INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|---|---|---|---|---|---|---|---|---|---|---|---|

T H È S E

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : « INFORMATIQUE : SYSTÈMES ET LOGICIELS »

préparée au laboratoire VERIMAG
dans le cadre de l'**École doctorale « MATHÉMATIQUES, SCIENCES ET TECHNOLOGIES DE L'INFORMATION, INFORMATIQUE »**

présentée et soutenue publiquement

par

Ludovic SAMPER

le 7 avril 2008

Titre :

Modélisations et Analyses de Réseaux de Capteurs

Directeurs de thèse :

Florence Maraninchi
Laurent Mounier

JURY

Marc Pouzet	Président
Isabelle Guérin-Lassous	Rapporteure
Robert de Simone	Rapporteur
Dominique Barthel	Examinateur
Florence Maraninchi	Directrice de thèse
Laurent Mounier	Directeur de thèse

Remerciements

Premièrement, je tiens à remercier Marc Pouzet de m'avoir fait l'honneur de bien vouloir présider le jury de ma thèse.
Je remercie Isabelle Guérin-Lassous et Robert de Simone d'avoir accepté d'être rapporteurs de ma thèse.
Merci à tous les trois pour l'intérêt que vous avez porté à ce travail, pour vos remarques qui m'ont permis de voir plus loin.
Bien entendu, sans mes encadrants rien de tout cela n'aurait été possible. Tous les doctorants n'ont pas un directeur de thèse disponible et motivé par leurs travaux. Je sais la chance que j'ai d'en avoir eu trois !
Je remercie Dominique Barthel, Florence Maraninchi et Laurent Mounier d'avoir entrepris ensemble des recherches sur la modélisation des réseaux de capteurs. Dominique Barthel a très vite compris l'intérêt des modèles formels. Cet enthousiasme a été très bénéfique pour mes travaux. Dominique m'a également beaucoup aidé par ses explications en électronique. J'ai eu la chance de bénéficier du recul de Florence Maraninchi qui m'a toujours proposé des perspectives de recherche pertinentes. Je remercie Laurent Mounier pour sa grande disponibilité et sa gentillesse.
Je tiens à remercier Abdelmalik Bachir et Louis Mandel pour les travaux de recherche fructueux que nous avons menés ensemble. Je remercie également Louis Mandel d'avoir toujours été très réactif aux problèmes que j'ai pu rencontrer avec son langage de programmation.
Wassim Znaidi, Kevin Baradon et Antoine Vasseur ont participé à ce travail pendant leurs stages respectifs. Je les remercie sincèrement.
Je remercie également Mischa Dohler, Christophe Dugas, Olivier Bezet ainsi que tous les membres du projet ARESA.
Merci à Hugo, Guillaume et Mathias pour ces années de thèse que nous avons surmontées ensemble.
Les laboratoires de Verimag et de France Télécom fournissent un cadre de travail privilégié, merci à tous. Vivent les séminaires thésards pendant lesquels on apprend plein de choses !
Je remercie Abdelmalik, Thomas, Laetitia, Laurent et Cyril pour la bonne ambiance qui régnait dans nos bureaux.
La réalisation de ce travail n'aurait pas été possible sans des moments de détente (et réciproquement), donc merci aux vttistes, aux randonneurs, aux footballeurs et aux autres.
Je remercie ma famille pour son soutien et la bonne organisation du pot de thèse.
Je remercie les auteurs des thèses desquelles j'ai copié les remerciements.
Je vous remercie par avance, lecteurs, de ne pas vous intéresser qu'au chapitre des remerciements.
Enfin, je remercie Amélie pour tout et le reste.

Table des matières

Chapitre 1

Introduction

Sommaire

1.1 L'essor des réseaux de capteurs

De nombreuses avancées techniques et technologiques dans les domaines de la micro-électronique, de la micro-mécanique, et des technologies de communication sans fil permettent de créer de petits objets communicants équipés de capteurs à un coût raisonnable. Ces nouveaux objets appelés nœuds ou capteurs sont équipés d'une unité de mesure (le ou les capteurs), d'une unité de calcul, de mémoires et d'une radio pour communiquer. Enfin, pour l'alimentation, ces nœuds possèdent une pile ou un système de récupération d'énergie dans l'environnement.

Cette technologie rend possible le déploiement de réseaux de capteurs sans fil. Les réseaux de capteurs ont de nombreuses perspectives d'application dans des domaines très variés : applications militaires, domotique, surveillance industrielle ou de phénomènes naturels, relevé de compteurs.

Chaque application a ses propres contraintes. Dans tous les domaines, le rôle d'un réseau de capteurs est cependant à peu près toujours le même, voir figure 1.1. Les nœuds doivent surveiller certains phénomènes grâce à leurs capteurs puis envoient les informations à un puits. Le puits (ou *sink* en anglais) est un nœud particulier doté d'une puissance de calcul supérieure et d'une quantité d'énergie potentiellement infinie. Ce puits peut être connecté à Internet ou possède un lien radio de type GSM ou GPRS qui lui permet d'envoyer les informations (données ou alertes) à un centre de contrôle pour l'utilisateur final. Il peut y avoir plusieurs puits mobiles ou fixes dans un réseau mais pour des raisons de coût, il y a de toutes façons beaucoup moins de puits que de nœuds. Les réseaux de capteurs qui sont l'objet de notre étude sont les systèmes constitués des nœuds et des éventuels puits, il ne s'agit pas d'étudier comment le puits est connecté à l'utilisateur final.

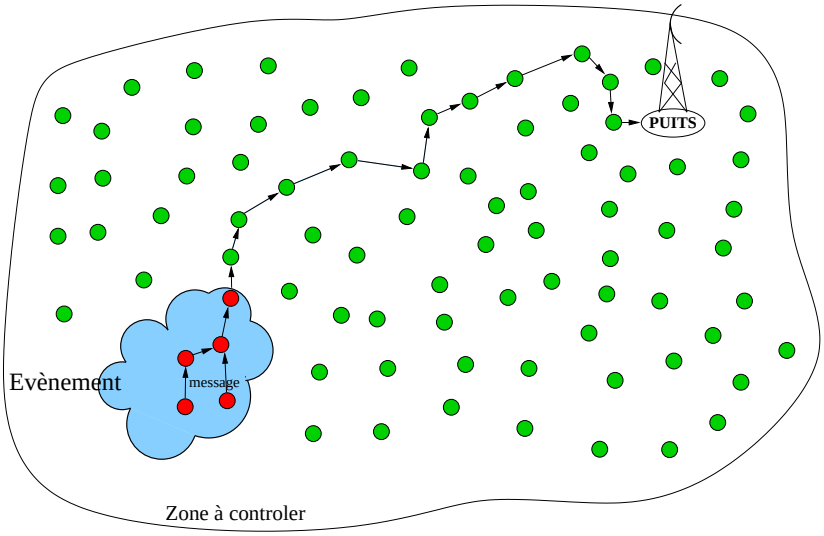


FIG. 1.1 – Schéma d'un réseau de capteur

Ce type d'application implique des contraintes communes à de nombreux réseaux de capteurs. Les capteurs sont déployés sur la zone à contrôler. Comme celle-ci est étendue et potentiellement difficile d'accès, ils doivent s'auto-organiser pour envoyer de proche en proche les messages jusqu'au puits. On ne remplace pas les nœuds qui ne fonctionnent pas ou qui n'ont plus d'énergie, il faut donc qu'ils vivent le plus longtemps possible avec une quantité d'énergie très faible étant donnée le coût élevé des piles. Pour certaines applications de surveillance où le trafic est très sporadique, on souhaite plusieurs dizaines d'années d'autonomie pour le réseau. Enfin, ces réseaux sont composés d'un très grand nombre de nœuds, plusieurs centaines voire plusieurs milliers de nœuds.

1.2 Difficultés de conception

Les acteurs industriels impliqués dans le domaine des réseaux de capteurs doivent être capables de développer rapidement des solutions fiables. Les entreprises qui conçoivent et déploient des réseaux de capteurs doivent le faire le plus rapidement possible pour faire face à la concurrence. L'enjeu économique lié à la conception des réseaux de capteurs est très important.

Cependant, concevoir un réseau de capteurs n'est pas une chose facile parce que ce sont des systèmes complexes qui combinent des caractéristiques propres aux systèmes distribués et aux systèmes embarqués. Les systèmes distribués sont difficiles à concevoir pour plusieurs raisons. Les communications entre les nœuds ne sont pas fiables, par exemple les nœuds d'un réseau communiquent à l'aide de radios. On peut difficilement définir l'état global du système, et enfin l'exécution des processus (ici les nœuds) est asynchrone. Quant aux systèmes embarqués, ils ont des ressources (calcul, mémoire) très contraintes parce qu'ils doivent respecter des contraintes de coût. Et pour concevoir les systèmes embarqués, il faut prendre en compte les interactions fortes entre le logiciel et le matériel.

En plus des contraintes cumulées des systèmes distribués et embarqués, les applications auxquelles sont dédiés les réseaux de capteurs imposent des exigences supplémentaires de fiabilité et surtout d'économie d'énergie. À cause des difficultés d'accès aux nœuds, un problème matériel ou logiciel sera plus difficile à régler dans les réseaux de capteurs. Pour ces systèmes, publier une mise à jour en

cas de problème est beaucoup plus difficile que pour les logiciels destinés aux ordinateurs de bureau. Et quand bien même celle-ci serait faisable, elle aurait un coût énergétique pénalisant pour la durée de vie du réseau. Aujourd'hui, de nombreux systèmes embarqués sont déjà contraints en énergie (téléphones, appareils photo), mais pour ces objets il s'agit simplement de maximiser le temps entre deux recharges de la batterie. Cette contrainte est beaucoup plus forte dans les réseaux de capteurs. Premièrement, on ne recharge pas un nœud qui n'a plus d'énergie parce ça coûterait aussi cher que de le remplacer par un nouveau nœud. Et deuxièmement, les clients demandent des garanties sur la durée de vie du réseau allant de 10 à 15 ans d'autonomie.

Voici, pour appuyer le besoin de méthodes de conception dédiées, quelques choix auxquels sont confrontés les concepteurs de réseaux de capteurs. Tout d'abord, il faut choisir les différents composants matériels qui constituent un nœud. Par exemple, il faut choisir un microcontrôleur basse consommation qui soit suffisamment puissant pour subvenir aux besoins de l'application. Le choix de la radio dépendra de la fréquence d'émission choisie qui, elle-même, est fonction de la portée souhaitée. Il faut également choisir ou concevoir les logiciels. Les protocoles de communication, le protocole d'accès au médium (MAC) ou celui de routage, influencent beaucoup la consommation. Le domaine de recherche qui consiste à inventer des protocoles dédiés aux réseaux de capteurs est très actif. De ces protocoles dépend directement le temps pendant lequel la radio émet ou reçoit et donc la consommation d'énergie. Pour tous les protocoles, il y a également souvent des paramètres à définir et ces paramètres peuvent interagir.

Tous ces choix dépendent bien sûr de l'application. Le choix du protocole de routage dépend du type de communication le plus couramment utilisé. Il est inutile de concevoir un protocole efficace point-à-point si ce mode de communication n'est jamais utilisé. Les choix de conception d'un réseau de capteurs dépendent naturellement de l'environnement physique dans lequel il est déployé.

Dans ce contexte, trouver une méthode pour atteindre la solution optimale en énergie est probablement hors de portée. Trouver des méthodes et outils d'aide à la conception des réseaux de capteurs, semble plus accessible. Un point important que doivent prendre en compte ces méthodes est la consommation d'énergie.

Une méthode de conception pourrait consister à construire le système complet puis à l'évaluer. Certes cette solution serait fiable puisqu'on évaluerait une solution complète mais, vu les contraintes de temps, elle paraît difficilement faisable. En effet, un réseau de capteurs peut nécessiter des solutions matérielles dédiées coûteuses à développer. Pour exécuter la même fonction, une solution matérielle dédiée peut être plus efficace en énergie. En contrepartie, elle est moins reconfigurable et sa conception conception est plus coûteuse. Ce coût élevé de conception ne permet pas de tester la solution matérielle pour décider si elle remplace avantageusement le logiciel. Il n'est pas envisageable non plus de comparer, en les testant, deux implantations matérielles différentes.

1.3 Techniques de conception existantes

On s'intéresse à une démarche de type prototypage virtuel. Elle consiste à construire un modèle du système pour pouvoir l'analyser dès les premières phases de conception.

Dans les réseaux, cette solution est déjà largement répandue. Les analyses utilisées sont principalement la simulation et des études utilisant des modèles mathématiques. Il apparaît cependant que les simulateurs de réseaux classiquement utilisés dans les réseaux ad hoc ne sont pas suffisants pour permettre une évaluation précise de la consommation d'énergie. Une des limitations est qu'ils ne prennent pas en compte la modélisation précise du matériel. Concernant les analyses mathématiques,

elles nécessitent des abstractions très fortes pour obtenir des résultats. Ces modèles ne modélisent donc pas de façon précise tous les comportements. Si les simulateurs permettent une modélisation précise des couches protocolaires puisqu'elles peuvent être simulées telles quelles en exécutant le code à embarquer sur les capteurs, il est beaucoup plus difficile de modéliser leurs comportements à l'aide de mathématiques. De telles analyses peuvent donc permettre de comprendre des problèmes locaux de collisions, ou de problèmes globaux en regardant le graphe formé par le réseau, mais il est hasardeux d'en déduire des informations sur la durée de vie du réseau.

Pour concevoir des systèmes embarqués critiques dans lesquels une erreur de conception peut avoir des conséquences dramatiques, on peut dans certains cas utiliser la vérification formelle. Cette technique permet de prouver des propriétés sur un modèle disposant d'une sémantique bien définie. L'avantage par rapport à d'autres techniques comme la simulation est l'exhaustivité. Par contre, une des limitations est le problème d'explosion d'états : les techniques de vérification exhaustive sont coûteuses et ne permettent donc pas de vérifier des systèmes de grande taille. Un autre problème pour le contexte des réseaux de capteurs est que la plupart de ces techniques ne prennent pas en compte les propriétés non fonctionnelles comme la consommation d'énergie.

L'objectif de la thèse est de proposer des méthodes de modélisation qui permettent de garantir la durée de vie d'un réseau dès les premières phases de conception.

1.4 Notre approche pour la modélisation

Pour répondre aux difficultés de conception des réseaux de capteurs, nous proposons de construire des modèles qui puissent être analysés. Nous avons d'abord proposé un modèle probabiliste. Représenter un système en utilisant des modèles probabilistes permet d'analyser l'évolution de différents indicateurs (consommation, collisions) en fonction de paramètres (taux d'erreur de transmission, nombre de messages). Nous avons modélisé et comparé différentes solutions d'un composant de réseau de capteurs à l'aide de ces modèles mathématiques. Un inconvénient de ces analyses est que le modèle est tellement simplifié qu'il est difficile de le relier à la réalité.

Nous proposons donc de construire des prototypes virtuels réalistes. Le but d'un prototype virtuel est d'évaluer dès que possible la solution que l'on souhaite déployer en utilisant un modèle opérationnel donc exécutable. Les modèles opérationnels sont conçus pour reproduire le comportement du vrai système, ils sont donc plus facilement proches de la réalité. Dans le cas des réseaux de capteurs où la difficulté principale est de concevoir des systèmes économes en énergie, le prototype virtuel doit permettre d'évaluer la durée de vie du réseau en calculant la consommation de chacun des nœuds.

Le formalisme du modèle que nous décrivons permet de modéliser finement la consommation d'énergie. De plus, notre modèle est global et précis. Il est global parce que nous ne savons pas, à l'avance, quels éléments doivent impérativement être modélisés et quels sont ceux qui sont superflus. A priori, tous les éléments ont une influence sur le comportement. Nous pensons, par exemple, que pour un réseau de capteurs l'environnement physique doit être pris en compte. Notre modèle est précis parce que les composants, notamment matériels, doivent être modélisés précisément pour que l'estimation de la consommation soit fiable. Pour ce faire, il contient des modèles opérationnels des composants matériels enrichis avec les consommations instantanées.

Pour construire ce prototype, nous avons modélisé les différents composants d'un réseau de capteurs puis nous les avons assemblés. C'est un des atouts de notre modèle, les composants sont une méthode de conception naturelle et efficace : pour concevoir un système complexe, on conçoit d'abord les différents composants puis on les assemble. De même, les composants sont pratiques pour construire des modèles globaux de systèmes complexes. Enfin, comparer deux solutions globales où seul un

composant diffère permet de comparer les deux choix possibles pour ce composant dans un contexte réaliste.

Grâce à un formalisme efficace, notre modèle est capable de simuler des réseaux de capteurs de plusieurs milliers de nœuds. La simulation est une analyse intéressante puisqu'elle permet de visualiser quelques exemples de comportements du réseau. On peut ainsi rapidement détecter des erreurs de conception ou de modélisation. Cependant, les simulations ne permettent de garantir aucune propriété.

Afin de proposer des analyses exhaustives, notre modèle dispose d'une sémantique formellement définie. Cependant, les techniques de vérification formelle ne peuvent s'appliquer sur des modèles de grande taille. Pour proposer ce type d'analyse et pouvoir garantir une durée de vie minimale du réseau, nous avons dû réduire la taille de notre modèle.

Nous avons inventé des modèles moins détaillés en nous servant de nos expériences de simulation de modèles complets. Grâce aux modèles de taille réduite ainsi construits, nous avons exhibé des scénarios de durée de vie pire-cas et montré que ce type d'analyse apporte bien des informations supplémentaires par rapport aux simulations. Nous avons prouvé des propriétés pour un modèle abstrait mais nous ne pouvons encore rien conclure pour le modèle initial plus proche de la réalité. En effet, même si nous avons utilisé le modèle détaillé pour construire le modèle abstrait, formellement aucun lien n'existe entre ces deux modèles.

Nous souhaiterions être sûrs que si la propriété est vraie sur le modèle abstrait alors elle est forcément vraie sur le modèle détaillé. Pour cela, les abstractions doivent satisfaire certaines contraintes. Comme les propriétés que nous souhaitons vérifier concernent la durée de vie pire-cas du réseau, il faut que le modèle abstrait consomme au moins autant que le modèle détaillé. De plus, nous avons construit un modèle global à partir de composants et un moyen d'abstraire ce modèle est d'abstraire certains de ces composants. Il faut alors s'assurer qu'en abstrayant un composant dans un modèle détaillé, on obtient bien un modèle plus abstrait. Nous proposons donc un cadre formel dans lequel on sait définir des abstractions qui satisfont ces deux contraintes.

Pour résumer, la technique de modélisation que nous proposons consiste à construire à l'aide de composants un modèle global et détaillé d'un réseau de capteurs. Le formalisme de ce modèle repose sur une sémantique formellement définie qui permet la modélisation de la consommation. Ce modèle est exécutable et donc simulable. Pour que des techniques de validation exhaustive soient réalisables, il est possible de faire des abstractions conservatives pour valider des propriétés faisant intervenir la durée de vie du réseau.

1.5 Cadre de la thèse

Ce document présente les travaux de thèse effectués dans le cadre d'un contrat CIFRE [1] entre l'entreprise France Télécom R&D et le laboratoire Verimag. À France Télécom R&D, nous travaillions au sein de l'équipe TECH/IDEA (anciennement TECH/ONE devenue par la suite TECH/MATIS). C'est la première collaboration entre cette équipe de France Télécom et Verimag, ces deux partenaires ne se connaissaient pas avant le début de la thèse mais ont choisi de travailler ensemble pour profiter de leur complémentarité.

La recherche effectuée dans l'équipe TECH/IDEA de France Télécom R&D vise à prévoir et construire les objets communicants de demain. Dans cette équipe, certains conduisent des recherches exploratoires, par exemple sur les techniques de communication sans fil, d'autres évaluent les terminaux existants et disponibles sur le marché. Cette équipe s'est naturellement intéressée aux réseaux de capteurs à partir de 2003. Abdelmalik Bachir a soutenu en 2007 sa thèse portant sur les protocoles

[1] Convention Industrielle de Formation par la Recherche

MAC et routage pour réseaux de capteurs, d'autres thèses sont en cours notamment sur les protocoles d'auto-organisation pour réseaux de capteurs. Pour aborder la question de la modélisation des réseaux de capteurs, France Télécom R&D a choisi de collaborer avec Verimag.

Le laboratoire Verimag propose des outils théoriques et techniques pour le développement de systèmes embarqués. Les techniques de modélisation développées à Verimag s'appuient sur une sémantique formelle qui permet de prouver des propriétés afin de garantir la sûreté de systèmes embarqués critiques. Parmi les domaines d'application des théories développées à Verimag, on peut citer l'avionique, le spatial ou les protocoles de communication. Avant le commencement de nos travaux de thèse, Verimag ne s'était jamais intéressé aux réseaux de capteurs. À Verimag, notre travail a été suivi par deux équipes : l'équipe Synchrone et l'équipe DCS (*Distributed and Complex Systems*). Comme son nom l'indique, l'équipe Synchrone est spécialisée dans les langages et outils de modélisation synchrones. L'équipe DCS conçoit, elle, des outils de modélisation asynchrone. Bénéficier de cette double compétence est particulièrement intéressant pour étudier les réseaux de capteurs qui sont des systèmes globalement asynchrones et localement synchrones, des GALS (pour *Globally Asynchronous, Locally Synchronous*).

Peu après le début de la thèse, France Télécom R&D, Verimag et d'autres partenaires ont répondu à un appel à projet RNRT[2]. Le projet ARESA [1] a été labellisé en juin 2006. Notre travail de thèse s'est donc naturellement inscrit dans le contexte de ARESA. Les autres partenaires sont les laboratoires Tima, LIG et le CITI qui travaillent respectivement sur le matériel, les protocoles de communication et l'auto-organisation dans les réseaux de capteurs ; et l'entreprise Coronis Systems. Coronis Systems déploie des réseaux de capteurs, sa présence dans le projet est donc très enrichissante puisqu'elle apporte une expérience industrielle des réseaux de capteurs.

1.6 Contenu du document

1.6.1 Résumé des contributions

Nous résumons ici les contributions présentées dans cette thèse.

- Nous avons proposé une modélisation mathématique pour analyser les performances de différentes variantes d'un protocole MAC, chapitre 3. Nous avons modélisé le système à l'aide de probabilités puis l'avons évalué, en fonction du taux d'erreur sur le canal, selon deux critères : la fiabilité et la consommation. Ce travail a été réalisé avec Abdelmalik Bachir pour mieux comprendre le fonctionnement de nouveaux protocoles MAC. Nous insistons ici sur notre contribution : la modélisation.
- Pour construire nos modèles à l'aide d'une sémantique formellement définie, nous avons étudié les techniques de modélisation de la consommation d'énergie utilisant le formalisme des automates. Nous avons comparé les différentes approches à la section 4.2.
- Nous avons conçu un simulateur dédié aux réseaux de capteurs. Ce simulateur, appelé GLONEMO, répond aux contraintes que nous nous étions fixées : il est global, précis et écrit dans un langage formellement défini, REACTIVEML. REACTIVEML est un langage conçu par Louis Mandel qui a également contribué à la réalisation GLONEMO. Ce travail est détaillé à la section 5.1.
- Nous avons montré qu'il est indispensable d'avoir un modèle d'environnement pour simuler un réseau de capteurs de manière réaliste. Section 5.2.
- Par la réalisation de ce simulateur, il nous est apparu que le langage synchrone REACTIVEML est particulièrement bien adapté à la programmation de simulateurs. Nous détaillons pourquoi

[2]Réseau National de Recherche en Télécommunications

section 5.4.
- Nous avons proposé un modèle très abstrait qui permet de vérifier des propriétés de durée de vie d'un réseau de capteurs. Ce travail montre la faisabilité des techniques de vérification formelle sur un modèle abstrait de réseau de capteurs en supposant que l'on sache obtenir de façon correcte un tel modèle. Plusieurs contributions ici : la modélisation d'un réseau à l'aide du formalisme asynchrone IF (section 6.2), la modification des outils d'exploration du graphe d'état pour obtenir les durées de vie pire cas (section 6.3).
- Nous avons transformé GLONEMO en LUSTRE pour avoir des modèles détaillés écrits en LUSTRE ce qui nous rapproche de la vérification formelle. Section 5.3.
- Nous avons proposé, section 7.2, une formalisation de la notion d'abstraction pour des modèles qui consomment de l'énergie. Notre formalisme permet de définir des abstractions de composants qui dépendent du contexte d'exécution. Si le modèle global garantit ce contexte d'exécution, il est alors possible de remplacer dans le modèle global un modèle détaillé d'un composant par un modèle abstrait pour obtenir un modèle global plus abstrait. Nous illustrons notre formalisme avec un exemple issu des réseaux de capteurs : composition d'un modèle de MAC avec des modèles plus ou moins abstraits de radios, section 7.1.

1.6.2 Plan du document

- Le chapitre 2 présente les réseaux de capteurs de façon plus précise que cette introduction. Il contient également les principales références bibliographiques concernant les travaux sur la modélisation de réseaux de capteurs
- Le chapitre 3 est une étude analytique qui nous a permis de comparer différents protocoles MAC.
- Le chapitre 4 synthétise les différentes techniques et théories de modélisation développées, entre autre, à Verimag. Section 4.2, nous étudions les différents moyens d'étendre ces techniques pour prendre en compte la consommation. La section 4.4 présente rapidement les outils de modélisation que nous utilisons par la suite. Après le chapitre 4, la partie III constitue le cœur de notre travail de thèse.
- Le chapitre 5 est dédié à nos travaux sur la simulation de réseaux de capteurs. Il détaille notre simulateur GLONEMO, puis montre l'importance d'inclure un modèle d'environnement dans un simulateur de réseaux de capteurs. La section 5.3 décrit la transformation de GLONEMO en un modèle LUSTRE. Section 5.4, nous montrons que REACTIVEML convient à la programmation de simulateurs en insistant sur les différences entre le mode de simulation à pas fixes de REACTIVEML et celui à événements discrets utilisé le plus souvent dans les simulateurs de réseaux.
- Le chapitre 6 décrit la modélisation et la vérification d'algorithmes de routage en IF. La section 6.2 explique comment nous avons modélisé de façon assez abstraite un réseau de capteurs à l'aide des primitives de communication asynchrone. La section 6.3 détaille l'algorithme de recherche du scénario de durée de vie pire cas.
- Le chapitre 7 introduit, via un exemple issu des réseaux de capteurs, un cadre formel qui permet d'écrire des relations d'abstraction prenant en compte l'énergie.
- Le chapitre 8 conclut cette thèse et présente les perspectives de recherche à ce travail.

Première partie

Contexte, travaux liés

Chapitre 2

Les réseaux de capteurs

Sommaire

Nous présentons ici les enjeux liés à la conception des réseaux de capteurs. Pour bien comprendre les difficultés de modélisation auxquelles doivent faire face les concepteurs de tels réseaux, la section 2.1 donne quelques exemples de réseaux de capteurs. Elle commence par l'introduction de quelques applications types puis explique les solutions protocolaires et matérielles que l'on trouve dans la littérature.

Nous en déduisons dans la section 2.2 quels sont les besoins pour le développement des réseaux de capteurs. Dans la section 2.3 nous présentons les différentes solutions existantes pour la modélisation des réseaux de capteurs. Pour chaque type de solutions, nous donnons quelques références bibliographiques. Étant donné le grand nombre de simulateurs par exemple, celles-ci ne seront pas exhaustives.

Après cet état de l'art, nous expliquerons notre approche pour aider à la conception des réseaux de capteurs : du prototypage rapide, réaliste et analysable, section 2.4.

2.1 Contexte des réseaux de capteurs

2.1.1 Applications

Détection de feux de forêt

Comme nous l'avons souligné en introduction, les réseaux de capteurs se prêtent particulièrement bien à des applications de remontées d'alarmes où la zone à surveiller est difficile d'accès. Les réseaux

de capteurs sont notamment pressentis pour aider à la surveillance et à la prévention des feux de forêt. L'application consiste à déployer un réseau de capteurs qui permet, d'alerter les secours en cas d'incendie, et d'évaluer le risque de départ de feux grâce à des relevés périodiques (température, humidité ...). Les nœuds d'un tel réseau sont donc équipés de capteur de température, d'humidité et infrarouge.

Sans les nouvelles solutions apportées par les réseaux de capteurs, une personne, placée sur un mirador, est chargée de surveiller la forêt en continu. Non seulement cette méthode n'est pas toujours faisable suivant la configuration de la forêt mais en plus la détection des premières flammes est tardive.

Dans cette application, le réseau a deux tâches. Une tâche consiste à envoyer périodiquement l'état de ses capteurs pour permettre la prévention. Il faut décider de la période nécessaire, envoyer trop souvent une information sur la température peut s'avérer inutile et coûteux en énergie. L'autre tâche consiste à envoyer une alarme en cas de détection de feux. Cette tâche est plus critique : il faut que l'alarme arrive au puits et qu'elle ne mette pas trop longtemps à arriver. Un puits est un nœud non contraint en énergie qui collecte les données et les envoie (via Internet par exemple) au centre de traitement.

Comme les nœuds sont inaccessibles, ou du moins on ne sait pas les localiser précisément, ils ne sont pas rechargés en énergie. Donc, une fois qu'un nœud a épuisé son énergie, il disparaît du réseau, et quand tous les nœuds n'ont plus d'énergie, le réseau est mort. En fait, le réseau est de moins en moins opérationnel au fur et à mesure que les nœuds disparaissent. Pour prolonger la vie du réseau, on peut vouloir ajouter de nouveaux nœuds, mais dans ce cas il faut un mécanisme d'auto-association pour que ces nouveaux nœuds intègrent le réseau.

Dans une telle application de surveillance d'environnement, il paraît très probable que les données des capteurs soient corrélées en temps et en espace. En effet, si un feux se déclenche, il va se propager ; donc lorsqu'un capteur détecte un incendie, il est fort probable que ses voisins aient également des alarmes à envoyer. Cette corrélation des stimuli pose plusieurs problèmes. Tout d'abord, les nœuds vont envoyer la même information plusieurs fois, ils vont donc dépenser inutilement leur énergie. De plus, plusieurs messages émis en même temps peuvent créer des collisions. Celles-ci sont souvent coûteuses en énergie puisqu'elles impliquent des retransmissions, elles peuvent même empêcher le message d'arriver à sa destination. Pour éviter ces problèmes de redondance et donc limiter les dépenses énergétiques des capteurs, certains évoquent la possibilité de n'activer que certains nœuds. L'idée est de n'activer simultanément que le nombre de nœud nécessaire pour surveiller la forêt. Un roulement doit s'effectuer pour que les nœuds actifs ne soit pas toujours les mêmes. C'est un challenge de concevoir de tels algorithmes de couverture de surface pour lequel des outils sont nécessaires.

Une autre application avec des contraintes semblables est celle qui consiste à contrôler les secousses sismiques dans les bâtiments. L'idée est de doter les bâtiments de capteurs dès la construction. Ceux-ci sont enfouis dans les fondations et détectent les différents efforts. Ils permettent donc de prévenir des affaiblissements de la structure du bâtiment dès les premiers effets. Ainsi, on pourra consolider le bâtiment avant que les fissures ne soient irréparables. Au vu du trafic, cette application a des contraintes semblables à la précédente, le trafic est très sporadique. De même, on peut difficilement remplacer les nœuds morts.

Télé-gestion de compteurs d'énergie, une application phare pour Coronis Systems

La relève automatique d'index de compteurs d'eau, de gaz d'électricité et de chaleur est une application qui émerge en Europe tandis que très largement exploitée depuis de nombreuses années aux USA en raison de la réglementation.qui a imposé un paiement mensuel des consommations réelles. La télé-relève (ou "metering" en anglais) consiste à doter les compteurs d'une fonction communicante

(PLC, radio, bus série) qui permet une supervision automatisée, au moyen d'équipements de relève mobiles (PDA, TSP, véhicule) ou bien grâce à des réseaux fixes, et ce, afin d'éviter une lecture manuelle des index.

Coronis Systems est une jeune société Montpelliéraine créée en 2000, spécialisée dans les technologies radio ultra basse consommation, longue portée et bas coût. Fort de l'expérience de ses fondateurs, Coronis Systems a élaboré et développé la technologie Wavenis.Avec les plateformes OEM développées sur la base du cœur de cette technologie, Coronis Systems se positionne en fournisseur de solutions radio-communicantes pour toutes les applications qui requièrent un faible volume d'information à transmettre, un faible trafic radio, et qui, pour les plus critiques, présentent de très fortes contraintes énergétiques associées à des conditions d'accès radio difficiles. L'un des marchés les plus matures dans ce domaine est la télé-relève des compteurs d'énergie avec des demandes considérables dans le secteur des particuliers mais aussi des industriels. La technologie Wavenis a été conçue et optimisée pour automatiser la supervision de réseaux radio de capteurs, encore désigné sous le terme générique de Machine-to-Machine (M2M).

Dans le cadre du projet ARESA, c'est avec Coronis Systems que le partenariat a été établi.

Prenons l'exemple de la ville des Sables d'Olonnes qui a été la première ville équipée de la technologie Wavenis couplée au réseau GSM pour assurer la supervision à distance des 25,000 compteurs d'eau. En partenariat étroit avec la SAUR, le réseau surveille 100% des compteurs d'eau de la ville depuis 2004. Il permet en toute priorité de relever périodiquement et plus fréquemment la consommation en eau des foyers sans avoir à mobiliser un technicien. Cette automatisation permet notamment d'établir des profils de consommateur grâce à des relèves quotidiennes au besoin, au lieu d'une à deux fois par an. Grâce à cette technologie radio bi-directionnelle et une gestion des alarmes spontanées, elle permet, en outre, de détecter des fuites d'eau dans le réseau ou chez les particuliers, dans le cas d'une consommation anormale.

Ce cas d'étude comporte de nombreuses contraintes que l'on retrouve dans les réseaux de capteurs. Essentiellement, les nœuds sont autonomes en énergie. En effet, même si l'on peut imaginer que les nœuds soient alimentés par une connexion au réseau électrique, pour limiter le coût du déploiement d'un tel réseau, ils sont autonomes. La solution consiste à remplacer les anciens compteurs purement mécaniques par un ensemble composé d'un compteur d'eau " nouvelle génération " capable de délivrer des informations (impulsions, bus série) et d'un émetteur-récepteur radio alimenté sur pile.

Cette limitation du budget énergie implique une autre contrainte typique des réseaux de capteurs : les nœuds envoient leurs messages en multi-sauts jusqu'au point de collecte quand ils ne sont pas à portée radio directe. Ils sont coopératifs : un compteur d'eau peut avoir à relayer les consommations d'autres compteurs bien que des équipements répéteurs soient préférés pour remplir cette fonction.

Généralement dans le metering, l'autonomie d'un réseau autonome doit atteindre plusieurs années pour en assurer la rentabilité. Les nouvelles exigences du marché portent désormais sur des autonomies de 10, 15 voire 20 ans. L'autonomie se calcule généralement au travers d'une analyse poussée du MTBF (Mean Time Between Failure) sur les cartes et les composants, avec des tests de vieillissement accéléré en laboratoire, une émulation du comportement radio (et donc de consommation d'énergie), et des tests de qualification poussée sur les piles en partenariat avec les fabricants de pile (SAFT par exemple).

Pour arriver à cette autonomie, pour éviter d'impacter significativement l'autonomie des modules radio couplés aux compteurs d'eau, Coronis Systems a élaboré un deuxième type de nœuds radio, disposant de plus d'autonomie énergétique (pile plus grosse, donc modules plus coûteux) et plus puissants (puissance radio 500mW) qui sont installés en extérieur sur des points hauts (lampadaires, pylônes électriques) pour favoriser la couverture radio à l'échelle d'un quartier, d'une ville. Ces nœuds sont uniquement exploités pour assurer la fonction de relais radio, leur rôle est donc de relayer les

messages des compteurs jusqu'à la Gateway Wavenis - GSM/GPRS.

Cette collaboration avec Coronis Systems nous a permis d'avoir une étude de cas concrète d'un réseau déjà déployé.

En juin 2007, Elster Group, le 1er fabricant mondial des compteurs d'énergie (1,6 milliards EUR de CA) a racheté Coronis Systems pour devenir le leader mondial des solutions metering AMI (Advanced Monitoring Infrastructure), ce qui illustre bien la pertinence de la technologie radio Wavenis de Coronis Systems pour les réseaux de capteurs.

Domotique

Un autre type d'application dans lequel les réseaux de capteurs émergent est la domotique. Dans ces applications, le réseaux de capteurs est déployé dans l'habitation. Le principe est que le réseau de capteur forme un environnement, dit environnement pervasif. Son but est de fournir toutes les informations nécessaires aux applications de confort, de sécurité et de maintenance dans l'habitat. Les capteurs sont des capteurs de présence, de son, ils peuvent même être équipés de caméras. Un tel réseau déployé doit permettre de créer une maison intelligente capable de comprendre des situations suivant le comportement des occupants et d'en déduire des actions.

Ces réseaux sont donc très hétérogènes, des éléments d'électroménager peuvent faire partie du réseau aussi bien que les ordinateurs personnels ou le routeur. Il se peut que certains éléments aient besoins d'être économes en énergie mais ça n'est pas le cas de tout le réseau. Il ne s'agit pas pour ces applications de réseaux grande échelle. Ces réseaux doivent être hautement reconfigurable : d'une part la topologie du réseau peut changer d'un jour à l'autre avec l'aménagement, d'autre part on peut avoir besoin de changer le type d'application pendant la vie du réseau.

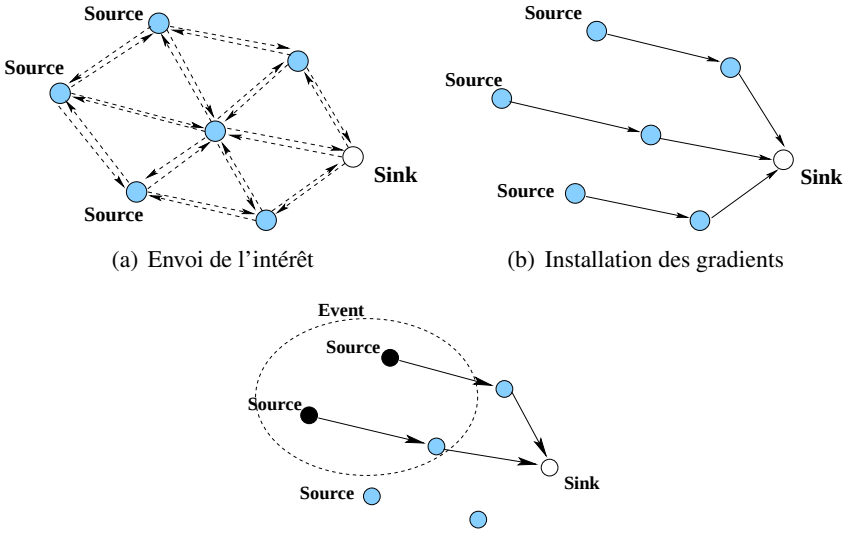
Le consortium Zigbee, s'appuyant sur les couches 1 et 2 standardisées IEEE802.15.4, répond assez bien aux contraintes de ces réseaux qui sont finalement assez éloignés des cas d'étude que l'on propose. Cependant, nos travaux sur la modélisation de systèmes économes en énergie pourront certainement servir également à ce genre d'application.

2.1.2 Protocoles de routage

Le routage consiste à trouver un chemin pour envoyer le message de la source à la destination. Dans le cadre des réseaux de capteurs, le routage doit être efficace en énergie. Pour cela, il faut bien sûr être capable de trouver une route qui ne coûte pas trop d'énergie, une route pas trop longue. Mais il faut aussi être capable de trouver ou de maintenir les routes sans dépenser trop d'énergie. Les protocoles dans lesquels on maintient à jour des tables de routage à l'aide d'envois périodiques de paquets *"hello"* ont un coût constant non négligeable. Ce coût constant est particulièrement pénalisant puisque l'on a des trafics très sporadiques : maintenir une table de routage, pour avoir des routes très efficaces, n'est pas intéressant si l'on n'utilise que très rarement ces routes.

Les protocoles de routage spécifiques aux réseaux de capteurs doivent tenir compte du type de communications induit par l'application. Outre le fait que la quantité de données échangées est très faible par rapport aux applications de types réseaux ad hoc, notons que le trafic est particulièrement prévisible puisqu'il va des nœuds vers le puits ou du puits vers les nœuds.

Nous ne faisons pas ici un état de l'art des protocoles de routages. Nous voulons seulement présenter des protocoles types des réseaux de capteurs. Nous détaillons ceux que nous avons choisis dans nos différents exemples de modélisation. Nous les avons choisi parce qu'ils sont représentatifs des protocoles de routage pour réseaux de capteurs.



(a) Envoi de l'intérêt (b) Installation des gradients

(c) Envoi des alarmes

FIG. 2.1 – Schéma du mécanisme de routage diffusion dirigée

Inondation

L'inondation ("*flooding*", en anglais) consiste à envoyer un message à tout le réseau. L'émetteur envoie le message à tous ses voisins. Chaque voisin envoie à son tour le message à tous ses voisins et ainsi de suite. Les nœuds vont donc recevoir le même message plusieurs fois de différents voisins. Pour éviter que le message ne se multiplie dans le réseau, chaque nœud ne le renvoie qu'une seule fois. Pour ce faire, chaque message envoyé en inondation a un identifiant unique. Les nœuds qui ré-émettent le message notent l'identifiant. S'ils reçoivent à nouveau un message avec cet identifiant, ils ne le renvoient pas.

Diffusion dirigée

L'algorithme de diffusion dirigée ("*directed diffusion*", en anglais) a été proposé en 2000 ([42]). Depuis, de nombreuses améliorations ont été proposées. Nous présentons rapidement son principe, le lecteur intéressé est encouragé à lire le papier [43].

Le principe de l'algorithme est le suivant : le puits envoie une requête à tout le réseau. Cette requête est envoyée à l'aide du mécanisme de routage précédent, l'inondation. Les nœuds concernés par cette requête répondent au puits en envoyant un message qui emprunte la route inverse. Pour cet algorithme, on suppose que les liens radio sont bidirectionnels. Il s'agit d'un algorithme local, les noeuds n'ont que la connaissance de leur voisinage. Pour joindre le puits, un noeud envoie son message au nœud duquel il a reçu en premier le message du puits. Chaque noeud a seulement besoin de savoir par quel voisin il pourra joindre le puits. Ce voisin ayant également connaissance d'un nœud grâce auquel il pourra joindre le puits, de proches en proches, le message arrivera à destination.

Le puits envoie donc sa requête à tous les nœuds du réseau à l'aide de l'inondation, voir figure 2.1. Les nœuds reçoivent alors le même message de plusieurs de leurs voisins. Ils mémorisent quel nœud leur a envoyé l'intérêt en premier. C'est à ce nœud qu'ils enverront les données (reçues ou mesurées localement) destinées au puits. Avoir des liens radio bidirectionnels est essentiel : on considère que

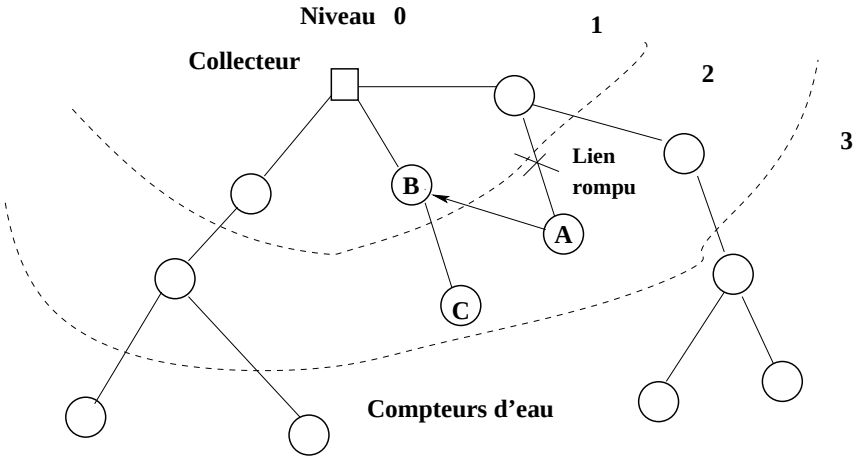


FIG. 2.2 – Routage utilisé par Coronis Systems pour un réseau de compteurs d'eau

le chemin le plus rapide du nœud au puits est aussi le chemin le plus rapide du nœud au puits. On dit que les nœuds installent des gradients, figure 2.1(b). Sur l'exemple représenté figure 2.1, l'intérêt ne concerne que les nœuds "Source", à gauche du réseau. La requête demande que l'on envoie une alarme si un événement est détecté. L'événement est ensuite détecté par trois capteurs (figure 2.1(c)). Mais seuls les deux nœuds concernés par l'intérêt vont envoyer un message. Le message est envoyé en suivant le gradient, c'est-à-dire au voisin par lequel le nœud a reçu la requête du puits. Celui-ci, même s'il n'est pas concerné par la requête, a établi un gradient et sait à quel nœud il doit envoyer le message. Et ainsi de suite, de voisin en voisin, jusqu'au puits.

