hardware/Software Systems*, Euromicro '95, Washington, DC, USA, septembre 1995. IEEE Computer Society Press. ISBN 0-818-67127-0, 978-0-818-67127-2. (cité pages 89 et 90)

- [82] Chih-Chieh HAN, Ram KUMAR, Roy SHEA, Eddie KOHLER et Mani SRIVASTAVA : « A dynamic operating system for sensor nodes ». In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, MobiSys '05, pages 163–176, New York, NY, USA, 2005. ACM. ISBN 1-931971-31-5. (cité page 40)
- [83] David HAREL : « Statecharts: A visual formalism for complex systems ». *Sci. Comput. Program.*, 8:231–274, juin 1987. ISSN 0167-6423. (cité page 37)
- [84] David HAREL et Amir PNUELI : *On the development of reactive systems*, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985. ISBN 0-387-15181-8. (cité page 16)
- [85] Karel HEURTEFEUX : *Protocoles Localisés pour Réseaux de Capteurs*. Thèse de doctorat, INSA de Lyon, novembre 2009. (cité page 17)
- [86] Karel HEURTEFEUX, Florence MARANINCHI et Fabrice VALOIS : « AreaCast: a Cross-Layer Approach for a Communication by Area in Wireless Sensor Networks ». In *Proceedings of the 17th IEEE International Conference on Networks*, ICON '11, Washington, DC, USA, décembre 2011. IEEE Computer Society. (cité page 17)
- [87] *Advanced Configuration and Power Interface Specification – revision 4.0a*. Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd. et Toshiba Corporation, avril 2010. <http://www.acpi.info/>. (cité page 5)
- [88] Jason HILL, Robert SZEWCZYK, Alec WOO, Seth HOLLAR, David E. CULLER et Kristofer PISTER : « System architecture directions for networked sensors ». *SIGOPS Oper. Syst. Rev.*, 34:93–104, novembre 2000. ISSN 0163-5980. (cité page 40)
- [89] Kirak HONG, Jiin PARK, Taekhoon KIM, Sungho KIM, Hwangho KIM, Yousun KO, Jongtae PARK, Bernd BURGSTALLER et Bernhard SCHOLZ : « TinyVM, an efficient virtual machine infrastructure for sensor networks ». In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 399–400, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-519-2. (cité page 44)
- [90] Jonathan W. HUI et David E. CULLER : « The dynamic behavior of a data dissemination protocol for network programming at scale ». In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, pages 81–94, New York, NY, USA, 2004. ACM. ISBN 1-58113-879-2. (cité page 40)
- [91] Ozlem Durmaz INCEL : « Survey Paper: A survey on multi-channel communication in wireless sensor networks ». *Comput. Netw.*, 55:3081–3099, septembre 2011. ISSN 1389-1286. (cité page 17)
- [92] *SensTools*. INRIA, 2008. <http://sensTOOLS.gforge.inria.fr/>. (cité pages 81 et 94)
- [93] Erwan JAHIER, Pascal RAYMOND et Philippe BAUFRETON : « Case studies with Lurette V2 ». *Int. J. Softw. Tools Technol. Transf.*, 8:517–530, octobre 2006. ISSN 1433-2779. (cité page 52)
- [94] B. JEANNET : « Dynamic Partitioning in Linear Relation Analysis: Application to the Verification of Reactive Systems ». *Form. Methods Syst. Des.*, 23:5–37, juillet 2003. ISSN 0925-9856. (cité page 52)
- [95] Philo JUANG, Hidekazu OKI, Yong WANG, Margaret MARTONOSI, Li Shiuan PEH et Daniel RUBENSTEIN : « Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with ZebraNet ». In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-X, pages 96–107, New York, NY, USA, 2002. ACM. ISBN 1-58113-574-2. (cité page 11)