Il existe aussi des mécanismes dit de renforcement pour consolider certaines routes. Différents mécanismes sont proposées, on ne les détaillera pas ici.

Le but de cet algorithme est donc de fournir un protocole de routage et d'organisation pour des applications dans lesquelles un puits pilote le réseau en envoyant des requêtes et en récupérant les données. Les requêtes peuvent être ponctuelles, par exemple pour demander le relevé de mesure à l'instant courant. Elles peuvent être périodiques, exemple : "envoyez-moi la température tous les jours". Enfin, elles peuvent dépendre des capteurs : "si la température excède un certain seuil, envoyez une alarme". Les auteurs appellent les paquets émis par le puits des intérêts ce qui montre bien que le puits est intéressé par une certaine information disponible à l'aide de son réseau.

La solution de Coronis Systems

Basée sur sa forte expérience du terrain, et afin d'optimiser la consommation d'énergie, Coronis Systems a élaboré une procédure semi-automatique d'installation des compteurs d'eau. En effet, selon la topologie et la géographie du terrain, les techniciens spécifient des conditions initiales qui peuvent varier d'un quartier à l'autre. De ce fait, l'algorithme de routage automatique pourra " court-circuiter " les 1eres itérations et converger plus rapidement vers l'appairage optimal en réduisant de ce fait le bilan énergétique lié à l'installation. Il faut noter qu'un mécanisme de routage et d'auto-organisation comme le précédent nécessite un certain nombre de messages lors des phases d'initialisation. Ces messages dépensent de l'énergie et impactent naturellement l'autonomie d'énergie des nœuds du réseau et du réseau dans sa globalité.

Néanmoins, grâce à une très grande sensibilité des récepteurs radio et une qualité de service (QoS)

définie avec les paramètres suivants (niveau dans le réseau, RSSI, énergie restante dans la pile, nb de devices attachés, Class of Device), le broadcast des messages est non seulement très sélectif pour restreintre les communications non désirables mais aussi permet d'élablir un lien radio robuste (avec 10dB de marge au dessus du seuil de sensibilité des récepteurs qui atteint -113dBm.

A ce stade de maturité des algorithmes d'auto-routage, sachant que les sites pilotes devront être éprouvés avant d'envisager des déploiements de masse à l'échelle de plusieurs centaines de milliers de points, la solution à ce jour préconisée par Coronis Systems consiste à installer et configurer l'infrastructure (gateway et répéteurs) du réseau " à la main " tout comme le font les opérateurs de télécom pour leur réseau GSM pour l'installation des " base stations ". Pour ce faire, l'agent déploie le réseau en configurant d'abord les nœuds (répteurs et capteurs à portée radio) près du point d'accès collecteur (gateway ou puits). C'est lui qui va, à l'aide d'un PC-portable (PDA avec compact flash Wavenis), donner l'ordre au capteur d'initialiser son routage. C'est ici que la connexion du capteur au réseau suit la procédure automatique selon l'algorithme de routage automatique Wavenis.

Le but de l'algorithme de Wavenis est d'avoir un routage en arbre dont la racine est le collecteur, voir figure 2.2. Wavenis définit un niveau de profondeur de l'arbre. Le puits est donc de niveau 0. Le nœud qui s'appaire à un niveau 0 devient un niveau 1. Le nœud qui s'appaire avec un niveau 1 devient un niveau 2 , etc. La profondeur maximale définie par Wavenis est 4, soit 4 sauts au maximum. L'algorithme évolue par itération. Le nœud que l'on veut configurer dans le résau envoie une requête du type : "Qui est de niveau X avec une QoS meilleure que Y ?". Seules les stations de niveau X qui reçoivent le message et qui offrent une meilleure qualité de service (QoS) que Y, répondent. Tous les autres modules s'interdisent de répondre. Si plusieurs stations répondent, le nœud va coisir son meilleur parent en fonction de la QoS (dépendant en toute première approximation de la puissance du signal reçu (RSSI, Receiver Signal Strength Indicator). A la première itération, soit X=0, ce qui revient à chercher en priorité le puits du réseau, soit le technicien aura " forcé " un niveau inférieur selon sa connaissance de l'infrastructure locale du réseau. Si le nœud ne trouve pas d'élément de niveau X avec la bonne QoS, il peut soit rechercher un élément de niveau X-1 (sans dégrader la QoS), ou bien relancer une recherche plus risquée en dégradant la QoS etc...

En cas de rupture d'un lien radio, si un nœud n'arrive plus à joindre le puits (ou collecteur), comme c'est le cas du nœud A sur la figure2.2, le nœud va chercher une nouvelle route. Cette opération se fait cette fois-ci automatiquement sans intervention humaine. Le nœud connaît son ancien rang, il ne va donc pas partir de zéro. Il va tout de suite rechercher un terminal de même niveau que celui auquel il était enrôlé auparavant, ici un terminal de niveau 1. S'il n'en trouve pas il cherchera un nœud de niveau supérieur.

La table de routage complète est mémorisée par la station de base dans un tableau. Ce nœud n'étant pas limité en énergie et mémoire, ça n'est pas un problème. Les compteurs ne mémorisent que leur route jusqu'au collecteur.

2.1.3 Protocoles d'accès au médium

Le rôle du protocole d'accès au médium (MAC pour "*Medium Access Control*", en anglais) est d'organiser l'accès au canal de communication. Pour les réseaux sans fil, cette tâche est difficile. Principalement deux raisons à cela : premièrement, le canal de communication est partagé entre les nœuds voisins ; et deuxièmement, le canal radio n'est pas fiable. En effet, un four à micro-ondes en marche ou un nouvel obstacle comme un camion peuvent changer complètement la topologie en créant des interférences ou en réduisant la portée des nœuds.

De même que pour les protocoles de routage, l'étude de nouveaux protocoles d'accès au médium dédiés aux réseaux de capteurs est un domaine de recherche très actif. Pour les réseaux de capteurs, il

faut compter avec les difficultés traditionnelles tout en minimisant la consommation d'énergie. Nous reviendrons plus loin sur les sources de consommation des nœuds, mais il faut noter qu'une radio de faible puissance consomme à peu près autant en émission, en réception et en écoute passive, c'est-à-dire quand elle est allumée pour sonder le canal. Par contre elle ne consomme pas si elle est éteinte.

Les causes de surconsommation sont donc :

- L'écoute libre ("*idle listening*"), lorsqu'un nœud a sa radio allumée parce qu'il attend un paquet où pour voir s'il y a du trafic alors qu'il n'y a rien.
- Les collisions entre trames causent des consommations inutiles. Les récepteurs et les émetteurs ont dépensé de l'énergie pour rien. Parfois les collisions entraînent des retransmissions.
- Quand un nœud reçoit une trame qui ne lui est pas destinée, il dépense de l'énergie inutilement. Ce cas de figure est appelé sur-écoute ("*overhearing*").
- Le protocole MAC lui-même peut avoir besoin de messages de contrôle qui contribuent à des pertes d'énergie. Les acquittements, par exemple, sont des paquets envoyés par le récepteur pour confirmer qu'il a bien reçu les données. Envoyer et recevoir ce paquet a un coût.

Pour réduire la consommation, le protocole MAC doit permettre de garder la radio éteinte le plus longtemps possible. Le cas parfait est de n'allumer la radio que pour recevoir et émettre le paquet de données utiles. Les nœuds ne communiquant que via leurs radios, c'est impossible en pratique. Dès lors, il existe principalement deux approches.

La première consiste à synchroniser les nœuds. SMAC [89] est typiquement un de ces protocole. Tous les nœuds s'allument en même temps, s'envoient les paquets nécessaires à maintenir leur synchronisation puis, si besoin, envoient les données. Ce protocole permet donc d'éteindre la radio régulièrement mais ça n'est peut-être pas encore suffisant s'il y a très peu de trafic. Dans ce cas, les nœuds s'allument périodiquement pour n'échanger que des paquets de contrôle afin de rester synchronisés, mais n'envoient pas de données. Le nombre de paquets émis contenant de l'information utile pour l'application devient faible par rapport au nombre total de paquets émis.

Nous détaillons plus la seconde méthode dite à échantillonnage de préambule parce que c'est celle que l'on a utilisée comme exemple. Mais tout d'abord, nous expliquons rapidement le mécanisme d'acquittement qui est utilisé dans de nombreux protocoles MAC.

Mécanisme d'acquittement

Le canal radio n'est pas fiable, des erreurs dues à des collisions ou à du bruit peuvent engendrer la perte d'un paquet émis. Pour éviter que trop de paquets soient perdus, les protocoles MAC implémentent souvent un mécanisme d'acquittement. Après avoir envoyé un paquet de données, le nœud récepteur attend un paquet d'acquittement (noté ACK pour *acknowledgment*) que son destinataire doit envoyer à la réception d'un paquet de données. Le paquet ACK est un tout petit paquet qui est émis juste après la réception du paquet de données. Comme le paquet de données a été reçu et que le paquet ACK est court, il a très peu de chances d'entrer en collision. Si l'émetteur ne reçoit pas l'acquittement, il considère donc que son paquet n'a pas été reçu. Il peut le cas échéant ré-émettre le paquet, après un certain délai par exemple. Ce mécanisme ne fonctionne plus lorsqu'un paquet est destiné à plusieurs récepteurs. En effet, dans ce cas, les multiples acquittements envoyés simultanément entreraient en collisions et le nœud émetteur du paquet ne pourrait en recevoir correctement aucun.

Protocoles à échantillonnage de préambule

Dans les protocoles à échantillonnage de préambule ("*preamble sampling MAC protocols*") les nœuds ne se synchronisent pas à l'aide de paquets de contrôle. À la place, ils sondent le canal

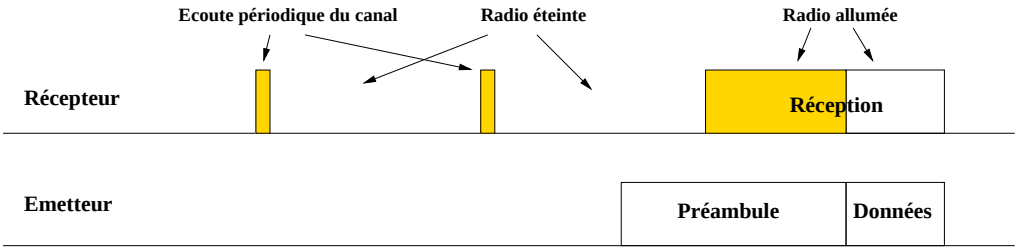


FIG. 2.3 – Protocole MAC à échantillonnage de préambule

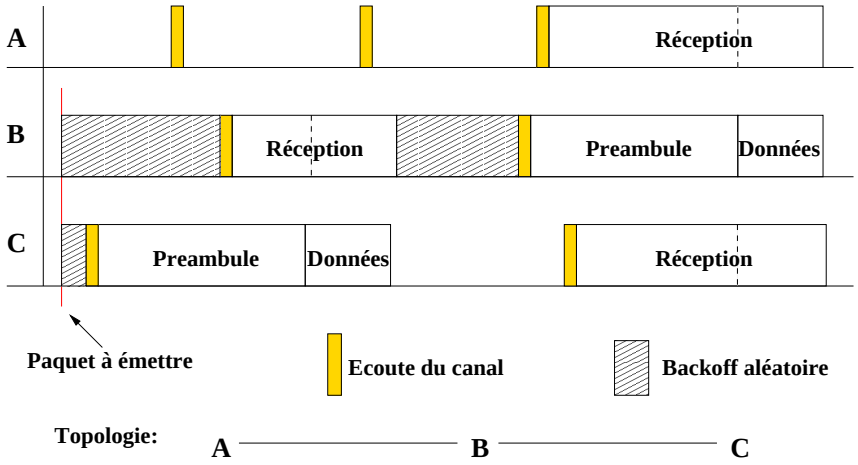


FIG. 2.4 – Protocole MAC à échantillonnage de préambule avec backoff

périodiquement pour tester la présence d'un signal. Un nœud qui veut émettre un paquet devra donc précéder l'envoi de son paquet de l'émission d'un préambule destiné à attirer l'attention du récepteur. La figure 2.3 illustre ce mécanisme avec un exemple. Les nœuds sondent donc périodiquement le canal avec une période fixe et uniforme dans tout le réseau. Notons Tw cette période en secondes. Ils peuvent cependant être décalés, un nœud ne sait donc pas quand son voisin allume sa radio pour tester la présence d'un signal, il sait seulement qu'il le fait toutes les Tw secondes. Pour lui envoyer un message en étant sûr qu'il pourra le recevoir, l'émetteur précède son paquet d'un préambule qui dure au moins Tw secondes. La longueur du préambule est telle que le récepteur sondera forcément le canal pendant cette période. En l'absence de trafic, les nœuds écoutent le canal toutes les Tw secondes et ré-éteignent leur radio. S'ils détectent un signal sur le canal ils continuent à écouter pour recevoir les données et ré-éteignent ensuite leur radio. Il se peut que le signal détecté ne soit finalement pas un préambule, dans ce cas le nœud éteint à nouveau sa radio.

Cette figure est très schématique. Pour éviter qu'un nœud envoie un message alors qu'une transmission est en cours, il doit écouter le canal avant toute transmission. Une collision peut aussi se produire si deux noeuds émettent un paquet en même temps. Ce cas de figure est probable parce que les nœuds peuvent avoir à transmettre un paquet qu'ils ont reçu en même temps ou encore parce qu'ils attendent la fin de la transmission en cours pour transmettre leurs paquets. Pour éviter que deux nœuds voisins transmettent un message au même moment, ils doivent attendre un délai aléatoire (appelé aussi *back off*) avant le début de leur transmission. La figure 2.3 montre un exemple où les nœuds B et C ont au même

moment un paquet à envoyer. Ils tirent au hasard un délai. Ici, l'attente de C vient à expiration avant celle de B. C écoute le canal et constate qu'il est libre. Il envoie donc son préambule suivi du paquet de données. Lorsque l'attente de B arrive à expiration, B sonde le canal, constate qu'il est occupé et reçoit le paquet de C. Lorsque le canal est à nouveau libre, il tire à nouveau un temps aléatoire à la fin duquel il envoie le paquet. A et C sont tous deux à portée de B, ils reçoivent donc tous les deux le paquet.

Les préambules longs sont coûteux en énergie. WiseMAC (Wireless Sensor MAC) [28] améliore ce point en réduisant dans certains cas la longueur des préambules. Afin d'éviter qu'un nœud n'ait à envoyer un préambule complet d'une longueur de Tw secondes, les nœuds mémorisent les périodes de veille de leurs voisins. Pour ce faire, les nœuds envoient dans l'acquittement la date de leur prochain réveil. Comme les nœuds se reveillent périodiquement, il est possible d'estimer la date des prochains réveils. Avec le temps et les dérives d'horloge, cette information est de moins en moins pertinente. En fait, un émetteur envoie le préambule le plus court possible tout en étant sûr de tomber pendant le réveil de son destinataire. Ce calcul prend en compte les dérives d'horloges. Si l'information est trop ancienne, le préambule envoyé doit durer Tw secondes. Il faut noter que cette amélioration est valable pour les transmissions à un seul destinataire (*unicast*) ; pour une émission vers plusieurs destinataires (*broadcast*), un long préambule, c'est-à-dire un préambule de T_w secondes, est utilisé.

De nombreuses améliorations ont été apportées à ce protocole par Abdelmalik Bachir. Nous en présentons certaines au chapitre 3. Pour plus de détails, une référence est la thèse de M. Bachir [5].

La technologie Wavenis

Wavenis implémente un protocole MAC à échantillonnage de préambule. La période de réveil, notée Tw plus haut, est programmable (10ms à 10s) et a été fixée à 1s pour les applications de *Metering*. Pour sonder le canal radio, les nœuds se réveillent pendant 500μs. Si aucune détection d'énergie n'est faite, le transceiver Wavenis rebascule immédiatement en mode veille. Dans le cas contraire, le récepteur restera allumé pendant 1,6ms pour vérifier la cohérence du signal radio entrant. S'il est incohérent, le transceiver Wavenis rebascule immédiatement en mode veille. Sinon, le signal sera traité dans le temps nécessaire à la transaction Tx/Rx.

Deux types de réseaux Wavenis ont été élaborés : réseaux non synchronisés et réseaux synchronisés. A court terme, tous les réseaux Wavenis seront synchronisés en raison des contraintes imposées par les normes radio définies par les autorités de régulation des télécoms tant en Europe qu'aux USA et en Asie. En conséquence, alors que la durée d'un préambule de réveil est de 1s dans le cas du metering pour des réseaux non synchronisés, il est drastiquement réduit à 50ms pour des réseaux synchronisés.

2.1.4 Matériel

Un nœud est composé de plusieurs parties qui consomment de l'énergie. Nous faisons un rapide résumé des différents éléments qui composent le nœud. Puis, nous présentons l'architecture de la solution de Coronis Systems. La partie capteur n'est pas générale, elle dépend de la grandeur que l'on a besoin de mesurer ; nous ne détaillons donc pas le fonctionnement du capteur.

Radio

Le module radio est particulièrement gourmand en énergie. Il est cependant possible d'éteindre complètement la radio pour économiser de l'énergie. Allumer la radio demande du temps et donc de l'énergie. Un délai est en effet nécessaire, à l'allumage, avant que la radio soit opérationnelle. Pendant ce temps, de l'énergie est consommée mais n'a pas d'utilité immédiate. Un compromis consiste à

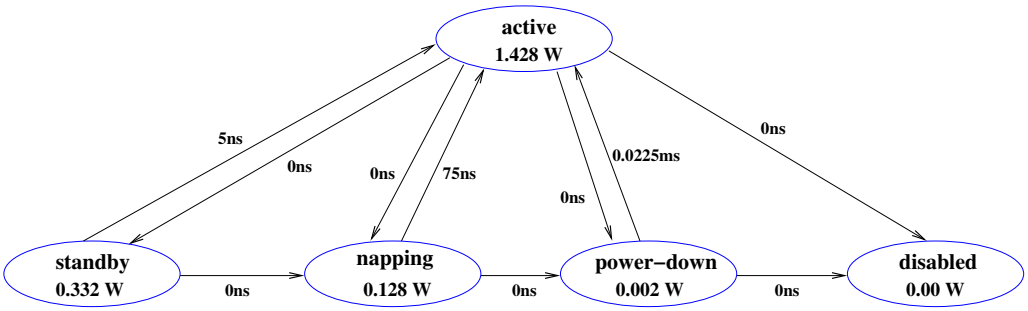


FIG. 2.5 – Modèle de mémoire DRAM (Ce modèle a été présenté par Delaluz et al., ce schéma est tiré de [24])

n'éteindre que certains éléments de la radio pour que la mise en route soit plus rapide. La radio consomme le plus quand elle est en émission ou en réception. Lorsqu'elle émet, la consommation de la radio dépend bien sûr de la puissance d'émission.

Mémoires

Plusieurs types de mémoires peuvent servir sur un système embarqué tel un nœud d'un réseau de capteurs. On peut classer les différents types de mémoires en deux catégories : les mémoires volatiles et les mémoires non volatiles. Les mémoires volatiles, type RAM (SRAM, SDRAM, DRAM, ...), perdent toutes les données si elles ne sont pas alimentées. Delaluz et al [24] proposent un modèle pour étudier la consommation d'énergie des modules DRAM. Le modèle proposé comporte 5 états, voir figure 2.5, à chaque état est associée une consommation statique et le délai pour arriver dans l'état *active* qui permet de lire ou d'écrire des données. Si l'on coupe complètement l'alimentation (état *disabled*) les données ne sont plus récupérables. Remarquons que l'automate de la figure 2.5 indique des consommations particulièrement élevées, mais il faut rappeler qu'il s'agit de 8 méga-octets de mémoire alors que, à titre d'exemple, le module de Coronis Systems ne contient que 1 kilo-octet de mémoire RAM. Parmi les mémoires non-volatiles, on compte les mémoires EPROM, EEPROM, Flash... Pour ces mémoires, les données ne sont pas perdues quand on coupe l'alimentation. En contrepartie, l'accès en lecture ou en écriture est coûteux en énergie.

Les plateformes Wavenis de Coronis Systems ne requièrent que 64kB de EEPROM ou FLASH pour implanter la stack locicielle Wavenis et 1kB de mémoire RAM pour les besoins de la stack et les paramètres liés à la calibration au moment du test industriel des cartes.

Microcontrôleur

Le microcontrôleur pilote toute l'activité du nœud. Son rôle est central. La consommation du microcontrôleur dépend de sa puissance de calcul. Plus un microcontrôleur calcule rapidement, plus il consomme. Cela va même plus loin : une tâche calculatoire effectuée lentement aura consommé moins d'énergie que la même tâche effectuée rapidement. Pour concevoir un système embarqué basse consommation, il faut donc choisir un microcontrôleur bien dimensionné, c'est-à-dire suffisamment rapide pour effectuer dans les temps les calculs demandés mais pas trop pour ne pas dépenser trop. Il est également possible de faire varier la vitesse du microcontrôleur pour qu'il soit moins consommateur d'énergie mais plus lent, ou plus rapide mais plus consommateur d'énergie. Pour faire varier la vitesse

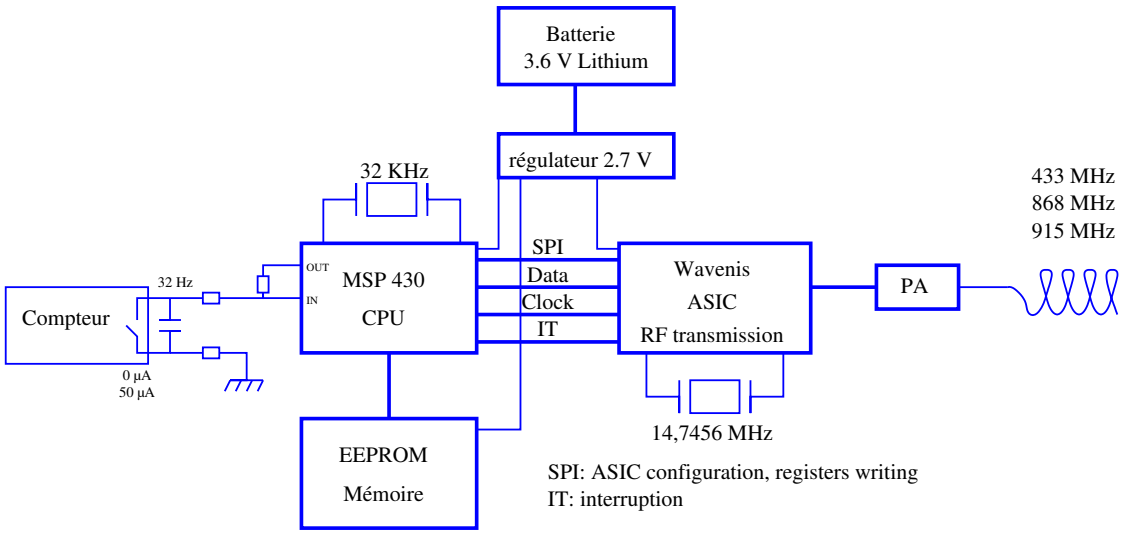


FIG. 2.6 – Architecture d'un compteur d'eau Wavenis

du microcontrôleur, une des méthodes consiste à faire varier la fréquence, ce qui permet d'ajuster la tension et donc de réduire la consommation.

L'architecture d'un nœud Wavenis

L'architecture d'un nœud Wavenis est représentée figure 2.6. Coronis Systems a développé un transceiver radio (ASIC RF) innovant pour atteindre les hautes performances RF et l'ultra basse consommation définie dans les spécifications de Wavenis. Il incorpore notamment des fonctions numériques qui optimisent les sollicitations du microcontrôleur (μC) externe. La plupart des plateformes de Coronis Systems associe le MSP430 de TI (*Texas Intrument*) en raison de sa basse consommation et des fonctions additionnelles offertes. Le tableau 1 synthétise les consommations des différents composants. Cependant ce microcontrôleur ne dispose pas de mécanisme de DVS (Dynamic Voltage Scaling). Il fonctionne à 4 MHz et consomme 1 mA en *Full Run*. En mode *Sleep*, seule une horloge fonctionne, il consomme alors 2μA. La mémoire RAM volatile est incluse dans le microcontrôleur. Pour ce nœud, elle est de 1 kB. Les plateformes de Coronis Systems contiennent également de la mémoire non-volatile : EEPROM. Ce composant permet de mémoriser les données de l'application, ici principalement les relevés des compteurs, qui seront ensuite envoyés de façon groupée au puits. Dans le cas des compteurs d'eau, l'index des compteurs est relevé toutes les heures et sa valeur est mémorisée dans l'EEPROM. Toutes les 24 heures, l'ensemble des valeurs est envoyé au puits. Le microcontrôleur pilote l'ASIC (Application Specific Integrated Circuit) chargé du module de transmission Radio Fréquence (RF).

Dans la future génération des plateformes Wavenis en cours de développement (échantillons fin Q4 2008), Coronis Systems élabore un SOC Wavenis (System On Chip) qui intègre un cœur de μC 32 bits ultra basse consommation et une version encore plus performante du transceiver RF Wavenis. L'économie d'énergie sera encore améliorée et une partie du protocole MAC sera câblée en dur dans le transceiver. La stratégie concernant l'évolution du partitionnement *hardware/firmware* a été soigneusement établie en tenant compte de tous les aspects technico-économiques et des risques liés aux designs du transceiver RF Wavenis dans un premier temps et du SOC actuellement.

Sur les plateformes actuelles, c'est bien le μC qui pilote la plupart des actions de la radio et la fonction RTC (Real Time Clock) de 32kHz qui cadence le réveil périodique de la RF. Toutes les secondes, une interruption de la RTC provoque l'allumage de la RF pour sonder le canal sans pour autant allumer tout le μC. C'est notamment sur cette mise en route que l'ASIC Wavenis est particulièrement économe en énergie. Les mécanismes décrits ci-dessous sont protégés par un brevet. L'allumage de la radio prend du temps et consomme de l'énergie. Cette dépense d'énergie est significative puisque l'opération d'allumage est réalisée par chaque nœud toutes les secondes pendant toute la durée de vie du réseau. Pour réduire ce coût, la mise en route a été optimisée et est effectuée par étapes. Les blocs fonctionnels de la RF sont allumés au fur et à mesure, selon leur caractéristique de durée d'allumage propre. Ceci a pour avantage d'éviter l'alimentation de blocs gourmands pendant toute la durée de la mise en route. La figure 2.7 représente la consommation en courant de la radio en fonction du temps pendant la mise en route. Les différentes étapes, notée de 1 à 4 sont :

1. démarrage de l'oscillateur hautes fréquences,
2. polarisation des étages hautes fréquences
3. démarrage du synthétiseur hautes fréquences
4. alimentation de toute la chaîne de réception

Pour l'émission, la stratégie d'allumage de la radio est la même, seul le courant I_4 peut changer puisqu'il dépend de la puissance d'émission. On comprend bien que par cette technique on n'arrive à la consommation maximale (I_4) qu'à la fin de la mise en route. Les différents délais en fonction des consommations sont précisés sur le tableau 2. Dans le tableau 1, la consommation pour le passage de veille à écoute correspond à une consommation moyenne. La radio est conçue pour émettre à une certaine fréquence, 433, 868 ou 915 MHz suivant la législation du pays auquel le réseau est destiné. Le composant PA pour Power Amplifier amplifie le signal radio avant son émission. Sur le tableau 1, la consommation de l'amplificateur de puissance (PA) est incluse dans la consommation de la radio en mode émission (25 mW) Les plateformes OEM sont généralement alimentées par une pile (non rechargeable) Chlorhide de Tionyle d'une capacité de 3,6A.h qui fournit une tension de 3.6 volts. Cette tension est régulée par un régulateur qui fournit une tension de 2.7 volts. Le coût de la batterie représente une part significative du coût global du nœud (environ 10-15%). C'est pour cette raison que l'on n'utilise pas de piles de plus grosse capacité. Comme la durée de vie des capteurs est de plusieurs années, il faut prendre en compte l'auto décharge des piles. Ce phénomène de vieillissement mis en avant par les fournisseurs de piles (SAFT) contraignent l'autonomie qu'à 60% de la capacité initiale de la pile.

2.2 Besoins pour la conception de ces systèmes

Nous pouvons maintenant mieux mesurer les difficultés auxquelles sont confrontés les concepteurs de réseaux de capteurs. Pour concevoir un réseau de capteurs, il faut choisir des solutions à différents niveaux. Au niveau du nœud de base, il faut choisir le matériel. Il faut aussi établir les méthodes de communication des nœuds. Il faut bien sûr que le réseau complet réponde aux besoins de l'application, c'est-à-dire qu'il remplisse son rôle.

Pour concevoir un réseau de capteurs économe en énergie, il faut s'assurer que les interactions de différents composants ne créent pas un effet néfaste qui engendrerait une surconsommation. Cependant, comme nous l'avons expliqué en introduction, tester la solution complète n'est pas possible. Il faut donc des modèles.

Composant	État	Consommation	Unité
Microcontrôleur MSP430	Veille	5.4	μW
	Actif	2.7	mW
Radio	Veille	1.35	μW
	Réception, Écoute	45.9	mW
	Émission	25	mW
	Passage de Veille à Écoute	8.02	mW
		(soit 31.28625 mJ en 3.9 ms)	
Régulateur de tension		2.7	μW
Capteur (compteur d'eau)	Pics de consommation	135	μW
	Reste du temps	0	μW

TAB. 2.1 – Consommation des composant d'un module Wavenis

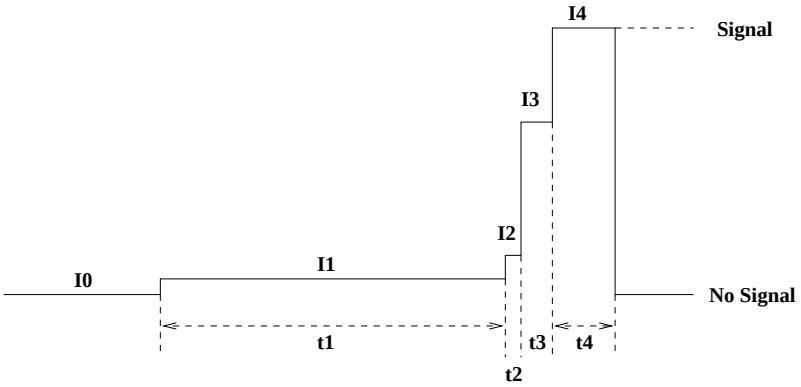


FIG. 2.7 – Consommation à la mise en route de la radio Wavenis

Tâche	Temps en μs	Consommation en mA	Consommation en mW
1	3000	0.1	0.27
2	150	0.25	0.675
3	250	11	29.7
4	500	17	45.9

TAB. 2.2 – Consommation à la mise en route de la radio Wavenis

Comment doivent être ces modèles ? Le concepteur d'un réseau de capteurs doit créer ou utiliser des solutions existantes pour des niveaux du nœud ou du réseau. Ces choix sont ensuite assemblés pour créer le réseau complet. En fait, le concepteur du réseau assemble des *composants*. Pour étudier et prévoir le comportement du réseau entier, il faut que le modèle du réseau soit *global*, qu'il prenne en compte tous les composants. À la conception du réseau, on a besoin d'étudier les interactions des différents composants. Il est intéressant de pouvoir comparer deux solutions complète où seul un composant change (le protocole MAC, par exemple). Un modèle à composants est un modèle dans lequel on peut facilement remplacer un composant par un autre.

Une solution globale pour la conception d'un réseau de capteurs peut nécessiter du matériel dédié. Étant donnés les coûts de conception de matériels dédiés, on ne peut pas concevoir un composant matériel avant de le tester dans la solution globale. Un outil de conception d'un réseau de capteur doit donc intégrer des modèles de composants matériels.

Ces modèles de composants doivent être suffisamment précis pour modéliser fidèlement la réalité. En plus d'être fidèle à la réalité, le modèle obtenu doit être analysable en pratique. En effet, un modèle très précis mais qui, pour des raisons pratiques, serait trop complexe pour être analysable, n'aurait aucun intérêt.

Pour les réseaux de capteurs, il y a un problème d'échelle qui ne dépend pas des analyses souhaitées. En effet, un réseau de capteurs est un système complexe qui comporte un très grand nombre de nœuds, il faut donc veiller à contrôler la complexité d'un modèle d'un tel système.

Les analyses que nous souhaitons faire ne sont pas forcément celles utilisées dans les approches existantes. Nous présentons notre approche section 2.4.

Quelles que soient les analyses souhaitées, pour développer un réseau de capteurs, il faut des informations sur la consommation d'énergie. On a également besoin d'informations sur le temps. Tout d'abord, il peut être confortable d'avoir une notion de temps pour estimer l'énergie consommée. De plus, pour économiser de l'énergie, on est parfois amené à allonger certaines tâches. En effet, le temps est souvent moins critique que l'énergie dans les réseaux de capteurs. Puisque des tâches sont retardées ou durent plus longtemps, il faut s'assurer que les exigences temporelles de l'application sont respectées. Le temps doit donc être pris en compte.

2.3 Méthodes usuelles de modélisation pour les réseaux de capteurs

Le prototypage virtuel que nous proposons et qui fait référence aux modèles utilisés dans les systèmes sur puce, n'est pas une idée nouvelle pour les réseaux de capteurs, ni pour les réseaux ad hoc en général. Cette méthode de développement consiste à créer un modèle du système que l'on veut développer. Une fois ce modèle créé, on obtient un **prototype** du système. Ce prototype virtuel est ensuite analysé et modifié jusqu'à ce qu'il respecte les spécifications du problème. On peut alors implémenter le prototype. Le prototypage virtuel permet de développer rapidement des systèmes complexes.

Nous faisons dans cette section un état de l'art des méthodes de prototypage virtuel utilisées pour modéliser les réseaux de capteurs. Tout d'abord nous présentons quelques simulateurs de réseaux. Même s'il existe des simulateurs spécifiques aux réseaux de capteurs, les simulateurs classiques sont parfois utilisés dans le contexte des réseaux de capteurs. Nous présentons aussi comment certains simulateurs prennent en compte une modélisation de l'environnement du réseau afin d'obtenir des simulations plus réalistes. Nous présenterons ensuite les méthodes d'analyses basées sur des modélisations mathématiques. Enfin, nous citerons les quelques travaux existants de modélisation formelle de réseaux de capteurs.

2.3.1 Les simulateurs de réseaux

Parmi les simulateurs de réseaux, on trouve les simulateurs de réseaux classiques, ces simulateurs ont été conçus pour modéliser et simuler des réseaux classiques : les réseaux filaires, les réseaux sans fil voire les réseaux ad hoc. Pour ces réseaux, la consommation d'énergie n'était pas encore la préoccupation majeure.

Simulateurs de réseaux généraux

Un des simulateurs de réseaux les plus utilisés est NS2 (The Network Simulator) [66]. À l'origine de NS2 est le projet VINT : en 2000, Breslau et al. [19] estiment qu'il faut un seul simulateur de réseaux pour la communauté scientifique. Ce souhait donne naissance au projet VINT. L'intérêt d'avoir un simulateur unique est essentiellement la facilité à comparer différentes solutions. NS2 est un simulateur à événements discrets. NS2 propose quatre niveaux d'abstraction (d'après [19]) ce qui permet d'adapter le simulateur aux différents intérêts. En effet, certains souhaiteront des informations bas-niveau, pour étudier par exemple l'effet d'un système multi-antennes alors que d'autres étudient les protocoles de routages et ne souhaitent que des informations au niveau réseau. Un simulateur qui calcule des informations trop précises passera moins bien à l'échelle qu'un simulateur dédié au routage. NS2 était d'abord destiné aux réseaux filaires ce qui explique certainement la simplicité des modèles de propagations radio de ce simulateur. Malgré les différents niveaux d'abstraction, NS2 est essentiellement utilisé par les gens qui s'intéressent aux protocoles de routage et/ou aux protocoles d'accès au médium. Le code de NS2 est ouvert, de cette façon, chacun peut ajouter sa contribution. Il existe donc une large bibliothèque de protocoles MAC et routage, ce qui permet en effet de comparer facilement ses dernières avancées avec l'état de l'art. Avec l'intérêt grandissant de la communauté de recherche en réseaux pour les réseaux de capteurs, NS2 est naturellement resté un simulateur très utilisé. Pour NS2, dans le cadre du projet VINT, l'outil Nam (Network ANimator) [30] permet de visualiser les simulations. Nam crée la visualisation après les simulations. NS2 génère des traces qui détaillent tous les événements de la simulation ; Nam utilise ces traces pour générer le "film" de la simulation. Générer la visualisation a posteriori permet de ne pas surcharger le coût de la simulation du coût de la sortie graphique. Mais une visualisation a posteriori ne permet pas d'agir sur la simulation pendant son exécution à partir des événements observés.

Cependant, selon nous, NS2 n'est pas le simulateur parfait. Tout d'abord, la modélisation sommaire des propagations radio est un point faible pour des réseaux dans lesquels toutes les communications sont des communications radio. En partie à cause des multiples contributions qui enrichissent NS2, ce simulateur est devenu très compliqué. Selon un utilisateur, il ne faut pas moins de six mois d'apprentissage pour maîtriser NS2. De plus, les composants qui constituent un réseau, ou les nœuds d'un réseau, ne sont pas toujours bien séparés dans NS2. Même s'il est logique, pour implémenter des protocoles qui contiennent des optimisations inter-couches ("cross-layer design"), de devoir modifier du code à différents niveaux, dans NS2 les couches n'étant pas clairement séparées, l'implémentation devient vite illisible. De plus, il est très difficile d'estimer le rapport entre les simulations effectuées par NS2 et la réalité. En effet, des améliorations sont proposées par de multiples contributeurs mais il est difficile de dire si elle sont valides par rapport à la réalité. NS2 est capable de simuler un réseau d'une centaine de nœuds maximum. Enfin, NS2 ne propose pas de modélisation de la consommation d'énergie. Pour estimer l'efficacité en énergie de leurs protocoles, les chercheurs font des abstractions assez brutales comme par exemple : compter le nombre de paquets émis/reçus, compter le temps pendant lequel la radio est allumée. De même, NS2 n'étant pas un simulateur dédié aux réseaux de capteurs, il ne propose pas de modèle d'environnement. Pour faire des simulations avec des communications on

peut choisir d'avoir un flux (TCP/IP, UDP...) ou bien on considère qu'une loi de Poisson modélise bien l'émission de paquets au niveau d'un nœud. Pour les réseaux de capteurs, ces deux approches sont tout à fait discutables.

Les auteurs de NAB (Network in A Box) [29], souhaitent répondre aux manquements de NS2 en proposant un simulateur qui passe à l'échelle, contenant un outils de visualisation inclus, et une architecture propre et flexible. Ils utilisent OCAML pour programmer leur simulateur. OCAML dispose d'un typage fort qui oblige à écrire les programmes de manière plus propre. Notamment grâce à OCAML, NAB passe mieux à l'échelle que NS2. Un outil de visualisation basé sur l'outil graphique de OCAML est disponible. NAB ne propose ni modèle d'environnement, ni modélisation de l'énergie. NAB n'est plus disponible.

Riley et al [35] proposent pour simuler des réseaux de grande taille, la *parallel discrete event simulation*. Autrement dit, ils proposent de distribuer les calculs du simulateur à événements discrets sur plusieurs machines. Le simulateur GTNetS [75] (Georgia Tech Network Simulator) de l'université de Georgia Tech est conçu avec ce paradigme. GTNetS n'est pas non plus dédié aux réseaux de capteurs.

Nous présentons maintenant aux simulateurs de réseaux de capteurs.

Simulateurs dédiés aux réseaux de capteurs

Avrora [84] est un simulateur pour réseaux de capteurs. Pour obtenir une simulation fiable, ce simulateur est *cycle accurate* ce qui signifie qu'il simule pour chaque nœud toutes les instructions qui s'exécutent sur ce nœud. Certes, cette approche offre une modélisation particulièrement précise mais on ne peut pas espérer qu'elle passe à l'échelle. Même dans le domaine des systèmes sur puce, une modélisation cycle-précis s'avère trop détaillée pour être utilisable en pratique. De plus, Avrora est écrit en Java et chaque nœud est implémenté par un *thread* (processus léger) Java ce qui impose une difficulté supplémentaire : il faut synchroniser les threads pour s'assurer qu'un nœud ne reçoive pas un paquet avant que son voisin ne l'ait envoyé.

Atemu [72] est aussi un simulateur pour réseaux de capteurs très précis. Ici, les nœuds sont émulés, c'est-à-dire que l'on exécute le code binaire de chaque nœud. Par contre les transmissions sans fil sont simulées. Une fois encore, il nous semble qu'une telle approche est trop précise. On pourrait cependant imaginer que l'on émule un nœud et que l'on simule moins finement les autres mais on ne peut émuler tous les nœuds d'un réseau comportant plusieurs centaines de nœuds.

TOSSIM [52] est le simulateur dédié à TinyOS. TinyOS [53, 83] est un système d'exploitation dédié aux réseaux de capteurs. TOSSIM essaye de tirer parti du mode d'exécution de TinyOS pour proposer un simulateur efficace et fiable. Le mode d'exécution de TinyOS est dirigé par les événements, ce mode d'exécution se calque bien sur un simulateur à événements discrets. TOSSIM contient un modèle abstrait de chaque composants du matériel d'un nœud. Pour une simulation TOSSIM, on utilise le même code que celui destiné au nœud cible mais cette fois-ci TOSSIM émule le comportement du matériel en utilisant les modèles des composants. Simuler exactement le code qui tournera sur les nœuds permet de tester l'implémentation finale des algorithmes. Cette notion d'abstraction du matériel est tout à fait intéressante, cependant TOSSIM dans sa première version ne permet pas d'estimer l'énergie consommée.

PowerTOSSIM [79] est l'extension de TOSSIM qui contient un modèle de consommation d'énergie. Pour les valeurs de consommation, les auteurs se sont basés sur le Mica2 (nœud développé à l'université de Berkeley). Les auteurs connaissent les consommations des différents composants de ce nœud suivant leurs états. Il faut donc connaître l'état de chaque composant d'un nœud pendant la simulation. Grâce au modèle de simulation basé sur TinyOS, on connaît immédiatement l'état des composants autres que le microcontrôleur puisque les changements d'états correspondent à des événements dans TinyOS et

donc dans TOSSIM. Plusieurs composants du nœud sont parfois abstraits dans TOSSIM par un seul composant. L'estimation de la consommation du microcontrôleur est plus délicate : il faut instrumenter le code pour être capable de compter le nombre d'exécution de chaque bloc d'instruction, et il faut faire correspondre chaque bloc d'instruction avec son code en assembleur. Lors de la simulation, on note le nombre de passage, d'exécution, de chaque bloc d'instructions. Sachant combien d'instructions élémentaires contient chaque bloc de base, on en déduit le nombre d'instructions effectuées par le microcontrôleur et donc sa consommation. Cette approche est intéressante mais elle ne permet pas de varier la précision du modèle de consommation. Enfin, les simulateurs TOSSIM et PowerTOSSIM ne conviennent que pour des applications écrites en TinyOS.

Les auteurs de Worldsens [32, 21] font le même constat que nous : il faut des outils de prototypage virtuel pour aider à la conception de réseaux de capteurs. Leur solution est de relier deux simulateurs : WSNet et WSim. WSNet est un simulateur à événements discrets à base de composants qui permet de simuler l'application, les protocoles de communication et les communications radio. Plusieurs modèles de radio sont disponibles dont certains relativement précis. WSim est un simulateur bas niveau, cycle-précis, dédié au matériel. Chacun de ces deux simulateurs peut être utilisé seul et Worldsens permet de les utiliser conjointement. En fait, selon les auteurs, leur simulateur s'utilise comme suit : on commence par faire les choix logiciels à l'aide de WSNet puis on fait les choix de matériel que l'on implémente sur WSim et Worldsens permet alors de simuler conjointement le matériel et le logiciel. Les deux simulateurs se synchronisent par rendez-vous. C'est WSNet qui gère les communications radio. Si WNet passe assez bien à l'échelle, WSim est à un niveau de détail auquel on ne peut espérer simuler un réseau d'une centaine de nœuds. La modélisation de l'énergie n'est pas explicitée mais à ce niveau de détail il doit être possible de l'estimer précisément. Cette approche a l'inconvénient d'offrir peu de granularité en termes de niveaux d'abstractions : on passe d'un simulateur haut niveau, tel les simulateurs de réseaux classiques, à un simulateur cycle précis.

Un autre état de l'art des simulateurs pour réseaux de capteurs est effectué dans [27].

Simulateurs prenant en compte un modèle d'environnement

Tous ces simulateurs sont dédiés aux réseaux de capteurs et cherchent donc à estimer de façon précise la consommation tout en restant capables de simuler des réseaux d'assez grande taille. Cependant, nous faisons remarquer section 5.2 qu'il faut inclure un modèle d'environnement pour que les simulations aient un trafic réaliste. En effet, dans un réseau de capteurs, le trafic dépend des capteurs et donc de l'environnement. Les simulateurs que nous venons de présenter ne contiennent pas de modèle d'environnement et il est difficile de savoir avec quel trafic sont générées les simulations. Nous faisons maintenant un état de l'art des simulateurs qui prennent en compte un modèle d'environnement.

Sridharan et al [82] proposent de connecter le simulateur d'environnements Matlab [62] au simulateur de réseaux de capteurs TOSSIM. L'expérience qu'ils ont faite est la simulation d'un réseau de capteurs dédié au contrôle de la structure d'un bâtiment. Pour cette application, Matlab fournit un bon modèle de l'environnement ; cependant Matlab convient beaucoup moins à la simulation d'environnements qui ne suivent pas des équations différentielles.

Outre ce lien entre Matlab et TOSSIM, les simulateurs de réseaux qui incluent un modèle d'environnement le font souvent en faisant une analogie entre la propagation des ondes radio et la propagation du phénomène à observer.

SensorSim [69] est un simulateur à événements discrets pour réseaux de capteurs. Pour simuler l'environnement, les nœuds de ce simulateur contiennent en plus du module radio un module *capteur* qui dispose d'une couche protocolaire ("*Sensor protocol stack*") qui reçoit des message venant d'un canal *capteur* ("*Sensor channel*"). La différence principale entre ce module *capteur* et le module radio

est que les nœuds ne peuvent que recevoir sur ce canal, ils ne peuvent pas émettre. Pour effectuer une simulation, il faut déterminer les caractéristiques de propagation du phénomène à observer sur le canal *capteur*.

J-Sim [81] s'inspire de SensorSim pour la modélisation de l'environnement. J-Sim contient donc un canal *capteur*. Le phénomène à capter est créé par un nœud particulier appelé "*Target node*", ce nœud envoie périodiquement des stimuli qui se propagent sur le canal "capteur". Deux modèles de propagation sont implémentés : un modèle de propagation sismique et un modèle de propagation acoustique.

Downard, dans [26], étend NS2 pour simuler des réseaux de capteurs. Ici aussi, l'auteur fait une analogie entre le canal *capteur* et le canal radio, mais il va plus loin dans cette analogie. Un canal radio est créé pour chaque phénomène. Il y a deux types de nœuds. En plus des nœuds classiques (les capteurs du réseaux), qui communiquent sur le canal radio et reçoivent des informations des canaux *capteurs*, Downard crée des nœuds PHENOM. Les nœuds PHENOM ne "communiquent" que sur un canal *capteur*. Ces nœuds émettent régulièrement un paquet indiquant leur présence. Les nœuds PHENOM disposent également d'une couche MAC et d'une couche routage. Pour éviter des collisions entre phénomènes qui ne seraient pas réalistes, la couche MAC utilisée est parfaite. Le routage détermine quand et avec quelle fréquence les nœuds PHENOM envoient les messages indiquant leur présence. C'est le protocole de routage qui détermine la propagation du phénomène. Selon nous, utiliser le modèle de propagation des ondes radio pour simuler la propagation de phénomènes quelconques n'est pas forcément très réaliste. Certains phénomènes à détecter, comme une cible par exemple, se déplacent mais ne se propagent pas et dans ce cas l'analogie avec la radio atteint déjà ses limites. Enfin, le modèle de propagation radio est une partie très coûteuse en calculs dans un simulateur de réseaux, dupliquer ce composant paraît inutilement coûteux. Dans le cas de cette dernière approche, on surcharge le simulateur de nouveaux nœuds comportant protocoles MAC et routage uniquement pour le composant environnement de la simulation.

2.3.2 Modélisation pour l'évaluation de performance

Ce type de modélisation vise à avoir des modèles analytiques. Ces modèles mathématiques doivent être facilement analysables. Une fois le modèle mathématique proposé, de nombreuses analyses sont possibles, ces analyses peuvent s'intéresser au cas moyen comme le font les simulations mais elles permettent aussi d'exhiber le pire cas du modèle par rapport à la propriété à valider. Ces modèles permettent ainsi une meilleure compréhension du système.

Une des techniques de modélisation pour l'évaluation de performances consiste à modéliser le système en utilisant des probabilités. Kleinrock et Tobagi [47], [85] ont fait des travaux précurseurs en modélisant les communications sans fil à l'aide de probabilités. Leur but était l'étude de protocoles MAC. Ce type d'étude peut également être utile dans le cas des réseaux de capteurs. Par exemple, chapitre 3, nous modélisons un canal radio entre deux nœuds avec l'hypothèse suivante : chaque bit envoyé a une probabilité constante et indépendante du temps d'être transmis correctement. Sous cette hypothèse nous comparons différents protocoles MAC afin de comparer leur robustesse et leur efficacité en énergie.

Demirkol et al [25] proposent une modélisation analytique de l'environnement. L'application qui les intéresse est la détection d'intrusion. Afin d'obtenir une évaluation de performance cohérente avec le déplacement de la cible, ils modélisent à l'aide de probabilités son déplacement dans le champ de capteurs. En fonction, entre autres, de la vitesse de la cible et du rayon de détection des capteurs ils obtiennent un modèle probabiliste du trafic.

De nombreux composants d'un réseau de capteurs peuvent donc être modélisés afin d'évaluer

leurs performances. Cependant, il est difficile voire impossible d'arriver à de tel modèles sans des simplifications drastiques. La question qui se pose alors est, quel lien existe-t-il entre le modèle abstrait sur lequel on a des résultats et la réalité ? Les auteurs de [70] proposent pour répondre à cette question de modéliser un ensemble de systèmes types à l'aides de différentes techniques et outils. Les systèmes considérés ne sont pas des réseaux de capteurs mais des systèmes temps réel distribués dans lesquels des unités de calculs exécutent des tâches concurrentes et communiquent. Le but de l'analyse est d'estimer les temps d'exécutions pire cas (*WCET*). Ils proposent un ensemble de systèmes types qui servent de bancs d'essais. Ensuite, ils modélisent et analysent ces systèmes à l'aide de différentes techniques chacune implémentant des abstractions propres. En outre, une modélisation à l'aide d'automates temporisés explore de façon exhaustive tous les cas possible du système. Cette analyse, plus coûteuse que les autres, sert de référence. Le résultat est le suivant : les différentes méthodes d'analyses fournissent des pire cas pessimistes que l'on ne peut pas comparer. Autrement dit, il est impossible de savoir à l'avance pour un cas d'étude particulier quelle analyse et donc quelles abstractions fourniront le résultat le plus fidèle. Ce résultat montre à quel point il est difficile de relier un résultat obtenu par une modélisation abstraite avec la réalité. Les auteurs recommandent aux concepteurs de tels systèmes d'utiliser plusieurs modélisations et abstractions différentes afin d'accroître la confiance en leurs résultats.

Les réseaux de capteurs sont également des systèmes complexes pour lesquels nous pensons qu'il faut être très prudent avec des modélisations mathématiques trop abstraites. Pour être utilisables en pratique, les modélisations mathématiques peuvent s'avérer éloignées de la réalité. Il est en effet difficile d'exprimer le comportement complexe d'un réseau de capteurs à l'aide de modèles mathématiques principalement pour deux raisons. Premièrement, un réseau de capteurs est un système qui comporte de nombreux éléments logiciels et matériels qu'il faut modéliser. De plus pour modéliser un réseau de capteurs, on est amené à modéliser des comportements physiques (communications radio par exemple) pour lesquels les modèles mathématiques existants sont complexes.

2.3.3 Modélisation pour la vérification formelle

Nous expliquons d'abord ce que l'on appelle vérification formelle. Puis, nous présentons les quelques approches de modélisation formelle de réseaux de capteurs.

Vérification formelle

On appelle vérification formelle, les techniques permettant de prouver une propriété concernant le comportement d'un programme. Si le programme ne respecte pas la propriété, ces techniques peuvent parfois fournir un contre-exemple ou donner des indications pour en trouver un. Le model-checking est une technique de vérification formelle dans laquelle on considère une représentation finie de l'ensemble des exécutions possibles du système. C'est le cas des programmes à mémoire bornée. En explorant la représentation finie du système, on peut vérifier que toutes les exécutions possibles sont correctes. Cependant, cette technique se heurte au problème d'explosion de la taille des modèles. Des abstractions permettent de réduire la taille de la représentation finie. Une abstraction doit être conservative pour qu'une preuve sur le modèle abstrait soit valide pour le modèle détaillé. La vérification formelle est utilisée pour les systèmes embarqués critiques, comme l'avionique ou les centrales nucléaires. Elle est également utilisée dans la conception de circuits et pour les protocoles de communication. Étant donnés son succès et l'intérêt grandissant pour les réseaux de capteurs, il est naturel de la retrouver pour la modélisation des réseaux de capteurs. Nous présentons ici les quelques approches de modélisation formelle de réseaux de capteurs dont nous avons connaissance.

Réseaux de capteurs et vérification formelle

Watteyne et al proposent dans [88] un protocole MAC temps réel pour réseaux de capteurs. Ce protocole doit garantir la durée maximum d'envoi d'un paquet d'un nœud vers le puits. Après une première validation par simulations, les auteurs vérifient la propriété à l'aide de l'environnement de modélisation et de validation formelle UPPAAL [86]. La vérification de la propriété est faite pour 7 topologies contenant jusqu'à 6 nœuds. Pour chacune des topologies étudiées, on vérifie que, quel que soit le nœud émetteur, le message arrive au puits à temps. Notre travail vise plutôt à vérifier des propriétés portant sur la durée de vie du réseau vu que l'énergie est critique pour les réseaux de capteurs. Cet exemple montre en tous cas l'intérêt des approches formelles pour les réseaux même s'il n'est pas spécifique aux réseaux de capteurs.

Les auteurs de [22] modélisent un nœud programmé en nesC avec HyTech [40]. nesC [36] est un langage C dédié qui permet d'écrire les applications TinyOS. HyTech est un outil qui permet de modéliser et d'analyser des automates hybrides. Les auteurs ont d'abord modélisé le comportement d'un nœud à l'aide d'automates hybrides. Grâce à ce modèle, il ont pu vérifier des propriétés de sûreté avec HyTech. La propriété vérifiée ne dépend pas non plus de l'énergie dans cette expérience. Puis, à l'aide de SHIFT [80], ils simulent un réseau d'automates hybrides qui correspond à leur réseau de capteurs. SHIFT est l'outil qui permet de simuler des automates hybrides programmés en HyTech. Lors de la simulation, l'énergie dépensée par les nœuds est une donnée qu'ils calculent ; mais il s'agit bien de simulation même si la simulation se fait sur un modèle d'automates.

Les auteurs de [67] se positionnent par rapport aux simulateurs de réseaux : ils souhaitent montrer que Real-Time Maude [68] est un bon outil pour modéliser, simuler et analyser un réseau de capteurs. Leur cas d'étude est un algorithme de couverture de surface (OGDC). Cet algorithme a déjà été simulé auparavant avec le simulateur NS2. Ölveczky et al. affirment qu'ils obtiennent en simulation avec Real-Time Maude les mêmes résultats que ceux obtenus avec NS2. Cependant leur modélisation se place à un niveau d'abstraction assez élevé qui permet de modéliser OGDC mais où les collisions et autres erreurs dues au canal radio ne sont plus prises en compte. Certes une telle approche évite le coût de l'apprentissage d'un simulateur mais ici il semble que la modélisation en Real-Time Maude soit moins précise que celle de NS2. L'autre intérêt d'utiliser RTMaude est de pouvoir faire de la vérification sur le même modèle puisque Real-Time Maude est pourvu d'un model-checker. Ils prouvent sur un réseau de 6 nœuds que toute la surface est couverte au premier passage de l'algorithme. Là aussi la propriété ne concerne pas du tout l'énergie. La consommation n'est d'ailleurs pas modélisée dans cette expérience.

Les expériences existantes de vérification formelle appliquée aux réseaux de capteurs restent isolées et sont peu nombreuses. De plus, la durée de vie qui est la préoccupation principale dans les réseaux de capteurs n'est pas prise en compte.

2.4 Notre approche : du prototypage rapide réaliste et analysable

Pour développer des réseaux de capteurs, il faut des modèles que l'on soit capable de construire rapidement. Le problème principal des réseaux de capteurs est leur durée de vie. La batterie des nœuds étant limitée et ayant une faible autonomie, il faut contrôler au mieux la consommation des capteurs pour augmenter la durée de vie du réseau. Pour les premières phases de conception du réseau, une estimation fiable et précise de la consommation d'énergie est nécessaire. Il faut donc un formalisme qui puisse modéliser le matériel et qui soit suffisamment expressif pour modéliser des comportements complexes.

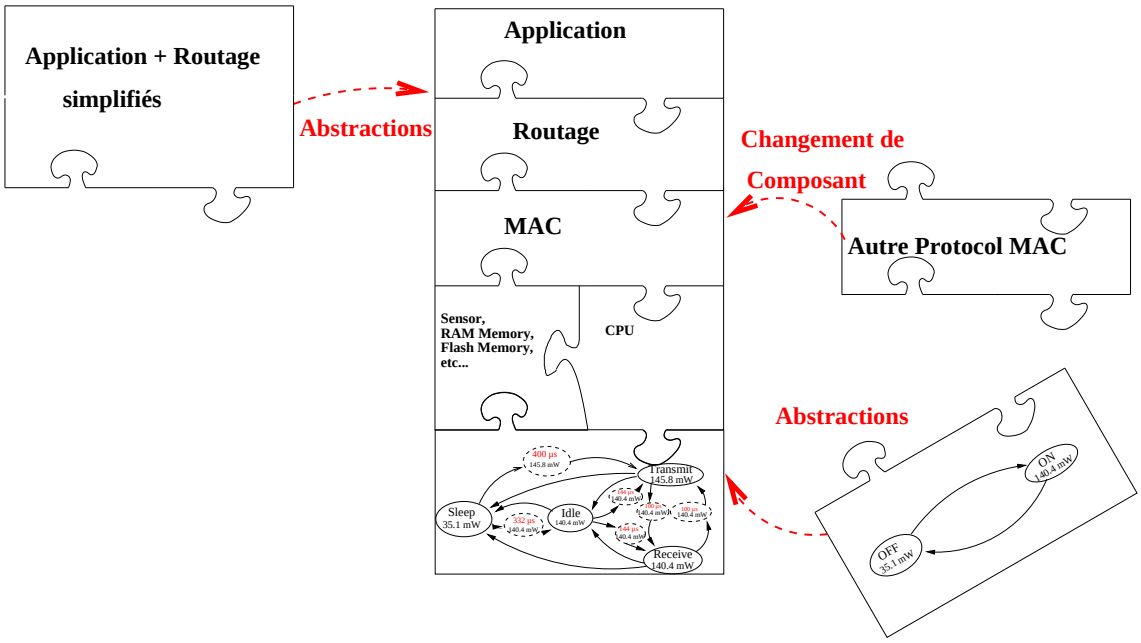


FIG. 2.8 – Modèle à composants et abstractions

Nous proposons également des modèles à composants. Par modèle à composants, nous entendons un modèle qui se construit tel un puzzle (figure 2.8) où les briques seraient des composants. Un tel modèle se construit plus facilement : il suffit de modéliser les différents composants un par un et de les assembler ensuite. Un autre intérêt du modèle à composants est qu'il permet de comparer les composants : en ne changeant qu'un composant du modèle global, on obtient deux modèles globaux où seul un composant a changé. Comparer ces deux modèles revient à comparer les deux composants dans un environnement réaliste.

De plus, nous souhaitons des modèles exécutables et ceci pour plusieurs raisons : selon nous, il est plus facile de valider un modèle lorsque l'on peut l'exécuter. En effet, l'exécution peut permettre de trouver rapidement des erreurs de modélisation ou de conception. L'exécution du modèle donne également un moyen de comparer le modèle à la réalité. Enfin, exécuter un modèle revient à faire des simulations ce qui a un intérêt en soi.

Cependant, la simulation a ses limites, notamment parce qu'une simulation n'est qu'une exécution possible du système et qu'il est très difficile de dire si cette exécution est représentative de la réalité. Nous souhaitons vérifier des propriétés concernant la consommation d'énergie sur notre système. Pour ce faire nous utiliserons des techniques de model-checking. Ces techniques se basent sur une sémantique formelle bien définie. Étant donné le coût élevé en mémoire et en temps de calcul de ces techniques, nous ne pouvons pas espérer valider un modèle aussi complexe que celui d'un réseau de capteurs détaillé à l'extrême. De telles techniques n'ont aucune chance d'aboutir si tous les composants de tous les nœuds sont décrits aussi précisément qu'en réalité. Nous pensons cependant qu'il est possible d'utiliser le model-checking pour vérifier des propriétés portant sur la durée de vie d'un réseau de capteurs abstrait. Les abstractions peuvent venir d'une bonne connaissance des réseaux de capteurs, elles peuvent également être le fruit de simulations. Mais aucune de ces techniques d'abstraction ne permet de relier formellement le modèle abstrait à la réalité. Une fois le modèle abstrait vérifié, que peut-on en déduire sur le modèle réaliste ?

Nous proposons donc un cadre qui permettent de faire des abstractions contrôlées. La première propriété que doit vérifier une abstraction est qu'elle doit être conservative. Ce qui signifie que si la propriété est vraie sur le modèle abstrait alors elle est forcément vraie sur le modèle plus détaillé. Comme nous avons construit des modèles détaillés fiables, si la propriété est vraie sur le modèle détaillé elle doit être vraie dans la réalité. La question clé dans les réseaux de capteurs est celle de l'énergie, les propriétés que nous souhaitons vérifier concernent donc l'énergie, elles peuvent être formulées de la manière suivante : *"Pour tous les comportements possibles, la durée de vie du réseau est au moins de t unités de temps."*. Pour que nos abstractions soient conservatives par rapport à des durées de vie pire cas, il faut et il suffit que les modèles abstraits consomment au moins autant que les modèles détaillés. S'il est délicat de proposer un modèle global d'un réseau de capteurs, il est au moins aussi délicat de proposer un modèle abstrait pour un réseau de capteurs. Par contre, on peut beaucoup plus facilement imaginer des abstractions pour chacun des composants de notre modèle global. Notre modèle à base de composants nous permet ici de construire un modèle global abstrait à partir de composants abstraits. Ceci requiert une propriété supplémentaire pour la composition des composants : elle doit préserver la relation d'abstraction. En d'autres termes, un modèle contenant un composant abstrait doit être une abstraction du même modèle contenant le composant original.

Deuxième partie

Modélisation, évaluation de performance

Chapitre 3

Évaluation de protocoles MAC à échantillonnage de préambule

Sommaire

Le travail présenté dans ce chapitre est un travail effectué avec Abdelmalik Bachir lors de sa thèse à France Télécom R&D. Ce travail nous a également conduit à déposer un brevet [6]. Le contenu de ce chapitre reprend essentiellement celui de la publication [7].

Ce chapitre s'intéresse aux protocoles d'accès au médium (protocoles MAC). Nous rappelons d'abord brièvement comment fonctionne un protocole à échantillonnage de préambule puis nous montrons quels sont ses inconvénients. Nous proposons alors quatre nouveaux protocoles pour lesquels de l'information est contenue dans le préambule. Ces protocoles sont des extension de MFP [2] : nous proposons deux types de préambules (MFP et DFP) et deux méthodes de réception du préambule en cas d'erreur : persistant ou non persistant. Nous proposons alors une étude mathématique de ces protocoles pour étudier leur efficacité en énergie et leur fiabilité dans le cas d'un canal radio non fiable.

Le chapitre est organisé comme suit : la section 3.1 rappelle ce que sont les protocoles à échantillonnage de préambule, rappelle leurs limites et introduit les protocoles que nous étudions ici. Nous présentons ensuite les hypothèses de modélisation et les résultats obtenus pour ces protocoles, section 3.2. La section 3.3 applique quelques exemples numériques à nos résultats théoriques afin de vérifier que les formules mathématiques obtenues sont conformes à l'intuition. Enfin la section 3.4 conclut le chapitre.

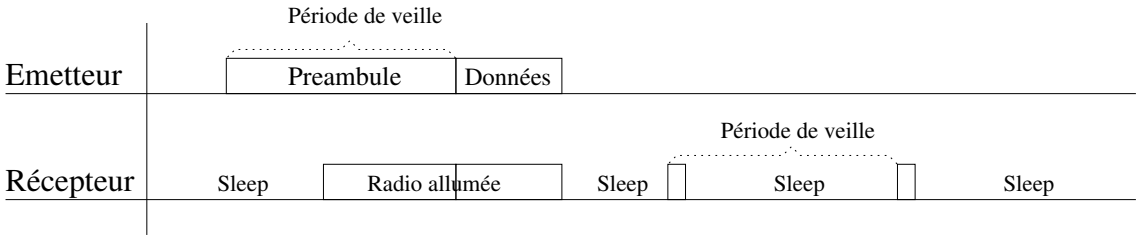


FIG. 3.1 – Protocole à échantillonnage de préambule

3.1 Protocoles à échantillonnage de préambule

3.1.1 Protocoles classiques à échantillonnage de préambule

Dans les protocoles à échantillonnage de préambule, les nœuds ne sont pas synchronisés, ils allument et éteignent périodiquement leur radio indépendamment les uns des autres. Chaque nœud (voir figure 3.1) se réveille périodiquement pour sonder le canal. S'il détecte un signal, il maintient sa radio allumée pour essayer de recevoir le paquet de données à venir. Pour émettre un paquet, les nœuds doivent envoyer le paquet précédé d'un préambule. Le rôle du préambule est d'attirer l'attention du récepteur pour que sa radio soit allumée au moment de l'envoi du paquet de données. Le préambule doit donc être aussi long que la période de veille, c'est-à-dire le temps entre deux réveils successifs.

3.1.2 Amélioration des protocoles à échantillonnage de préambule

L'inconvénient majeur des protocoles à échantillonnage de préambule est que le nœud qui détecte le préambule maintient sa radio allumée jusqu'à la réception du paquet de données. En effet, le préambule ne contient aucune information, donc en particulier, il ne contient aucune information indiquant le moment où le paquet de données sera envoyé.

Pour outrepasser ce défaut, nous proposons d'envoyer un préambule contenant des informations. Le préambule est remplacé par une suite de paquets qui peuvent être décodés par le récepteur avant la réception du paquet de données. Dans [2], Abdelmalik Bachir présentait déjà le protocole MFP, pour *Micro-Frame Preamble*, dans lequel le préambule est remplacé par une suite de petits paquets qui indiquent le temps qu'il reste avant le paquet de données ainsi que certaines informations sur le paquet de données à venir. Nous étendons ici la notion avec d'autres propositions pour les informations contenues dans le préambule.

Nous proposons quatre variantes de protocoles MAC à préambule à base de paquets. Ces variantes sont issues des stratégies qu'un nœud peut adopter à la réception et à l'émission. Nous proposons deux stratégies pour la réception et deux stratégies pour l'émission. Ces différents choix au niveau du récepteur et de l'émetteur se combinent si bien que l'on peut compter quatre protocoles différents. Nous présentons dans les parties 3.1.3 et 3.1.4 le choix de l'émetteur et celui du récepteur respectivement.

3.1.3 Transmission : MFP ou DFP

Les paquets envoyés à la place du préambule peuvent être des micro-paquets ou des paquets de données copies conformes du paquet de données lui-même.

Dans MFP (Micro Frame Preamble) [2], le nœud émetteur envoie à la place du préambule une suite de petits paquets contenant des informations sur le paquet de données à venir. Nous utilisons le terme de *micro-trame* pour désigner ces paquets. Un nœud qui capte le préambule, décode une

micro-trame de laquelle il déduit quand le paquet de données sera transmis et s'il est intéressant de le recevoir. Les informations contenues dans une micro-trame sont : un numéro de séquence, l'adresse de destination et un numéro hachage des données à venir. Le numéro de séquence correspond au nombre de micro-trames qui seront encore transmises avant le paquet de données. Le récepteur en déduit quand le paquet de données sera émis, il peut donc éteindre sa radio ce qui évite de perdre de l'énergie en recevant des micro-trames inutiles. L'adresse de destination permet d'éviter à un nœud la réception d'un paquet qui ne lui est pas adressé. Le numéro de hachage permet d'identifier des données et donc d'éviter de recevoir à nouveau un paquet déjà reçu (en cas d'inondation par exemple). Pour cela, les nœuds doivent mémoriser le numéro de hachage de chaque paquet reçu.

L'autre possibilité pour l'émetteur est appelée DFP (pour *"Data Frame Protocol"*). Dans DFP, le préambule est remplacé par des copies du paquet de données. L'émission du préambule suivi du paquet de données est donc remplacée par l'émission d'une suite de paquets de données. L'avantage de DFP est que le nœud qui se réveille pour sonder le canal reçoit directement le paquet de données, il n'a pas besoin de se réveiller à nouveau pour recevoir les données. Cependant, il est ici impossible d'éviter la réception d'un paquet de données non-souhaité (déjà reçu ou adressé à un autre nœud).

3.1.4 Réception : persistant ou non-persistant

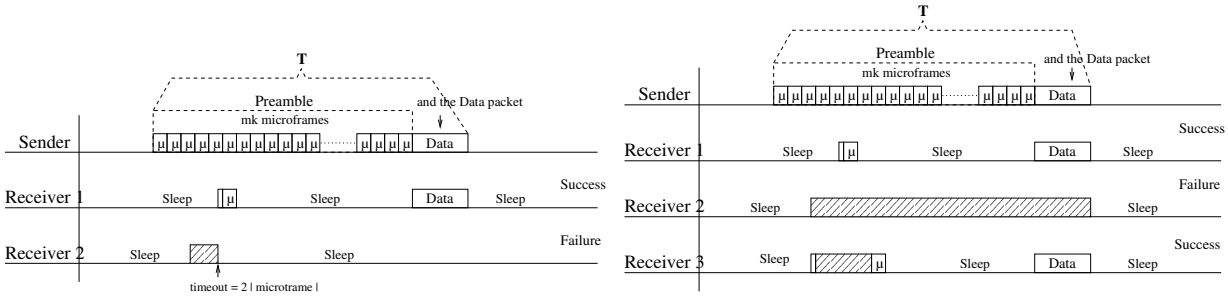
Dans les protocoles à échantillonnage de préambule, les nœuds allument périodiquement leur radio pour sonder le canal. Le temps pendant lequel la radio est allumée est le temps jugé nécessaire pour décider si un préambule est envoyé ou pas.

Si le préambule est une suite de petits paquets (MFP), alors ce temps est au moins égal à la durée de réception d'une micro-trame (un petit paquet). Mais en général, il faut plus que cette durée pour pouvoir décoder une micro-trame. Si l'on suppose que le lien radio est parfait, il faut que le nœud soit éveillé pendant une durée équivalente à l'émission de deux micro-trames pour pouvoir, toujours, décoder au moins une micro-trame. Le cas le pire, où cette durée est nécessaire, se produit quand le nœud se réveille juste après l'émission du premier bit d'une micro-trame. Dans ce cas, le nœud ne peut pas décoder la première micro-trame et doit rester éveiller jusqu'à la fin de la micro-trame suivante pour la décoder.

Cependant, si le lien radio n'est pas parfait, il peut y avoir des erreurs ou des collisions qui brouillent la micro-trame suivante empêchant le nœud de la décoder. Le nœud comprend cependant qu'il y a de l'activité sur le canal mais ne peut en déduire d'autres informations. Dans ce cas, le nœud récepteur a deux options :

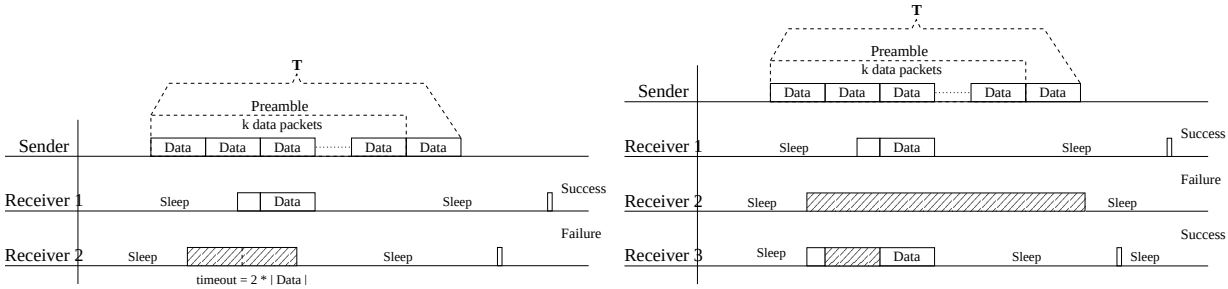
1. être *persistant* et maintenir la radio allumée jusqu'à la réception correcte d'une micro-trame où jusqu'à ce qu'il n'y ait plus d'activité sur le canal radio.
2. ou *non persistant* et éteindre sa radio après un certain temps que nous définissons comme le délai nécessaire pour recevoir certainement une micro-trame, quand il n'y a pas d'erreurs sur le canal. Pour MFP, ce temps correspond à deux fois le temps d'émission d'une micro-trame ; pour DFP, ce temps correspond à deux fois le temps d'émission d'un paquet de données.

La figure 3.2 montre un exemple pour les protocoles MFP-persistant (figure 3.2(b)) et MFP-non-persistant (figure 3.2(a)). Les différents choix possibles du nœud récepteur que nous avons appelés persistant et non-persistant s'applique également au protocole DFP. Dans ce cas, le temps nécessaire à la réception d'un paquet sachant que le nœud peut démarrer son écoute à tout moment est deux fois la durée d'émission d'un paquet. Nous étudierons donc également les protocoles DFP-persistant ((figure 3.3(b)) et DFP-non-persistant (figure 3.3(a)).



(a) Non Persistant (b) Persistant

FIG. 3.2 – MFP



(a) Non Persistant (b) Persistant

FIG. 3.3 – DFP

3.2 Modélisation et évaluation des différents protocoles

3.2.1 Modélisation du système

Problématique et hypothèses

Nous considérons un lien radio entre deux nœuds. Nous cherchons à estimer le coût en énergie d'une transmission d'un paquet de données d'un nœud à l'autre via ce lien radio. Nous définissons le coût d'une telle transmission comme étant la somme des coûts de l'émission et de la réception. Nous comparons les quatre protocoles décrits précédemment. Le but de l'étude est de savoir quels protocoles il faut choisir suivant la fiabilité souhaitée et l'état du canal radio.

En fait, nous cherchons à minimiser ici, le temps pendant lequel la radio des nœuds est allumée, qu'elle soit en réception ou en émission. Ce temps n'est pas directement proportionnel à la consommation d'énergie puisque d'autres éléments matériels des nœuds consomment de l'énergie. Cependant la radio consomme beaucoup dès qu'elle est allumée, il est donc nécessaire de mesurer le temps pendant lequel elle est allumée.

Nous modélisons les imperfections du canal radio en supposant que chaque bit transmis a une probabilité d'erreur constante et indépendante du temps. Ce modèle est appelé *"Binary Symmetric Channel (BSC)"*. Notons que cette modélisation du canal prend mal en compte les erreurs dues aux collisions. En effet lors d'une collision, le canal est brouillé pendant un certain temps et donc la probabilité d'erreur n'est pas indépendante du temps.

Soit p la probabilité qu'une micro-trame soit corrompue. Nous supposons que les micro-trames ont une longueur et une durée de transmission fixes et nous définissons l'unité de temps comme le

temps nécessaire à l'émission d'une micro-trame. Nous supposons que les paquets de données ont également une longueur fixe, et nous supposons qu'ils sont m fois plus longs que les micro-trames. Nous introduisons la constante k telle que $m \times k$ micro-trames constituent le préambule. $m \times k$ est donc la période de veille des nœuds, voir figure 3.2. La constante k nous permet de comparer les protocoles MFP et DFP, dans DFP l'émetteur envoie k paquets de données dans ce qui correspond au préambule.

En résumé, voici les différents paramètres que nous utilisons dans notre étude :
- p, la probabilité d'erreur d'une micro-trame
- 1, la longueur d'une micro-trame
- m, longueur d'un paquet de données
- $m \times k$, le nombre de micro-trame du préambule et donc aussi la longueur du préambule

Fiabilité du lien

Dans les protocoles à échantillonnage de préambule, le nœud récepteur envoie un message d'acquittement (ACK pour *Acknowledgement*) à l'émetteur juste après la transmission pour accuser réception du paquet de données. Si l'émetteur ne reçoit pas d'acquittement, il ré-émet jusqu'à recevoir un acquittement ou jusqu'à ce qu'un nombre maximum d'émissions ait été atteint. Nous notons n le nombre maximum de ré-émissions et p_f la probabilité d'erreur d'une transmission unique. Nous appelons *transmission unique*, l'émission d'un préambule suivi du paquet de données alors qu'une *transmission* est l'émission d'un paquet et peut éventuellement nécessiter plusieurs (re)transmissions. Nous définissons la fiabilité du lien comme étant la probabilité de réussite d'une transmission. Sachant que pour une transmission, on autorise un maximum de n transmissions uniques, on obtient pour la fiabilité (R pour *"Reliability"* en anglais) :

$$R = 1 - p_f^n. \tag{3.1}$$

Coût de transmission

Nous supposons que l'énergie dépensée pendant une transmission est proportionnelle au temps passé à émettre un signal. Soit T la durée d'une transmission unique :

$$T = mk + m \tag{3.2}$$

et T_{tx} la durée d'une transmission :

$$\begin{aligned} T_{tx} &= (1-p_f)T + p_f(1-p_f)2T + \cdots + p_f^{n-2}(1-p_f)(n-1)T + p_f^{n-1}nT \\ &= (1-p_f)\sum_{i=1}^{n-1} p_f^{i-1}iT + p_f^{n-1}nT \\ &= \frac{1-p_f^n}{1-p_f}T \end{aligned} \tag{3.3}$$

Coût de réception

Nous utilisons la même technique qu'au paragraphe précédent, pour exprimer le temps pendant lequel le récepteur a sa radio allumée. Soit S (respectivement F) une variable aléatoire qui représente le

temps passé en mode réception pour une transmission unique dans le cas où cette transmission unique a réussie (respectivement a échoué). Nous pouvons donc exprimer T_{rx}, la durée de réception :

$$\begin{aligned} T_{rx} &= (1-p_f)S + (1-p_f)p_f[F+S] + \cdots + (1-p_f)p_f^{n-1}[(n-1)F+S] + p_f^n nF \\ &= (1-p_f)\left(\sum_{i=0}^{n-1} p_f^i[iF+S]\right) + p_f^n nF \\ &= \frac{1-p_f^n}{1-p_f}[p_f F + (1-p_f)S] \end{aligned} \tag{3.4}$$

3.2.2 Évaluation

Pour évaluer nos protocoles, nous allons déterminer les valeurs de p_f, F, et S pour chacun des protocoles DFP non-persistant (npDFP), MFP non-persistant (npMFP), DFP persistant (pDFP) et MFP persistant (pMFP).

DFP non-persistant

Dans DFP non-persistant, le nœud qui sonde le canal ne reste pas plus de deux fois la longueur d'un paquet éveillé. Pour qu'une transmission unique réussisse, il faut que le récepteur reçoive correctement le paquet de données qui suit son instant de réveil. Soit q la probabilité qu'un paquet de données soit corrompu.

$$q = 1-(1-p)^m \tag{3.5}$$

Donc, ici

$$p_f = q \tag{3.6}$$

Dans les protocoles à échantillonnage de préambule, le récepteur est susceptible de sonder le canal à n'importe quel instant du préambule. Dans DFP, le préambule est constitué d'une suite de paquets de données, le récepteur se réveille donc forcément pendant l'émission d'un de ces paquets mais il y a autant de chances qu'il se réveille au début, au milieu ou à la fin de l'émission d'un paquet. S'il se réveille pendant l'émission d'un paquet, il est obligé d'attendre la fin de ce paquet pour commencer à décoder correctement le suivant. Le temps qu'il passe forcément radio allumée avant le début du paquet suivant est compris entre 0 et m. Ce temps est donc U_m où U_m est une variable qui suit une loi uniforme sur l'intervalle $[0, m[$. Dans le cas d'une réception correcte, le premier paquet entièrement reçu n'est pas corrompu et donc :

$$S = U_m + m \tag{3.7}$$

Dans le cas où la réception échoue, la durée de réception F n'est pas fixe, deux cas se produisent. Si le récepteur se réveille pendant la transmission de la dernière trame de données alors il ne peut pas recevoir le paquet de données en cours d'émission, il échoue à recevoir le paquet suivant qui est le dernier paquet de cette transmission unique. En effet, puisqu'il s'est réveillé à la fin du préambule, ce paquet ne correspond plus au préambule. Après ce paquet, le canal est à nouveau libre, le récepteur éteint donc sa radio. Ainsi, le récepteur éteint sa radio avant la durée maximum de deux paquets de données ($2 \times m$). Dans ce cas $F = U_m + m$. À l'inverse, si le récepteur se réveille avant le dernier paquet de données du préambule alors il reste au moins deux paquets à envoyer (au moins le dernier

paquet du préambule et le paquet de données lui-même). Sachant que la transmission échoue, le récepteur a éteint sa radio parce que le délai de $2 \times m$ a expiré. La probabilité que le récepteur se réveille pendant le dernier paquet de données est $1\ k$. Donc

$$F = k - \frac{1}{k} \times 2m + \frac{1}{k} \times (U_m + m) \tag{3.8}$$

MFP non-persistant

Pour recevoir correctement une transmission unique dans MFP non-persistant, il faut que le récepteur reçoive correctement la micro-trame qui suit son réveil et qu'il reçoive correctement le paquet de données. Donc :

$$\begin{aligned} p_f &= 1 - (1-p)(1-q) \\ &= 1 - (1-p)^{m+1} \end{aligned} \tag{3.9}$$

Pour des raisons de simplicité, nous ne considérons pas le cas où le récepteur se réveille durant la dernière micro-trame. Donc le récepteur reçoit correctement une micro-trame ou son timer expire pendant le préambule. Cette hypothèse simplifie l'analyse et n'a que peu d'effets sur les résultats. Donc, en cas de réception correcte d'une micro-trame, le temps S passé en réception est égale au temps nécessaire à la réception d'une micro-trame plus le temps de réception d'un paquet :

$$S = U_1 + 1 + m \tag{3.10}$$

où U_1 est une variable aléatoire suivant une loi uniforme sur $[0, 1[$.

Deux raisons peuvent impliquer l'échec d'une transmission unique. Soit le récepteur n'a pas réussi à décoder une micro-trame. Dans ce cas, il ne se réveille pas pour les données puisqu'il ne sait pas quand le paquet sera émis. Soit il a correctement décodé une micro-trame, mais le paquet de données était corrompu. La probabilité qu'une micro-trame soit corrompue est p. Nous obtenons donc :

$$F = p \times 2 + (1-p) \times (U_1 + 1 + m) \tag{3.11}$$

DFP persistant

Dans DFP persistant, une transmission unique échoue quand le récepteur ne reçoit aucun paquet correctement. Tous les paquets de données reçus entre le réveil du récepteur et la fin du préambule plus le paquet de données lui-même sont corrompus. C'est ce qui arrive au récepteur 2 de la figure 3.3(b). Le nœud récepteur peut se réveiller n'importe quand pendant l'émission du préambule, il est donc susceptible de recevoir j paquets de données, j allant de 1 à k. Par exemple, s'il se réveille pendant le premier paquet de données, il reste éveillé pendant k paquets de données, correspondant aux $k-1$ paquets du préambule plus 1 paquet après le préambule. La probabilité que le nœud récepteur se réveille pendant un des paquets est la même pour tous les paquets, soit $1/k$. On a donc,

$$\begin{aligned} p_f &= \frac{1}{k}q^k + \frac{1}{k}q^{k-1} + \cdots + \frac{1}{k}q \\ &= \frac{q}{k}\left(\frac{1-q^k}{1-q}\right) \end{aligned} \tag{3.12}$$

Pour exprimer F et S, nous introduisons $X \in \{0, \ldots, k-1\}$, une variable aléatoire discrète qui exprime le nombre de paquets corrompus du préambule reçus.

$$X = (X|_{\textit{failure}})p_f + (X|_{\textit{success}})(1-p_f). \tag{3.13}$$

Où, $X|_{\textit{failure}}$ (respectivement $X|_{\textit{success}}$) est une variable aléatoire discrète exprimant le nombre de paquets du préambule reçus corrompus sachant que cette transmission unique a échoué (respectivement réussi). Nous exprimons donc F et S comme :

$$F = U_m + m \times (X|_{\textit{failure}}) + m \tag{3.14}$$
$$S = U_m + m \times (X|_{\textit{success}}) + m \tag{3.15}$$

Pour obtenir $P[X=j]$ pour $j = 0, \ldots, k-1$, nous utilisons :

$$P[X=j] = P[X \geq j] - P[X \geq j+1] \tag{3.16}$$

Pour exprimer $P[X \geq j]$, nous introduisons $Z \in \{1, \ldots, k\}$, une variable aléatoire qui représente le numéro du paquet envoyé au moment où le récepteur allume sa radio. Nous considérons donc que les paquets sont numérotés comme suit : le premier paquet du préambule à le numéro 1 et le dernier paquet a le numéro k. Z est uniforme, $P[Z=j] = 1/k$. Nous avons donc,

$$\begin{aligned} P[X \geq j] &= \sum_{i=1}^{k} P[X \geq j | Z=i] P[Z=i] \\ &= \underbrace{\frac{1}{k}q^j + \cdots + \frac{1}{k}q^j}_{\text{si le récepteur se réveille avant la position } k-j} + \underbrace{\frac{1}{k}0 + \cdots + \frac{1}{k}0}_{\text{sinon}} \\ &= \frac{k-j}{k}q^j \end{aligned} \tag{3.17}$$

Donc,

$$\begin{aligned} P[X=j] &= \frac{k-j}{k}q^j - \frac{k-(j+1)}{k}q^{j+1} \\ &= \frac{q^j}{k}\Big[k(1-q) + q - j(1-q)\Big] \end{aligned} \tag{3.18}$$

Il reste maintenant à exprimer $X|_{\textit{failure}}$. $X|_{\textit{failure}} \in \{0, \cdots, k-1\}$. $X|_{\textit{failure}}$ est uniforme sur $[0, k-1]$. En effet, la probabilité que le récepteur écoute j paquets de données sachant l'échec de la transmission unique est exactement la probabilité que le récepteur se réveille pendant la transmission du paquet numéro $k-j$. La probabilité associée à cet événement est $1/k$. Donc $P[X|_{\textit{failure}} = j] = 1/k$. Nous pouvons donc en déduire $X|_{\textit{success}}$:

$$X|_{\textit{success}} = \frac{X - (X|_{\textit{failure}})p_f}{1-p_f} \tag{3.19}$$

MFP persistant

La probabilité de succès d'une transmission unique pour le protocole MFP persistant ne dépend que de la réussite du paquet de données qui suit le préambule composé de micro-trames. Donc,

$$p_f = q \tag{3.20}$$

Pour calculer F et S, nous introduisons $Y \in \{0, \cdots, mk-1\}$, une variable aléatoire qui correspond au nombre de micro-trames reçues, corrompues ou pas.

$$F = U_1 + Y + m \tag{3.21}$$
$$S = U_1 + Y + m \tag{3.22}$$

U_1 correspond au temps pendant lequel le récepteur doit écouter avant le début d'une micro-trame et m est la durée d'émission d'une micro-trame. Notons que Y ne dépend pas du succès de la transmission unique puisque celle-ci ne dépend que du succès de l'émission du paquet de données. Donc $F = S$ dans les deux équations précédentes. Pour exprimer Y, nous utilisons à nouveau la relation :

$$P[Y = j] = P[Y \geq j] - P[Y \geq j+1] \tag{3.23}$$

Pour calculer $P[Y \geq j]$, nous procédons comme section 3.2.2, c'est-à-dire, nous introduisons Z qui exprime la position de la micro-trame pendant laquelle le récepteur se réveille. Z est uniforme sur $\{1, \cdots, mk\}$, donc $P[Z = j] = 1/mk$. Nous obtenons donc :

$$\begin{aligned} P[Y \geq j] &= \sum_{i=1}^{k} P[Y \geq j | Z = i] P[Z = i] \\ &= \underbrace{\frac{1}{mk}p^{j-1} + \cdots + \frac{1}{mk}p^{j-1}}_{\text{si le récepteur se réveille avant la micro-trame numéro } mk-j} + \underbrace{\frac{1}{mk}0 + \cdots + \frac{1}{mk}0}_{\text{sinon}} \\ &= \frac{mk-j}{mk}p^{j-1} \end{aligned} \tag{3.24}$$

Nous obtenons donc :

$$P[Y = j] = \frac{1}{mk}\left[(mk-j)p^{j-1} - (mk-j-1)p^{j}\right] \tag{3.25}$$

3.3 Résultats numériques

Dans cette section, nous avons tracé les courbes correspondant à nos calculs pour certaines valeurs des paramètres. Nous traçons donc l'évolution du temps passé en transmission, du temps passé en réception, de la somme de ces deux temps et enfin de la fiabilité en fonction de la probabilité d'erreur d'une micro-trame. La durée pendant laquelle les radios sont en mode émission ou réception permet de comparer l'efficacité énergétique des différents protocoles. La fiabilité est un paramètre indispensable puisque le but est non seulement de concevoir des protocoles efficaces en énergie mais aussi des protocoles fiables. Un protocole qui ne consomme pas d'énergie mais n'envoie aucun paquet ne présente que peu d'intérêt.