- [96] Raja JURDAK, Peter CORKE, Dhinesh DHARMAN et Guillaume SALAGNAC : « Adaptive GPS duty cycling and radio ranging for energy-efficient localization ». In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems, SenSys '10*, pages 57–70, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0344-6. (cité page 11)
- [97] Gilles KAHN et David MACQUEEN : « Coroutines and Networks of Parallel Processes ». Rapport technique, Laboria, University of Edinburgh, 1976. (cité page 51)
- [98] Oliver KASTEN : *A State-Based Programming Model for Wireless Sensor Networks*. Thèse de doctorat, ETH Zurich, Zurich, Switzerland, juin 2007. (cité page 37)
- [99] Oliver KASTEN et Kay RÖMER : « Beyond event handlers: programming wireless sensors with attributed state machines ». In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks, IPSN '05*, Piscataway, NJ, USA, 2005. IEEE Press. ISBN 0-7803-9202-7. (cité page 37)
- [100] Kevin KLUES, Vlado HANDZISKI, Chenyang LU, Adam WOLISZ, David E. CULLER, David GAY et Philip LEVIS : « Integrating concurrency control and energy management in device drivers ». *SIGOPS Oper. Syst. Rev.*, 41:251–264, octobre 2007. ISSN 0163-5980. (cité pages 45 et 107)
- [101] Kevin KLUES, Chieh-Jan Mike LIANG, Jeongyeup PAEK, Răzvan MUSĂLOIU-E, Philip LEVIS, Andreas TERZIS et Ramesh GOVINDAN : « TOSThreads: thread-safe and non-invasive preemption in TinyOS ». In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems, SenSys '09*, pages 127–140, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-519-2. (cité page 43)
- [102] Donald E. KNUTH : *Fundamental Algorithms*, volume 1 de *The Art of Computer Programming*, section 1.4.2: Coroutines. Addison-Wesley, Reading, MA, USA, 3^{ème} édition, juillet 1997. ISBN 0-201-89683-4, 978-0-201-89683-1. (cité pages 37 et 83)
- [103] Kurtis KREDO, II et Prasant MOHAPATRA : « Medium access control in wireless sensor networks ». *Comput. Netw.*, 51:961–994, mars 2007. ISSN 1389-1286. (cité page 17)
- [104] Philip LEVIS et David E. CULLER : « Maté: a tiny virtual machine for sensor networks ». *SIGOPS Oper. Syst. Rev.*, 36:85–95, octobre 2002. ISSN 0163-5980. (cité page 44)
- [105] Philip LEVIS, David GAY et David E. CULLER : « Bridging the Gap: Programming Sensor Networks with Application Specific Virtual Machines ». Rapport technique UCB/CSD-04-1343, EECS Department, University of California, Berkeley, 2004. (cité page 44)
- [106] Philip LEVIS, Nelson LEE, Matt WELSH et David E. CULLER : « TOSSIM: accurate and scalable simulation of entire TinyOS applications ». In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems, SenSys '03*, pages 126–137, New York, NY, USA, 2003. ACM. ISBN 1-58113-707-9. (cité page 40)
- [107] Philip LEVIS, Sam MADDEN, Joseph POLASTRE, Robert SZEWCZYK, Kamin WHITEHOUSE, Alec Woo, David GAY, Jason HILL, Matt WELSH, Eric BREWER et David E. CULLER : « TinyOS: An Operating System for Wireless Sensor Networks ». In Werner WEBER, Jan M. RABAEY et Emile H.L. AARTS, éditeurs : *Ambient Intelligence*. Springer-Verlag, 2005. ISBN 978-3-540-23867-6. (cité page 40)
- [108] Philip LEVIS, Neil PATEL, David E. CULLER et Scott SHENKER : « Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks ». In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association. (cité page 44)