Pour tracer les figures 3.4 à 3.7 nous avons utilisé les équations 3.1, 3.3 et 3.4 avec pour p_f, F et S les formules obtenues pour chacun des protocoles. Le nombre de retransmissions n a été choisi égal à

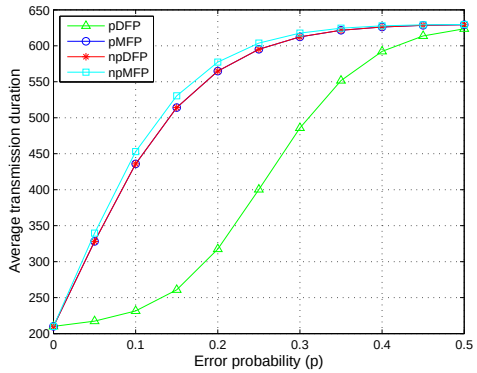


FIG. 3.4 – Temps d'émission moyen

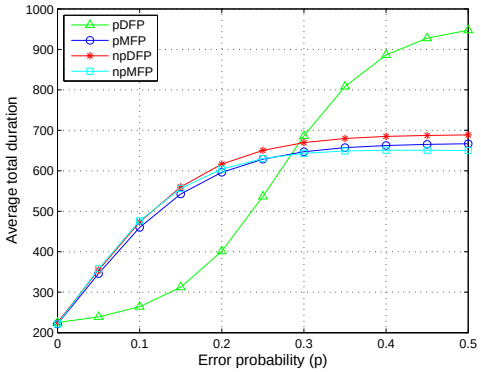


FIG. 3.5 – Temps moyen d'émission et réception

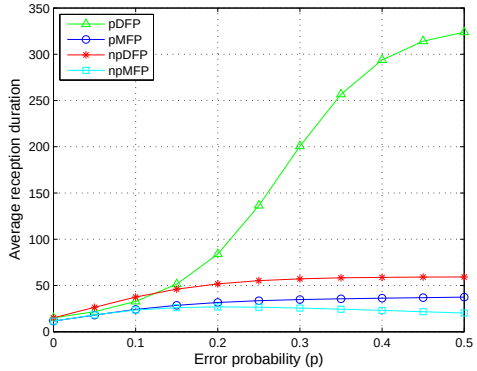


FIG. 3.6 – Temps de réception moyen

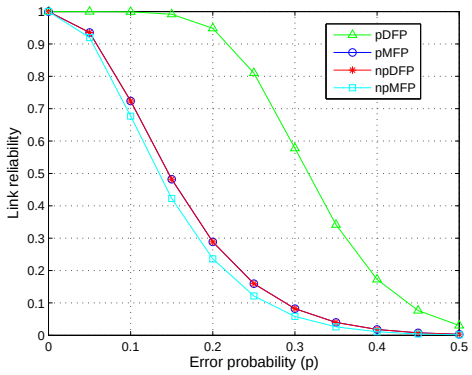


FIG. 3.7 – Fiabilité du lien

3. Nous considérons que les paquets de données sont dix fois plus longs que les micro-trames, soit $m = 10$. La période de veille est de 200 unités de temps, ce qui veut dire qu'un préambule est composé de 200 micro-trames pour MFP et de 20 paquets de données pour DFP. Soit, $k = 20$. Nous faisons varier la probabilité p qu'une micro-trame soit erronée entre 0 et 0.5. Nous rappelons que l'unité de temps est le temps nécessaire à l'émission d'une micro-trame.

Sur la figure 3.4, on peut constater que les durées de transmission des protocoles pMFP et npMFP sont les mêmes. En effet pour ces deux protocoles, la probabilité de réussite d'une transmission unique est la même, elle est égale à la probabilité de réception correcte d'un paquet de données. Dans les deux cas, l'émetteur émet autant de transmissions uniques. Cette remarque est confirmée figure 3.7, les deux protocoles ont la même fiabilité.

Notons également que la durée moyenne d'émission est légèrement supérieure pour le protocole npMFP. C'est parce que la probabilité d'erreur d'une transmission unique est légèrement plus grande pour ce protocole : pour qu'une transmission unique réussisse il faut non seulement que le paquet de données soit bien reçu mais également qu'une micro-trame soit bien décodée. Ce protocole nécessite donc en moyenne plus de retransmissions.

C'est pour le protocole pDFP que l'émetteur passe en moyenne le moins de temps en émission (figure 3.7). Car, pour ce protocole, une transmission unique a le moins de chance d'échouer. Ce résultat apparaît à nouveau figure 3.7 : pDFP a la meilleure fiabilité.

Sur la figure 3.6, on remarque que le temps de réception pour le protocole npMFP atteint un maximum (vers $p = 0.3$). Nous expliquons ce résultat comme suit : pour p faible, le récepteur reçoit correctement la micro-trame du préambule, il peut donc se réveiller plus tard pour le paquet de données ; à l'inverse lorsque la probabilité d'erreur est trop grande, il n'arrive en général pas à décoder la micro-trame, il n'allume donc pas sa radio pour recevoir le paquet.

Sur la figure 3.6, il apparaît aussi que la durée passée en réception pour pDFP augmente avec le taux d'erreurs. En effet, pour un taux d'erreur élevé, le récepteur doit recevoir plus de paquets avant d'en décoder au moins un. Cependant ce temps est borné puisque le nombre de transmissions est limité à n (3 sur les courbes).

3.4 Conclusion

Dans ce chapitre, nous avons proposé une analyse mathématique de protocoles d'accès au médium. ces protocoles (np-DFP, np-MFP, p-DFP et npMFP) sont des protocoles à échantillonnage de préambule dans lesquels le préambule a été remplacé par une suite de paquets qui contiennent de l'information. Pour cette étude nous avons modélisé un canal radio non fiable à l'aide du modèle BSC : la probabilité d'erreur binaire est constante et indépendante du temps. Cette modélisation permet de prendre en compte les erreurs dues au canal radio. Il apparaît qu'aucun protocole n'est toujours meilleurs que les autres, on peut donc imaginer que les nœuds émetteurs changent leur préambule (MFP ou DFP) ou que les nœuds récepteurs s'adaptent (persistant ou non-persistant) en fonction de l'état du canal.

Ce chapitre fournit également un exemple de modélisation mathématique d'un réseau de capteurs pour analyser des performances. Un tel travail permet d'obtenir rapidement des résultats concernant un problème précis. Cependant cette modélisation n'est pas globale. Au niveau du nœud, on ne prend en compte que le protocole et au niveau du réseau on ne considère qu'un seul lien. L'abstraction faite pour estimer l'efficacité en énergie consiste à ne compter que le temps passé en émission ou en réception. Cette abstraction n'est pas infondée, mais il faut être conscient que toutes les sources de consommations n'ont pas été prises en compte.Enfin, une telle analyse permet de comprendre certains aspects des protocoles mais ne permet pas de déduire s'ils interagiront bien avec les autres protocoles.

Nous proposons de construire des abstractions contrôlées. Dans le chapitre suivant, nous introduisons les techniques de modélisation formelle que nous utilisons par la suite.

Troisième partie

Modèles réalistes et modèles analysables

Chapitre 4

Méthodes de modélisation formelles

Sommaire

À la fin du chapitre 2, section 2.4, nous avons présenté une approche pour le prototypage rapide des réseaux de capteurs. Elle consiste à construire un modèle exécutable du réseau à partir de composants. Nous voulons aussi être capable d'effectuer des analyses automatiques sur ce modèle afin de vérifier des propriétés concernant la durée de vie du réseau.

Dans ce chapitre, nous présentons les outils et formalismes qui nous ont permis de mener à bien cette approche. Les méthodes de modélisation que nous avons utilisées se basent toutes sur des automates. C'est un formalisme qui convient bien à nos besoins. Tout d'abord, un automate est un modèle mathématique qui permet de formaliser la notion de comportement d'un système. Un comportement du système est représenté par une séquence d'exécution de l'automate. De plus, ils sont suffisamment expressifs pour décrire les objets que nous souhaitons modéliser. Nous montrons notamment comment ils peuvent être étendus pour modéliser le temps et l'énergie. Enfin, ils permettent de décrire un système de manière compositionnelle : il est possible de composer deux automates afin de construire un automate-produit qui représente à la fois le fonctionnement des deux automates. Il existe différentes notions de produit.

Le chapitre est organisé comme suit. Dans la section 4.1, nous présentons les automates, expliquons à quoi correspond le produit d'automates présentons les produits existants. Nous discutons section 4.2

de la modélisation par des automates de la consommation d'énergie. La section 4.3 présente les analyses que l'on sait faire sur ces modèles. Enfin, dans la section 4.4, nous décrivons les langages de modélisation basés sur les formalismes présentés section 4.1. Ces langages sont ceux que nous utilisons dans les chapitres suivant pour modéliser le réseau de capteurs.

4.1 Modèles à automates

Nous cherchons ici à donner une idée intuitive de la sémantique des formalismes utilisés. Nous restons donc volontairement informel.

4.1.1 Les automates

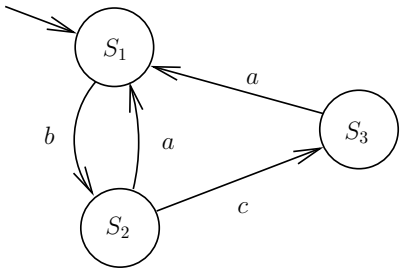


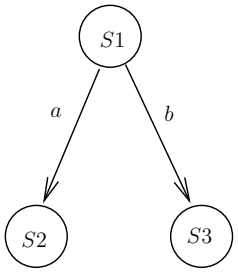
FIG. 4.1 – Un exemple d'automate

La figure 4.1 est un exemple d'automate. Un automate est constitué d'états (ici S_1, S_2 et S_3) et de transitions entre ces états. Parmi les états, il y a un (ou plusieurs) état initial (ici S_1). Toutes les séquences d'exécution commence par un état initial. On désigne souvent cet état par une petite flèche. Les transitions sont des triplets contenant un état de départ, un état d'arrivée et une étiquette (ou label). On attribut des sens différents aux étiquettes (ici a, b et c) suivant le formalisme considéré. Elles peuvent correspondre à une action, à une instruction ou à une communication. Une séquence d'exécution est une suite d'états et de transitions commençant par un état initial et telle que deux états consécutif sont reliés par unetransition. Par exemple $S_1 \xrightarrow{b} S_2 \xrightarrow{a} S_1 \xrightarrow{b} S_2 \xrightarrow{c} S_3$ est une séquence d'exécution de l'automate de la figure 4.1.

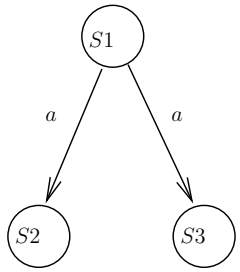
Avant d'aborder la question du produit d'automate, nous présentons rapidement sur un exemple deux propriétés : le déterminisme et la réactivité.

Déterminisme et réactivité

Déterminisme. Un système peut être déterministe ou non déterministe. Un système est dit déterministe si son comportement est complètement déterminé par les entrées (externes) qui lui sont fournies. Un automate est déterministe si pour chaque état au plus une transition est exécutable à la fois dans un état donné. C'est-à-dire que lorsque l'on est dans un état, on n'a pas le choix des transitions que l'on peut emprunter pour aller dans un autre état. Par exemple, l'automate de la figure 4.2(a) est déterministe. Au contraire, s'il existe un état dans lequel plusieurs transitions peuvent être valides en même temps alors l'automate est dit non-déterministe. L'automate de la figure 4.2(b) est indéterministe parce que si a est reçu dans l'état S_1, on peut aller dans l'état S_2 ou dans l'état S_3 et ce choix est interne au système.



(a) Exemple d'automate déterministe.

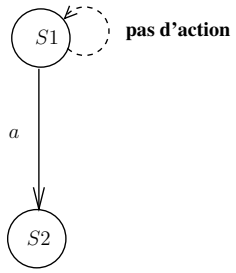


(b) Exemple d'automate non-déterministe.

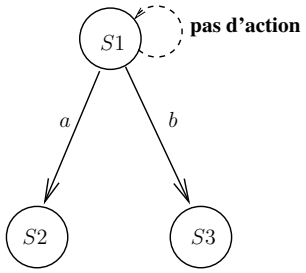
FIG. 4.2 – Déterminisme

On souhaite parfois n'avoir que des automates déterministes par exemple pour modéliser le fonctionnement de machines déterministes. Souvent, une propriété souhaitée pour un programme informatique est le déterminisme. Il existe des outils qui permettent d'implémenter un programme décrit par un automate. Si l'on souhaite que l'implantation finale du programme - sur une machine physique - soit déterministe alors il faut que l'automate de départ le soit aussi. À l'inverse, les automates indéterministes peuvent servir à modéliser un ensemble de comportements. Plusieurs transitions valides d'un automate non déterministe permettent alors de modéliser différents cas possibles. L'indéterminisme peut par exemple modéliser différentes durées, des ordres d'exécution différents ou des pannes qui peuvent surgir à tout moment.

Réactivité. Une autre propriété des automates est la réactivité. Un automate est dit réactif ou complet si, pour tous les états, quelle que soit l'entrée reçue, il existe au moins une transition dont la condition d'activation est vraie. Si aucune entrée n'est reçue, on reste dans l'état courant, ce qui revient à des boucles sans action sur les états. Si les conditions sont la réception de signaux, pour dire si un automate est réactif, il faut savoir quels sont les signaux susceptibles d'être reçus. Supposons, figure 4.3, que l'ensemble des signaux est $\{a, b\}$. Alors l'automate de la figure 4.3(a) n'est pas réactif alors que celui de la figure 4.3(b) l'est.



(a) Exemple d'automate non réactif.



(b) Exemple d'automate réactif.

FIG. 4.3 – Réactivité

Automates étendus. Les automates peuvent être étendus de différentes manières suivant les besoins.

Les transitions peuvent être étiquetées par des signaux d'entrées/sorties. Figure 4.4(a), la transition

est étiquetée par a/b, ce qui signifie que dans $Q1$, si a est reçu, on émet b et on va dans $Q2$. Ce type de formalisme met en évidence les communications entre l'automate et son environnement, il est utilisé pour modéliser des systèmes communicants ou des systèmes réactifs.

Il est aussi possible d'étiqueter les transitions par des conditions et des actions. Dans ce cas, il faut que la condition d'une transition soit vraie pour que l'on puisse la prendre, lorsqu'une transition est franchie, l'action est réalisée. Les conditions (appelées aussi gardes) et les actions font intervenir des variables. Les valeurs des variables conditionnent la transition et les actions sont des affectations de variables. L'automate de la figure 4.4(b) a deux états et une transition condition/affectation. Donc si dans l'état $S1$, $x < 10$ alors $x := x + 1$ et l'état de contrôle devient $S2$. Si les variables ont des domaines de définition finis, il est possible d'écrire l'automate expansé dans lequel chaque état ne contient qu'une valeur possible pour chacune des variables du système. Cependant, dès que les programmes sont assez gros et que l'ensemble de valeur des variables est plus grand que celui des booléens, un automate interprété tel que celui représenté figure 4.4(b) est beaucoup plus pratique à manipuler que l'automate expansé. Pour des ensembles de valeurs infinis (si x appartient à l'ensemble des réels par exemple), on ne peut plus construire l'automate expansé correspondant.

(a) Exemple d'automate : entrées/sorties.

(b) Exemple d'automate : condition/affectation.

FIG. 4.4 – Exemples d'automates

4.1.2 Composition d'automates

Un critère important mis en avant dans l'approche que nous proposons est la modularité. C'est-à-dire la contruction du modèle d'un système à partir de la composition des sous-systèmes qui le constituent. Formellement, on a besoin d'une notion de composition des automates. La composition d'automates a plusieurs sens de modélisation que nous avons besoin de prendre en même temps en compte dans le produit : l'exécution en parallèle et les communications entre automates. L'automate produit est une représentation de l'ensemble des séquences d'exécution du système global. Enfin, nous expliquerons en quoi le temps intervient dans le produit et quels sont les choix possibles.

Modèles d'exécution en parallèle

Le produit d'automates modélise l'exécution en parallèle des différents automates. On peut distinguer intuitivement deux grands modes d'exécution en parallèle : synchrone et asynchrone.

Synchrone. Tous les composants effectuent leurs actions en même temps. On dit qu'ils sont synchrones, quand un composant effectue une transition, les autres aussi. Il y a donc une notion d'horloge

globale. Par exemple, figure 4.5, nous avons représenté le produit synchrone de deux composants. La transition de $S1Q1$ à $S2Q2$ effectue à la fois les actions a et b. Ce produit modélise bien les circuits synchrones où toutes les bascules sont cadencées par une seule et même horloge.

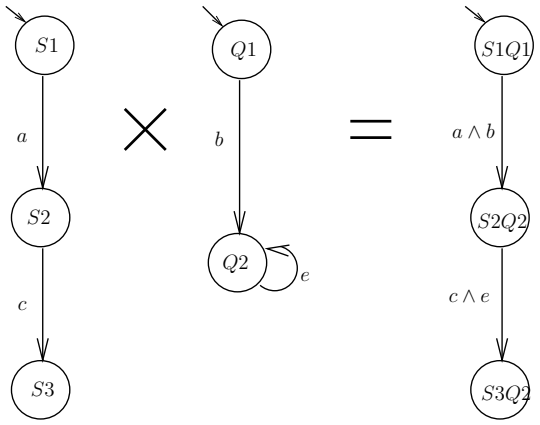


FIG. 4.5 – Produit synchrone

Asynchrone. Les différentes actions ont lieu indépendamment les unes des autres dans n'importe quel ordre. Ce produit est appelé produit asynchrone. Le produit asynchrone des mêmes composants est dessiné figure 4.6. Ce type de produit peut être utilisé pour modéliser l'exécution de processus indépendants sur une machine. L'entrelacement représente tous les ordonnancements possibles.

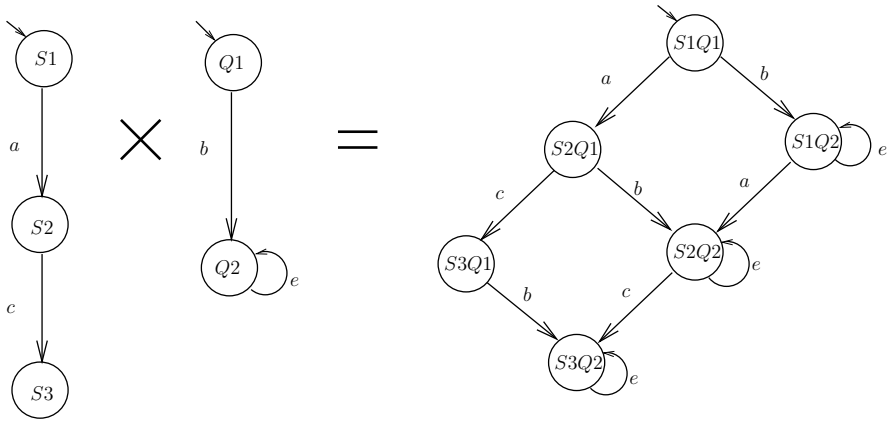


FIG. 4.6 – Produit asynchrone

Ces deux produits sont commutatifs et associatifs ce qui est bien sûr une condition nécessaire à la composabilité de nos modèles. Cependant, nous n'avons pas évoqué ici les communications entre les automates. En effet, les automates émettent et reçoivent des signaux, ces signaux peuvent être échangés avec l'environnement mais ils peuvent aussi provenir d'autres composants. Les communications entre composants doivent être prises en compte lors du produit d'automates. Dans la section suivante, nous présentons différents modes de communication existants.

Modes de communication

Un second aspect à intégrer dans un modèle à composants est le mode de communication utilisé entre les différents composants. Nous présentons ici quelques types de communication que l'on retrouve dans les langages de programmation et de spécification classiques.

Diffusion synchrone. Pour ce mode de communication, lorsqu'un message est envoyé, tous les autres le reçoivent. L'émetteur du signal n'est pas bloqué : il envoie son message et peut continuer à s'exécuter normalement. Que le récepteur soit prêt à recevoir le message ou occupé, le comportement de l'émetteur ne change pas.

Rendez-vous. Le rendez-vous est un mode de communication dans lequel les processus s'attendent. Sur la figure 4.7, les deux automates ont rendez-vous sur le signal c. Les notations $!c$ et $?c$ signifient respectivement, émettre c et recevoir c. Donc, dès que l'un d'eux doit émettre ou recevoir c, il est bloqué tant que l'autre n'est pas prêt à émettre ou recevoir c. Par exemple, sur la figure 4.7, tous les ordonnancements sont possibles pour les exécutions des transitions a et b ce qui crée sur l'automate-produit les états $S2Q1$ et $S1Q2$. Par contre, arrivés dans $S2$ et $Q2$, les automates doivent s'attendre. L'émission et la réception de c a lieu au même moment.

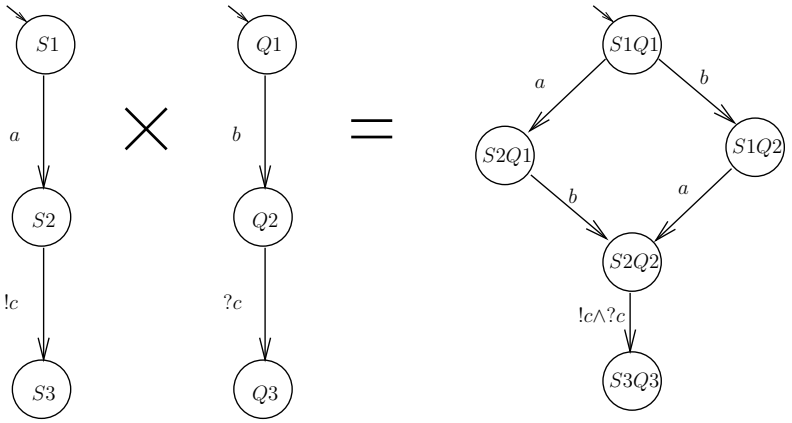


FIG. 4.7 – Illustration du mode de communication par rendez-vous. Ici les deux automates ont rendez-vous sur c.

Ce rendez-vous entre deux composants est appelé binaire. De la même façon, il existe un rendez-vous n-aires lorsque plusieurs automates ont rendez-vous sur un même signal. Dans ce cas, il doivent attendre d'être tous présents au rendez-vous pour effectuer en même temps leurs transitions.

Échange de messages. L'échange de message peut se faire via une mémoire partagée. Dans ce cas, l'émission d'un message revient à ajouter une valeur dans la mémoire. L'émission n'est donc pas bloquante, par contre un processus qui attend un message est bloqué : tant qu'il ne reçoit pas le message, il ne peut pas effectuer la transition. La figure 4.8 illustre ce mode de communication. L'automate Q (de droite) est bloqué dans l'état $Q2$ tant que c n'a pas été émis. Par contre l'émetteur (S) peut effectuer la transition $!c$ quand il veut indépendamment du récepteur (Q).

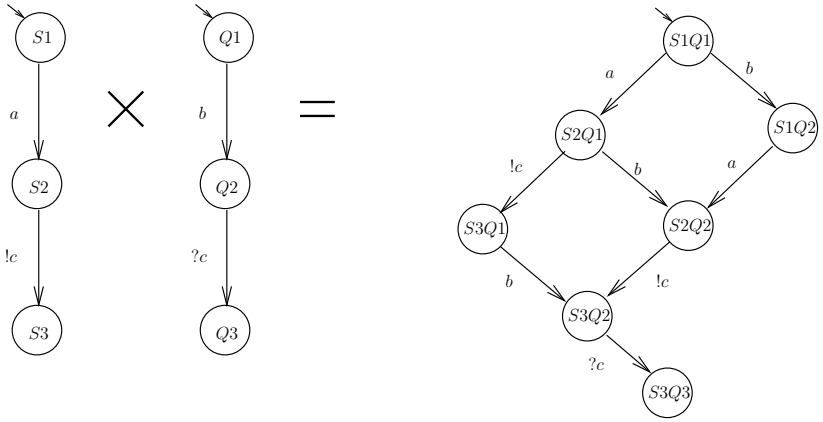


FIG. 4.8 – Illustration du mode de communication par échange de message.

Modèle du temps

Le temps est un élément que l'on a souvent besoin de prendre en compte. Tout d'abord, on est parfois amené à parler de temps pour programmer des systèmes embarqués, c'est le cas pour les protocoles de communications qui utilisent des délais (aléatoires ou fixes). On peut également avoir besoin de modéliser le temps pour vérifier des propriétés. Dans certains systèmes réactifs qui interagissent avec l'environnement, le temps de réponse du système est critique : on ne peut tolérer de réponses qui arrivent trop tard. Ou encore, même si le temps de réponse n'est pas critique, il peut être nécessaire de savoir estimer sur les modèles le temps d'exécution. Enfin, la consommation d'énergie est liée au temps : pour une puissance de consommation fixe, l'énergie et le temps sont proportionnels. Donc afin de concevoir des modèles sur lesquels on pourra estimer la consommation d'énergie, nos modèles doivent prendre en compte le temps.

Le temps peut être modélisé par une variable particulière, *horloge*. La valeur de cette variable augmente quand le temps s'écoule. Le choix qui est fait dans les systèmes de modélisation que nous avons utilisés est que le temps ne s'écoule que sur les états, les transitions, elles, sont instantanées. Ça n'est pas une limite puisque : pour modéliser une transition qui prend du temps, il suffit d'ajouter un état intermédiaire. Par exemple, si la transition t prend une unité de temps, on ajoute un état T et on remplace $Q \xrightarrow{t} S$ par $Q \longrightarrow T \longrightarrow S$.

Pour un système dans lequel il n'y a qu'un seul automate, ce modèle convient. Les difficultés interviennent à partir du moment où plusieurs automates évoluent en parallèle. En effet, dans ce cas, il ne faut pas oublier que le temps physique que l'on modélise s'écoule à la même vitesse dans tous les automates. La variable *horloge* doit avoir la même valeur dans tous les processus.

Pour les automates que nous manipulons, le temps s'écoule sur les états, mais pour faire le produit asynchrone, on passe par une représentation où le temps est étiqueté sur les transitions. Sur la figure 4.9, nous avons décortiqué le produit de deux automates où le temps est discrétisé et l'étiquette T représente l'écoulement d'un pas de temps. D'un point de vue modélisation, ces automates signifient que les états $S1$ et $Q1$ ne prennent pas de temps, le temps s'écoule d'une durée d'une unité de temps dans $S2$ et $Q2$. Les deux automates doivent donc s'attendre dans les états S_2 et Q_2 avant de pouvoir effectuer les transitions suivantes. Techniquement, pour faire le produit, on transforme chacun des automates en ajoutant une transition τ qui représente l'écoulement d'un pas de temps. Puis, on fait le produit asynchrone avec un rendez-vous sur les transitions τ. En effet, le temps ne peut pas s'écouler dans un

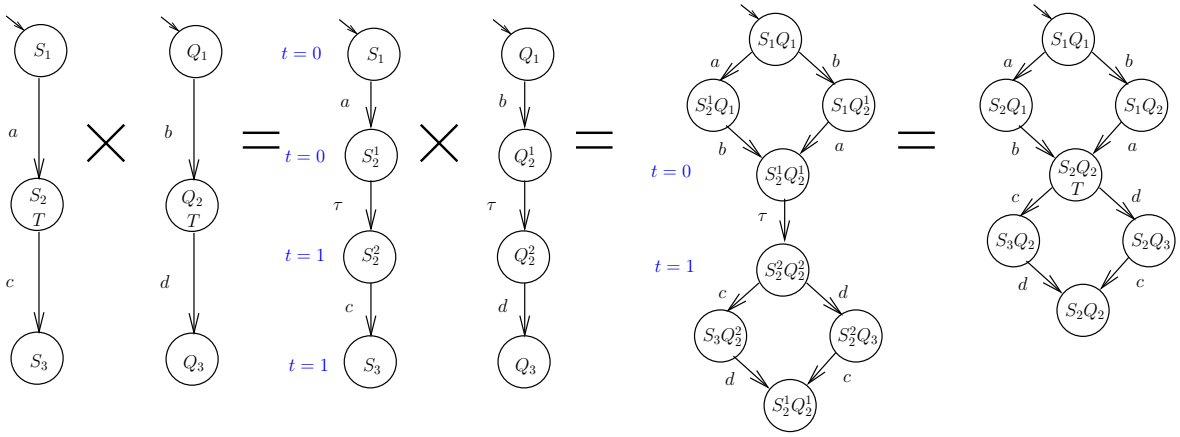


FIG. 4.9 – Produit asynchrone avec synchronisation sur le temps. τ représente l'écoulement d'une unité de temps.

automate sans qu'il s'écoule dans l'autre. Il est ensuite possible de revenir à une représentation du temps sur les états.

Dans le cas du produit synchrone illustré par la figure 4.5, on considère que le temps s'écoule sur chacun des états. C'est pourquoi l'automate-produit correspond à l'exécution simultanée des deux automates. Dès qu'on tire une transition de S, on tire en même temps une transition de Q. Le temps est donc modélisé de manière implicite dans ce produit. Toutes les combinaisons de ces modèles d'exécution, de communication et du temps ne sont pas toujours possibles. Le produit asynchrone et le produit synchrone sont deux combinaisons possibles et cohérentes de caractéristiques. Nous les présentons dans les deux sections suivantes (4.1.3 et 4.1.4).

4.1.3 Produit asynchrone

Une première façon de composer les automates est le produit asynchrone dont nous présentons les caractéristiques maintenant.

- Le modèle d'exécution est celui présenté figure 4.6. Chaque automate effectue ses actions indépendamment des autres. Pour la composition de deux automates, le parallélisme est modélisé par l'entrelacement des actions. Ce modèle d'exécution peut donc créer de l'indéterminisme.
- Les communications entre les processus se font via des files d'attente, des partages de variables ou des rendez-vous. Le rendez-vous est un mode de communication synchrone qui ne correspond pas à la diffusion synchrone. En effet, la diffusion synchrone ne bloque pas l'émetteur alors que le rendez-vous si.
- Pour prendre en compte le temps, on considère certaines actions τ qui correspondent à l'écoulement du temps. Afin que le temps s'écoule à la même vitesse dans les tous processus, les processus ont rendez-vous sur les actions τ. On fait donc un produit avec un rendez-vous n-aire sur τ, exactement comme sur la figure 4.9.

Ce produit ne gère que le temps discret. Il est cependant possible de prendre en compte un temps continu grâce aux automates temporisés. Les automates temporisés [4] sont des automates classiques enrichis d'horloges qui évoluent de manière continue et synchrone avec le temps physique. Comme pour le temps discret, le temps s'écoule dans les états, les transitions restent instantanées. Les horloges interviennent sur les gardes (c'est-à-dire dans les conditions qui permettent - ou pas - de prendre une

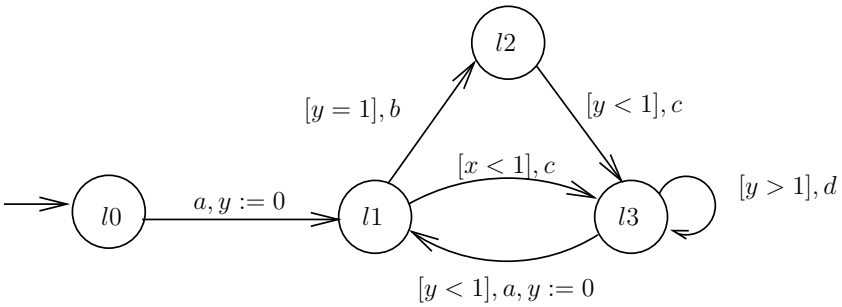


FIG. 4.10 – Exemple d'automate temporisé issu de [4]. x et y sont des horloges et a, b, c, d des actions.

transition). On peut également effectuer des opérations sur les horloges (c'est pour cela qu'il peut y avoir plusieurs horloges). Un exemple d'automate temporisé est présenté figure 4.10. Cependant, nous rappelons qu'un des intérêts du formalisme des automates est la possibilité d'effectuer des analyses automatiques sur les modèles. Pour ces analyses, notamment pour la *vérification par modèle* (nous y revenons section 4.3), une représentation finie du modèle est nécessaire. Il faut donc être capable de créer à partir d'un automate avec des horloges, un système de transition fini. Dans les automates temporisés de Alur et Dill [4], quelques contraintes sont imposées sur les horloges afin de permettre une représentation finie de l'espace d'états. Les horloges évoluent dans l'ensemble des réels positifs $(\mathbb{R}_+)$[1]. Voici les contraintes imposées pour la manipulation des horloges qui permettent de représenter de manière finie l'ensemble des comportements d'un automate temporisé :

- Les gardes sur les horloges sont des expressions booléennes contruites en utilisant les opérateurs booléens habituels ($\neg$, $\wedge$ et $\vee$) à partir des atomes de la forme $x - y \bowtie k$ où x et y sont des horloge, $k \in \mathbb{Z}$ est une constante et $\bowtie$ un opérateur relationnel quelconque ($<, \leq, =, \neq, \geq, >$).
- Les seules actions que l'on s'autorise sur les horloges sont des remises à zéro (*reset*). Il est possible de remettre à zéro une ou plusieurs horloges à la fois.

Avec ces contraintes, il est possible de construire l'automate des régions associé à un automate temporisé. L'idée des régions est de partitionner l'ensemble des valeurs possible pour les horloges en un nombre fini de classes. Chaque région correspond à un ensemble (éventuellement infini) de valeurs possible pour les horloges, et à chaque région ne correspond qu'un unique état de contrôle. C'est possible parce qu'on ne compare les valeurs des horloges qu'à des nombres entiers positifs et qu'il existe forcément un nombre entier maximal auquel une horloge est comparée. La représentation par régions est également introduite dans [4], et [14] présente la vérification par automates temporisés.

Cependant, le produit asynchrone d'automates temporisés peut conduire à des blocages ([15]). Or, afin de construire des modèles réalistes, il est indispensable d'avoir un modèle compositionnel. Les automates temporisés avec urgence [11, 12] permettent d'éviter les blocages. Ils sont une extension des automates temporisés où les transitions ont un attribut supplémentaire qui permet de modéliser la priorité de la transition par rapport à l'avancement du temps. Dans ce formalisme, une transition peut-être urgente (*eager*), retardable (*delayable*) ou paresseuse (*lazy*). Une transition urgente est immédiate dès qu'elle est exécutable, le temps ne progresse plus tant qu'une telle transition est exécutable. Au contraire, une transition paresseuse n'est jamais urgente, le choix entre attendre ou exécuter la transition est non-déterministe. Enfin, une transition retardable est une combinaison des deux précédentes : on peut laisser s'écouler le temps tant que la garde reste vrai mais dès que l'écoulement du temps la rend

[1] Sans perte de généralité, on peut considérer des automates temporisés où les horloges évoluent sur $\mathbb{Q}_+$ voire sur $\mathbb{N}$.

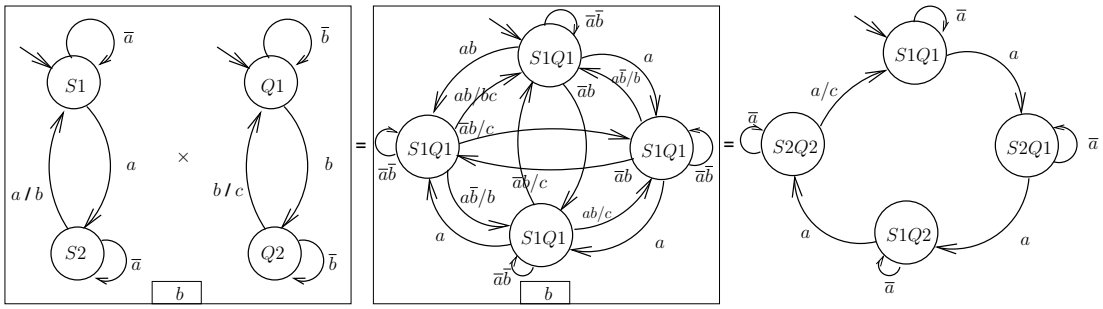


FIG. 4.11 – Produit synchrone à la Argos de deux automates. Première étape : produit cartésien de tous les états. Deuxième étape : encapsulation par b.

non exécutable, on est obligé de la franchir (elle devient donc urgente). Le langage de modélisation IF, présenté section 4.4.1, est basé sur le modèle des automates temporisés avec urgence.

4.1.4 Produit synchrone

Le produit synchrone est celui illustré sur la figure 4.5, page 63 et sur la figure 4.11. Tous les automates tirent une transition en même temps. On dit qu'à chaque instant les processus réagissent. Lorsque deux processus réagissent, on ne discerne pas lequel des deux a exécuté sa transition en premier, ils les ont effectuées simultanément. Ce mode d'exécution de plusieurs processus en parallèle fournit gratuitement une notion de temps. En effet, puisqu'"à chaque instant les processus réagissent", on a bien une notion d'instant et donc une notion de temps. Ce mode d'exécution revient en fait à avoir un processus `temps` qui émet des **tops** d'horloge. À chaque **top** les processus exécutent une transition. Si on peut relier le temps de réaction au temps physique, on obtient une modélisaton du temps. Pour cela, il faut considérer que l'intervalle de temps entre deux **tops** est fixe. Cette hypothèse peut tout à fait être respectée en modélisation. On a ainsi une notion de temps, gratuitement, sans aucune variable supplémentaire. Mais contrairement au formalisme des automates temporisés, le temps est discret. En effet, la plus petite unité de temps exprimable est la durée représentée par un instant logique.

Pour communiquer, les automates utilisent la diffusion synchrone instantanée. Ce qui veut dire qu'à chaque instant, à chaque transition, des signaux peuvent être émis. Les émissions de signaux n'ont pas de destinataires désignés et ne sont pas rémanentes. Tout message émis peut être reçu par n'importe quel processus à l'instant courant mais n'est plus visible dès l'instant d'après.

La figure 4.11 est un exemple de produit synchrone à la ARGOS [60]. Les processus n'échangent que des signaux non-valués. À chaque instant un signal est soit présent, soit absent. La syntaxe utilisée est aussi celle d'ARGOS : l'étiquette "a/b" sur une transition signifie que l'on prend cette transition si a est présent et que dans ce cas, on émet b. $\overline{a}$ dénote l'absence de a. On fait donc le produit synchrone de deux automates. L'automate de gauche (S) est un compteur de a modulo 2, il émet un b tous les deux a. L'automate de droite (Q) est, lui, un compteur de b modulo 2, il émet un c tous les deux b. Pour composer ces deux automates en ARGOS, la première étape est le produit cartésien de tous les états, on obtient ainsi l'automate central. Supposons maintenant que le signal b ne puisse provenir d'aucun autre automate, dans ce cas on *encapsule* par b : on supprime les transitions non cohérentes. Une transition non cohérente est par exemple une transition conditionnée par la présence de b alors que le signal b n'est pas émis ou inversement une transition conditionnée par l'absence d'un signal qu'elle émet. On obtient finalement l'automate de droite qui est un compteur de a modulo 4, il émet un c tous les quatre

a.

Le fait que tout message émis puisse être reçu par n'importe quel processus à l'instant courant mais ne soit plus visible dès l'instant d'après pose le problème de la causalité. En effet, au cours d'un même instant, les processus émettent des signaux qui dépendent des signaux reçus. Pour l'exemple précédent (figure 4.11), tout se passe bien. Nous illustrons maintenant le problème de causalité au travers de deux exemples simples.

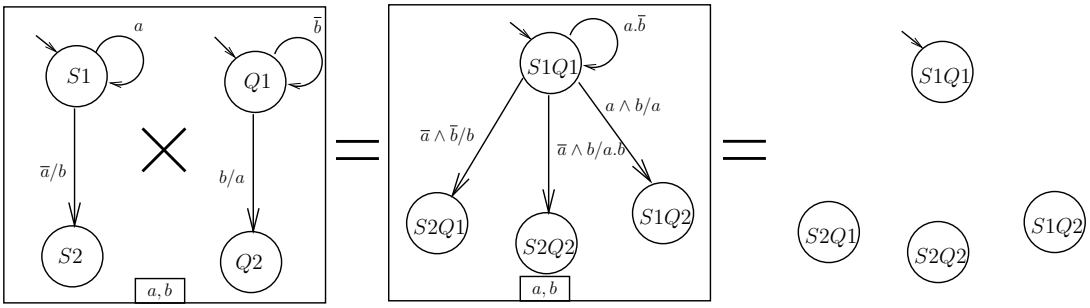


FIG. 4.12 – Mise en évidence du problème de causalité. Supprimer les transitions non-cohérentes peut créer de la non-réactivité.

Regardons, figure 4.12, la transition de l'état $S1Q1$ à $S2Q2$. Pour prendre cette transition, il faut que a ne soit pas présent et que b le soit. Dans ce cas, a et b sont émis. Donc l'absence de a conditionne une transition qui produit a, le tout dans le même instant. Ce problème est appelé problème de *causalité*.

Le problème de causalité est intrinsèquement lié au fait qu'au même instant logique les entrées dépendent des sorties. Supposons que les signaux a et b ne puissent venir d'ailleurs. En ARGOS, cela ce traduit par l'encapsulation : on élimine les transitions non-cohérentes. On obtient donc l'automate de droite. Plus aucune transition n'est accessible depuis l'état initial, y compris la transition qui boucle sur l'état $S1Q1$. L'automate n'est plus réactif.

Supprimer les transitions non-cohérentes peut aussi créer du non-déterminisme. C'est ce que nous illustrons figure 4.13. Pour couper les transitions, on a également supposé que les signaux émis ne peuvent provenir que des deux automates, S ou Q.

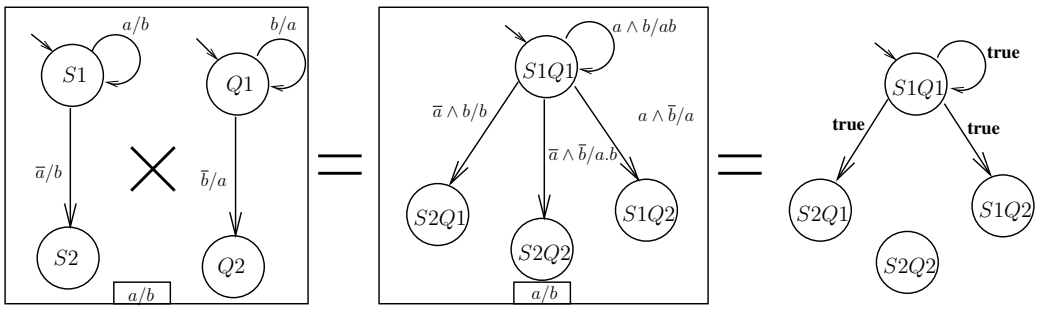


FIG. 4.13 – Mise en évidence du problème de causalité. Supprimer les transitions non-cohérentes peut créer du non-déterminisme.

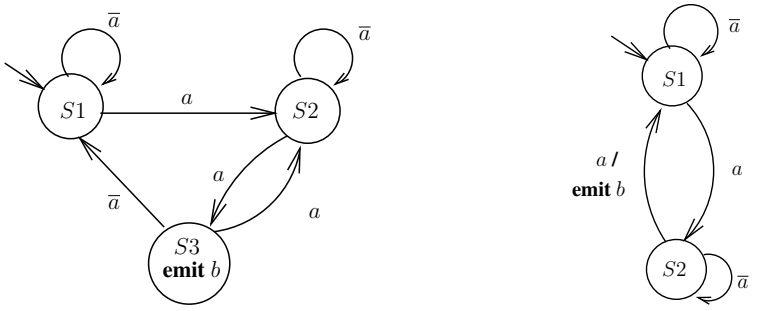
S'il n'y a pas de dépendance cyclique instantanée entre les variables, il n'y a pas de problème de causalité. Dans les exemples précédents, le cycle était de longueur deux : le statut de a dépend de

celui de b qui dépend de celui de a. Cependant, pour des cycles plus longs le problème reste le même. Nous présentons maintenant les différentes solutions qui permettent d'éviter les dépendances cycliques instantanées.

Remarque 1 *Nous avons présenté le produit synchrone et le problème de causalité au travers d'*ARGOS*, mais il est important de bien comprendre que le problème de causalité ne dépend pas du langage. En* ARGOS*, ce problème se manifeste à l'encapsulation : l'automate produit peut devenir non-déterministe ou non-réactif. Mais la cause profonde est bien les cycles de dépendance instantané de signaux.*

Cycles de dépendance instantanée interdits. En LUSTRE, une analyse de causalité est faite statiquement. Elle vérifie que tous les cycles de dépendance de données sont coupés par un retard. Nous revenons plus loin, section 4.4.2, sur le langage LUSTRE. En LUSTRE, il est possible de faire référence à la valeur de la variable à l'instant précédent en utilisant le mot clé `pre`. `pre x` est la valeur du signal `x` à l'instant précédent. Pour qu'un programme soit accepté par le compilateur LUSTRE, il faut et il suffit qu'il y ait au moins un `pre` sur chaque cycle de dépendance. Cette solution n'est pas idéale puisqu'il peut y avoir des cas où la dépendance cyclique est seulement syntaxique. Par exemple si a est produit par : $a := a$ or $\overline{a}$. Dans ce cas a est calculable et $a =$`vrai`, alors que ce programme est rejeté par le compilateur LUSTRE.

Les machines de Moore. Les automates que nous avons montrés jusqu'ici sont des machines de Mealy. Au chapitre 7, nous définissons formellement les machines de Mealy, leur caractéristique principale est que les entrées et les sorties se trouvent sur les transitions. La conséquence immédiate est que les sorties dépendent des entrées. En effet, à chaque instant, suivant les entrées, une transition est empruntée et des sorties sont produites. Au contraire, les machines de Moore produisent leurs sorties sur les états. Pour les automates de Moore, chaque état est étiqueté par la valeur des signaux de sortie. Les sorties dépendent donc des entrées de l'instant précédent. Il n'est donc pas possible de créer des cycles. Ce modèle de calcul est cependant moins pratique puisqu'il introduit un retard à chaque émission de signal. Par exemple, figure 4.14, l'automate de Moore est moins performant que l'automate de Mealy qui réalise la même fonction. Tout d'abord, il nécessite un état de plus mais surtout les b sont émis un instant plus tard.



(a) Automate de Moore (b) Automate de Mealy

FIG. 4.14 – Compteurs de a modulo 2. Les deux automates émettent un b quand deux a ont été reçus.

Le modèle réactif synchrone. Enfin, une solution intermédiaire entre les machines de Moore et les machines de Mealy est celle du modèle réactif synchrone de Boussinot [13]. La caractéristique principale de ce modèle est l'ajout d'un retard pour la réaction à l'absence d'un signal. Par contre la réaction à la présence d'un signal reste instantanée. Il y a donc une dissymétrie entre le test de présence et celui de l'absence. L'automate de la figure 4.15(a) est une machine de Mealy qui ne peut s'écrire tel quel dans le modèle réactif. Dans le modèle réactif, on peut seulement exprimer l'automate de la figure 4.15(b). L'automate de la figure 4.15(b) fonctionne comme une machine de Mealy sur la transition conditionnée par la présence de a et comme une machine de Moore sur la transition conditionnée par l'absence de a.

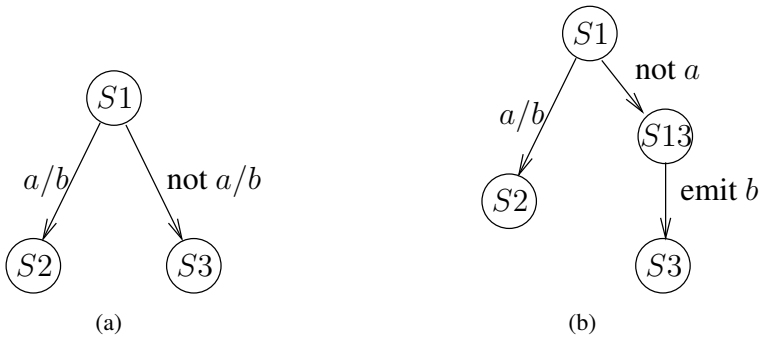


FIG. 4.15 – Modèle réactif : on introduit un retard dans la réaction à l'absence d'un signal. La transition de $S1$ à $S3$ prend donc un instant de plus.

Cette première caractéristique seule ne suffit pas à éliminer le problème de causalité. La seconde caractéristique du modèle réactif est de toujours privilégier l'absence d'un signal. Donc lorsque deux transitions étiquetées avec les conditions a et $\overline{a}$ sont possibles, c'est la seconde transition, $\overline{a}$, qui est exécutée. Ainsi, les réactions sont forcément déterministes. Reprenons l'exemple de la figure 4.13. Tout d'abord, l'ajout d'un retard à la réaction à l'absence crée des états supplémentaire, $S1'$ et $Q1'$, sur les automates S et Q. Voir figure 4.16. Mais là encore, après composition des automates, on obtient du non-déterminisme. Dans l'état $S1Q1$, deux transitions sont possibles. C'est grâce à la deuxième caractéristique que l'indéterminisme est levé : privilégier la réaction à l'absence revient à forcer la transition vers $S1'Q1'$ et donc à éliminer l'indéterminisme.

Enfin, la dernière caractéristique du modèle réactif synchrone est d'ajouter un retard pour la récupération de la valeur du signal. Dans les exemples précédents, les signaux émis sont toujours des booléens. L'information concernant la valeur d'un signal est donc équivalente à celle de sa présence : si le signal est présent, le booléen vaut `vrai`, s'il est absent, il vaut `faux`. Cependant, les signaux ne sont pas toujours des booléens. Dans le cas général, les signaux peuvent donc être présents ou absents et s'ils sont présents, ils ont une valeur. Ajouter un retard pour la récupération de la valeur d'un signal va dans le même sens que l'ajout d'un retard pour la réaction à l'absence. L'idée est que pour être sûr que le signal est absent, il faut attendre la fin de l'instant, de même pour connaître la valeur d'un signal, il faut attendre la fin de l'instant.

Le modèle réactif fournit toujours un ordonnancement cohérent et sans échec. D'un point de vue théorique, ce modèle est moins expressif que les machines de Mealy, il se situe dans la classe des machines de Moore. D'un point de vue pratique, ce modèle est très confortable à utiliser pour des applications où avoir une notion de temps est fondamentale mais où le temps n'est pas critique. REACTIVEML est un langage réactif synchrone. Nous présentons plus en détail REACTIVEML à la

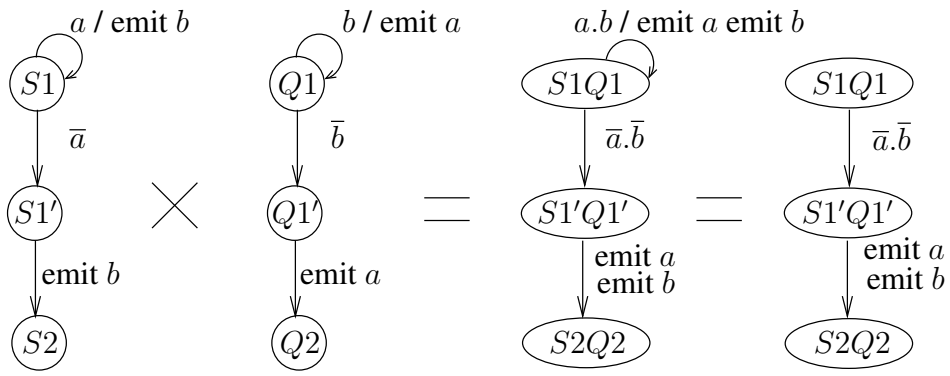


FIG. 4.16 – Grâce aux deux caractéristiques du modèles réactif, les programmes sont forcément déterministes.

section 4.4.3.

4.1.5 Système globalement asynchrone, localement synchrone

Un réseau de capteurs est un système Globalement Asynchrone, Localement Synchrone (GALS). En effet, les nœuds sont des systèmes sur puces contenant une horloge interne et fonctionnent donc de manière tout à fait synchrone. À l'inverse, les capteurs sont très peu synchronisés puisqu'ils ne communiquent que par voie radio et n'ont pas d'horloge en commun.

Le formalisme asynchrone convient bien pour modéliser les communications et interactions entre les nœuds d'un réseaux de capteurs.

À l'inverse, une modélisation à l'aide d'automates synchrones convient bien pour modéliser finement le comportement d'un nœud. Même s'il faut faire attention aux effets de synchronisation entre les nœuds qui pourraient exister dans le modèle et qui ne correspondraient pas à la réalité, il est possible de construire un modèle global d'un réseau de capteurs dans un formalisme synchrone. Pour éviter les effets de synchronisation, on peut décaler les horloges des nœuds à l'initialisation ou encore prendre en compte les dérives d'horloges qui ont lieu au cours de la vie du réseau.

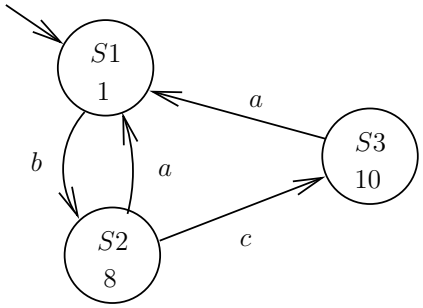
4.2 Extensions pour la modélisation de l'énergie

4.2.1 Les techniques de modélisation de l'énergie

Consommation sur les états de l'automate

Qu'en est-il de la modélisation de la consommation d'énergie ? Nous avons vu que les modèles à base d'automates permettaient de prendre en compte le temps. Celui-ci s'écoule dans les états de l'automate. Un système consomme de l'énergie quand il consomme du temps. Il n'est en effet pas possible, physiquement, de consommer de l'énergie en un temps nul. C'est pourquoi il est logique de modéliser la consommation d'énergie sur les états de l'automates. Dans ce cas, les états sont étiquetés avec des puissances. En intégrant la puissance consommée avec le temps passé dans l'état on obtient l'énergie consommée. Dans le cas synchrone, le temps passé dans chaque état est discret. Les étiquettes correspondant à la consommation sont donc l'énergie dépensée pendant la durée d'une unité de temps. Plus la durée représentée par un instant logique est courte, plus cette énergie s'approche

d'une puissance. Pour obtenir la quantité d'énergie dépensée pour une exécution, il faut sommer toutes les énergies dépensées dans chacun des états parcourus pendant l'exécution. Autrement dit, si les états sont étiquetées avec des puissances, il faut multiplier les puissances par le temps passé dans les états puis sommer les consommations obtenues. La figure 4.17(a) est un exemple d'automate où les états sont étiquetés par des consommations. Le tableau 4.17(b) fournit un exemple de trace de cet automate pour un mode d'exécution synchrone. On a considéré que les consommations correspondent en fait à l'énergie consommée pendant un instant logique. Il s'agit donc de consommations en joules (et pas en watts) qu'il suffit de sommer pour obtenir la consommation d'une exécution.



(a) Automate

entrées		b	a	-	b	c
consommation instantanée	1	8	1	1	8	10
énergie consommée	1	9	10	11	19	29

(b) Exemple de trace et consommation associée

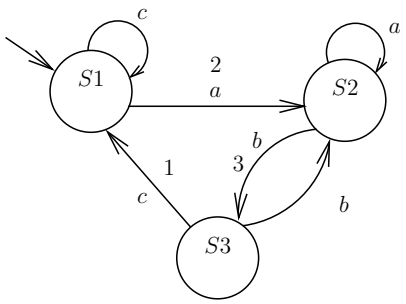
FIG. 4.17 – Modélisation de la consommation sur les états avec un automate synchrone et exemple de trace associée.

Il est également possible d'étiqueter les états d'un automate asynchrone avec des puissances, mais pour ensuite calculer le coût d'une exécution, il faut savoir combien de temps on a passé dans chaque état. Pour les automates synchrones la notion de temps est implicite, mais ça n'est pas le cas pour les automates asynchrone. Pour ceqs derniers, il faut une notion de temps explicite ; on parle alors d'automates temporisés. Il est donc possible d'étiqueter les états d'un automate asynchrone temporisé avec des consommations. À l'exécution, on peut estimer le temps passé dans chaque état et donc l'énergie consommée. Mais dans ce cas, se pose le problème de la représentation de l'espace d'état d'un tel automate. Section 4.2.2, page 76 nous abordons cette question.

Énergie sur les transitions de l'automate

Il est également possible de construire des modèles à base d'automates où les informations concernant la consommation se situent sur les transitions. Dans ce cas, les transitions sont étiquetées avec des énergies et non plus des consommations puisque de telles transitions permettent de modéliser une consommation brutale d'énergie en un temps nul. Ceci permet de prendre en compte dans un modèle assez abstrait des consommations qui semblent instantanées au vu de la précision du modèle. Afin d'obtenir la quantité d'énergie dépensée pour une exécution d'un tel automate, il faut sommer les énergies dépensées sur chacune des transitions empruntées. Voir figure 4.18 un exemple d'automate. Ici, nous n'avons pas besoin d'hypothèse concernant le mode d'exécution (synchrone ou asynchrone) pour calculer le coût d'une exécution.

Remarque : quand l'énergie n'est modélisée que sur des transitions d'un automate, la consommation devient complètement indépendante du temps. Ici, figure 4.18(a), il n'y a aucune information sur l'écoulement du temps.



(a) Automate

entrées		a	a	b	-	c
dépense d'énergie	-	2	-	3	-	1
énergie consommée	0	2	2	5	5	6

(b) Exemple de trace et consommation associée

FIG. 4.18 – Modélisation de la consommation sur les transitions et exemple de trace associée.

Il est également possible de modéliser des consommations à la fois sur les états et sur les transitions d'un automate. Dans ce cas, le coût d'une exécution est la somme des énergies dépensées sur les transitions et sur les états. Pour ces dernières, il faut savoir combien de temps l'automate a passé dans chacun des états.

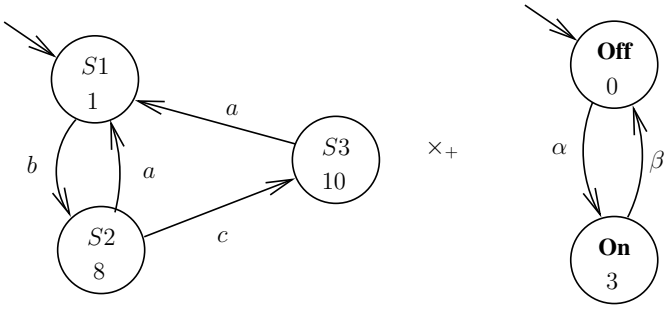
Fonction de composition

Pour un système construit à partir de plusieurs automates dont on fait le produit, une fonction de composition des étiquettes représentant la puissance et/ou l'énergie consommée est nécessaire. Nous distinguons deux cas. Supposons plusieurs composants comportant chacun des informations de consommation (consommation sur les états et/ou énergies sur les transitions). Pour des composants fonctionnant sur la même batterie (dans le contexte des réseaux de capteurs, il pourrait s'agir d'un modèle de radio et d'un modèle de micro-controlleur d'un même nœud), la somme des consommations a un sens. Dans ce cas, nous proposons que les étiquettes concernant l'automate-produit soient la somme des étiquettes de chacun des automates. Voici un exemple, figure 4.19. Pour cet exemple, le mode d'exécution est le modèle synchrone. À chaque instant logique, les deux automates peuvent recevoir des entrées.

Le deuxième cas concerne des automates qui ne fonctionneraient pas sur les mêmes batteries, par exemple deux nœuds d'un réseau de capteurs. Dans ce cas, la somme des énergies consommées n'est plus une information intéressante. Pour estimer la durée de vie du système, on a besoin d'information concernant les consommations des deux automates. C'est donc le n-uplet avec les consommations de chacun des nœuds qui convient. Sur la figure 4.20, on exécute en parallèle deux automates identiques, mais leurs entrées sont différentes. Pour cet exemple aussi, il s'agit du mode d'exécution synchrone.

Remarque : lorsque l'on compose deux automates, garder le n-uplet des consommations permet de garder toute l'information sur les consommations de chacun des automates du produit. On peut donc également garder le n-uplet lorsque les différents automates consomment sur la même batterie puis ne faire la somme qu'à la fin. Cependant, le n-uplet rend le produit d'automates non-commutatif ; il faut donc l'utiliser avec prudence.

Recharge. Ce type de modélisation à base d'automates interprétés permet également de modéliser des systèmes qui se rechargent. Il suffit de considérer des consommations (ou des énergies) négatives.

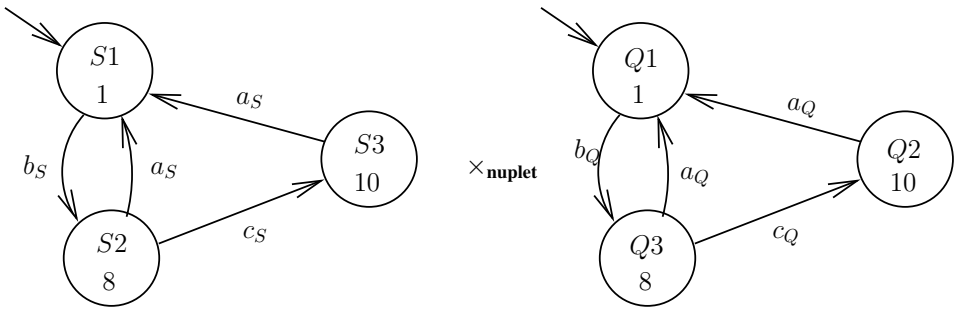


(a) Automates

entrées		b	a	-	b	c
		α	-	β	α	-
consommation instantanée	$1+0$	$8+3$	$1+3$	$1+0$	$8+3$	$10+3$
énergie consommée	1	12	16	17	28	41

(b) Exemple de trace et consommation associée

FIG. 4.19 – Produit de deux automates qui consomment sur la même batterie.



(a) Automates

entrées		b_S	a_S	-	b_S	c_S
		b_Q	c_Q	c_Q	c_Q	a_Q
consommations instantanées	$(1,1)$	$(8,8)$	$(1,10)$	$(1,10)$	$(8,10)$	$(10,1)$
énergies consommées	$(1,1)$	$(9,9)$	$(10,19)$	$(11,29)$	$(19,39)$	$(29,40)$

(b) Exemple de trace et consommations associées

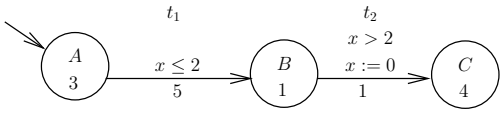
FIG. 4.20 – Produit de deux automates qui consomment sur des batteries différentes.

4.2.2 Discussions

Représentation discrète ou continue de l'énergie ? Avec le formalisme synchrone, le temps est forcément discret. L'énergie consommée, qui augmente proportionnellement au temps dans chaque état ou qui est incrémenté brutalement sur les transitions, est discrète aussi.

Dans le formalisme asynchrone, les automates temporisés permettent de travailler avec du temps dense. Se pose alors le problème de la représentativité. Les automates temporisés permettent moyennant quelques contraintes sur les horloges de représenter du temps dense de manière finie. Qu'en est-il de l'énergie ?

Les LPTA [31, 8, 9] (pour *Linearly Priced Timed Automata*) sont des automates temporisés enrichis avec des informations de coût sur les états et sur les transitions. Sur les états, le coût augmente linéairement avec le temps. Les transitions ont un coût fixe.



(a) Un exemple de LPTA.

$$(A,0) \xrightarrow{\epsilon(1.5),4.5} (A,1.5) \xrightarrow{t_1,5} (B,1.5) \xrightarrow{t_2,1} (C,1.5)$$

(b) Exemple d'exécution pour cet automate.

trace	$\epsilon(1.5)$	t_1	t_2
coût associé	4.5	5	1
coût total	4.5	9.5	10.5

(c) Exemple de trace et consommations associées

FIG. 4.21 – Un exemple de LPTA issu de [9].

La figure 4.21(a) est un exemple de LPTA et la figure 4.21(b) représente une exécution possible de cet automate. Sur la figure 4.21(b), la première transition correspond au passage du temps : on reste 1.5 unités de temps dans l'état de contrôle A. Puisque l'état A coûte 3 unités de coût par unité de temps, cette transition coûte 4.5 unités de coût. Puis, pour cette exécution, on franchit les transitions t_1 et t_2 sans passer de temps dans B et C, le coût n'est donc que celui des transitions. Le tableau 4.2.2 récapitule les coûts dépensés.

Pour de tels systèmes de transitions, les comportements sont représentés par un graphe d'états contenant un état de contrôle, les valuations des horloges et le coût. On a vu qu'il était possible de représenter les comportements des automates temporisés de manière finie. Les auteurs des LPTA[2] proposent des représentations finies du comportement des LPTA qui permettent de trouver le coût minimal pour un comportement souhaité. Ces représentations, qui sont basées sur des régions [9] semblables à celles utilisées pour les automates temporisés, ou sur des zones [49](plus efficace), permettent uniquement de conserver le coût minimal possible pour arriver dans un état de contrôle. Or, cette information ne convient pas pour notre problématique : nous voulons garantir une durée de vie minimale d'un système, c'est donc la consommation pire cas qui nous intéresse, c'est-à-dire le coût

[2]Si la notation LPTA a été introduite par Behrmann et al, l'idée d'étiqueter les automates avec des coûts n'est pas nouvelle et est tout à fait naturelle, leurs contributions sont surtout les représentations de ces LPTA.

maximum. Une piste de recherche consiste à adapter les représentations en régions ou en zones de façon à garder l'information contenant le coût le plus élevé.

Dans cette thèse, nous n'avons cependant pas suivi cette piste. Un formalisme qui permet de représenter de manière symbolique l'énergie est intéressant puisqu'il permet de réduire l'espace d'état. Cependant, les représentations proposées pour les LPTA sont très coûteuses en calculs y compris pour des systèmes de taille modeste. Or, pour les réseaux de capteurs, le problème du passage à l'échelle est fondamental. Plutôt que de chercher une représentation symbolique de l'énergie, le but premier est de réduire la taille du modèle. Il faut donc faire des abstractions. Nous proposons au chapitre 7, un cadre pour des abstractions sur des modèles qui consomment de l'énergie. Notons également que trouver une représentation symbolique des coûts est une piste de recherche orthogonale à celle qui consiste à faire des abstractions.

De plus, il n'est pas fondamental d'avoir une précision infinie quant à la dépense énergétique. En fait, on a besoin de pouvoir représenter une énergie dense uniquement pour des automates temporisés : l'énergie consommée sur les états augmente linéairement avec le temps ; dans les automates temporisés, le temps est dense donc l'énergie l'est aussi. Or, les automates temporisés se composent de façon asynchrone. Pour les automates synchrones, le temps est discret, donc l'énergie aussi. Un réseau de capteurs est un système globalement asynchrone et localement synchrone (4.1.5), donc une modélisation asynchrone d'un réseau de capteurs est une modélisation assez abstraite et dans ce cas, on peut se contenter de modéliser l'énergie consommée uniquement sur les transitions. De cette façon, l'énergie devient une grandeur discrète. C'est d'ailleurs ce que nous avons fait au chapitre 6. Nous expliquons maintenant pourquoi pour une modélisation abstraite d'un réseau de capteurs, modéliser la consommation d'énergie uniquement sur les transitions n'est pas une contrainte trop importante.

Modéliser la consommation sur les transitions ou sur les états ? Le formalisme asynchrone convient bien pour modéliser les communications et interactions entre les nœuds d'un réseaux de capteurs puisque les communications radio ne synchronisent pas les nœuds. Par contre, il est moins pratique pour modéliser le fonctionnement interne des nœuds. Le formalisme asynchrone convient donc bien pour réaliser un modèle abstrait d'un réseau de capteurs. Dans un tel modèle, il y a des actions consommatrices d'énergie qui prennent très peu de temps par rapport à l'échelle de temps du système global. Dans le modèle, on néglige donc le temps pris par ces actions, mais pas leurs coûts. Ces actions se modélisent bien dans un automate par des transitions étiquetés avec des énergies. Considérons par exemple un modèle au niveau réseau des interactions entre les nœuds. Dans ce modèle, on s'intéresse aux messages échangés en faisant abstraction des protocoles d'accès au canal (protocoles MAC). Une émission de paquet prend un temps négligeable à ce niveau d'abstraction, elle est modélisée par l'envoi d'un signal sur une transition. Pour ce modèle, on souhaite prendre en compte l'énergie sans prendre en compte le temps de l'émission d'un paquet alors on étiquette la transition avec le coût d'émission d'un paquet.

À l'inverse, le modèle synchrone est proche de la réalité concernant les communications au sein d'un nœud. Les nœuds d'un réseau de capteurs sont des systèmes qui contiennent une horloge. Une modélisation synchrone d'un réseau est un modèle fin parce que la durée qui s'écoule lors de l'exécution d'un instant est relativement courte. À la limite, on peut avoir un modèle dans lequel un instant logique correspond à un seul cycle d'horloge. Dans ce cas, comme physiquement rien ne se passe de plus court qu'un cycle, on n'a pas de raison de vouloir modéliser des consommations d'énergie "instantanées". Pour nos modèles de réseaux de capteurs, nous n'allons pas jusqu'à un modèle aussi précis qui serait inutilisable en pratique mais les instants logiques correspondent tout de même à des temps relativement courts. Des consommations sur les transitions correspondraient à des actions qui consomment de

l'énergie en moins de temps qu'un instant logique. Dans la pratique, de telles actions consomment trop peu d'énergie pour que l'on ait besoin de les prendre en compte. Dans le formalisme synchrone, nous n'avons donc pas ressenti le besoin de modéliser une dépense d'énergie sur les transitions.

4.3 Techniques d'analyse formelle

En modélisant le système par des automates, on obtient le *graphe d'état* qui représente l'ensemble des comportements du système. C'est l'automate obtenu en faisant le produit de tous les automates qui constituent le système. Sur ce graphe de multiples analyses sont possibles. Tout d'abord, on peut parcourir aléatoirement le graphe à partir d'un état initial, ce qui revient à *simuler* le système. Il est également possible de guider l'exploration du graphe en utilisant des séquences particulières. Enfin, il est possible de vérifier exhaustivement une propriété, c'est la *vérification par modèle* ou *model-checking*.

Il existe deux types de propriétés [48], les propriétés de *vivacité* (*liveness*) et les propriétés de *sûreté* (*safety*)[3]. Les propriétés de vivacité sont de la forme "quelque chose (de bon) arrivera un jour" alors que les propriétés de sûreté sont de la forme "quelque chose (de mauvais) n'arrivera jamais". Il est donc possible de contredire une propriété de sûreté avec une trace finie alors que ça n'est pas le cas pour une propriété de vivacité. De plus, pour les systèmes critiques, ce sont les propriétés de sûreté qui sont intéressantes. En effet, il n'est pas très utile de savoir que "en cas de problème, le système d'alarme se déclenche", il est préférable d'être sûr que " au plus t secondes après un problème, le système d'alarme se déclenche forcément". Pour ces raisons, ce sont des propriétés de sûreté que l'on cherche à prouver.

Prouver une propriété de sûreté revient à prouver la non-accessibilité d'états dans le graphe d'état. On étiquette les états dans lesquels la propriété est violée puis on cherche à prouver qu'il n'existe pas de chemin entre un état initial et un état étiqueté. Si un chemin est trouvé, la propriété est violée et le chemin exhibé fournit un contre-exemple. En effet, un chemin dans un graphe d'état correspond à une exécution possible du système. Puisque les automates fournissent une représentation finie du système, il est possible de prouver qu'aucun état étiqueté n'est accessible depuis un état initial. Cependant, ces algorithmes sont très coûteux parce que l'on se heurte au problème d'*explosion d'état*. Il faut donc faire des *abstractions*. Une abstraction est une approximation qui permet de réduire la taille du modèle. Une abstraction correcte doit être conservative, c'est-à-dire que si la propriété à vérifier est vraie sur le modèle abstrait alors elle l'est sur le modèle détaillé. Au contraire si on prouve que la propriété est violée sur le modèle abstrait, on ne peut rien conclure. Pour une propriété de sûreté, le modèle abstrait ne doit qu'ajouter des comportements.

4.4 Les implémentations

Nous présentons maintenant succintement des langages de programmation qui sont basés sur les formalismes que nous avons présentés section 4.1. Ces langages ont été utilisés dans nos travaux exposés aux chapitres 5 et 6.

4.4.1 IF

Le langage de modélisation IF [16, 87] est basé sur le modèle des automates temporisés avec urgence (voir section 4.1.3).

[3] Toute propriété est la conjonction d'une propriété de sûreté et d'une propriété de vivacité.

Un système IF est constitué de processus. Chaque processus (`process`) a un identifiant unique (son `pid`). Les processus contiennent des variables (`var`) locales typées parmi lesquelles il peut y avoir des horloges. Les processus sont constitués d'un ensemble d'états de contrôle et d'un ensemble de transitions entre les états. Un état initial est marqué par l'attribut `#start`. Un état est stable ou instable : on ne peut pas interrompre l'exécution d'un processus dans un état instable. Les transitions sont instantanées, leur priorité par rapport au temps dépend de l'attribut (`eager`, `delayable` ou `lazy`) qui leur est attaché explicitement, voir section 4.1.3. Les transitions sont conditionnées par des gardes (`provided`) qui peuvent porter sur les horloges et sur les valeurs des variables du processus. L'effet des transitions est décrit par des programmes séquentiels à base d'actions élémentaires.

Pour communiquer, les processus peuvent utiliser des variables globales. Les variables (`var`) globales sont déclarées avant tous les processus. Un autre moyen de communiquer est les signaux. Les signaux sont typés. Les processus peuvent s'envoyer des signaux soit directement (en utilisant les `pid` de l'émetteur et du destinataire) soit en passant par des canaux de communication (les `signalroute`). Un `signalroute` est un canal de communication dans lequel l'émetteur et le ou les destinataires sont prédéfinis. Le comportement d'une telle route est défini par sa politique d'ordonnancement des messages (`fifo` ou multi-ensemble [`multiset`]), sa connectivité (point-à-point [`peer`], `multicast` ou `unicast`), sa fiabilité (avec perte [`lossy`] ou sans [`reliable`]) et le délais d'acheminement (immédiat [`urgent`] ou retardé [`delay`]).

Concernant les données, la notation IF propose des types prédéfinis comme les booléens, les entiers, les réels, les `pid` et les horloges (`clock`). Il est également possible de construire des types structurés (par exemple un tableau). Les `procedures` permettent d'inclure et de manipuler du code externe.

Les observateurs (`observer`) sont des processus particuliers qui peuvent accéder à toutes les variables, au contenu des files d'attentes et à tous les états de contrôles de chacun des processus du système. Un observateur réagit de manière synchrone (c'est-à-dire avec une plus grande priorité que tous les autres processus) aux événements et aux changements de valeurs dans les composants du système. Les observateurs permettent d'exprimer de manière exécutable une propriété à vérifier, d'exprimer des restrictions sur l'environnement et c'est un moyen de contrôler l'exploration du système.

Outils d'analyses disponibles [18, 17].

Une fois qu'un système est modélisé en IF sous la forme d'un système de transitions étiquetées, il est alors possible de construire le graphe d'état et de le parcourir. La figure 4.22 décrit comment s'articulent différentes analyses. À partir du fichier source IF (qui décrit le système : les processus, les signaux etc), on construit grâce au compilateur **if2c** un ensemble de librairies C++. Cette étape permet de construire une représentation des états. Un état du système IF consiste en l'état de chacun des processus, les valeurs de chacunes des variables et de tous les éléments présents dans toutes les files d'attentes, canaux de communication et signaux et les valeurs des horloges. Deux types de représentations des horloges permettent de représenter du temps dense (à l'aide de régions) ou du temps discret. À partir d'un système donné, le compilateur **if2c** construit une fonction qui donne accès aux états initiaux du système et une fonction qui à partir d'un état donné fournit les états successeurs accessibles. Ces fonctions peuvent être manipulées de manière générique, elles portent le même nom quel que soit le système IF. Grâce à ces fonctions (implémentées en librairies C++), on peut construire un programme d'exploration qui permet de parcourir le graphe d'état. Plusieurs types d'explorations sont implémentées dans la boîte à outils IF. Le simulateur interactif exécute le système pas à pas, à chaque pas il propose les différentes transitions exécutables et l'utilisateur choisit d'en exécuter une. Le simulateur aléatoire choisit aléatoirement les transitions qu'il exécute (parmi celle qui sont exécutables). Il est aussi possible de générer le graphe complet (simulation exhaustive). Enfin, nous avons utilisé

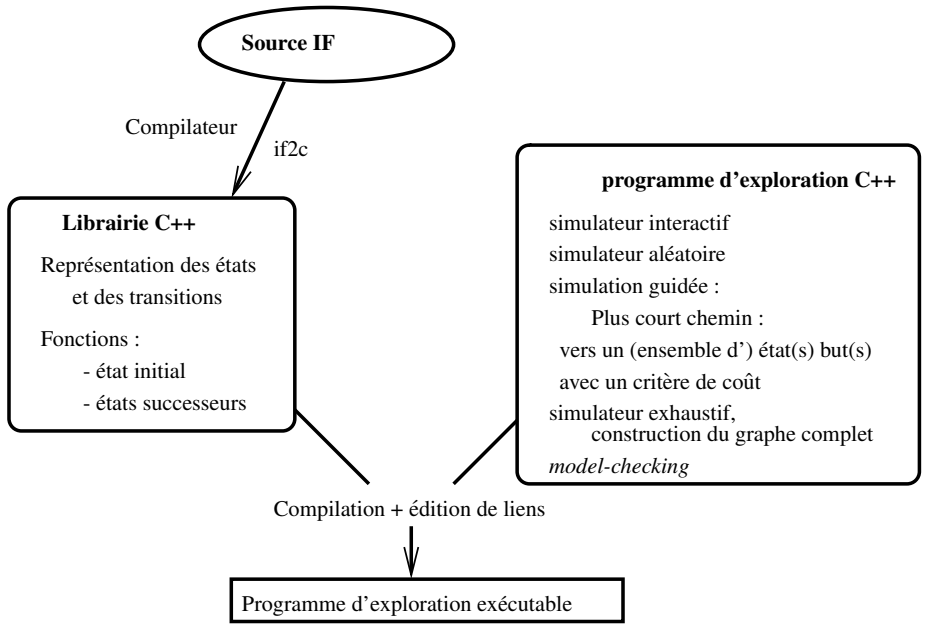


FIG. 4.22 – Outils pour l'exploration de systèmes de transitions étiquetées IF.

(chapitre 6) la simulation guidée. Cet outil permet de guider l'exécution du système vers un état ou un ensemble d'états buts en minimisant un coût. Les algorithmes de plus court chemin implémentés sont Dijkstra et A*, ils permettent de trouver le plus faible coût de l'exécution d'un état initial à un état but. Pour utiliser cet outil il faut exprimer les états buts et le critère de coût. Enfin pour l'algorithme de recherche de plus court chemin A*, une heuristique peut être utilisée afin de privilégier l'exploration vers les chemins les plus prometteurs, c'est-à-dire les chemins qui semblent atteindre un état but au coût minimal. Cette heuristique doit également être décrite par l'utilisateur. L'implémentation du critère discriminant les états buts, du critère de coût et de l'heuristique en C++ est forcément à la charge de l'utilisateur puisque ces critères ne peuvent être exprimés qu'avec les variables du programme IF. Le programme C++ d'exploration (aléatoire, guidée, exhaustive, interactive) est ensuite compilé en un exécutable. Pour le compiler, il faut faire le lien avec les librairies crées à partir du code source IF et qui sont utilisées dans le code du programme d'exploration.

D'autres outils sont disponibles pour des programmes IF mais nous ne les avons pas utilisés donc nous n'en parlons pas ici.

4.4.2 LUSTRE

LUSTRE [38][4] est un langage **synchrone**. Comme pour les automates synchrones que nous avons présentés précédemment, dans un langage synchrone, il existe une horloge globale qui active tous les composants en même temps. À chaque top d'horloge, ceux-ci calculent leurs sorties à partir des entrées et des variables locales puis on passe à l'instant (au top) suivant.

LUSTRE est un langage **fonctionnel**. Les programmes décrivent la fonction d'un système plutôt que la manière de l'obtenir. En particulier, les structures impératives telles que l'affectation ou des boucles itératives, n'existent pas en LUSTRE.

[4]La présentation du langage LUSTRE est inspirée de celles des thèses de Jan Mikáč [63] et Lionel Morel [64].

LUSTRE est un langage **flot-de-données**. Les entrées, sorties et variables locales que manipule un programme LUSTRE sont des flots, c'est-à-dire des suites infinies de valeurs.

Le langage LUSTRE

Les types. En LUSTRE il y a trois types de base : les booléens (`bool`), les entiers (`int`) et les réels (`real`). Il est possible de construire des types tableaux à partir des types de base. Par exemple, si T est un type quelconques, et n une constante définie statiquement, T^n est le type tableau de taille n d'éléments de type T.

Les flots. LUSTRE ne manipule que des flots. Les flots sont des suites infinies de valeurs d'un type donné. Par exemple si **X** est une variable[5], **X** est la suite de valeurs $X_0, X_1, ..., X_i, X_{i+1}, ...$ Tous les X_i ont le même type, celui du flot **X**. X_i représente la valeur du flot **X** à l'instant i. Les constantes sont des flots particuliers, ainsi la constante **1** dénote la suite $(1, 1, 1, ...)$ Les opérateurs sont donc également des opérateurs sur les flots. Par exemple si **X**$= (X_i)_{i\in\mathbb{N}}$ et **Y**$= (Y_i)_{i\in\mathbb{N}}$ alors **X**+**Y**$= (X_i + Y_i)_{i\in\mathbb{N}}$. Ou encore, l'expression **if C then E else F** définit le flot **X** tel que $\forall i \in \mathbb{N}$, **if** C_i **then** $X_i = E_i$ **else** $X_i = F_i$.

Les opérateurs temporels. `pre` permet de faire référence au passé d'un flot. Si $X = (X_i)_{i\in\mathbb{N}}$ `pre`$(X) = (X_{i-1})_{i\in\mathbb{N}}$ où X_{-1} est une valeur indéfinie. Donc `pre`(X) =(`nil`,$X_0, X_1, ..., X_i, ...$). L'opérateur d'initialisation $\rightarrow$ permet alors de définir la valeur d'un flot à l'instant initial. Soit $Y = (Y_i)_{i\in\mathbb{N}}$, $X \rightarrow Y = (X_0, Y_1, ..., Y_i, ...)$.

Les nœuds. L'unité de base d'un programme LUSTRE est un nœud (défini dans le langage par le mot clé `node`). Un nœud contient un en-tête (qui sert de signature : nom du nœud, liste des variables) et un corps. Le corps du nœud est une suite d'équations de la forme <*variable*>=<*expression*>. Les expressions portent sur les flots définis localement par le nœud (les entrées, les sorties et les éventuelles variables locales) et sur les éventuelles constantes globales. La liste des équations forme un système. Pour assurer l'existence d'une solution, deux conditions sont imposées :

- Pour chaque variable de sortie et pour chaque variable locale, il doit y avoir exactement une équation.
- Afin d'éviter les dépendances circulaires instantanées, chaque cycle de dépendance entre variables doit contenir au moins un opérateur `pre`. C'est le retard dont nous parlions à la section 4.1.4, page 68.

Un exemple de programme LUSTRE : l'intégrateur. Le programme LUSTRE suivant (figure 4.23) est constitué d'un seul nœud. Les programmes LUSTRE peuvent être représentés sous forme de circuits synchrones, ce qui fait mieux apparaître l'aspect flot-de-données. La figure 4.24 est le schéma du circuit correspondant au nœud **Integr**.

Le nœud **Integr** calcule la somme des entrées du flot **e**. À l'instant i, $s_i = \sum_{k=0}^{i} e_k$. Voici un exemple de trace du nœud **Integr** :

e	2	1	15	−3	...
p	?	2	3	18	...
s	2	3	18	15	...

[5]Nous appelons "variable" tout flot de base, qu'il s'agisse d'une entrée, d'une sortie ou d'une variable locale.

```
node Integr(e:real) returns (s:real)
var p:real
let
  s = e -> (e+p);
  p = pre s;
tel
```

FIG. 4.23 – Un nœud LUSTRE.

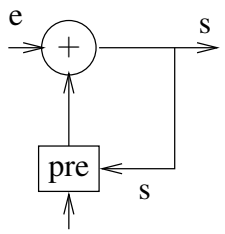


FIG. 4.24 – Le circuit correspondant.

Programme LUSTRE. Un programme LUSTRE est constitué de nœuds, de déclarations globales et de fonctions externes. Il est possible de faire appel à des fonctions externes implémentées dans un autre langage, pour cela il suffit de déclarer le profil d'une telle fonction. Ensuite, la fonction peut être appelée dans un nœud comme n'importe quelle autre expression pour l'affectation d'une variable.

Compilation. Pour compiler un programme LUSTRE, on commence d'abord par le représenter "à plat", sous la forme d'un seul nœud. C'est le produit d'automates synchrones présenté plus haut qui permet de réaliser cette opération. Cette étape produit un ficher "**.ec**" (pour *expanded code*). Ensuite, l'algorithme suivant permet de générer du code séquentiel.

```
initialisation
tant que vrai faire
  - lire les entrées
  - calculer les sorties
  - mettre à jour les mémoires
fin tant que
```

FIG. 4.25 – Séquentialisation d'un programme LUSTRE en boucle simple

La boucle infinie (**tant que** `vrai`) correspond au passage du temps. Cet algorithme fournit une solution au système décrit par les équations du nœud. C'est parce que chaque variable est définie par exactement une équation et qu'il n'y a pas de dépendances cycliques instantanées que cet algorithme fournit une solution du système d'équations.

Comme on l'a vu, un programme LUSTRE est un circuit logique, on peut donc aussi synthétiser un programme LUSTRE afin d'obtenir une implantation matérielle du programme.

Les horloges. Un programme LUSTRE tel qu'on l'a décrit jusqu'ici est un programme synchrone dans lequel tous les nœuds fonctionnent à la même cadence. Il est également possible, en LUSTRE, de tenir compte de sous-systèmes qui évolueraient à des vitesses différentes. Pour cela, il y a les opérateurs **when** (d'échantillonnage) et **current** (de sur-échantillonnage).

Si **E** est un flot quelconque et **B** un flot de booléens alors **E when B** dénote la suite de valeurs extraites de **E** quand **B** est vrai.

Si **X** est un flot sur l'horloge **B** alors **current X** est un flot sur l'horloge globale plus rapide. Les trous sont comblés avec la dernière valeur connue :

E	E_0	E_1	E_2	E_3	E_4	E_5	...
B	t	f	t	t	f	f	...
X = E when B	E_0		E_2	E_3			...
current X	E_0	E_0	E_2	E_3	E_3	E_3	...

Techniques et outils d'analyse disponibles pourLUSTRE

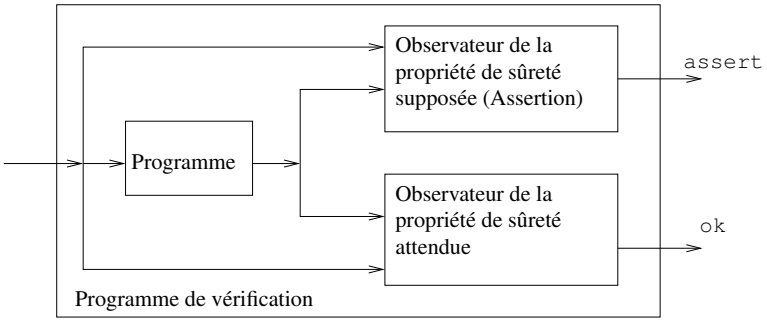


FIG. 4.26 – Schéma de preuve LUSTRE.

Pour vérifier des programmes synchrones, on utilise des *observateurs*. Un observateur est un programme synchrone qui scrute les variables du programme (à vérifier) et produit une sortie booléenne qui reste vraie tant que la propriété attendue est satisfaite. L'avantage principal est que les propriétés peuvent être programmées dans le même langage que le système à valider.

Le schéma de preuve est celui de la figure 4.26. La propriété à vérifier peut n'être vraie que sous certaines hypothèses. Intuitivement, ces hypothèses représentent le contexte ou l'environnement dans lequel le programme sera utilisé normalement. C'est dans cet environnement que le programme est censé fonctionner correctement. On a donc besoin d'un observateur pour vérifier les hypothèses. En LUSTRE, on appelle assertion l'observateur des hypothèses. Le rôle du model-checker est de prouver "quelle que soit une séquence d'entrée, aussi longtemps que `assert` reste vraie, alors la sortie `ok` reste vraie".

Plusieurs outils de vérification automatique sont disponibles pour les programmes LUSTRE. LESAR [39] est un model-checker dédié à LUSTRE. La partie booléenne du programme est complètement reflétée dans LESAR mais tout le reste est abstrait, notamment les variables numériques. NBAC [46] prend également en compte les variables numériques (NBAC utilise les polyèdres convexes pour représenter une abstraction de l'ensemble des valeurs des variables numériques). ASPIC [37] est un outil qui peut aussi vérifier des programmes numériques (contrairement à NBAC et LESAR, ASPIC n'est pas dédié à LUSTRE). ASPIC ne gère pas les booléens. L'avantage de ASPIC par rapport à NBAC est qu'il calcule de meilleures abstractions concernant les valeurs numériques lorsque celles-ci varient de façon linéaire.