- [109] David LINDEN et Thomas B. REDDY : *Handbook of batteries*. McGraw-Hill handbooks. McGraw-Hill, 3^{ème} édition, 2002. ISBN 978-0-071-35978-8. (cité page 20)
- [110] Tim LINDHOLM et Frank YELLIN : *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2^{ème} édition, 1999. ISBN 0-201-43294-3. (cité page 43)
- [111] Konrad LORINCZ, Bor-rong CHEN, Jason WATERMAN, Geoff WERNER-ALLEN et Matt WELSH : « Resource aware programming in the Pixie OS ». In *Proceedings of the 6th ACM Conference on Embedded network sensor systems*, SenSys '08, pages 211–224, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-990-6. (cité page 41)
- [112] Florence MARANINCHI : « Operational and Compositional Semantics of Synchronous Automaton Compositions ». In *Proceedings of the 3rd International Conference on Concurrency Theory*, CONCUR '92, pages 550–564, London, UK, 1992. Springer-Verlag. ISBN 3-540-55822-5. (cité pages 54, 55 et 86)
- [113] Florence MARANINCHI et Nicolas HALBWACHS : « Compiling ARGOS into Boolean Equations ». In *Proceedings of the 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 72–89, London, UK, 1996. Springer-Verlag. ISBN 3-540-61648-9. (cité page 86)
- [114] Florence MARANINCHI et Yann RÉMOND : « Argos: an automaton-based synchronous language ». *Comput. Lang.*, 27(1-3):61–92, 2001. ISSN 0096-0551. (cité pages 6, 51, 54, 86 et 88)
- [115] Hervé MARCHAND, Patricia BOURNAI, Michel Le BORGNE et Paul Le GUERNIC : « Synthesis of Discrete-Event Controllers Based on the Signal Environment ». *Discrete Event Dynamic Systems*, 10:325–346, octobre 2000. ISSN 0924-6703. (cité page 58)
- [116] William P. McCARTNEY et Nigamanth SRIDHAR : « Abstractions for safe concurrent programming in networked embedded systems ». In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, pages 167–180, New York, NY, USA, 2006. ACM. ISBN 1-59593-343-3. (cité page 43)
- [117] Fabrice MÉRILLON et Gilles MULLER : « Dealing with Hardware in Embedded Software: A General Framework Based on the Devil Language ». In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, LCTES '01, pages 121–127, New York, NY, USA, 2001. ACM. ISBN 1-58113-425-8. (cité page 25)
- [118] Fabrice MÉRILLON, Laurent RÉVEILLÈRE, Charles CONSEL, Renaud MARLET et Gilles MULLER : « Devil: an IDL for hardware programming ». In *Proceedings of the 4th Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 2–2, Berkeley, CA, USA, 2000. USENIX Association. (cité page 24)
- [119] David MONNIAUX : « Verification of device drivers and intelligent controllers: a case study ». In *Proceedings of the 7th ACM & IEEE International Conference on Conference on Embedded Software*, EMSOFT '07, pages 30–36, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-825-1. (cité page 23)
- [120] MOTOROLA, INC : *M68HC11 Reference Manual*. Motorola, Inc., 1995. (cité page 13)
- [121] Mohammad Mostafizur Rahman MOZUMDAR, Luciano LAVAGNO et Laura VANZAGO : « A comparison of software platforms for wireless sensor networks: MANTIS, TinyOS, and ZigBee ». *ACM Trans. Embed. Comput. Syst.*, 8:12:1–12:23, février 2009. ISSN 1539-9087. (cité page 42)
- [122] Christopher NITTA, Raju PANDEY et Yann RAMIN : « Y-Threads: Supporting Concurrency in Wireless Sensor Networks ». In Phillip GIBBONS, Tarek ABDELZAHER, James ASPNES et Ramesh RAO, éditeurs :

- Distributed Computing in Sensor Systems*, volume 4026 de *Lecture Notes in Computer Science*, pages 169–184. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-35227-3. (cité page 43)
- [123] Gordon PACE, Nicolas HALBWACHS et Pascal RAYMOND : « Counter-example generation in symbolic abstract model-checking ». *Int. J. Softw. Tools Technol. Transf.*, 5:158–164, mars 2004. ISSN 1433-2779. (cité page 108)
- [124] Massoud PEDRAM et Qing WU : « Design considerations for battery-powered electronics ». In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, DAC '99, pages 861–866, New York, NY, USA, 1999. ACM. ISBN 1-58113-109-7. (cité page 20)
- [125] *The I²C-bus specification*. Philips Semiconductors, 2000. <http://www.nxp.com/documents/other/39340011.pdf>. (cité page 13)
- [126] Joseph POLASTRE, Robert SZEWCZYK et David E. CULLER : « Telos: enabling ultra-low power wireless research ». In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press. ISBN 0-7803-9202-7. (cité page 14)
- [127] Barry PORTER et Geoff COULSON : « Lorien: a pure dynamic component-based operating system for wireless sensor networks ». In *Proceedings of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*, MidSens '09, pages 7–12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-851-3. (cité page 40)
- [128] Dumitru POTOP-BUTUCARU, Stephen A. EDWARDS et Gérard BERRY : *Compiling Esterel*. Springer Publishing Co., Inc., 1^{ère} édition, 2007. ISBN 0-387-70626-7, 978-0-387-70626-9. (cité page 89)
- [129] Steffen PROCHNOW, Claus TRAULSEN et Reinhard von HANXLEDEN : « Synthesizing safe state machines from Esterel ». In *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool support for Embedded Systems*, LCTES '06, pages 113–124, New York, NY, USA, 2006. ACM. ISBN 1-59593-362-X. (cité page 37)
- [130] William PUGH : « Fixing the Java memory model ». In *Proceedings of the ACM 1999 Conference on Java Grande*, JAVA '99, pages 89–98, New York, NY, USA, 1999. ACM. ISBN 1-58113-161-5. (cité page 43)
- [131] Joseph RAHMÉ, Nicolas FOURTY, Khaldoun Al AGHA et Adrien van den BOSSCHE : « A Recursive Battery Model for Nodes Lifetime Estimation in Wireless Sensor Networks ». In *Proceedings of the Wireless Communications and Networking Conference*, WCNC '10, pages 1–6, Washington, DC, USA, avril 2010. IEEE Computer Society. (cité page 21)
- [132] Raj RAJKUMAR, Kanaka JUVVA, Anastasio MOLANO et Shuichi OIKAWA : « Resource kernels: A resource-centric approach to real-time and multimedia systems ». In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, pages 150–164, 1998. (cité page 39)
- [133] Daler N. RAKHMATOV et Sarma VRUDHULA : « Energy management for battery-powered embedded systems ». *ACM Trans. Embed. Comput. Syst.*, 2:277–324, août 2003. ISSN 1539-9087. (cité page 20)
- [134] Peter J. G. RAMADGE et W. Murray WONHAM : « The control of discrete event systems ». *Proceedings of the IEEE*, 77(1):81–98, janvier 1989. ISSN 0018-9219. (cité page 56)
- [135] Ravishankar RAO, Sarma VRUDHULA et Naehyuck CHANG : « Battery optimization vs energy optimization: which to choose and when? ». In *Proceedings of the 2005 International Conference on Computer-Aided Design*, ICCAD '05, pages 439–445, Washington, DC, USA, novembre 2005. IEEE Computer Society. ISBN 0-7803-9254-X. (cité page 21)