4.4.3 REACTIVEML

REACTIVEML[6] [54, 55, 57, 58] est une implantation du modèle réactif synchrone de Boussinot (présenté succinctement section 4.1.4). REACTIVEML étend le langage généraliste OCAML [51, 50] avec des constructions concurrentes basées sur le modèle réactif. Tous les programmes OCAML sont

[6]La présentation de REACTIVEML s'inspire du chapitre de 2 de la thèse de Louis Mandel [55].

des programmes valides en REACTIVEML (dans l'implantation actuelle, les objets, les labels et les foncteurs ne sont pas encore intégrés).

REACTIVEML étend OCAML avec une notion de temps logique, de parallélisme et des signaux de communication. REACTIVEML permet de déclarer des processus qui s'exécutent pendant plusieurs instants logiques alors que les fonctions OCAML sont instantanées. Nous présentons maintenant à l'aide d'exemples simples les principales structures de REACTIVEML.

Les principales structures du langage

Compilation. Le compilateur de REACTIVEML génère, à partir de code REACTIVEML, un fichier source OCAML qu'il faut encore compiler avec le compilateur OCAML pour obtenir un fichier exécutable.

Temps. Le mot clé `pause` permet de faire passer le temps : l'exécution de `pause;` suspend le temps jusqu'à l'instant suivant. Par exemple le processus suivant affiche `hello` au premier instant et `world` au second :

```
let process hello_world =
  print_string "hello␣";
  pause;
  print_string "world"
```

Le processus `hello_world_loop` affiche `hello world` à chaque instant. L'instruction `loop` permet de répéter en boucle une série d'instruction. Il faut cependant que le corps de la boucle ne soit pas instantané pour garantir la réactivité du programme. En effet, s'il y a une boucle instantanée dans un programme, l'exécution d'un instant ne se termine jamais, et donc les instants ne peuvent pas passer : le temps "s'arrête".

```
let process hello_world_loop =
  loop
    print_string "hello␣world";
    pause;
  end
```

Parallélisme. Le symbole `||` est l'opérateur de composition parallèle synchrone. L'exécution du processus `hello_world2` affiche : `hello hello` au premier instant et `worldworld` au deuxième. Les processus qui prennent du temps sont déclarés avec le mot clé `process`. Pour exécuter un tel processus, le mot-clé est `run`.

```
let process hello_world2 =
  run hello_world || run hello_world
```

Communication. Dans l'exemple précédent, il n'y a pas de communication entre les différents processus, dans le modèle réactif synchrone, les processus peuvent communiquer. Dans REACTIVEML, les communications se font via des signaux qui peuvent être valués, ou pas.

Voici deux exemples de programmes qui utilisent des signaux non-valués (ou du moins qui ne se préoccupent pas de leurs valeurs).

```
let process is_present x =
  present x then print_string "Present"

let process is_absent x =
  present x else print_string "Absent"
```

L'instruction `present` permet de tester la présence d'un signal. Le processus `is_present` affiche `Present` si le signal `x` est présent. Le processus dual, `is_absent`, a une sémantique un peu différente : si le signal `x` est présent rien n'est affiché, et si `x` est absent l'affichage de `Absent` est retardé d'un instant. En effet, le modèle réactif (voir paragraphe 4.1.4) sur lequel est basé REACTIVEML introduit un retard à la réaction à l'absence afin de s'affranchir du problème de causalité présenté plus haut.

Les instructions `emit` et `await` permettent d'émettre et d'attendre un signal.

Il est également possible de définir des signaux valués. Mais dans ce cas, il faut préciser à la déclaration du signal, quelle est la fonction de combinaison ; en effet, plusieurs émissions peuvent avoir lieu sur le même signal au même instant. Dans ce cas le processus qui attend le signal recevra comme valeur la combinaison de tous les signaux émis. Pour définir un signal on utilise l'instruction suivante :

```
signal <name> default <value> gather <function> in ...
```

Par exemple, le signal `sum` calcule la somme des signaux émis.

```
signal sum default 0 gather (+) in ...
```

Si au cours d'un instant, une seule émission (`42`) a lieu, la valeur récupérée est `(+) 0 42 = 0 + 42 = 42`. Car `0` est la valeur par défaut.

La récupération de la valeur d'un signal est retardé d'un instant pour éviter les problèmes de causalité. Par exemple, le programme suivant est causal :

```
await s(x) in emit s(1+x)
```

Donc, si `s = 42` pendant l'instant courant, le programme émet `s(43)` à l'instant suivant. Pour écrire ce même programme en LUSTRE, il aurait fallu introduire un retard de façon explicite : le programme suivant n'est pas correct en LUSTRE :

```
let
  s = 1 + s;
tel
```

Par contre, en introduisant un `pre`, on casse le cycle de dépendance instantanée et le programme devient correct :

```
let
  s = 1 + pre s;
tel
```

Le modèle réactif synchrone décrit implicitement le second programme.

Une fonction de composition souvent utilisée lorsque plusieurs valeurs peuvent être émises au même instant sur le même signal est la concaténation. Ainsi, les processus qui reçoivent la valeur du signal, reçoivent la liste des valeurs émises et c'est à eux de combiner comme ils le souhaitent les éléments de la liste.

```
signal s default [] gather fun x y -> x :: y in ...
```

Mode d'exécution

En REACTIVEML, contrairement à LUSTRE, les programmes sont ordonnancés dynamiquement. Pour exécuter la composition parallèle de plusieurs processus, des optimisations sont possibles. En LUSTRE, à la compilation le programme est transformé en un seul nœud LUSTRE, à l'exécution il n'y a donc plus de choix possibles. Quel que soit le choix de l'ordonnanceur, le résultat d'un programme REACTIVEML est le même, REACTIVEML est bien déterministe. Prenons un exemple.

```
(await s1)
||
(await s0; emit s1)
||
(for i = 1 to 1000 do pause done emit s0)
```

Au 1000ième instant, `s0` est émis, donc au 1001ième instant `s1` est émis. Si aucune optimisation n'est implémentée, à chaque instant on teste la présence de `s0` puis celle de `s1`. C'est ce qui se passe en LUSTRE. Avec l'optimisation **inter-instants** implémentée dans REACTIVEML, une liste d'attente est créée pour chaque signal. Pour chaque signal `s`, on met dans sa liste d'attente les processus qui sont en attente de `s`. Ici, au premier instant le premier processus (en haut) est mis dans la liste de `s1` et celui du milieu dans celle de `s0`. Tant que `s1` n'est pas émis on ne regarde plus le premier processus, on ne teste plus la présence de `s1`. Idem pour `s0`. Quand `s0` est émis, on débloque les processus qui étaient dans la file d'attente de `s0`, ici le processus du milieu ; `s1` est alors émis à l'instant suivant, ce qui débloque le premier processus.

Pour cet exemple, l'optimisation inter-instants permet d'économiser $1000+999$ tests en contrepartie de la création de deux listes. Cependant, suivant les programmes, le surcoût dû à la création des listes peut s'avérer plus coûteux que le gain. Intuitivement, c'est le cas quand les signaux sont émis très souvent. Dans ce cas, mieux vaut effectuer le test à chaque fois puisqu'il a de bonnes chances de s'avérer positif.

Nous avons expliqué rapidement l'optimisation inter-instants parce que, selon nous, elle explique en partie la différence d'efficacité entre LUSSENSOR et GLONEMO, voir section 5.3.3. D'autres optimisations de l'ordonnancement contribuent certainement à l'efficacité du compilateur REACTIVEML et donc de notre simulateur GLONEMO mais nous ne les présentons pas ici. Le lecteur intéressé pourra lire [55].

4.4.4 LUCKY

Les différents formalismes synchrones que nous avons présentés sont tous déterministes. C'est une propriété fondamentale attendue pour les systèmes réactifs opérationnels. Cependant, au cours du développement et de la validation de ces systèmes, exprimer et simuler du non-déterministe peut s'avérer très utile. Nous présentons rapidement LUCKY [74,76,73,44], un langage qui permet de décrire des systèmes non-déterministes avec une approche synchrone des réactions. Nous avons utilisé LUCKY au chapitre 5 pour décrire le comportement de l'environnement. LUCKY est en fait le langage cible de LUTIN. LUTIN est un langage de plus haut niveau mais comme nous avons programmé directement en LUCKY, nous le présentons pas ici.

Principes du langage

LUCKY est un langage synchrone qui permet d'exécuter des systèmes décrits de manière non déterministe. Le but de LUCKY n'est pas de savoir construire un graphe d'état représentant tous les

états du système, mais de savoir exécuter un système de transitions. Comme en LUSTRE, il existe une notion de temps logique discret : à chaque instant, on effectue une transition (instantanée) sur laquelle on calcule les valeurs des variables et on change (éventuellement) d'état. La différence principale avec LUSTRE est l'indéterminisme des automates LUCKY. Celui-ci peut provenir de deux sources :

1. Les équations qui permettent de calculer les valeurs des variables peuvent avoir plusieurs solutions. Un solveur booléen et numérique génère une solution possible du système d'équations. La seule restriction est que les contraintes numériques doivent être linéaires.
2. Plusieurs états de contrôle peuvent être accessibles depuis un état donné. Dans ce cas, l'exécution d'un programme LUCKY choisit un état parmi les états accessibles. Il est également possible de mettre des poids sur les transitions. Ces poids s'utilisent comme des probabilités, les transitions auxquelles on a attribué des poids forts sont plus probables que celles qui ont des poids faibles. La différence avec les probabilités est que l'on n'impose pas que la somme des poids soit normalisée ce qui facilite la construction du produit.

Comme dans les autres formalismes synchrones, les automates LUCKY peuvent communiquer par diffusion synchrone. La composition parallèle d'automates se fait avec le produit synchrone classique[7].

Quelques éléments de syntaxe

La syntaxe complète de LUCKY est disponible dans [45].

Pour un système LUCKY, il y a trois catégories de variables, les entrées (`inputs`), les sorties (`outputs`) et les variables locales. Les états sont appelés `nodes`. Il faut définir l'ensemble des états initiaux, `start_node`, cet ensemble ne doit pas être vide. Il faut alors définir les transitions (`transition`) entre les états (`etat1 -> etat2`), elles ont plusieurs attributs : les poids (`~weight`) et les conditions (`~cond`). Les conditions peuvent faire intervenir toutes les variables ainsi que leurs valeurs précédentes (`pre pre x` dénote la valeur de `x` deux coups auparavant) et peuvent aussi utiliser la structure conditionnelle `if then else`. Bien entendu, le solveur ne fournit des solutions que pour les valeurs courantes des variables.

connexion à REACTIVEML

Il est désormais possible d'utiliser un programme LUCKY dans un programme REACTIVEML. Pour ce faire, voici comment on le déclare dans le programme REACTIVEML :

```
external.luc cloud_lucky
  {wind_x : float; wind_y : float;}
  {cloud_x: float; cloud_y: float;} = ["cloud.luc"]
```

Ici, un processus `cloud_lucky` est créé à partir du fichier LUCKY `cloud.luc`. Ce processus a les entrées `wind_x` `wind_y` et produit les sorties `cloud_x` et `cloud_y`. Pour exécuter un tel processus en REACTIVEML, on utilise la commande :

```
run (cloud_lucky Luc4ocaml.StepInside (wx, wy) (x, y))
```

Les sorties produites (`x` et `y`) le sont suivant les contraintes du programme LUCKY avec les entrées (`wx` et `wy`) fournies par le programmes REACTIVEML.

[7] Un autre produit est possible. Comme les modèles LUCKY que nous avons créés sont constitués d'un seul automate, nous ne détaillons pas cet aspect.

Chapitre 5

Modélisation dans l'approche synchrone : le simulateur GLONEMO

Sommaire

Dans ce chapitre nous proposons un modèle de réseaux de capteurs qui prend en compte des modèles précis de chacun des composants du système. Ce modèle global et précis permet une modélisation fiable de la consommation d'énergie. De plus, dans notre modèle les composants sont clairement définis. Nous avons utilisé le langage réactif synchrone REACTIVEML, pour décrire notre modèle. Nous avons donc obtenu un modèle exécutable ce qui permet d'effectuer des simulations. Nous avons nommé ce simulateur GLONEMO, pour GLObal NEtwork MOdel.

Dans la section 5.1, nous présentons GLONEMO au travers d'un cas d'étude. Ensuite, section 5.2, nous montrons grâce à des simulations GLONEMO qu'il est indispensable de prendre en compte

l'environnement pour modéliser un réseau de capteurs. Nous avons ensuite transformé GLONEMO en un modèle LUSTRE, nous décrivons LUSSENSOR et les difficultés rencontrées pour construire un tel modèle section 5.3. Section 5.4, nous expliquons pourquoi REACTIVEML est un langage de programmation qui convient à la réalisation de simulateurs.

5.1 Le simulateur de réseaux de capteurs, GLONEMO

Le contenu de cette section provient en partie de la référence [78].

5.1.1 Exemple

Nous présentons GLONEMO au travers d'un exemple de réseau de capteurs. Le cas d'étude que nous avons choisi est représentatif des réseaux de capteurs.

Le but du réseau est de détecter la présence d'un nuage radioactif. Les nœuds disposent de capteurs pour détecter les radiations. Si le nuage est détecté, ils doivent alerter le puits (point de collecte des informations).

Le protocole de routage qui permet de trouver une route entre le capteur qui donne l'alerte et le puits est la diffusion dirigée ("*directed diffusion*"). Nous avons présenté ce protocole section 2.1.2.

Le protocole d'accès au medium (MAC) que nous avons choisi est un protocole à échantillonnage de préambule. Il s'agit du protocole décrit section 2.1.3 et présenté figure 2.4. Selon nous, ce type de protocole est représentatif des protocoles d'accès au médium pour réseaux de capteurs : c'est un protocole conçu pour fonctionner avec peu d'énergie et qui permet d'éviter un certain nombre de collisions grâce à un système d'émission après un délai aléatoire. Pour décrire ce type de protocole, notre formalisme doit permettre de prendre en compte le temps et des opérations aléatoires.

Le modèle de propagation que nous avons considéré pour cet exemple est celui du "disque unité"[1]. Dans ce modèle, tous les nœuds qui sont à une distance inférieure ou égale à r de l'émetteur reçoivent le paquet émis, les autres ne reçoivent rien. r représente la portée radio maximale. Pour les collisions, nous avons considéré que dès que la réception de deux signaux se chevauche, le récepteur ne peut en décoder aucun. Dans ce cas, il reçoit un bruit équivalent à un préambule. Nous avons aussi considéré qu'un nœud qui émet ne peut rien recevoir.

Un modèle précis d'un réseau qui permet d'effectuer des analyses de consommation doit prendre en compte une modélisation du matériel. Si l'on considère que la plus grande partie de l'énergie dépensée par un nœud est utilisée pour communiquer (c'est la cas selon [3]), un modèle de consommation de la radio est indispensable. Pour ce modèle, on ne peut pas négliger l'énergie et le temps nécessaire pour passer d'un état de consommation à l'autre. Nous montrons comment notre formalisme permet de prendre en compte le temps et l'énergie consommés par les différents éléments matériels.

5.1.2 Formalisme du modèle

Voici le formalisme que nous avons choisi pour programmer GLONEMO. Tous les éléments ont été détaillés au chapitre 4.

Notre modèle est constitué d'automates interprétés. Ces automates communiquent via des signaux. Ces signaux sont diffusés de façon synchrone, chaque automate peut recevoir les signaux envoyés et tous les automates réagissent en même temps aux signaux. Le modèle de compostion des automates est

[1] Ce modèle est celui impléménté dans la première version. Depuis que Olivier Bezet a repris GLONEMO, un modèle plus réaliste a été implémenté

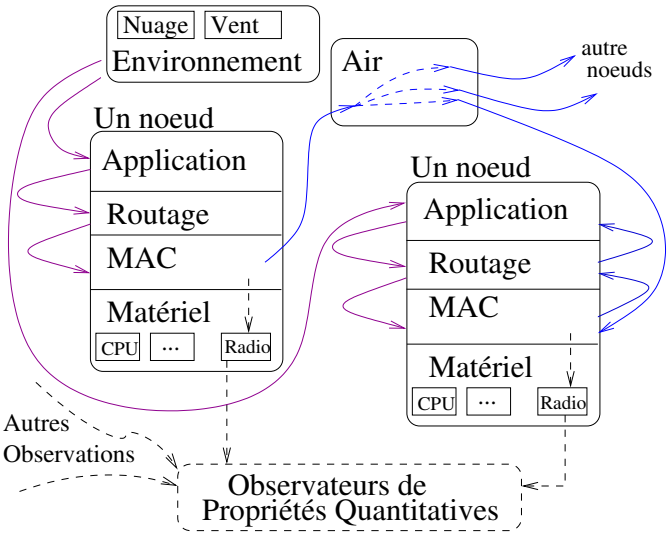


FIG. 5.1 – Processus et Communications

le modèle réactif synchrone (section 4.1.4, page 71). Nous avons ajouté sur les états des étiquettes qui correspondent à l'énergie consommée par le modèle quand l'automate se trouve dans cet état. Nous avons implémenté notre modèle à l'aide de REACTIVEML (section 4.4.3). Nous avons utilisé LUCKY (section 4.4.4) pour modéliser le comportement non-déterministe de l'environnement.

5.1.3 Un modèle global : démonstration sur notre exemple

Nous décrivons la modélisation de notre cas d'étude rappelé section 5.1.1 à l'aide du formalisme de la section 5.1.2.

Principes

Notre modèle global est un ensemble de processus communicants écrits en REACTIVEML ou en LUCKY. La figure 5.1 montre les processus et les informations qu'ils échangent. Voici les éléments qui composent le modèle :

- Un modèle d'un nœud, exprimant le comportement fonctionnel et les propriétés de consommation. Ce modèle a une instance pour chaque nœud du réseau. Toutes les instances (représentant tous les nœuds du réseau) sont des processus qui évoluent en parallèle.
- Un modèle du médium, c'est-à-dire du canal radio. Ce modèle contient à la fois l'information sur la topologie du réseau et les hypothèses qui sont faites sur la propagation des ondes radio. C'est ici que sont prises en compte les collisions. Ce processus reçoit potentiellement des signaux des couches MAC de tous les nœuds et envoie des signaux destinés aux couches MAC de certains nœuds (suivant la topologie, les collisions...).
- Un ensemble d'observateurs. Ce sont des processus qui ne modifient pas le comportement du système mais observent les états courant des différents processus pour en déduire toutes sortes d'informations (consommation, nombre de collisions...).
- Le modèle de l'environnement, écrit en LUCKY. C'est un processus comme les autres qui envoie des signaux à la partie application de chacun des nœuds.

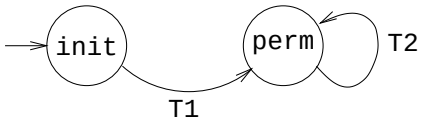


FIG. 5.2 – L'automate décrit par le processus `wind` en LUCKY

Le modèle d'un nœud est décrit plus loin. Un processus `node`[2] est la composition parallèle d'un processus `application` qui décrit le fonctionnement de l'application, d'un processus `mac` et d'un processus `routing` qui représentent les comportements fonctionnels des protocoles MAC et routage. Pour émettre un paquet, des signaux sont envoyés, comme décrit figure 5.1, du processus `application` jusqu'au processus `air` en passant par les processus `routing` et `mac`. En cas de réception c'est le processus `mac` qui émet un signal à destination du `routing` puis éventuellement vers le processus `application`.

De plus, ce sont ces modèles qui pilotent les modèles de consommation. Par exemple le modèle de la radio, figure 5.6, est piloté par le processus `mac`. Enfin pour chaque nœud, un processus contrôle l'état de la batterie et stoppe l'exécution du nœud dès qu'il n'a plus d'énergie.

Environnement

L'environnement qu'il faut prendre en compte pour notre cas d'étude est le nuage radioactif ou polluant que le réseau est chargé de détecter. Nous modélisons cet environnement par un nuage qui se déplace sous l'effet du vent. Notre modèle est constitué de deux processus, un processus `wind` qui modélise le comportement du vent et un processus `cloud` qui modélise le mouvement d'un nuage sous l'effet d'un vent à deux dimensions.

Voici le programme LUCKY correspondant au processus `wind` :

```
inputs { }

outputs {
   wind_x : float;
   wind_y : float;
}

start_node { init }

transitions {
 init -> perm // transition T1
   ~cond
     wind_x = 0.0 and wind_y = 0.0;

 perm -> perm // transition T2
   ~cond
     abs (wind_x - pre wind_x) < 1.0 and
     abs (wind_y - pre wind_y) < 1.0 and
     abs wind_x < 5.0 and abs wind_y < 5.0
```

[2]Les noms écrits en caractères `typewriter` sont des variables de l'implémentation.

```
}
```

Ce processus n'a pas d'entrées et produit les signaux `wind_x` et `wind_y` qui représentent un vent à deux dimensions dont la vitesse et la direction ne varient pas trop brutalement. Ce programme définit l'automate de la figure 5.2. Les sorties doivent vérifier les contraintes exprimées sur les transitions, T1 et T2, de l'automate. Ces contraintes sont désignées par le mot clé `~cond` dans le code LUCKY. Ici pour que le vent ne varie pas trop brutalement, on impose que la valeur courante de `wind_x` (respectivement `wind_y`) ne s'éloigne pas trop de la valeur l'instant précédent, `pre wind_x` (respectivement `pre wind_y`).

Le nuage est modélisé par un disque qui se déplace sous l'effet du vent. Voici le code du processus `cloud`:

```
inputs {
  wind_x : float;
  wind_y : float;
}

outputs {
  cloud_x: float;
  cloud_y: float;
}

start_node { init }

transitions {
 init -> perm // transition T1
   ~cond
     cloud_x = 0.0 and cloud_y = 0.0;

 perm -> perm // transition T2
   ~cond
     (if wind_x >= 0.0
      then ((cloud_x - pre cloud_x) >= 0.0
        and (cloud_x - pre cloud_x) <= wind_x)
      else ((cloud_x - pre cloud_x) <= 0.0
        and (cloud_x - pre cloud_x) >= wind_x))
    and
     (if wind_y >= 0.0
      then ((cloud_y - pre cloud_y) >= 0.0
        and (cloud_y - pre cloud_y) <= wind_y)
      else ((cloud_y - pre cloud_y) <= 0.0
        and (cloud_y - pre cloud_y) >= wind_y))
}
```

Le processus `cloud` reçoit les signaux provenant du processus `wind` et produit les coordonnées du centre du disque nuage. Il définit le même automate (figure 5.2) dans lequel `wind_x` et `wind_y` sont les entrées et `cloud_x` et `cloud_y` sont les sorties. À l'instant initial, le nuage a la position $(0.0, 0.0)$ puis il se déplace sous l'effet du vent : ses coordonnées (`cloud_x`, `cloud_y`) évoluent en fonction de celles du vent.

Application

Le processus `application` reçoit des signaux du nuage. En fonction de ces signaux, il décide s'il faut envoyer une alarme et à qui - typiquement au puits. Ce processus compare la position du nœud avec celle du nuage pour savoir s'il est sous le nuage. Pour éviter d'envoyer trop de paquets qui surchargeraient le canal radio, il n'envoie une alarme que s'il détecte un front montant, c'est-à-dire que s'il n'était pas sous le nuage à l'instant précédent et qu'il y est à l'instant courant. Dans ce cas, il crée un paquet. Le paquet contient des champs tels que le destinataire, le contenu, etc. Le processus `application` envoie alors ce paquet au processus `routing` chargé de trouver une route vers la destination. L'envoi de ce paquet fait référence à l'émission de signaux REACTIVEML entre les processus et pas à l'émission de messages via les ondes radio.

Routage

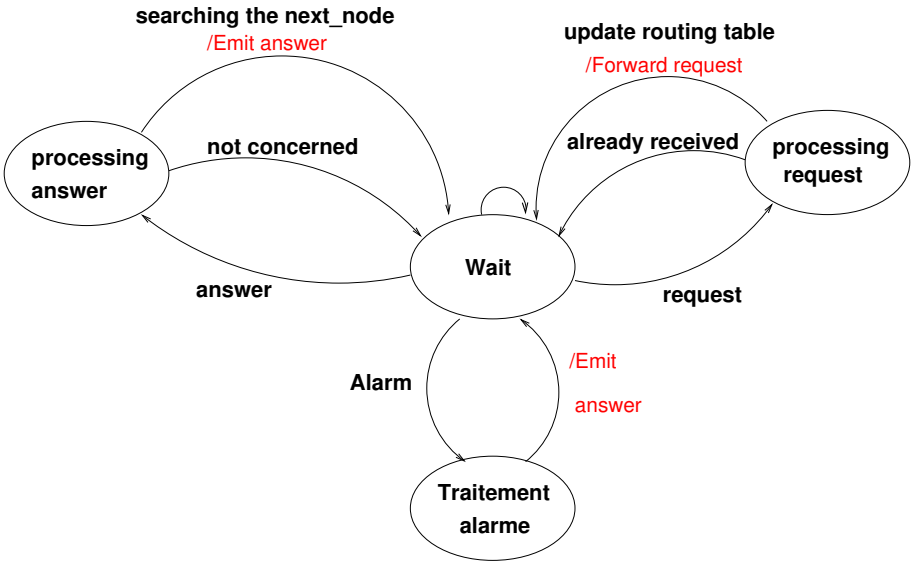


FIG. 5.3 – Protocole de routage

Les protocoles de routage implémentés sont l'inondation et la diffusion dirigée. Nous avons présenté ces protocoles au chapitre 2, sections 2.1.2 et 2.1.2, page 23. Ici, nous nous contentons de rappeler ces algorithmes puis nous expliquons rapidement l'implémentation.

Le puits envoie à tout le réseau des requêtes. Ces requêtes sont réémises à l'ensemble du réseau par tous les nœuds. Ces requêtes permettent aux nœuds de savoir par lequel de leurs voisins ils pourront joindre le puits. En effet, chaque nœud tient à jour une table de routage contenant l'identifiant du voisin par lequel il a reçu la requête (aussi appelée intérêt) en premier. On appelle ce nœud `next_node`. Lorsque le nœud a un message à envoyer au puits, il l'envoie à son `next_node` qui l'enverra lui-même à son `next_node`, etc.

Nous avons décrit cet algorithme par un automate (voir figure 5.3). Le processus `routing` correspondant au protocole de routage dans GLONEMO est cet automate implémenté en REACTIVEML. Il communique avec le processus `mac` et avec le processus `application`. Les signaux de sorties émis vers le processus `mac` sont écrits après un " / " sur les transitions. L'automate du routage est dans l'état `Wait` quand rien ne se passe. Les données des paquets reçus au niveau MAC sont des entrées

du processus `routing`. Deux cas sont possibles : “requête” ou “réponse”, soit il s’agit d’une requête provenant du puits (éventuellement après plusieurs retransmissions), soit il s’agit d’une réponse à une requête.

Dans le premier cas, on teste si la requête a déjà été reçue. Si oui, on l’ignore. Sinon, les tables de routage sont mises à jour : l’émetteur de la requête est mémorisé. C’est le `next_node`. Puis, le paquet est transmis au processus `mac` chargé de le ré-émettre à l’ensemble de ses voisins (en *broadcast*).

Dans le second cas, il s’agit d’une réponse à une requête : par exemple le nœud voisin a une alarme à envoyer au puits. Dans ce cas on traite cette réponse : si l’on est destinataire du message, on envoie au processus `mac` le paquet à envoyer où le champ destinataire est le `next_node`. Sinon, le paquet est ignoré.

Enfin, le processus `routing` peut recevoir un signal de l’application qui veut envoyer un paquet. Dans notre cas, il peut s’agir d’une alerte provenant du capteur qu’il faut remonter au puits. Le processus `routing` traite alors l’alerte en envoyant un signal au protocole MAC indiquant qu’il faut envoyer un paquet. Le destinataire de ce paquet est le `next_node`.

MAC

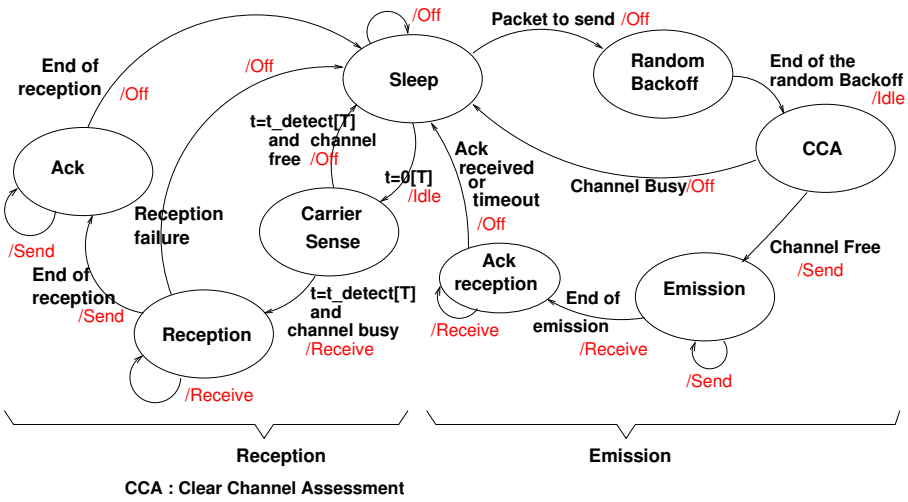


FIG. 5.4 – Protocole MAC

Le protocole MAC est le protocole à échantillonnage de préambule détaillé section 2.1.3, page 26. Nous avons également implémenté un mécanisme d’acquittement (voir section 2.1.3 page 26). La figure 5.4 est l’algorithme MAC décrit par un automate. Le processus `mac` est cet automate implémenté en REACTIVEML. Le processus `mac` exploite des signaux provenant du canal radio, comme par exemple, *channel free*, *channel busy*. Il reçoit également des signaux provenant du processus `routing` du type *packet-to-send*. Le processus `mac` émet également des signaux pour le processus `routing` et vers le canal radio. Enfin, les signaux de sortie du `mac` sont également exploités par le modèle de consommation de la radio.

Modélisation du canal radio

Le canal radio est modélisé par un processus `Air`. Ce processus calcule les positions de tous les nœuds. Il en calcule donc pour chaque nœud l’ensemble de ses voisins, c’est-à-dire l’ensemble des

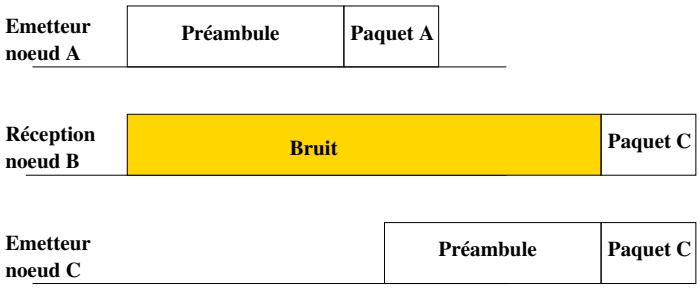


FIG. 5.5 – Propagation radio

nœuds à portée radio. Cette opération n'est réalisée qu'une fois parce que nous faisons l'hypothèse que les nœuds sont fixes. Ce processus reçoit l'ensemble des signaux radio émis : les préambules et les paquets. Dès le début de l'émission d'un message, il envoie aux nœuds voisins de l'émetteur l'information indiquant qu'ils reçoivent un signal. Et c'est seulement à la fin de l'émission que l'information indiquant le contenu du paquet est émise aux nœuds qui n'ont pas subi de collisions. Les récepteurs qui ont reçu un autre signal ont simplement reçu un bruit qui a duré autant que toutes les émissions. Si c'est un préambule, ce signal est considéré comme du bruit. Par exemple, sur la figure 5.5 le nœud B est à portée radio des nœuds A et C qui émettent chacun un préambule suivi d'un paquet. Le nœud récepteur (nœud B) va recevoir le premier préambule (qui est équivalent à un bruit). Le paquet A entre en collision avec le préambule du nœud C, le nœud B continu de recevoir du bruit. Enfin, quand le nœud C envoie son paquet, celui n'est pas brouillé, le nœud B le reçoit donc correctement. Il faut bien comprendre que ce schéma (figure 5.5) correspond à ce que le processus `Air` envoie au nœud, mais il se peut que le nœud B ne capte pas le paquet C si par exemple il a éteint sa radio entre temps.

Modélisation du matériel

Le modèle présenté pour l'application et les protocoles est constitué des algorithmes qui seraient réellement implémentés sur les nœuds du réseaux. Pour modéliser le matériel, nous aurions pu utiliser une description précise comme par exemple le modèle VHDL (*VHSIC hardware description language*). Cependant ce niveau de détail rendrait le modèle trop complexe pour être utilisé en pratique. De plus, une modélisation aussi précise n'est pas toujours nécessaire. Nous présentons ici une modélisation assez abstraite du matériel.

Nous modélisons la consommation à l'aide d'automates dont les états sont étiquetés par des consommations. Ces modèles ont été présentés chapitre 4 et ils sont formalisés chapitre 7. En fonction du temps passé dans un état et de la puissance consommée sur cet état, on en déduit l'énergie dépensée.

Radio. Le modèle de consommation implémenté est l'automate présenté figure 5.6. Cet automate modélise les différents états de la radio, cependant les valeurs de consommation indiquées prennent en compte la consommation de tout le nœud. Ces consommations ont été mesurées sur le module MC13192 de Freescale [33], mais nous ne voulons pas discuter ici de la justesse des valeurs numériques. Les états principaux de l'automate sont les états `sleep`, `transmit`, `receive` et `idle`. Les changements d'état sont parfois coûteux en énergie et en temps. Nous modélisons ce surcoût en ajoutant un état supplémentaire. Ces états, dessinés avec des pointillés, sont transitoires. Pour chaque transition par ce type d'état, le temps passé dans ces états est fixe.

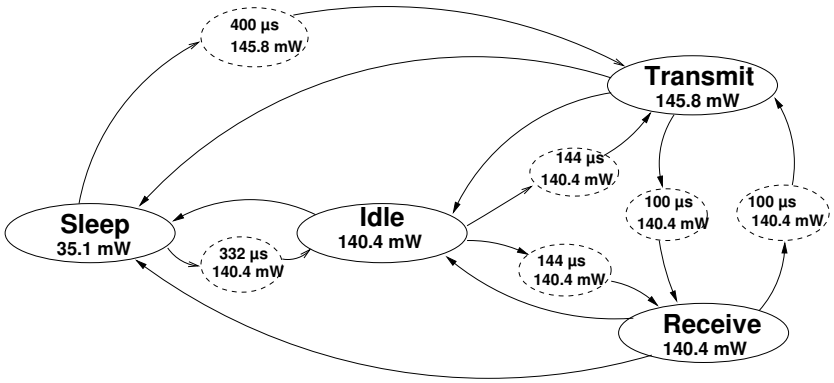


FIG. 5.6 – Modèle de consommation de la radio

Dans l'état `sleep`, la radio est éteinte et consomme presque zéro. Dans l'état `idle`, la radio consomme autant qu'en réception, mais il n'y a aucun signal sur le canal. La radio est dans cet état lorsqu'elle sonde le canal ou lorsqu'elle attend un signal. Cet état correspond à de l'écoute inutile que les concepteurs de protocoles cherchent à réduire. Dans l'état `transmit`, le nœud émet un signal sur le canal radio. La consommation associée dépend de la puissance de transmission. Ce modèle implique donc que les nœuds émettent tout le temps à la même puissance. Nous pourrions imaginer qu'un nœud puisse changer dynamiquement sa puissance de transmission. Notre modèle pourrait aisément intégrer ce type de nœud qui auraient dans ce cas plusieurs états `transmit` ayant chacun des consommations différentes. La seule contrainte étant que le nombre d'état reste fini. Dans l'état `receive`, le nœud reçoit et décode un signal radio. Nous modélisons l'émission d'un préambule ou d'un paquet de données par le même état.

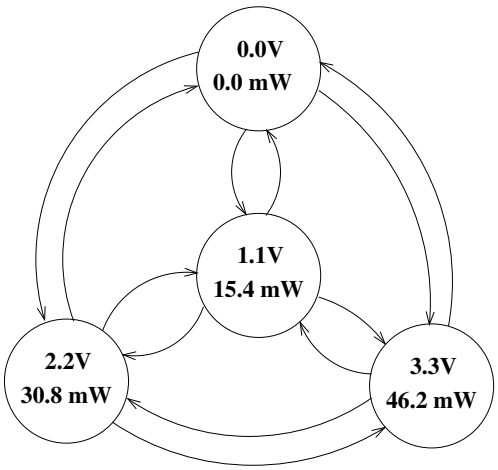


FIG. 5.7 – Modèle de consommation du CPU avec DVS

Extension : CPU et consommation des mémoires. Les valeurs de notre modèle de consommation de la radio (figure 5.6) prennent en compte la consommation totale du nœud. Il serait cependant préférable de pouvoir découpler les consommations des autres éléments matériels, tels que mémoires

et microcontrôleur, si par exemple le microcontrôleur fonctionne de façon indépendante de la radio. En fait, nous savons modéliser la consommation d'autres éléments matériels, mais il est difficile de piloter ces modèles. Cette difficulté ne vient pas de la modélisation mais de la difficulté de piloter les éléments matériels dans le but de réduire leur consommation. Nous expliquons maintenant pourquoi. Nous avons par exemple implémenté un modèle de consommation d'un processeur dont la tension varie. Nous avons supposé que le microcontrôleur peut être alimenté par trois tensions différentes : 1.1, 2.2 ou 3.3 Volts plus un état dans lequel il est complètement éteint. Nous avons donc obtenu l'automate de la figure 5.7. Ici, il s'agit du *Dynamic Voltage Scaling (DVS)*, évoqué section 2.1.4 qui permet d'ajuster la vitesse et la consommation du microcontrôleur. Nous aurions pu de même modéliser d'autres techniques qui permettent de varier la consommation du microcontrôleur, comme par exemple la variation de fréquence (*dynamic frequency scaling*). Une fois qu'un tel modèle de consommation est inclus dans notre modèle global, il faut piloter ce modèle de consommation. Pour le modèle de la radio (figure 5.6), c'était assez facile : l'automate de la figure 5.6 est piloté par le protocole MAC (figure 5.4). Pour le microcontrôleur, c'est beaucoup plus compliqué de savoir quand changer d'état. Mais, il est important de comprendre que ce n'est plus un problème de modélisation mais bien un problème de conception. En effet, il est difficile de concevoir un système capable d'adapter la tension/fréquence du microcontrôleur de manière dynamique et automatique. Une solution consisterait à analyser statiquement le code afin d'ajouter les instructions de changement de vitesse. Dans ce cas, le modèle serait immédiat puisque l'on introduirait ces instructions dans notre modèle (au niveau application par exemple) afin de changer d'état dans l'automate de la figure 5.7. D'autres solutions se basent sur des mesures dynamiques de l'activité du CPU. Dans tous les cas, la difficulté intrinsèque consiste à trouver une solution pour faire varier la consommation du microcontrôleur, pas à la modéliser.

Nous pouvons également introduire un modèle de consommation des mémoires : par exemple, pour la mémoire DRAM, celui proposé par Delaluz et al [24] et présenté figure 2.5, page 29. De même nous pourrions modéliser la consommation de la mémoire non volatile (flash) par un automate. Mais, comme pour le microcontrôleur, la difficulté est de savoir instrumenter le code pour piloter ces modèles de consommation. Et ici aussi, cette difficulté est un problème de conception mais pas de modélisation.

5.1.4 Interface graphique

Selon nous, un outil qui permet de visualiser les simulations est presque indispensable. En effet, un tel outil permet non seulement de mieux comprendre le fonctionnement global du réseau de capteurs mais il est également une aide précieuse à l'implémentation du simulateur. En effet, une représentation graphique du réseau offre une vue globale beaucoup plus lisible qu'un fichier contenant des séquences de caractères représentant l'exécution du système.

Nous avons tiré parti de REACTIVEML pour écrire cette interface graphique. Tout d'abord, grâce au modèle d'exécution synchrone, il est confortable d'écrire le code qui génère l'affichage comme un processus qui s'exécute en parallèle. De plus, comme REACTIVEML est un langage réactif synchrone, ce processus peut être créé pendant la simulation - on pourrait même en créer plusieurs. On peut éventuellement le supprimer afin d'accélérer la simulation en cours. Comme tous les autres, ce processus réagit à chaque instant ce qui permet à l'affichage d'être synchrone avec l'exécution du simulateur. REACTIVEML permet d'interagir avec l'exécution du programme en envoyant - via le clavier ou la souris - des signaux. C'est pourquoi notre interface graphique n'est pas un simple afficheur mais il permet véritablement d'interagir avec la simulation en créant par exemple plus de vent. C'est également par ce type d'interactions que l'on peut créer ou supprimer le processus chargé de l'affichage. Nous revenons sur ces aspects section 5.4.

La figure 5.8 est une capture d'écran de l'afficheur pour une simulation GLONEMO de 1000 nœuds.

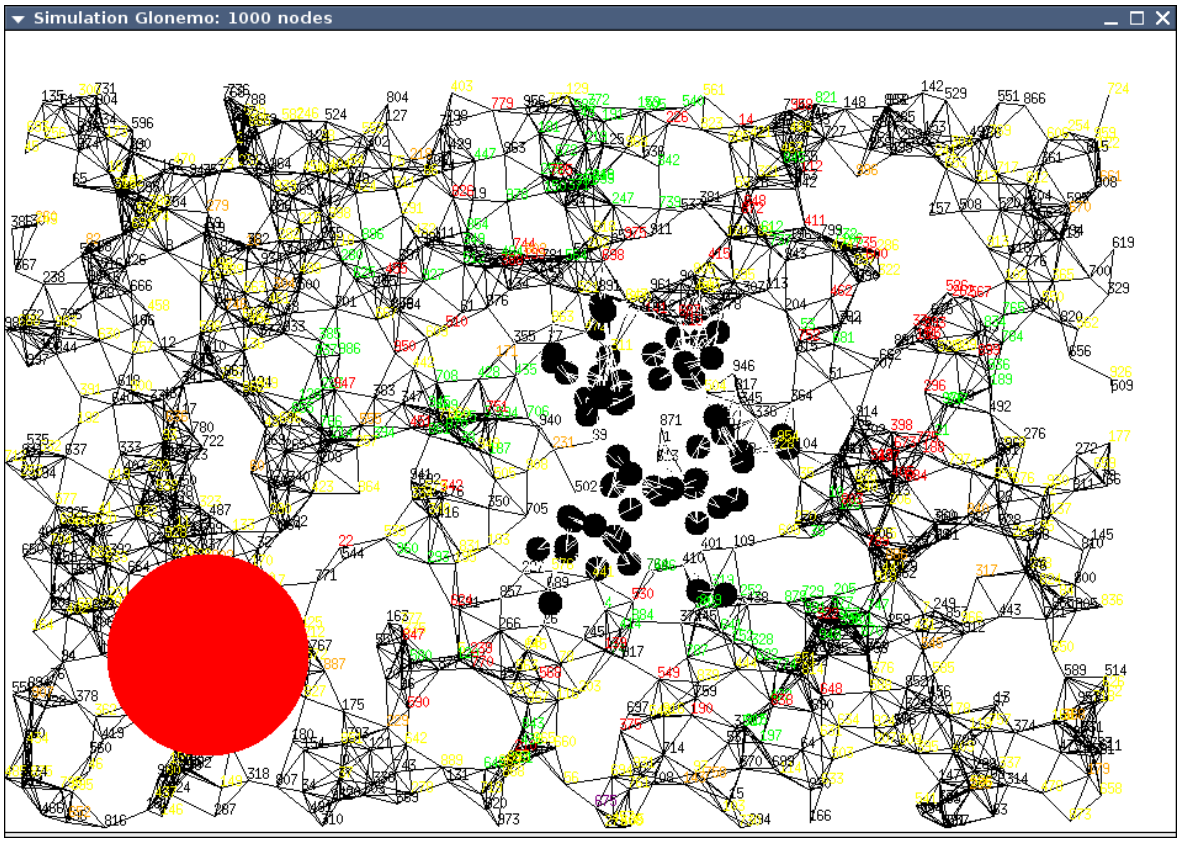


FIG. 5.8 – Capture d'écran de l'interface graphique du simulateur. Le disque rouge représente un nuage toxique. Les disques noirs sont des nœuds qui n'ont plus d'énergie.

Légende des couleurs :
- `Sleep` : noir
- `Wakeup` : jaune
- `Idle` : orange
- `Idle_to_Transmit` : rouge foncé
- `Sleep_to_Transmit` : magenta foncé
- `Idle_to_Receive` : vert foncé
- `Transmit` : rouge
- `Transmit_to_Receive` : jaune foncé
- `Receive` : vert
- `Receive_to_Transmit` : jaune foncé

Même si nous modifions facilement cet affichage suivant les besoins, nous présentons ici rapidement à quoi il correspond. Tout d'abord, les nœuds sont représentés par leur identifiant. En effet, chaque instance du processus `node` a un identifiant unique[3]. Un segment entre deux nœuds signifie que les nœuds sont à portée radio[4]. Le disque rouge est le nuage. Ce qui nous intéresse dans nos simulations est la consommation, celle-ci est fonction de l'état dans lequel se trouve la radio. Pour afficher cette information, les nœuds changent de couleur en fonction de l'état dans lequel se trouve leur radio. La légende des couleurs est sous la figure 5.8. Nous pouvons ainsi suivre la propagation des messages dans le réseau. Lorsqu'un nœud n'a plus d'énergie, l'instance du processus correspondant à ce nœud est supprimée. De cette façon, le nœud n'interagit plus avec le réseau ce qui est réaliste. Au niveau de l'affichage, un disque noir remplace l'identifiant du nœud et les lien avec ses voisins sont effacés.

5.2 Expérience GLONEMO : importance de l'environnement

Le contenu de cette section est basé sur celui de la référence [77]. Nous avons effectué des expériences à l'aide de GLONEMO, et nous avons observé que l'environnement a une forte influence sur le comportement d'un réseau de capteurs. Dans cette section, nous illustrons cet aspect en comparant le comportement du même réseau (même matériel, mêmes protocoles, même application) sous deux modèles d'environnement différents. Un des modèles d'environnement est celui présenté dans la section précédente, le nuage implémenté en LUCKY génère sur les capteurs des stimuli corrélés temporellement et spatialement. L'autre modèle d'environnement que nous proposons génère des stimuli indépendants temporellement et spatialement. Ce type de modèle pourrait être utilisé pour modéliser le trafic de réseaux ad hoc dans lesquels les stimuli ne sont effectivement pas corrélés.

La section 5.2.1 décrit le second modèle d'environnement dans lequel la génération de paquets est modélisé par une loi de Poisson. Ce modèle est classiquement utilisé dans les réseaux ad hoc. La section 5.2.2 discute de la difficulté de comparer des simulations effectuées avec deux modèles d'environnements différents. La section 5.2.3 présente les résultats obtenus et la section 5.2.4 conclut.

5.2.1 Un autre modèle d'environnement : loi de Poisson

Modéliser le trafic dans un réseau signifie modéliser les émissions de paquets. Il ne s'agit pas de modéliser les éventuelles retransmissions de paquets mais seulement de modéliser le fait qu'un nœud a un paquet à envoyer à un moment. Dans la réalité, c'est l'application ou un éventuel utilisateur qui génère un paquet. On dit qu'un paquet à émettre arrive au niveau du nœud. Modéliser le trafic est un sujet qui a été longuement étudié, notamment pour les réseaux classiques, voir par exemple [34].

Dans la section précédente, pour modéliser le trafic, nous avons modélisé l'application et son environnement. En exécutant à la fois le modèle du réseau et son environnement, nous avons pu générer des paquets sans avoir à modéliser le trafic comme un élément supplémentaire. Une abstraction couramment utilisée pour modéliser le trafic dans les réseaux sans modéliser l'environnement est d'utiliser un modèle Poissonnien. On suppose dans ce cas que les temps d'inter-arrivées sont exponentiellement distribués avec un paramètre λ. Soit A_n le temps entre l'arrivée du nième et du (n+1)ième paquet, $P(A_n \leq t) = 1 - exp(-\lambda t)$. C'est cette loi que nous allons utiliser pour nos réseaux de capteurs.

Dans les réseaux de capteurs, à l'exception du puits, tous les nœuds ont le même rôle. Il y a bien sûr des nœuds qui ont plus de paquets à envoyer parce que se trouvant à proximité du puits, ils doivent

[3] Attention, ça ne veut pas dire que le réseau de capteurs dispose d'un identifiant unique pour chaque nœud. C'est le simulateur qui a un identifiant pour chaque nœud, pas forcément le réseau simulé.

[4] Dans les versions plus récentes de GLONEMO qui implémentent un modèle de canal probabiliste, l'épaisseur du lien est fonction de la qualité du lien radio.

faire le lien entre les nœuds qui sont à la périphérie et le puits. Cependant ce phénomène ne concerne pas la modélisation de l'environnement ni le taux d'arrivée des paquets à émettre. Le paramètre λ sera donc le même pour tous les nœuds. En effet, nous supposons que le nuage radioactif à détecter peut apparaître n'importe où sur le réseau avec la même probabilité.

Dans ce modèle, on modélise la détection du danger (c'est-à-dire le nuage toxique) au niveau d'un nœud par un événement. Pour générer ces événements, chaque nœud démarre un compte à rebours à la fin duquel l'événement est généré. L'initialisation des comptes à rebours suit une loi exponentielle. Ainsi, la détection d'un danger au niveau d'un nœud est indépendante de celle de ses voisins. De même, la détection d'un danger au niveau d'un nœud est indépendante des détections antérieures de ce nœud.

La loi qui génère le prochain instant où un danger est détecté est la suivante :

$$A_n \quad = \quad \lfloor -\lambda \times log(x) \rfloor$$

où x suit une loi uniforme sur $[0, 1]$. Notre modélisation fonctionne avec un temps discret, cet intervalle de temps doit donc être un nombre entier ; c'est pourquoi nous prenons la partie entière (notée $\lfloor\ \rfloor$).

5.2.2 Comparaison des deux modèles

Nous avons exécuté des simulations pour deux réseaux identiques en ne changeant que le module qui génère le trafic, c'est-à-dire les alarmes. Afin d'observer l'influence du modèle de l'environnement, nous avons besoin de comparer les simulations avec le nuage LUCKY et les simulations avec les processus de Poisson. Afin que cette comparaison ait du sens, il nous faut choisir un critère. Ça n'est pas une tâche facile.

On peut choisir de comparer des simulations où le nombre d'alarmes envoyées pendant toute la durée de vie du réseau est le même pour les deux modèles d'environnement. Les alarmes générées par la modélisation LUCKY seront plus denses puisque le nuage active plusieurs nœuds à la fois.

Nous avons réglé le paramètre λ de la loi exponentielle afin d'assurer ce critère de comparaison. λ est un nombre d'instants de simulation qui correspond à un temps simulé de 1000 secondes. Pour maîtriser le nombre d'alarmes générées par le nuage LUCKY, nous avons la possibilité d'agir sur les simulations pendant leur exécution en créant plus de vent pour déplacer le nuage ou simplement en détruisant le nuage.

5.2.3 Résultats

Nous avons effectué des simulations avec les deux modèles d'environnement. Le réseau est toujours le même, il est affiché figure 5.9. Les nœuds sont équipés de piles. Mis à part le puits qui n'est pas contraint en énergie, les nœuds ont tous la même autonomie. Nos simulations s'arrêtent lorsque plus un seul nœud n'a d'énergie.

Voici les paramètres de notre simulation :

- capacité des piles : 50 J (soit 13.9 mWh)
- 100 nœuds : 1 puits et 99 capteurs
- Les nœuds sont disposés de façon aléatoire suivant une loi uniforme sur un carré de 700 unités de côté
- Rayon de transmission = 120 unités

Afin d'évaluer la qualité de service d'un réseau dédié à la remontée d'alarmes, un bon critère est le nombre de paquets reçus par le puits par rapport au nombre d'alarmes émises. Pour ces expériences λ correspond à 1000 secondes de temps simulé et le nuage ne fait que traverser le réseau. Les résultats sont résumés dans le tableau 5.1. Nous observons que le réseau fonctionne relativement bien lorsque les

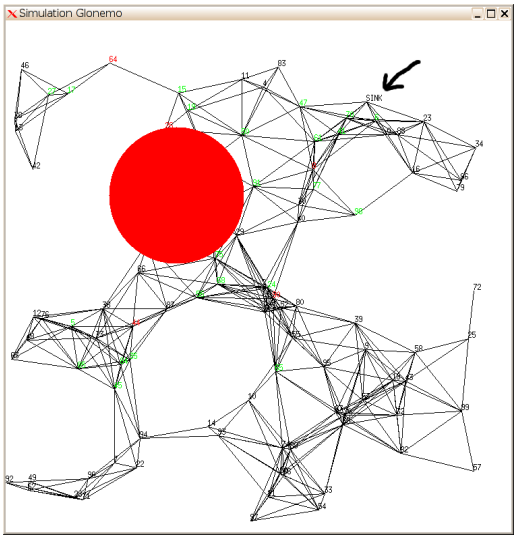


FIG. 5.9 – Une simulation GLONEMO activée avec le nuage LUCKY.

alarmes sont générées suivant une loi de Poisson. Le puits reçoit plus de 60 pourcents des alarmes. Au contraire, avec notre modélisation de l'environnement, ce taux chute. Lorsque c'est un nuage qui active les nœuds, le puits reçoit moins de 10 pourcents des alarmes ! Si les alarmes générées par les nœuds du réseau ne sont pas toujours reçues par le puits, c'est à cause des collisions. En effet, le nombre de collision est bien plus élevé lorsque le réseau est stimulé par notre modèle d'environnement. Le nombre de collisions qui se produisent lors de la propagation des intérêts est fixe : pour une simulation donnée (avec une graine donnée) l'inondation des messages "intérêts" se fait toujours de la même manière quelle que soit le modèle d'environnement. Ici, on a compté 189 collisions. Par contre, pour la remontée d'alarme (ligne "données seulement") de nombreuses collisions ont lieux avec notre nuage alors qu'elle sont peu nombreuse si l'environnement est simplement modélisé par un processus de Poisson.

		processus de Poisson	environnement LUCKY
Alarmes :	alarmes émises	76	210
	alarmes reçues	48	15
	taux correspondant	0.63	0.07
Nombre de collisions :	propagation des intérêts	189	189
	données seulement	29	547
	total	218	736

TAB. 5.1 – Résultats : nombre de collisions

Nous allons étudier où ces collisions se produisent. Les figures 5.10, 5.11, 5.12 et 5.13 sont des captures d'écran de différentes simulations. Chaque série de quatre images correspond à une simulation avec un environnement particulier. Pour chaque simulation (correspondant à un modèle d'environnement) les quatre images (de gauche à droite) correspondent à des moments différents au cours de la simulation. Le temps s'écoule de gauche à droite : à gauche c'est une capture d'écran qui

est prise dès que les premières collisions apparaissent alors que à droite de nombreux capteurs n'ont plus d'énergie.

Les cercles qui apparaissent autour des nœuds pendant les simulations indiquent le nombre de collisions. Un nœud peut être entouré de 1 à 4 cercles s'il a subi 10, 20, 30 ou plus de 40 collisions. On dit qu'un nœud subit une collision lorsqu'il reçoit en même temps au moins deux signaux radio. Dans ce cas, il ne peut en décoder aucun.

Les différentes figures correspondent à différents modèles d'environnement ou comportements de l'environnement. La figure 5.10 correspond à une simulation dans laquelle l'envoi de paquets est modélisé par des processus de Poisson. Pour les figures 5.11, 5.12 et 5.13, l'environnement est modélisé par un nuage disque qui se déplace sous l'effet du vent. Mais pour ces trois cas, on le contraint différemment : figure 5.11 le nuage ne survole que le coin en bas à droite. Sur la simulation de la figure 5.12 le nuage peut aller partout alors que sur celle de la figure 5.13 il peut aller partout sauf dans la région du puits.

Enfin, lorsqu'un nœud n'a plus d'énergie, il n'interagit bien sûr plus avec le reste du réseau et pour le visualiser nous affichons un disque noir.

Sur la figure 5.10, les nœuds sont activés par des processus de Poisson de paramètre $\lambda = 92000$ instants $= 92$ secondes. Donc en moyenne, les nœuds ont un paquet à envoyer toutes les 92 périodes de veille. En effet le protocole MAC à échantillonnage de préambule a un cycle de veille-écoute d'une seconde. C'est-à-dire que les nœuds sondent le canal périodiquement toutes les secondes. Ici, la plupart des collisions ont lieu le long des routes qui mènent au puits. Nous observons d'ailleurs l'effet entonnoir : certains nœuds sont sur la route d'une bonne partie du réseau. Ces nœuds doivent acheminer beaucoup plus de trafic que les autres et donc ils meurent en premier. De même, ils subissent plus de collisions que le reste du réseaux.

Au contraire, pour les simulations comprenant un modèle d'environnement plus réaliste, les collisions ont plutôt tendance à se produire où est passé le nuage. Par exemple figure 5.11, la grande majorité des collisions a lieu dans le coin en bas à droite là où est passé le nuage. Au passage du nuage des nœuds voisins sont activés en même temps et envoient donc des alarmes en même temps créant ainsi de multiples collisions. Ce sont ces nœuds, dans le coin en bas à droite, qui meurent en premier. Les autres nœuds du réseau dépensent très peu d'énergie puisqu'ils n'ont rien à faire : ils ne font que sonder le canal périodiquement. Même les nœuds près du puits meurent assez tardivement. Ces nœuds doivent acheminer les alarmes produites dans le coin en bas à droite, mais comme de nombreuses collisions se sont produites, il ne reste que peu de paquets à router vers le puits. Enfin, on remarque sur la dernière figure (figure 5.11, droite) que trois nœuds n'ont plus d'énergie et n'ont subi que très peu de collisions. Ces nœuds ne se situent pas dans la zone survolée par le nuage. Ils se trouvent très certainement sur la route qui va du coin en bas à droite au puits. Ces nœuds ont probablement dépensé leur énergie en relayant des messages destinés au puits. N'ayant qu'un émetteur potentiel, c'est-à-dire envoyant des messages du coin alerté, ils n'ont pas subi de collisions.

Sur les figures 5.12 et 5.13 on observe le même genre de phénomène : les collisions se produisent où passe le nuage. L'effet d'entonnoir n'est pas la cause principale des collisions.

Comme le tableau 5.1, le tableau 5.2 compare deux autres simulations effectuées avec nos deux modèles d'environnement. Ici, on s'attache à comparer les durées de vie des réseaux. Nous avons pris trois critères pour évaluer la durée de vie, le réseau est mort quand : 1) le premier nœud est mort, 2) plus aucun nœud n'a d'énergie ou 3) quand le réseau n'est plus connexe. Les temps sont indiqués en nombre d'instants. Finalement les durées de vie sont assez proches et même si l'environnement nuage-vent génère plus de paquets, les durées de vie sont sensiblement supérieures. En fait, avec le nuage LUCKY, le réseau fonctionne tellement mal que les nœuds ne dépensent même pas leur énergie. Dans le cas du nuage, les paquets générés par la détection du nuage interfèrent rapidement. Ces paquets

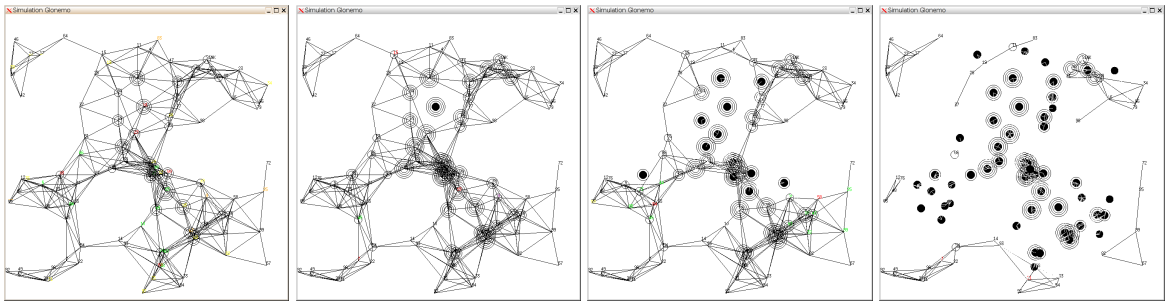


FIG. 5.10 – Représentations d'une simulations GLONEMO où chaque capteur est activé par un processus de Poisson

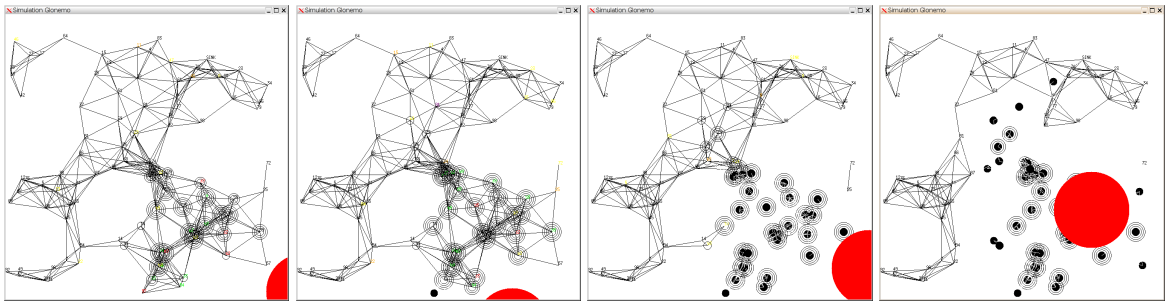


FIG. 5.11 – Représentations d'une simulations GLONEMO où les capteurs sont activés par le nuage, celui-ci n'active le réseau que sur le coin en bas à droite.

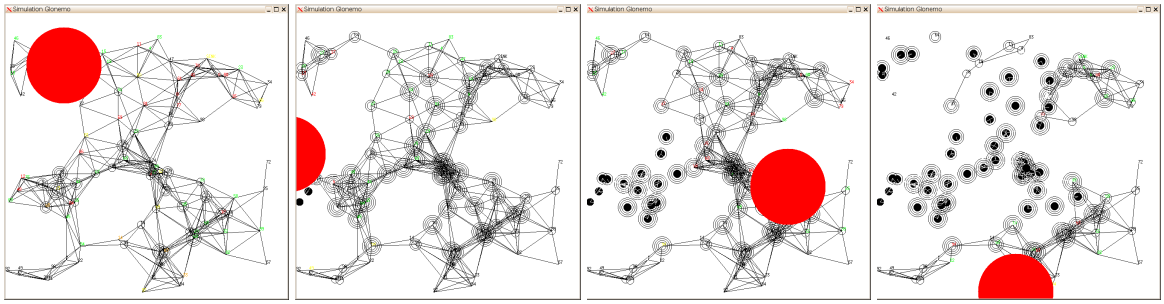


FIG. 5.12 – Représentations d'une simulations GLONEMO où les capteurs sont activés par le nuage. Le nuage peut être partout.

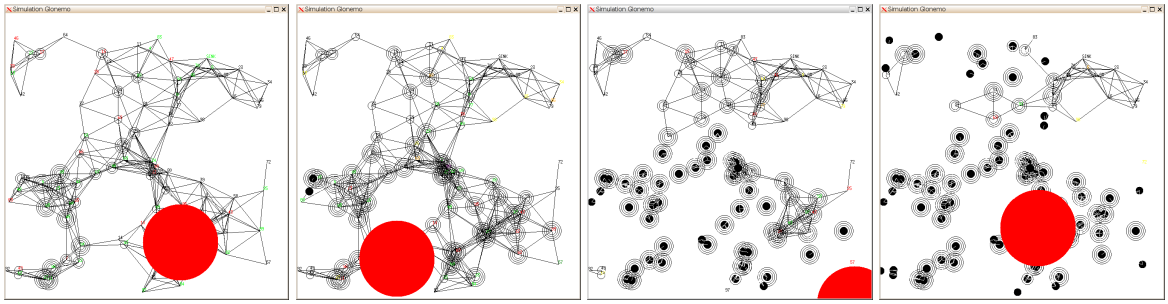


FIG. 5.13 – Représentations d'une simulations GLONEMO où les capteurs sont activés par le nuage. Le nuage peut être partout sauf sur le puits.

		Processus de Poisson	Environnement LUCKY
Alarmes :	alarmes émises	77	148
	alarmes reçues	44	15
	taux correspondant	0.57	0.10
Durée de vie pour différents critères :	premier nœud mort	67676	83775
	tous les nœuds sont morts	92790	93839
	réseau déconnecté	84681	85924

TAB. 5.2 – Résultats : durée de vie du réseau

ne créent donc pas beaucoup de trafic puisque les collisions ont lieu dès les premières émissions. Ainsi les nœuds n'ont pas à dépenser leur énergie pour acheminer le trafic vers le puits. À l'inverse, lorsque la génération d'alarmes suit une loi de Poisson, la majeur partie des alarmes arrive finalement au puits (57% dans ce cas). Chacune de ces alarmes a nécessité de nombreuses retransmissions afin d'être acheminée au puits. Pour propager une seule alarme, de nombreux nœuds ont collaboré et ont donc consommé de l'énergie.

Nous avons souligné les comportements différents de deux réseaux identiques fonctionnant avec un modèle d'environnement plus ou moins réaliste. Nous n'avons comparé que quelques critères comme l'efficacité du réseaux, le nombre de collision ou la durée de vie. Ce dernier critère ne semble pas si discriminant à première vue mais il faut bien le relier aux deux autres pour comprendre que même si les durées de vie sont proches le réseau n'offre pas du tout la même qualité de service suivant le modèle d'environnement. Nous pourrions observer d'autres indicateurs afin de comparer les comportements, mais ceux-ci ont suffit à montrer que le réseau réagit vraiment différemment suivant l'environnement.

5.2.4 Conclusion

Nous avons montré sur un exemple simple l'importance de prendre en compte un modèle d'environnement précis dans la modélisation des réseaux de capteurs. En fait, tous les simulateurs de réseaux contiennent un modèle d'environnement mais habituellement ce modèle est très abstrait. Ces modèles sont trop éloignés de la réalité dans le cas des réseaux de capteurs. Pour obtenir des conclusions fiables de simulation de réseaux de capteurs, il faut une modélisation réaliste de l'environnement.

Nous avons étudié ici des réseaux de *capteurs* seulement. Dans le cas de réseaux de capteurs *et actionneurs*, le réseau agit sur l'environnement. Le réseau et l'environnement forment alors un système bouclé tel un système de contrôle. Les propriétés à vérifier dans ce cas concernent tout le système, réseau et environnement. Il serait d'autant plus nécessaire pour un simulateur dédié à ce type de réseaux de modéliser à la fois le réseau et l'environnement.

À l'aide de LUCKY, nous pouvons modéliser dans GLONEMO des comportements non-déterministes. Nous pensons que c'est une bonne approche pour modéliser l'environnement de façon réaliste tout en évitant de décrire finement les phénomènes physiques mis en jeu.

5.3 LUSSENSOR

5.3.1 Un modèle en LUSTRE

Nous avons présenté le langage LUSTRE au chapitre précédent. Plusieurs outils de vérification automatique développés à Verimag (et présentés aussi au chapitre 4) sont dédiés à des programmes

LUSTRE. Afin de proposer des techniques d'analyse autre que la simulation pour les réseaux de capteurs, nous avons entrepris de modéliser un réseau de capteurs en LUSTRE. Ce travail a été facilité grâce au modèle GLONEMO. En effet, GLONEMO est un modèle global de réseaux de capteurs qui a déjà de nombreuses caractéristiques que l'on souhaite pour notre modèle en LUSTRE. GLONEMO est un modèle global à composants écrit dans un formalisme synchrone.

Nous présentons donc ce modèle, les principales difficultés rencontrées ainsi que les différences avec GLONEMO. LUSSENSOR est le nom de ce modèle qui a été réalisé par Kevin Baradon et Antoine Vasseur pendant leur stage à Verimag. Les détails ont été publiés dans [61].

5.3.2 Structure de LUSSENSOR

Même si REACTIVEML et LUSTRE sont deux langages synchrones, il n'existe pas d'outil de transformation automatique de REACTIVEML vers LUSTRE, y compris pour un sous-ensemble de REACTIVEML qui ne contiendrait pas de création dynamique. La transfomation de GLONEMO en LUSSENSOR a été faite de façon quasi-systématique à la main. La structure de LUSSENSOR est donc la même que celle de GLONEMO, voir section précédente.

Pour les mêmes raisons que pour GLONEMO, nous voulons un modèle à composants. Les composants correspondent aux mêmes fonctions que dans GLONEMO, voir figure 5.1, page 91. Mais ici les composants ne sont plus des processus mais ce sont des *nœuds*[5] LUSTRE (le mot clé est `node`). Chaque élément matériel qui consomme de l'énergie est un composant ainsi que chaque protocole logiciel. Tous ces *nœuds* sont connectés entre eux pour former un capteur.

Enfin, un *nœud* `channel` connaît la topologie du réseau et gère ainsi les communications radio, c'est ce *nœud* qui détermine quel capteur reçoit quel paquet et les éventuelles collisions. Contrairement à son équivalent en REACTIVEML, ce *nœud* a été particulièrement pénible à programmer. LUSSENSOR est programmé en LUSTRE-V4. Cette version de LUSTRE est plutôt dédiée à la conception de systèmes sur puces. Les boucles sont dépliées statiquement. L'algorithme de gestion des communications radio implémenté dans le canal nécessite des structures de programmation assez complexes comme des boucles. Pour implémenter cet algorithme en LUSTRE-V4, il a fallu exprimer ces boucles uniquement à base de matrices. Il ne s'agit pas d'une difficulté intrinsèque liée à LUSTRE, mais plutôt des structures de programmation du langage. Par exemple LUSTRE-V6 contient des structures de contrôle plus fonctionnelles qui auraient facilité l'implémentation du canal cependant LUSTRE-V6 n'est pas aussi bien connecté aux outils de test et de vérification.

Le modèle de l'environnement n'est pas modélisé en LUSTRE. C'est le modèle LUCKY qui est utilisé. En fait, dans LUSSENSOR, il y a des entrées correspondant aux signaux provenant de l'environnement. Ces entrées ont vocation à être connectées aux sorties du modèle de l'environnement écrit en LUCKY. Cependant, il est également possible d'activer ces entrées à la main via une interface ou statiquement.

LUSTRE ne permet pas de créer dynamiquement des processus. Il est cependant possible de faire un modèle dans lequel des nœuds sont ajoutés ou supprimés pendant l'exécution. Le nombre de nœuds maximum doit être connu à l'avance. Si n est le nombre maximum de nœuds, il faut exécuter LUSSENSOR avec n instances du *nœud* capteur. Le *nœud* capteur a alors deux états, un état `off` dans lequel il ne fait rien et un état `on` dans lequel il fonctionne normalement. Ajouter un nouveau nœud dans le réseau se fait en activant un *nœud* inactif, c'est-à-dire en le faisant passer de l'état `off` à `on`.

Le formalisme synchrone de LUSTRE crée des programmes complètement déterministes. Cependant, les protocoles de communications des réseaux de capteurs peuvent utiliser des valeurs aléatoires.

[5] Nous notons *nœud* en italique pour parler d'un nœud LUSTRE, à ne pas confondre avec un capteur, c'est-à-dire un nœud du réseau.

C'est le cas du protocole MAC : un émetteur choisit de façon aléatoire dans un intervalle le moment où il enverra son message. Ce mécanisme permet d'éviter des collisions. Pour ces valeurs aléatoires, on utilise des entrées externes qui peuvent provenir d'un générateur aléatoire quelconque. Nous aurions pu utiliser localement un appel à une fonction C externe, mais en vue de faire des analyses, c'est mieux de considérer une valeur aléatoire comme une entrée. En effet, plutôt que de considérer une fonction externe gérée comme une boîte noire, nous pouvons faire des abstractions sur les valeurs possible de cette entrée. Nous savons par exemple que cette entrée prend ses valeurs dans un intervalle donné.

Le modèle global, LUSSENSOR, consiste en environ 1500 lignes de codes LUSTRE.

5.3.3 Différences entre GLONEMO et LUSSENSOR

Nous avons deux programmes sensiblement identiques programmés tous deux dans un formalisme synchrone. L'un est écrit en REACTIVEML, l'autre en LUSTRE. Il est intéressant de voir quelles sont les différences et d'essayer de comprendre d'où elles viennent.

La première observation est la vitesse d'exécution. LUSSENSOR est beaucoup plus lent que GLONEMO. Si avec GLONEMO il est possible de simuler assez rapidement des réseaux de plusieurs centaines de nœuds, ça n'est pas le cas pour LUSSENSOR. Pour un réseau d'une dizaine de nœuds à peine, l'exécution est très lente. Cette différence n'est pas due à la précision des modèles, GLONEMO et LUSSENSOR sont aussi précis. Nous expliquons cette différence par les modes d'exécution qui sont différents dans REACTIVEML et dans LUSTRE. En effet, comme présenté au chapitre 4, dans LUSTRE, l'ordonnancement est défini statiquement à la compilation alors que dans REACTIVEML, l'ordonnancement se fait dynamiquement, à l'exécution du programme. L'ordonnancement dynamique permet des optimisations, notamment l'optimisation **inter-instants** (expliquée section 4.4.3, page 86). Cette optimisation, implémentée dans REACTIVEML, permet une exécution plus efficace dans le cas où le programme contient des structures du type `await s` et que le signal `s` n'est pas émis souvent. Pour nos simulateurs, c'est le cas par exemple voici un extrait de code du processus `routing` :

```
loop
  await self.application_to_routing (p3) in
  ...
end
```

Cette instruction exprime l'attente, au sein d'un nœud, de paquets de niveau 3 (routage) provenant de l'application. Elle est censée être effectuée à chaque instant pour chaque nœud et il existe d'autres instructions similaires dans ce même processus `routing` mais aussi pour le processus `mac`. On comprend donc que ce simple test de présence du signal devient coûteux pour le simulateur s'il est exécuté à chaque instant pour chaque nœud, c'est ce qui se passe dans LUSSENSOR. Dans GLONEMO, l'optimisation inter-instants de REACTIVEML permet de limiter le nombre de tests : la première fois que l'on rencontre (à l'exécution) une telle instruction, on la place dans une file d'attente (une par signal) et c'est seulement quand le signal est émis que l'on vient activer les éléments qui sont dans la file d'attente. Nos simulateurs étant à grains fins, de très nombreux instants logiques séparent deux émissions de paquets. Par exemple, supposons qu'un paquet est produit chaque minute par un nœud du réseau, et qu'il faut 100 instants logiques pour simuler une seconde du réseau. Dans LUSSENSOR, entre l'émission de deux paquets, le processus dédié au routage devra tester à chaque instant pendant 100×60 secondes $= 6000$ instants la présence du signal `p3`. Alors que dans GLONEMO, ce test aura eu lieu au maximum deux fois[6] au premier instant qui suit l'émission d'un paquet puis plus jamais

[6] Pour comprendre pourquoi ce test peut avoir lieu deux fois et pas une seule au cours d'un instant logique, voir la thèse de Louis Mandel [55], chapitre 6.

jusqu'à l'émission du paquet. Pour nos simulateurs, ce cas de figure est présent plusieurs fois dans les différents protocoles, de plus il faut multiplier ce gain par le nombre de nœuds simulés.

REACTIVEML est un langage de haut niveau qui laisse la possibilité au programmeur de construire et de manipuler toutes les structures de données qu'il veut, ça n'est pas le cas de LUSTRE. Il est donc plus facile d'implémenter des algorithmes efficaces dans REACTIVEML que dans LUSTRE. C'est certainement aussi pour ça que GLONEMO est plus efficace que LUSSENSOR.

Une sortie graphique a été implémentée pour LUSSENSOR. Cette afficheur a été implémenté en langage C. Elle a été relativement pénible à implémenter puisqu'il fallait joindre les sorties du code LUSTRE et le code C de l'affichage. De plus cet affichage graphique permet également d'agir sur certaines entrées, notamment celles de l'environnement si celui-ci n'est pas déjà simulé par LUCKY. On active ces entrées via des boutons disponibles sur l'interface. Contrairement à GLONEMO, on ne peut pas modifier le comportement de la simulation en cliquant sur la représentation graphique du réseau.

Avant de faire cette expérience, il était prévisible que GLONEMO, outre sa facilité d'implémentation et de modification, reste un simulateur plus efficace. Cependant, si nous avons mené à bien cette expérience c'est afin de tirer parti des outils d'analyse existants pour LUSTRE. Maintenant que nous avons implémenté les modèles de consommation en LUSTRE, il est possible d'utiliser les outils d'analyse disponibles pour les comparer. Ainsi, nous pouvons prouver qu'un modèle est une abstraction d'un autre s'il fait la même chose et qu'il consomme plus. Nous formalisons cette notion au chapitre 7.

5.4 REACTIVEML, un langage efficace pour la programmation de simulateurs

Outre l'intérêt d'avoir un modèle de réseaux de capteurs écrit en LUSTRE afin de bénéficier des outils de vérification automatique disponibles pour LUSTRE, nous avons pu comparer deux programmes "identiques" écrit en REACTIVEML ou en LUSTRE. Il apparaît que GLONEMO est nettement plus efficace que LUSSENSOR. Dans cette section nous donnons des arguments en vu de montrer que REACTIVEML est un langage de programmation qui convient pour programmer rapidement des simulateurs efficaces. Cette argumentation se base non seulement sur GLONEMO mais aussi sur un simulateur dédié à l'étude de protocoles de communication pour des réseaux mobiles ad hoc [56].

Nous avons présenté chapitre 2, section 2.3.1, certains simulateurs de réseaux et/ou de réseaux de capteurs. Cette taxonomie n'était pas exhaustive, en effet il existe de nombreux simulateurs issus de la recherche académique ou produits de l'industrie. Cependant il arrive fréquemment que, pour tester leur solution, les concepteurs de réseaux de capteurs développent un simulateur dédié à leur étude. Il y a au moins deux raisons pour préférer développer soi-même un simulateur. Tout d'abord, l'utilisation d'un simulateur de réseaux requiert un apprentissage qui peut s'avérer très long. De plus, développer sont propre simulateur permet de ne simuler que le nécessaire. Avec un simulateur développé pour un problème précis, on évite d'avoir des éléments du simulateur inutilisés ou laissés à leur valeur par défaut comme c'est parfois le cas pour des simulateurs génériques. Enfin, même si certains simulateurs proposent parfois plusieurs niveau de détail, avec un simulateur dédié on a le niveau de détail que l'on souhaite pour le problème étudié. Certes, avoir un simulateur générique pour toute la communauté de recherche en réseau permet de comparer facilement les solutions proposés par les chercheurs. C'est l'idée du projet VINT [19]. Notre propos n'est pas d'aller à l'encontre de cette idée, il s'agit simplement de dire que quitte à développer, soi-même, un simulateur *à la main*, autant le faire en REACTIVEML.

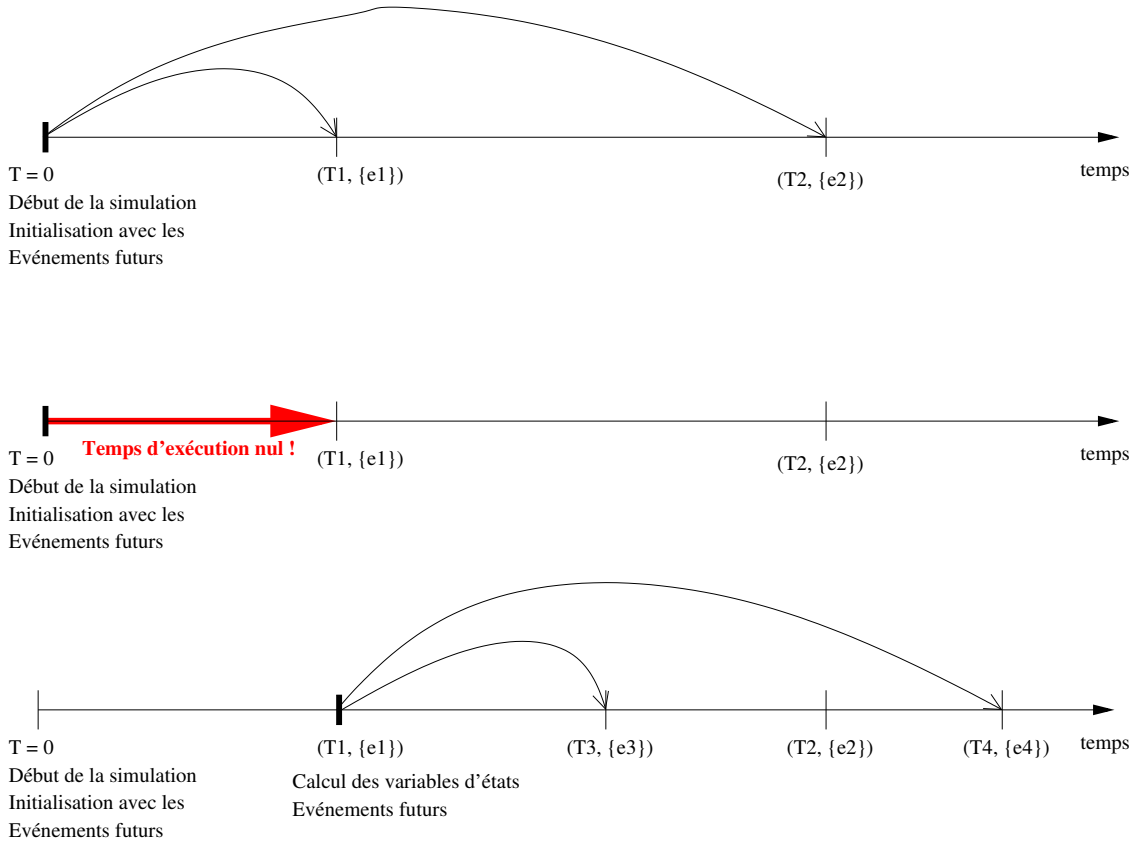


FIG. 5.14 – Mode d'exécution d'un simulateur à événements discrets

5.4.1 Deux types de simulateurs : à événements discrets ou à pas fixe

Tout d'abord notons une différence fondamentale entre les simulateurs de réseaux comme NS2 [66] ou OPNET [20] et les simulateurs écrits en REACTIVEML : l'exécution de la simulation. Les simulateurs classiques sont des simulateurs à événements discrets alors que les simulateurs écrits avec REACTIVEML profitent du mode d'exécution synchrone et sont des simulateurs à pas fixes.

Simulation à événements discrets

Schématiquement, un simulateur à événements discrets fonctionne avec un échéancier discret comme représenté figure 5.14[7]. À l'initialisation de la simulation, les premiers évènements sont créés et sont ordonnés sur l'échéancier par date d'exécution croissante. La date d'exécution correspond à la date à laquelle l'événement serait exécuté dans le système réel. Une fois que tous les événements sont placés, on passe à l'événement suivant, c'est-à-dire l'événement suivant sur l'échéancier. Cet événement, ici e_1, va à son tour créer d'autres événements (e_3 et e_4). À ces événements sont associées leurs dates d'exécution (T_3 et T_4). Les événements sont ordonnés sur l'échéancier en fonction de leurs dates d'exécution. Ici, on a $T_1 < T_3 < T_2 < T_4$. Une fois que l'événement e_1 est traité, on passe à l'événement suivant, ici e_3 et ainsi de suite... Pour gérer le temps, une horloge doit être simulée par le moteur d'exécution.

[7] Figure reprise d'un cours de Fabrice Valois donné à l'INSA de Lyon

Approche réactive synchrone

Section 4.1.4, nous avons présenté l'approche réactive synchrone. Nous expliquons ici comment s'écrit un simulateur de réseaux avec un langage de programmation réactif synchrone. L'approche réactive synchrone permet de décrire le fonctionnement des nœuds indépendamment les uns des autres. Que ce soit dans le simulateur de Farid Benbadis et Louis Mandel [56] ou dans GLONEMO, on programme les nœuds puis on les exécute en parallèle.

Pour réaliser un simulateur avec REACTIVEML ou avec un autre langage réactif synchrone, le procédé est toujours le même. Prenons l'exemple de GLONEMO.

```
let process main_node self cloud_x cloud_y =
 signal kill_node in
   do
     run (mac_node self)
     ||
     run (network_node self)
     ||
     run (check_energy self kill_node)
     ||
     run (application_interest self cloud_x cloud_y)
   until kill_node done
   done

let process main =
   for i = 0 to Array.length nodes - 1 dopar
     run (main_node nodes.(i) nuage_x nuage_y)
```

Le processus `main` exécute tous les nœuds en parallèle. Les nœuds sont rangés dans un tableau (`nodes`). Pour chacun des nœuds, on exécute le processus `main_node` qui est lui-même l'exécution en parallèle des différents composants qui constituent un nœud. Le processus `check_energy` calcule l'énergie dépensée par le nœud, et quand celle-ci dépasse la capacité de la batterie, il émet le signal `kill_node` qui a pour effet d'arrêter l'exécution du processus `main_node` correspondant à ce nœud.

L'approche synchrone permet de gérer le temps sans aucun surcoût : chaque pas de calcul correspond à un intervalle fixe de temps simulé. Dans GLONEMO, un instant vaut de 10 millisecondes à 1 milliseconde suivant la précision désirée. Mais cette précision est forcément constante durant toute la simulation.

5.4.2 Intérêt de l'approche à événements discrets

L'approche de simulation à événements discrets permet de considérer du temps continu. En effet même si l'échéancier (figure 5.14) est à événements **discrets**, le temps lui est **continu**. On peut placer un événement à n'importe quelle date sur l'échéancier du moment qu'il est possible d'ordonner sa date d'exécution avec celles des événements qui sont déjà sur l'échéancier. Bien sûr, plus les événements sont rapprochés, plus la simulation prendra du temps. De cette façon, on peut choisir d'être très précis par moment, pour détailler finement le comportement d'une action (l'envoi d'un message par exemple), sans pour autant que ça ralentisse la simulation lorsque rien ne se passe. Avec ce mode d'exécution, lorsque rien ne se passe, il n'y a pas d'événements sur l'échéancier pendant un long intervalle de temps simulé, et donc la simulation est instantanée. En REACTIVEML pour être plus précis, il faut augmenter le nombre de pas de simulation correspondant à un temps simulé fixe. Mais s'il ne se passe rien pendant

par exemple 1 heure de temps simulé, il faut quand même exécuter tous les processus pendant 360000 instants (si un instant correspond à 10 ms de temps simulé). En fait, en REACTIVEML, l'optimisation inter-instants (voir section 4.4.3), permet de diminuer le coût de l'exécution de processus en attente. C'est certainement en partie grâce à cette optimisation que cette limitation intrinsèque aux simulateurs à pas fixes n'est pas pénalisante pour les simulateurs écrits en REACTIVEML. Nous le montrons à la section 5.4.6 où nous étudions les performances de GLONEMO.

Certains pourraient craindre que ce modèle d'exécution synchrone engendre une synchronisation artificielle des nœuds, cependant il est facile de contourner ce problème en implémentant une dérive d'horloge aléatoire au niveau de chacun des nœuds.

5.4.3 Intérêts de l'approche réactive synchrone

Observateurs synchrones

L'intérêt de faire des simulations est de récupérer des informations sur l'exécution de la simulation. Les informations que l'on cherche à extraire dépendent du but de la campagne de simulation. Par exemple, un indicateur digne d'intérêt peut être le nombre de collisions subies par les nœuds. Puisque les indicateurs contrôlés ne sont pas toujours les mêmes, l'utilisateur du simulateur doit être à même de choisir les informations qu'il veut observer. Pour ce faire, l'approche synchrone est très pratique : il suffit d'ajouter un processus qui s'exécute en parallèle et qui peut recevoir les signaux émis sans modifier la simulation. Un tel processus est un observateur. Grâce au formalisme synchrone, on est sûr qu'un observateur ne risque pas de modifier la simulation.

Dans le cas d'un simulateur à événements discrets, il faut placer une nouvelle tâche sur l'échéancier. La difficulté est donc de savoir quand il faut placer ce nouvel événement sur l'échéancier.

Création dynamique

Contrairement au modèle d'exécution synchrone "à la LUSTRE", le modèle réactif synchrone de Boussinot (comme REACTIVEML, par exemple) permet de créer dynamiquement des processus pendant la simulation. Il est par exemple possible de créer, pendant l'exécution de la simulation, des nœuds qui interagissent naturellement avec le reste du réseau.

Affichage, et interactions avec la simulation

Nous avons déjà évoqué l'importance de l'outil d'affichage pour les simulateurs de réseau. Dans de nombreux simulateurs à événements discrets, un fichier détaillant tous les événements qui se sont produits est généré durant la simulation. Ce fichier sert ensuite à afficher le déroulement de la simulation. En quelques sortes, la simulation est "rejouée". Bien sûr, on pourrait exécuter à l'aide d'un *pipe* la commande : `mon_simulateur | mon_afficheur` de sorte que l'outil d'affichage s'exécute en même temps que la simulation. Mais pour autant, l'afficheur n'afficherait pas l'état de la simulation en "temps réel", le *pipe* est un mode de communication asynchrone.

Le modèle réactif synchrone permet d'utiliser un observateur pour gérer l'affichage de la simulation. Ainsi l'affichage correspond exactement à la simulation qui est en train de s'exécuter. Grâce au formalisme synchrone, on est sûr que l'ajout en parallèle de cet observateur ne modifie pas l'exécution de la simulation.