- [136] Ravishankar RAO, Sarma VRUDHULA et Daler N. RAKHMATOV : « Battery Modeling for Energy-Aware System Design ». *Computer*, 36:77–87, décembre 2003. ISSN 0018-9162. (cité page 20)
- [137] REAL TIME ENGINEERS LTD. : « FREERTOS », 2008. (cité page 39)
- [138] Nicholas RESCHER et Alasdair URQUHART : *Temporal logic*. Springer-Verlag New York, Inc., New York, NY, USA, 1971. ISBN 3-211-80995-3, 978-0-387-80995-3. (cité page 25)
- [139] Dennis M. RITCHIE et Ken THOMPSON : « The UNIX time-sharing system ». *SIGOPS Oper. Syst. Rev.*, 7:27–, janvier 1973. ISSN 0163-5980. (cité page 42)
- [140] Leonid RYZHYK, Peter CHUBB, Ihor KUZ, Etienne LE SUEUR et Gernot HEISER : « Automatic device driver synthesis with termite ». In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP '09*, pages 73–86, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. (cité page 25)
- [141] Ludovic SAMPER : *Modélisations et Analyses de Réseaux de Capteurs*. Thèse de doctorat, Institut National Polytechnique de Grenoble, avril 2008. (cité page 11)
- [142] Klaus SCHNEIDER et Jens BRANDT : « Performing causality analysis by bounded model checking ». In *Proceedings of the 8th International Conference on Application of Concurrency to System Design, ACSD '08*, pages 78–87, Washington, DC, USA, juin 2008. IEEE Computer Society. (cité page 89)
- [143] Jacob SORBER, Alexander KOSTADINOV, Matthew GARBER, Matthew BRENNAN, Mark D. CORNER et Emery D. BERGER : « Eon: a language and runtime system for perpetual systems ». In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems, SenSys '07*, pages 161–174, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-763-6. (cité page 41)
- [144] Phillip STANLEY-MARBELL et Liviu IFTODE : « Scylla: a smart virtual machine for mobile embedded systems ». In *Proceedings of the 3rd IEEE Workshop on Mobile Computing Systems and Applications*, pages 41–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0816-2. (cité page 45)
- [145] Eliana STAVROU et Andreas PITSILLIDES : « A survey on secure multipath routing protocols in WSNs ». *Comput. Netw.*, 54:2215–2238, septembre 2010. ISSN 1389-1286. (cité page 17)
- [146] *M25P80*. STMicroelectronics, décembre 2002. <http://www.datasheetcatalog.org/datasheet/stmicroelectronics/8495.pdf>. (cité page 15)
- [147] Jun SUN, Wanghong YUAN, Mahesh KALLAHALLA et Nayeem ISLAM : « HAIL: a language for easy and correct device access ». In *Proceedings of the 5th ACM International Conference on Conference on Embedded Software, EMSOFT '05*, pages 1–9, New York, NY, USA, 2005. ACM. ISBN 1-59593-091-4. (cité page 25)
- [148] Robert SZEWCZYK, Eric OSTERWEIL, Joseph POLASTRE, Michael HAMILTON, Alan MAINWARING et Deborah ESTRIN : « Habitat monitoring with sensor networks ». *Commun. ACM*, 47:34–40, June 2004. ISSN 0001-0782. (cité page 11)
- [149] *TSL2550 Ambient Light Sensor with SMBus Interface*. TAOS Inc., octobre 2007. <http://www.taosinc.com/ProductDetails.aspx?id=133>. (cité page 15)
- [150] Olivier TARDIEU et Robert de SIMONE : « Loops in estereel ». *ACM Trans. Embed. Comput. Syst.*, 4:708–750, novembre 2005. ISSN 1539-9087. (cité page 89)
- [151] *MSP430x1xx Family, User's Guide*. Texas Instruments, 2006. <http://focus.ti.com/lit/ug/slau049f/slau049f.pdf>. (cité pages 14 et 32)

- [152] Claus TRAULSEN, Torsten AMENDE et Reinhard von HANXLEDEN : « Compiling SyncCharts to Synchronous C ». In *Proceedings of the Design, Automation and Test in Europe Conference, DATE '11*, pages 563–566, Washington, DC, USA, mars 2011. IEEE Computer Society. ISBN 978-1-61284-208-0. (cité page 85)
- [153] VARTA Microbattery GmbH. « LIP 423450 AJL, data sheet — Rechargeable Lithium-Ion Prismatic », 2011. http://www.varta-microbattery.com/en/mb_data/documents/data_sheets/DS56424.pdf. (cité pages 15 et 19)
- [154] Reinhard von HANXLEDEN : « SyncCharts in C: a proposal for light-weight, deterministic concurrency ». In *Proceedings of the 7th ACM International Conference on Conference on Embedded Software, EMSOFT '09*, pages 225–234, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-627-4. (cité page 82)
- [155] Shaojie WANG, Sharad MALIK et Reinaldo A. BERGAMASCHI : « Modeling and Integration of Peripheral Devices in Embedded Systems ». In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1, DATE '03*, pages 10136–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1870-2. (cité page 25)
- [156] Yin WANG, Stéphane LAFORTUNE, Terence KELLY, Manjunath KUDLUR et Scott MAHLKE : « The theory of deadlock avoidance via discrete control ». In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '09*, pages 252–263, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. (cité pages 107 et 113)
- [157] Matt WELSH et Geoff MAINLAND : « Programming sensor networks using abstract regions ». In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association. (cité page 42)
- [158] Geoffrey WERNER-ALLEN, Konrad LORINCZ, Matt WELSH, Omar MARCILLO, Jeff JOHNSON, Mario RUIZ et Jonathan LEES : « Deploying a Wireless Sensor Network on an Active Volcano ». *IEEE Internet Computing*, 10:18–25, mars 2006. ISSN 1089-7801. (cité page 11)
- [159] Andrew WHITAKER, Marianne SHAW et Steven D. GRIBBLE : « Scale and performance in the Denali isolation kernel ». In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation, OSDI '02*, pages 195–209, New York, NY, USA, 2002. ACM. ISBN 978-1-4503-0111-4. (cité pages 5, 47 et 64)
- [160] Andrew K. WRIGHT et Matthias FELLEISEN : « A syntactic approach to type soundness ». *Inf. Comput.*, 115:38–94, novembre 1994. ISSN 0890-5401. (cité page 43)
- [161] M. Aykut YIGITEL, Ozlem Durmaz INCEL et Cem ERSOY : « QoS-aware MAC protocols for wireless sensor networks: A survey ». *Comput. Netw.*, 55:1982–2004, juin 2011. ISSN 1389-1286. (cité page 17)
- [162] Heng ZENG, Carla S. ELLIS, Alvin R. LEBECK et Amin VAHDAT : « ECOSystem: managing energy as a first class operating system resource ». *SIGOPS Oper. Syst. Rev.*, 36:123–132, octobre 2002. ISSN 0163-5980. (cité page 41)
- [163] Manchun ZHENG, Jun SUN, Yang LIU, Jin Song DONG et Yu GU : « Towards a Model Checker for NesC and Wireless Sensor Networks ». In *Proceedings of the 13th International Conference on Formal Engineering Methods, ICFEM '11*, pages 372–387, London, UK, octobre 2011. Springer-Verlag. ISBN 978-3-642-24558-9. (cité page 107)
- [164] Hubert ZIMMERMANN : « OSI Reference Model — The ISO Model of Architecture for Open Systems Interconnection ». *IEEE Trans. Comm.*, 28(4):425–432, avril 1980. (cité page 94)

Résumé

Le travail présenté dans cette thèse porte sur la conception de logiciels pour systèmes embarqués. Outre les contraintes de programmation provenant des faibles quantités de mémoire et capacité de calcul, ces plateformes matérielles ne disposent parfois que de peu d'énergie pour fonctionner. Les applications usuelles de ces systèmes imposent de plus des objectifs en matière de réactivité et de durée de vie. Par ailleurs, quelques-unes des ressources fournies sont partagées entre les composants, qu'il s'agisse de l'énergie délivrée par une batterie, ou encore des bus de communication qui les relient. Il est donc nécessaire de pouvoir assurer des propriétés globales portant sur l'ensemble de la plate-forme, telles que le contrôle des accès aux bus, ou encore la maîtrise de la puissance électrique consommée. Cependant, les pilotes des différents périphériques sont d'ordinaire programmés individuellement. La connaissance nécessaire à l'implantation d'une politique de contrôle global est alors distribuée parmi diverses portions du logiciel.

Nous exposons une solution au problème du contrôle global des ressources, basée sur une vue centralisée des états des composants matériels de la plate-forme. Bâtie sur un principe de para-virtualisation, notre approche consiste en l'introduction d'une couche de contrôle ; le système d'exploitation invité est adapté afin de communiquer avec le matériel à l'aide de celle-ci. La couche de contrôle incorpore les pilotes des périphériques, conçus à partir d'automates dont les états correspondent aux modes de fonctionnement ou de consommation du composant géré. Un contrôleur est ajouté, dont le rôle est d'assurer les propriétés globales. L'ensemble de ces automates est programmé à l'aide d'un langage synchrone, puis compilé en code séquentiel.

Nous proposons une implantation de la couche de contrôle pour une architecture de nœuds de réseaux de capteurs sans fil, qui constitue une plate-forme représentative des systèmes embarqués contraints. Nous évaluons qualitativement et quantitativement ce prototype afin de montrer la viabilité de l'approche. Son impact sur le reste du logiciel est également apprécié, que celui-ci soit construit selon un modèle d'exécution purement événementiel ou multi-fils. Enfin, nous passons en revue plusieurs extensions possibles, et identifions quelques bonnes pratiques pour son usage dans d'autres contextes.

Mots-clés : Programmation synchrone, Systèmes d'exploitation, Systèmes embarqués, Contrôle global

Abstract

This thesis is about the design of software for embedded systems. The hardware platforms usually employed in these systems provide a limited amount of memory, computational power and energy. The software they execute is then constrained by such limited resources. Usual applications involve further objectives, such as reactivity and lifetime. In addition, these platforms comprise shared resources like buses or even the energy provided by a battery. Hence, global properties concerning the whole platform must be enforced, for instance to control concurrent accesses to a bus or power consumption. As device drivers are commonly developed individually, the knowledge necessary to implement global control policies is distributed among several pieces of software.

We propose a global control approach, based on a centralized view of the devices' states. Built upon para-virtualization principles, it operates on the hardware/software interface. It involves a simple adaptation of the guest operating system, to communicate with the hardware via a control layer. The control layer itself is built from a set of simple automata: the device drivers, whose states correspond to functional or power consumption modes, and a controller to enforce global properties. All these automata are programmed using a synchronous language, and compiled into a single piece of sequential code.

As a suitable representative of embedded systems hardware, we choose the node of a wireless sensor network. To show that our approach is practical, we propose a proof-of-concept implementation of the control layer to manage this platform, and evaluate it both qualitatively and quantitatively. We also demonstrate its use and benefits with an event-driven or multithreading operating system, and estimate the impact of the adaptation on guest software. Finally, we audit several extensions and draw guidelines for its use in other contexts.

Keywords: Synchronous Programming, Operating Systems, Embedded Systems, Global Control