De plus, l'interface graphique offre un bon moyen d'interagir avec la simulation qui n'est possible que si l'affichage est synchrone avec la simulation en cours. Il est par exemple possible de créer un nœud durant la simulation en cliquant. Nous avons également implémenté un processus qui crée un

paquet à envoyer pour le nœud le plus proche du clic de la souris. Une telle fonction aurait été difficile à implémenter avec un simulateur à événements discret parce qu'il faut être capable de dater le clic. Par exemple si un clic est envoyé (voir figure 5.14) pendant le traitement de T_1, on sait simplement le placer entre T_1 et T_2 mais il est difficile de savoir s'il faut le placer sur l'échéancier avant ou après T_3. Il est donc impossible de lui donner une date exacte.

5.4.4 Confort de programmation

Selon nous, le modèle de programmation synchrone qui offre gratuitement la notion de temps, est très confortable pour programmer un simulateur de réseaux. Ce modèle permet de ne plus avoir à gérer la notion de temps.

Par exemple une difficulté que l'on rencontre pour décrire un réseau à l'aide d'un simulateur à événements discrets et à laquelle on n'est pas soumis dans un simulateur à pas fixe est le respect de la *causalité*. Prenons un exemple. Si l'on crée deux événements qui arrivent au même moment ((e_1, T), (e_2, T)) on ne sait pas lequel sera exécuté en premier. Donc, si l'on souhaite créer deux événements consécutifs qui sont sensés arriver au même moment, on est obligé de les dater différemment. Cet artifice de simulation ne correspond pas à la réalité que l'on souhaite simuler. De plus, la nécessité d'implémenter ce type d'artifices traduit les précautions qui doivent être prises pour obtenir des simulations réalistes.

Le modèle de programmation synchrone réactif ôte ce souci au programmeur. En effet, en REACTIVEML, on implémente une séquence d'événements à exécuter pendant un seul instant logique. Le processus `seq` exécute consécutivement les tâche e_1 puis e_2 à chaque instant.

```
let process seq =
 loop
   e1;
   e2;
   pause;
 end
```

Même si ce type d'artifice ne remet pas toujours en cause le réalisme des simulations, il constitue une difficulté qui doit être prise en compte par le programmeur et que le formalisme synchrone réactif permet d'éviter.

5.4.5 Autres qualités de REACTIVEML

REACTIVEML est un langage à la ML pour la programmation réactive. Nous avons expliqué les avantages du modèle de programmation réactif synchrone pour la programmation de simulateurs, mais un autre atout de REACTIVEML est qu'il s'appuie sur OCAML. Nous ne sommes pas les premiers à penser que OCAML est un langage qui convient à la programmation de simulateurs. En 2004 des chercheurs de l'EPFL ont développé le simulateur de réseaux NAB [29] en OCAML pour justement bénéficier des avantages de ce langage. Mais NAB était un simulateur à événements discrets qui ne bénéficiait pas des atouts de la programmation réactive.

Tout comme OCAML, REACTIVEML est un langage doté d'un **système de typage fort**. Les contraintes qu'impose un système de typage fort obligent le programmeur à une certaine rigueur de programmation, le code obtenu est donc plus propre. De plus, les techniques classiques de preuves de sûreté du typage [71] permettent de détecter des bogues à la compilation. Par exemple, dans GLONEMO, nous avons créé (entre autres) les types `packet2` et `packet3` correspondant respectivement aux

paquets de la couche MAC et de la couche réseau. En bref, un `packet3` est encapsulé dans un `packet2` avant d'être envoyé -via le canal radio- au nœud voisin. À la programmation des confusions auraient pu être possibles entre ces deux types de paquets. Le typage fort permet de détecter dès la compilation les éventuelles erreurs et de les déboguer rapidement.

OCAML est un **langage fonctionnel** ce qui facilite la programmation modulaire [41]. OCAML étant un langage généraliste, il est doté de quelques **librairies**. La librairie `Graphics`, notamment, nous a permis d'implémenter très facilement l'interface graphique de GLONEMO. Enfin, OCAML s'appuie sur un **compilateur efficace**.

Connexion à LUCKY. Pour simuler un réseau, il peut être nécessaire de modéliser de façon abstraite des systèmes physiques complexes. Les simulateurs de réseaux classiques permettent seulement d'utiliser la fonction de génération de nombres aléatoires du langage dans lequel ils sont programmés. Avec LUCKY, on peut programmer facilement des comportements non-déterministes complexes. Dans REACTIVEML, on bénéficie de cet atout car REACTIVEML est connecté à LUCKY : les processus REACTIVEML peuvent interagir avec des processus LUCKY.

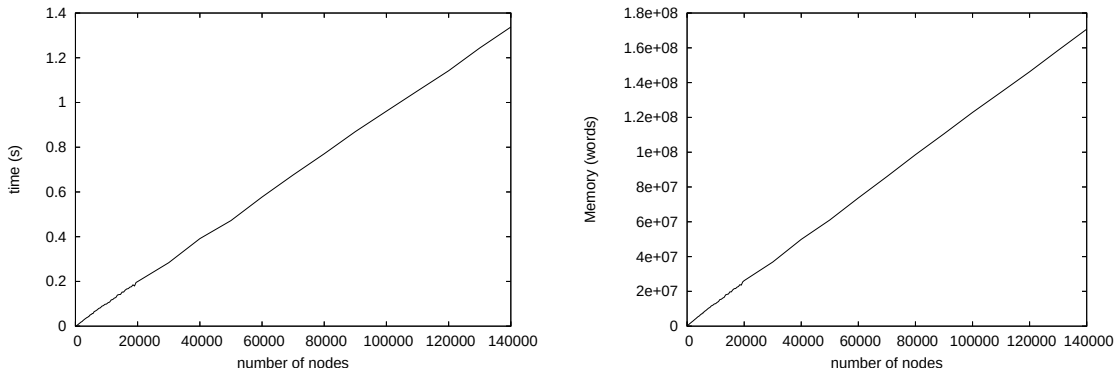
5.4.6 Passage à l'échelle : exemple de GLONEMO

Nous avons montré que le modèle de programmation réactif synchrone est adapté à la programmation de simulateurs. De plus, REACTIVEML hérite des atouts de OCAML comme la programmation fonctionnelle et le typage fort. REACTIVEML est donc un langage confortable pour programmer des simulateurs. Mais est-ce que REACTIVEML permet de programmer des simulateurs efficaces, c'est-à-dire capable de simuler des réseaux de grande taille ? A priori, le confort de programmation permet de se concentrer sur l'implantation d'algorithmes efficaces. Il faut cependant que le surcoût du langage de haut niveau reste raisonnable. Nous analysons à présent les performances de notre simulateur GLONEMO.

Il serait intéressant de pouvoir comparer GLONEMO à d'autres simulateurs afin d'évaluer son efficacité. Cependant, il est difficile d'avoir accès à un simulateur équivalent. Les simulateurs sont différents suivant leur utilisation. On a besoin de simulateurs pour comparer deux algorithmes (MAC, routage, organisation etc). Dans ce cas, les modèles de chacun des composants ne sont pas au même niveau d'abstraction dans le simulateur. En effet, l'algorithme à étudier doit être complètement modélisé mais les autres composants, notamment les protocoles au-dessus (au sens de la pile protocolaire) n'ont pas besoin d'être aussi bien détaillés. Un simulateur peut aussi servir à évaluer une solution de déploiement d'un réseau de capteurs dans son ensemble. Dans ce cas, les utilisateurs sont plutôt des industriels qui souhaitent pour déployer un réseau de capteurs quelques garanties quant à sa durée de vie. Dans ce cas, le simulateur doit modéliser toute la pile protocolaire au même niveau d'abstraction. GLONEMO se range plutôt dans cette deuxième catégorie mais il est difficile de trouver des outils semblables permettant une comparaison : les outils utilisés par les industriels sont rarement libres. Ce qui montre qu'il n'existe pas vraiment d'outils auxquels nous pouvons comparer GLONEMO et que ceux qui existent ne sont pas toujours accessibles.

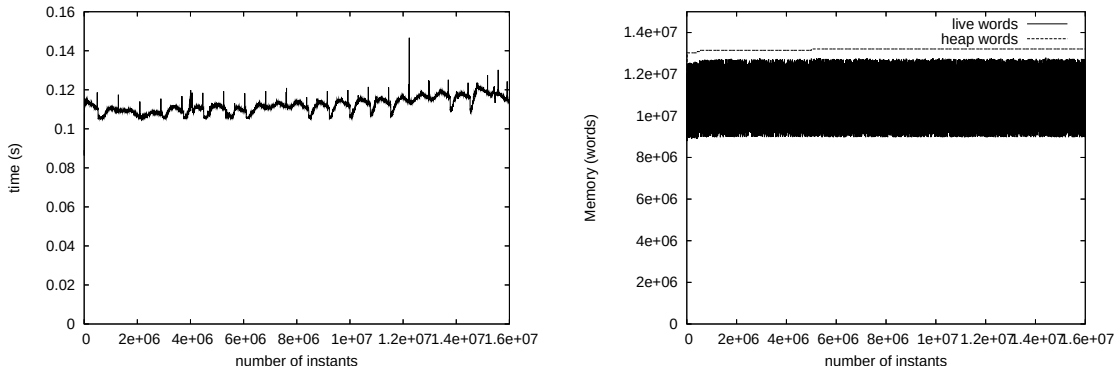
Afin de modéliser précisément la consommation d'énergie, GLONEMO est un simulateur à grain fin. Dans GLONEMO l'exécution d'un instant logique représente 10^{-2} secondes[8]. Donc pour simuler le fonctionnement du réseau pendant une heure, il faut exécuter 360000 instants. Pour un réseau de 10000 nœuds, une telle simulation prend environ 11 heures.

[8] C'est un paramètre que nous pouvons faire varier, 10^{-2} secondes est la valeur typique que nous donnons ici à titre d'exemple.



(a) Temps de simulation d'un instant logique. (b) Utilisation de la mémoire.

FIG. 5.15 – Temps de simulation et mémoire utilisée en fonction du nombre de nœuds.



(a) Temps de simulation d'un instant logique. (b) Utilisation de la mémoire.

FIG. 5.16 – Simulation d'un réseau de 10000 nœuds pendant 20 jours.

Pour étudier l'impact du nombre de nœuds sur l'efficacité de la simulation, nous avons mesuré le temps de simulation d'un instant et la mémoire nécessaire en fonction du nombre de nœuds, figure 5.15. Ces deux paramètres augmentent linéairement avec le nombre de nœuds. C'est une bonne nouvelle puisque ça montre que le moteur de simulation n'ajoute pas de surcoût : le coût de simulation de n nœuds est exactement égal à n fois le coût de simulation d'un nœud. Pour simuler un réseau de 140000 nœuds, il faut compter 1.4 secondes par instant de simulation. Simuler un si grand réseau prend donc du temps mais vu qu'il ne faut que 700 MB de mémoire, c'est faisable sur un ordinateur actuel.

Pour montrer la fiabilité de REACTIVEML au travers de GLONEMO, nous avons tracé le temps de simulation d'un instant logique et la mémoire utilisée au cours de la simulation, pour une simulation de 16000000 instants, figure 5.16. Le temps d'exécution d'un instant logique (figure 5.16(a)) reste constant au cours de la simulation, il y a bien sûr des pics qui correspondent certainement à des moments d'activité accrue du réseaux mais la vitesse de simulation ne diminue pas au cours de la simulation. Enfin, la mémoire nécessaire pour effectuer une simulation est constante (figure 5.16(b)). De cette façon, on peut présager que si la mémoire est suffisante au début de la simulation, elle le sera tout au long de celle-ci.

Chapitre 6

Modélisation haut niveau avec l'approche asynchrone

Sommaire

Dans le chapitre précédent, nous avons présenté une modélisation précise d'un réseau de capteurs. Ce modèle précis nous permet de bien estimer la consommation d'énergie. Cette modélisation effectuée dans un formalisme exécutable permet de simuler le modèle pour étudier le comportement du réseau de capteurs. Cependant, même après de nombreuses simulations, on n'est jamais sûr du comportement du réseau. En effet, il se peut qu'il existe un cas de fonctionnement pour lequel la durée de vie du réseau est beaucoup plus faible que pour les simulations effectuées. Les simulations peuvent ne pas représenter l'ensemble des cas possibles. C'est pourquoi nous pensons qu'il est intéressant de proposer une approche exhaustive qui garantisse une durée de vie minimale du réseau.

Les techniques de model-checking sont implémentées dans différents outils, elles sont utilisées pour la vérification de systèmes logiciels ou matériels. Ces techniques ont permis, pour des systèmes critiques, la vérification de propriétés non triviales. Pour proposer une approche exhaustive complémentaire à une analyse du prototype à l'aide de simulation, nous allons utiliser le model-checking. L'étude de cas que nous présentons consiste à comparer deux protocoles de routage. Nous avons choisi le formalisme asynchrone IF pour notre modélisation. Un réseau de capteurs étant un système globalement asynchrone et localement synchrone (voir section 4.1.5 page 72), un formalisme asynchrone convient bien pour une modélisation haut niveau. En effet, les communications entre les nœuds se font via la radio, elles sont donc plutôt asynchrones. De plus, IF permet de générer le graphe d'état du modèle afin d'exécuter les techniques de model-checking. Une des limitations des techniques d'exploration exhaustive d'un modèle est la taille du modèle. En effet, ces techniques peuvent s'avérer particulièrement coûteuses pour des systèmes complexes. Un réseau de capteurs est un système trop complexe pour que l'on puisse espérer analyser exhaustivement le comportement d'un modèle de réseau de capteurs détaillé à

l'extrême. Pour cette expérience nous avons modélisé le réseau de façon assez abstraite ; les abstractions faites sont inspirées de simulations GLONEMO.

Dans ce chapitre, nous comptons deux contributions principales. Tout d'abord, nous proposons une modélisation d'un réseau de capteur à l'aide de IF. Nous proposons un modèle assez abstrait des couches basses qui soit suffisamment pertinent pour permettre de tirer des conclusions sur les protocoles de routage. De plus, cette modélisation prend en compte la consommation d'énergie. L'autre contribution consiste à trouver le scénario correspondant à la durée de vie minimale. Le contenu de ce chapitre est publié dans la référence [65]. Le plan du chapitre est le suivant : dans la section 6.1 nous présentons rapidement le cas d'étude, cet exemple est à peu près le même que celui du chapitre 5. La section 6.2 détaille l'implémentation en IF. La section 6.3 explique comment nous avons calculé la durée de vie pire cas. Enfin, la section 6.4 présente des résultats numériques.

6.1 Cas d'étude

Nous proposons dans ce chapitre de calculer la durée de vie pire cas, donc la plus courte, d'un réseau de capteurs. Le but est de comparer deux protocoles de routage. Pour montrer la faisabilité des techniques à base de model-checking pour l'analyse de réseaux de capteurs, nous insistons sur le fait que notre cas d'étude est représentatif d'un réseau de capteurs type.

L'application est la même que celle implémentée dans GLONEMO, il s'agit d'une application de remontée d'alarme semblable à la détection de feux de forêt décrite section 2.1.1. Ici, on considère que le réseau est chargé de détecter un nuage de pollution. L'environnement est donc ce nuage qui se déplace sous l'effet du vent. Comme expliqué section 5.2, il est important qu'une modélisation du phénomène à détecter soit prise en compte dans la modélisation d'un réseau de capteurs.

Les deux protocoles de routage que l'on cherche à comparer sont l'inondation et la diffusion dirigée. Ils sont tous deux détaillés section 2.1.2. Pour obtenir des comparaisons valables, il faut que le reste du modèle soit identique.

Nous supposons que le protocole MAC est un protocole dédié aux réseaux de capteurs et donc conçu pour économiser l'énergie. Ce type de protocole éteint la radio le plus souvent possible ce qui crée des délais supplémentaires dans les communications. Nous avons considéré que les paquets envoyés à un voisin (*unicast*) sont acquittés (voir section 2.1.3 page 26 une explication du mécanisme d'acquittement) alors que ceux destinés à tous les voisins (*broadcast*) ne le sont pas.

L'implémentation de ce cas d'étude en IF est décrite section suivante. Les protocoles de routage sont traduit en IF de façon fidèle alors que les autres éléments du réseaux ont été abstraits afin de diminuer la complexité du modèle et de rendre possible son analyse par model-checking.

6.2 Modélisation en IF

Le réseau de capteurs que nous considérons est formé d'un réseau de *nœuds* qui communiquent via des *liens radio* et réagissent à un *environnement*. Nous proposons un modèle formel de chacun de ces trois composants.

Modélisation de l'environnement. Dans cette application, l'environnement stimule les capteurs en envoyant des signaux indiquant que le niveau de pollution a dépassé un certain seuil. Dans la réalité le niveau de pollution suit certaines lois spécifiques quant à son déplacement ou à sa répartition spatiale. Nous avons choisi ici de modéliser cet environnement par un processus IF (`Environment`) dédié

s'exécutant en parallèle de l'application. Ce processus modélise le comportement d'un nuage polluant comme suit :

- il envoie un message `envInput` à un certain nœud *n*, indiquant que pour ce nœud le niveau de pollution a dépassé le seuil au-delà duquel il doit envoyer une alarme.
- puis, le processus `Environment` choisit le nœud qui sera le prochain nœud à être stimulé par le nuage de pollution. Ce nœud est choisi de façon non déterministe parmi les voisins de *n* (y compris le nœud *n* lui-même).

La période d'exécution de ce comportement cyclique est un paramètre du modèle. Changer cette constante de temps revient à modifier la vitesse de déplacement du nuage. En effet, plus la période est faible plus le nuage se déplace souvent. On peut considérer que si le nuage se déplace sous l'effet du vent, cette constante de temps permet de modéliser différents profils de vent.

Modélisation d'un nœud. Les nœuds du réseau sont modélisés par des processus IF appelés `Node`. Chaque nœud du réseau est représenté par une instance du processus `Node`. Le puits est une instance particulière du processus `Node` que l'on distingue grâce à son numéro d'instance. Le nœud puits est l'instance 0 du processus `Node`. Suivant le protocole de routage considéré, un nœud suit la description informelle suivante :

- Pour l'inondation, à la réception d'un message `envInput`, chaque nœud envoie à tous ses voisins ("*broadcast*") un paquet `Alarm`. Ce paquet `Alarm` contient un identifiant unique. À la réception d'un paquet `Alarm`, le nœud vérifie à l'aide de l'identifiant qu'il n'a pas déjà reçu cette alerte. S'il l'a déjà reçue, il ne fait rien. Sinon, il la ré-émet à l'ensemble de ses voisins.
- Pour le protocole de diffusion dirigée, le nœud puits initialise la construction de routes en envoyant un message `Interest` à tout le réseau. Ce message est envoyé selon le protocole de routage inondation décrit plus haut. De cette façon, chaque nœud mémorise l'identifiant du nœud dont il a reçu le message `Interest` en premier dans un champ "next neighbor". Si le nœud est sollicité par un signal `envInput`, il envoie un message (en "*unicast*") à son voisin "next neighbor". Ce message `Alarm` atteint le puits en étant ré-émis par chaque nœud à son "next neighbor" selon le même mécanisme.

Modélisation des communications radio. Nous prenons en compte la topologie du réseau et les délais et erreurs de communication dûs au canal radio à l'aide d'un processus `Topology`. Ce processus contient une matrice booléenne représentant les relations de voisinage pour tous les nœuds. Cette matrice carrée contient autant de ligne que de nœuds et l'élément (i, j) est vrai si et seulement si le nœud numéro j est à portée radio du nœud numéro i. La topologie qui initialise cette matrice est fournie par l'utilisateur. La topologie ne change pas au cours de la vie du réseau. Nous modélisons les communications de deux façons différentes, pour les messages envoyés en *broadcast* ou en *unicast* :

- L'émission d'un paquet en *broadcast* consiste pour l'émetteur à envoyer un paquet à l'ensemble de ses voisins en une seule émission. Pour ce faire, nous avons utilisé en IF une boucle *while* dans laquelle l'émetteur envoie un signal à chacun de ses voisins. Du point de vue du réseau, les messages sont envoyés simultanément. En cas de collision, si un des récepteurs est déjà en communication, il ne reçoit pas le message. En effet, les paquets envoyés en *broadcast* ne sont pas acquittés.
- À l'inverse, les paquets envoyés à un seul voisin (en *unicast*) sont acquittés. Donc tant qu'il y a des collisions, l'émetteur ré-émet son paquet. Les ré-émissions se font après un délai non-déterministe compris dans un certain intervalle. En IF, nous avons utilisé des transitions retardables (*delayable transitions*) pour modéliser les ré-émissions. L'énergie et le temps nécessaires

à l'émission d'un paquet *unicast* dépendent du nombre de retransmissions.

Temps. Le processus `Topology` contient une horloge globale appelée `Lifetime`. Cette horloge démarre quand le réseau commence à s'exécuter. Elle fournit donc le temps écoulé depuis l'initialisation du réseau. Elle permet de calculer la durée de vie du réseau, voir section 6.3.

Énergie. Notre modèle d'un nœud doit prendre en compte la consommation d'énergie. Contrairement à la modélisation plus fine présentée au chapitre 5, ici nous ne calculons pas l'énergie consommée à l'aide d'un modèle de puissance instantanée, mais bien avec des coûts énergétiques associés à certaines actions. Nous supposons ici que seules les communications consomment. En fait, nous avons affecté à chaque action : émission et réception, un coût. Ce coût dépend du type d'émission, *broadcast* ou *unicast*. Pour une émission ou réception ce coût est fixe, mais en cas de ré-émission, le coût de la ré-émission est à nouveau pris en compte. On ne compte pas l'énergie dépensée par le puits. Pour les autres nœuds, quand ils n'ont plus d'énergie, ils sont morts, c'est-à-dire qu'ils n'effectuent plus aucune action.

6.3 Calcul de la durée de vie pire cas

Dans cette section, nous expliquons l'approche utilisée pour calculer la durée de vie pire cas du réseau à partir de la spécification IF.

La sémantique opérationnelle de IF permet de représenter le comportement modélisé par la spécification IF à l'aide d'un *graphe d'état*. Ce graphe est composé d'états et de transitions. Dans un état du graphe est encodé l'état de contrôle et les différentes valeurs des variables pour chaque processus IF. Les transitions correspondent aux opérations effectuées sur le modèle IF. Ce graphe d'état peut être calculé à la volée par le moteur de simulation IF. Plusieurs outils sont alors disponibles pour explorer ce graphe et permettent différentes méthodes de validation (voir section 4.4.1 page 79). On peut par exemple, faire des simulations interactives, de la génération de test ou encore du model-checking.

La durée de vie pire cas est la durée de vie la plus courte. Nous ramenons le calcul de durée de vie pire cas à un calcul de plus court chemin dans le graphe d'état. Le scénario pire cas correspond dans le graphe d'état au plus court chemin entre l'état initial (ou le réseau démarre) et un état final. Les états finals sont ceux où le réseau est décrit comme mort. Nous proposons différents critères pour déclarer que la vie du réseau est terminée, vus qu'ils sont indépendants du calcul du pire cas, nous les présentons plus loin, section 6.4.1. La notion de longueur de chemin dans le graphe d'état s'entend par rapport à l'horloge `Lifetime`. La valeur de l'horloge `Lifetime` est associée à chaque état du graphe d'état. Elle représente le temps écoulé depuis l'initialisation du réseau. Le scénario pire cas est donc celui où un état final est atteint avec la plus petite valeur de l'horloge `Lifetime`. C'est donc le plus court chemin, au sens de l'horloge `Lifetime`, entre l'état initial et un état final.

Pour calculer le plus court chemin, nous avons construit un algorithme à partir de l'algorithme classique A*. L'algorithme est présenté figure 6.1. Les spécificités introduites pour notre cas de figure concernent essentiellement l'heuristique qui guide l'exploration.

Algorithme A*
p : état
Q : ensemble de paires (état, durée)
V : ensemble d'états (visités)
begin
$Q \leftarrow$ (état_initial, 0) ; $V \leftarrow \emptyset$
while $Q \neq \emptyset$ **do**
  extraire une paire (p, d) de Q t.q. d est minimal
  if $p \notin V$ and not is_goal(p) **then**
    foreach q in succ(p)
      $Q \leftarrow Q \cup \{(q, cost(q))\}$
  else
    if is_goal(p) **then**
      return d
    endif
endwhile
end

FIG. 6.1 – Algorithme A* de recherche de plus court chemin

A* est un algorithme qui peut s'effectuer à la volée sans avoir a priori la connaissance du graphe entier. Ainsi, l'exploration du graphe et la recherche du plus court chemin se font conjointement. L'exploration du graphe se fait en suivant le chemin le plus prometteur. On explore le graphe à partir de l'état initial. Une liste (Q) contient les états atteints qui n'ont pas déjà été explorés. Au début Q ne contient que l'état initial, ensuite il faut choisir parmi cette liste le nœud "qui semble" rapprocher le plus d'un état final. Pour ce faire, on associe aux états explorés une fonction $cost$. Cette durée, $cost(p)$, est une borne inférieure du plus court chemin passant par l'état p. Autrement dit, le plus court chemin passant par l'état p est de longueur supérieure ou égale à $cost(p)$. L'exploration se fait donc en explorant d'abord l'état dont la fonction $cost$ est la plus faible. Pour que l'heuristique soit correcte, il faut que la fonction $cost(p)$ soit forcément plus petite que le plus court chemin passant par p de sorte que si l'on trouve, par un autre état p', un chemin de longueur inférieure à $cost(p)$ menant à un état final on sait qu'il est inutile d'explorer les successeurs de p. Nous calculons la fonction $cost(p)$ comme suit. $cost(p)$ est une durée, c'est la somme de d et d'. d est la valeur de `Lifetime` à l'état p, c'est-à-dire le temps nécessaire pour atteindre l'état p. Et d' est un minorant de la distance de p à un état final. d' correspond au temps mis par le nœud le plus faible de l'état p pour se décharger complètement s'il ne fait que réaliser l'action la plus coûteuse possible. L'action la plus coûteuse possible est celle qui fait dépenser à un nœud du réseau le maximum d'énergie en un minimum de temps. Soit a cette action. Notons e_a le coût énergétique de a et d_a sa durée. a est telle que le rapport $\frac{e_a}{d_a}$ est maximal.

Plus formellement, voici le calcul de $cost(p)$:

```
cost(p) =
let
   d = la valeur de Lifetime pour l'état p
   e = l'énergie du nœud le plus faible dans l'état p
   x = e/e_a
   d' = x × d_a
in
   d + d'
```

Enfin, il suffit de mémoriser les états qui constituent le plus court chemin pour avoir en plus de la durée de vie pire cas, le scénario pire cas.

6.4 Résultats expérimentaux

6.4.1 Critère de durée de vie

C'est assez difficile de décider si le réseau est mort ou s'il répond encore aux besoins de l'application. Il y a une phase de transition pendant laquelle certains nœuds n'ont plus d'énergie mais le réseau peut encore répondre à certaines requêtes. Puis, même s'il y a encore quelques nœuds actifs, le réseau ne peut plus répondre à aucune requête. Il n'y a donc pas un critère général qui permette de décider que le réseau est mort ou pas. Cependant pour notre étude, il est nécessaire d'établir un critère discriminant pour décider quand la vie du réseau est terminée et quelle est sa durée de vie pire cas. Afin d'avoir un critère irréversible, nous supposons que les nœuds ne se rechargent pas, une fois qu'ils n'ont plus d'énergie, ils sont définitivement morts. Nous proposons deux critères.

- Critère A : un nœud n'a plus d'énergie. Dès qu'un nœud a épuisé sa batterie, le réseau est déclaré mort.
- Critère B : le réseau est déconnecté, au moins deux partitions apparaissent. Puisque nous supposons qu'il n'y a qu'un seul puits dans le réseau, si le réseau est déconnecté alors certains nœuds ont encore de l'énergie mais ne peuvent plus communiquer avec le puits. Précisons que pour le protocole diffusion dirigée, lorsqu'un nœud meurt, le puits essaie de réparer le réseau en envoyant un nouvel intérêt. Si le réseau n'est pas déconnecté une nouvelle route est trouvée, sinon le réseau est mort.
- Un autre critère pourrait être : une proportion des nœuds du réseau n'a plus d'énergie. Ce critère n'est pas fondamentalement différent du critère A, nous ne l'avons donc pas implémenté.

6.4.2 Expérimentations

Nous avons fait des expériences d'analyse de la durée de vie pire cas pour différentes topologies allant de quatre à onze nœuds. Les topologies sont représentées figure 6.2, par exemple le réseau à quatre nœuds que nous avons étudié est celui contenant les nœuds **0**, **1**, **2** et **3** du réseau représenté figure 6.2. Comme expliqué précédemment, notre technique de recherche du scénario pire cas est basée sur l'exploration partielle d'un graphe d'état. La taille de ce graphe, et donc le temps de calcul, dépend beaucoup de la taille du réseau étudié. La machine utilisée pour ces expériences est un PC équipé d'un processeur de 3.2Ghz et de 2GB de mémoire RAM. Pour un réseau comportant quatre nœuds, le calcul du scénario pire cas prend moins de trois minutes, alors que pour un réseau contenant onze nœuds, il faut plus de deux jours. En fait, à partir de sept nœuds, la mémoire vive n'est plus suffisante pour calculer le plus court chemin. Le temps de calcul devient donc arbitrairement long puisque la machine

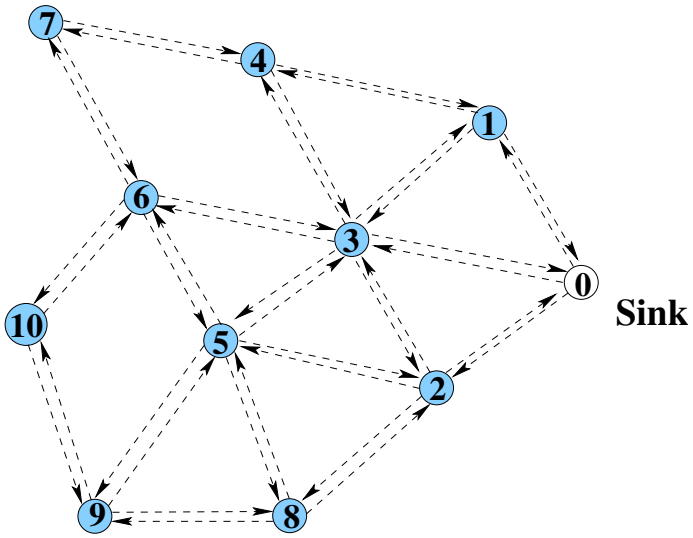


FIG. 6.2 – Les topologies testées, de 4 à 11 nœuds

utilise de la mémoire virtuelle (*swap*). Les résultats présentés plus loin concernent donc un réseau de six nœuds, le réseau de la figure 6.2 avec les nœuds **0** à **5**.

Pour nos expérimentations, il faut choisir les consommations associées aux différentes actions. Nous avons fixé le coût d'une transmission à 3 unités d'énergie et celui d'une réception à 2 unités d'énergie. Ces choix sont arbitraires, nous aurions bien pu prendre d'autres valeurs. Les batteries des nœuds contiennent, à la mise en route, 40 unités d'énergie. Un nœud ne peut donc pas envoyer plus de 14 émissions pendant toute sa vie ce qui est très peu. En effet, nous étudions le comportement du réseau pendant un intervalle de temps assez court. Il faudrait extrapoler les résultats obtenus pour en déduire la *vraie* durée de vie du réseau. Cependant, le cas d'étude consiste à comparer des protocoles et pas à obtenir une valeur exacte pour la durée de vie du réseau. Remarquons que les simulateurs font le même type d'abstraction : le réseau n'est simulé que pendant une courte durée qui ne correspond pas à la durée de vie complète du réseau.

Au niveau du temps, une transmission peut prendre de 1 à 3 unités de temps et si une retransmission est nécessaire (en cas de collision), la transmission dure 5 unités de temps supplémentaire. Le nuage envoie des signaux indiquant à un capteur un niveau de pollution accru toutes les 7 unités de temps.

	Critère A	Critère B
Inondation	14	21
Diffusion dirigée	41	52

TAB. 6.1 – Durée de vie minimum

Sur le tableau 6.1 sont affichées les durées de vie (en unités de temps) du réseau pour les deux protocoles étudiés et pour nos deux critères. Comme attendu, le protocole diffusion dirigée permet de garantir une durée de vie plus longue que le protocole inondation et ce pour nos deux critères. C'est logique puisque pour envoyer une alarme au puits en utilisant l'inondation, de nombreux messages sont générés. De plus, le critère B est plus long a atteindre, c'est logique aussi puisque si le réseau est déconnecté (critère B) alors au moins un nœud est mort (critère A).

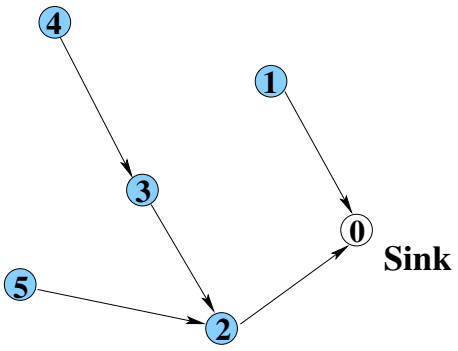


FIG. 6.3 – Installation des gradients pour le protocole diffusion dirigée dans le scénario pire cas.

6.4.3 Importance du pire cas

Pour souligner l'intérêt d'avoir des outils de modélisation qui permettent d'étudier les comportements pire cas, nous présentons deux expériences. Tout d'abord nous montrons que les durées de vie pire cas obtenues sont bien différentes des cas moyens puis nous montrons qu'il peut être intéressant d'obtenir le scénario pire cas.

Pire cas contre cas moyen. Pour souligner les différences entre le cas pire et le cas moyen, nous avons comparé, tableau 6.2, les durées de vie pire cas avec les durées de vie moyennes. Le simulateur IF, de la boîte à outils IF, permet de générer une séquence aléatoire dans le graphe d'état. Cette séquence correspond à une exécution possible du système. Générer de telles séquences revient à simuler notre modèle. Sur le tableau 6.2, le cas moyen correspond à la moyenne des durées de vie effectuées sur dix simulations. Cette expérience montre bien que l'on n'aurait pas pu déduire le pire cas de simulations.

	Pire cas	Cas moyen
Inondation	14	24
Diffusion Dirigée	41	101

TAB. 6.2 – Calcul de la durée de vie pour le critère A

Scénario pire cas. Nous avons montré que notre modélisation permet d'obtenir la durée de vie pire cas, mais nous pouvons également obtenir le scénario pire cas qui mène à la durée de vie la plus courte. Nous pensons que cette information est très pertinente pour l'analyse d'un réseau. La figure 6.3 montre par exemple l'installation des gradients qui mène au pire cas de consommation. Les nœuds ne choisissent pas le voisin auquel il vont envoyer leurs paquets en direction du puits comme on s'y attendait. Le concepteur de ce protocole de routage espère plutôt que les nœuds choisissent un chemin proche du plus court chemin pour atteindre le puits. Ici, avec nos hypothèses de modélisation, nous observons que ce choix surprenant de gradients réduit la durée de vie de presque 60% (table 6.2). Une telle information est pertinente, elle peut notamment aider à la conception de protocoles de routage.

6.5 Conclusion

Nous avons proposé une méthode pour calculer la durée de vie pire cas d'un réseau de capteurs. Il faut d'abord modéliser le réseau de capteurs à l'aide d'un formalisme suffisamment expressif et d'une sémantique opérationnelle bien définie. Ensuite, tous les comportements de cette spécification peuvent être modélisés par un graphe d'état et trouver le scénario pire cas revient à exécuter un algorithme de plus court chemin sur ce graphe. Nous avons illustré cette approche sur une étude de cas représentative : à l'aide de IF, nous avons comparé deux protocoles de routage.

Chapitre 7

Cadre formel pour des abstractions modulaires prenant en compte l'énergie

Sommaire

Dans le chapitre précédent, nous avons modélisé de façon très abstraite un réseau de capteurs. Cette modélisation permet de représenter l'ensemble des comportements possibles du système avec un graphe d'état de taille modéré et donc de pouvoir prouver des propriétés sur l'ensemble des comportements (à l'aide de vérification par modèles). Le chapitre 5 présentait un modèle assez détaillé d'un réseau de capteurs que l'on était capable d'exécuter (de simuler) mais dont il est inenvisageable d'explorer de façon exhaustive le graphe d'état. Nous avons donc besoin de faire des abstractions contrôlées. Pour vérifier des propriétés portant sur la durée de vie, un modèle abstrait doit toujours surestimer la consommation. Le but du présent chapitre est de proposer un cadre qui définisse formellement ce qu'est une abstraction dans ce contexte. Nous nous plaçons dans le cas d'automates synchrones qui ne consomment que sur les états.

La section 7.1 introduit avec des exemples les besoins que nous avons pour notre relation d'abstraction. Les exemples sont issus du domaine des réseaux de capteurs : on compose un protocole

MAC avec différents modèles de consommation de radio plus ou moins abstraits. La section 7.2 définit formellement notre notion d'abstraction. Ensuite, nous revenons sur l'exemple section 7.3. La section 7.4 commente certains choix que nous avons faits.

7.1 Motivations et exemples

Composants

Tous les modèles de réseaux de capteurs que nous avons construits ont été faits à partir de composants. On ne saurait pas créer un réseau de capteurs d'un bloc, en un seul composant ; de même, on ne sait pas modéliser un réseau de capteurs sans passer par des modèles de chacun des composants. S'il est difficile de concevoir un réseau de capteurs d'un coup, c'est également difficile de concevoir un modèle abstrait de réseau de capteurs en un seul bloc. Il semble plus aisé de concevoir des abstractions pour chacun des composants puis d'en déduire ainsi un modèle abstrait du système tout entier. Pour pouvoir construire un modèle global abstrait à partir de modèles des différents composants, il faut que la composition de composants préserve la relation d'abstraction. En notant $A \preceq B$ la relation "B *est une abstraction de* A" et $\|$ la compostion, nous avons besoin d'une relation du type :

$$A \preceq B \Longrightarrow A \parallel M \preceq B \parallel M$$

Enfin, pour concevoir des abstractions efficaces en pratique, nous nous sommes apperçus qu'il est utile de tenir compte de la manière dont les composants sont utilisés dans le système global.

Automates synchrones

Nous avons choisi de nous intéresser d'abord au cas d'automates synchrones qui ne consomment de l'énergie que sur les états. Il s'agit donc d'automates semblables à celui représenté figure 4.17 page 73. Le formalisme synchrone présente l'avantage de modéliser le temps de façon implicite avec les instants logiques. S'intéresser prioritairement au formalisme synchrone pour définir des abstractions est justifié puisque nous avons réalisé LUSSENSOR, un modèle global détaillé de réseau de capteurs en LUSTRE, et LUSTRE est un langage de modélisation synchrone. Nous avons expliqué section 4.2.2 page 77 que dans le cadre synchrone modéliser la consommation d'énergie uniquement sur les états suffisait.

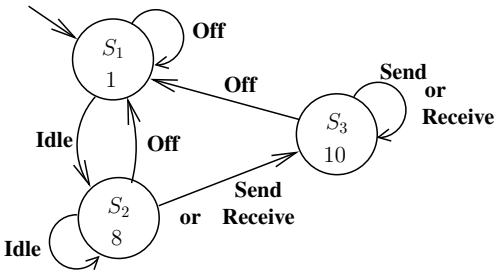
Abstraction

Nous souhaitons définir une relation d'abstraction pour des modèles de consommation. Une abstraction valide d'un modèle est une abstraction conservative pour la propriété à vérifier. De cette façon, si la propriété est vraie sur le modèle abstrait, elle l'est aussi sur le modèle plus détaillé ; au contraire, si la propriété est fausse sur le modèle abstrait, on ne peut pas conclure. Nous souhaitons vérifier des propriétés pire-cas comme par exemple : "dans le pire des cas, la durée de vie du système est supérieure à T" ou "durant t unités de temps, au pire le système consomme E unités d'énergie". Donc, nos modèles sont étiquetés avec des consommations pire-cas, c'est-à-dire des majorants de la consommation. Une abstraction valide est donc telle que le modèle abstrait consomme au moins autant que le modèle détaillé. C'est donc une condition pour la définition de notre abstraction, si B est une abstraction de A, noté $A \preceq B$, alors B consomme au moins autant que A.

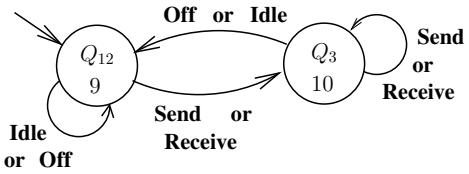
7.1.1 Quelques exemples de modèles de radios

Prenons quelques exemples d'automates qui modélisent la consommation d'une radio. Ces automates sont semblables à celui implémenté dans GLONEMO, figure 5.6 page 97, mais nous les avons simplifiés afin de faciliter cette présentation. Les entrées de ces modèles sont les signaux *Idle, Off, Receive* et *Send* qui proviennent tous du protocole MAC. Ces modèles de radio sont uniquement des modèles de consommation et ne produisent pas de signaux de sortie.

Sur les figures suivantes (7.1, 7.2 et 7.3), l'automate de droite est censé remplacer celui de gauche dans un modèle plus abstrait. Dans ces exemples, on cherche à remplacer deux états (S_1 et S_2) par un seul (Q_{12}). Nous illustrons grâce à ces exemples quels sont les besoins que l'on a pour notre relation d'abstraction.



(a) Modèle détaillé, R_1.



(b) Modèle abstrait, R_2.

entrées		*Idle*	*Off*	-	*Idle*	*Send*
énergie consommée par R_1	1	9	10	11	19	29
énergie consommée par R_2	9	18	27	36	45	55
conso$(R_1) \leq$ **conso**(R_2)	true	true	true	true	true	true

(c) Exemple de trace et consommation associée

FIG. 7.1 – Une abstraction brutale

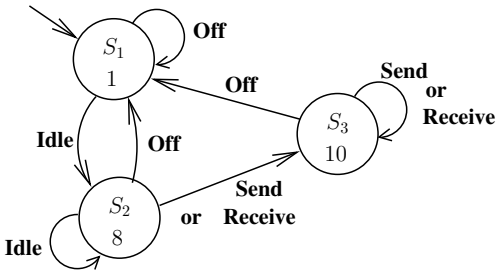
Pour la figure 7.1, nous avons pris l'exemple d'une trace (fig. 7.1(c)) pour laquelle la consommation calculée par l'automate R_2 est toujours supérieure à celle calculée par R_1. En fait, ici l'état Q_{12} consomme plus que les états S_1 et S_2. Donc pour toutes les traces d'entrées, la consommation de R_2 sera toujours supérieure à celle de R_1. D'après les critères que nous avons évoqués, R_2 est une abstraction valide de R_1. Cependant, définir la relation d'abstraction par $R_1 \preceq R_2 \Leftrightarrow \forall t, \text{conso}(R_1, t) \leq \text{conso}(R_2, t)$ est trop fort. En effet, il existe une trace t telle que l'automate R_1 (respectivement R_2) reste tout le temps dans l'état S_1 (respectivement Q_{12}), donc conso(S_1)$\leq$ conso(Q_{12}). De même, il existe une trace qui reste presque tout le temps (sauf l'instant initial) dans les états S_2 et Q_{12}, donc, conso(S_2)$\leq$ conso(Q_{12}). En fait, si la relation d'abstraction implique que la relation $\text{conso}(R_1, t) \leq \text{conso}(R_2, t)$ doit être vraie pour toute trace t, alors dès qu'un état du modèle abstrait remplace plusieurs états du modèle détaillé, cet état consomme plus que tous ceux qu'il remplace. Ici, $\text{conso}(Q_{12}) \geq \max\{\text{conso}(S_1), \text{conso}(S_2)\}$.

Une telle relation d'abstraction ne permet pas de prendre en compte l'utilisation du modèle. En replaçant les états S_1 et S_2 correspondant aux modes "éteint" (*Off*) et "écoute libre" (*Idle*[1]) par un seul état qui consomme au moins autant que l'état le plus cher (*Idle*), on ignore le fait qu'on passe peu de temps dans l'état *Idle*. Dans les réseaux de capteurs tout est fait pour éteindre la radio le plus souvent et

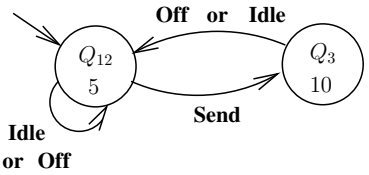
[1]Rappel : dans l'état *Idle*, noté ici S_2, la radio est allumée mais ne reçoit rien.

donc passer le moins de temps possible dans l'état *Idle*. On ne peut pas espérer prouver des propriétés de durée de vie intéressantes sans considérer que la radio est très souvent éteinte. Il faut prendre en compte le contexte dans lequel est utilisé le composant.

Nous voudrions accepter des abstractions plus fines pour lesquelles un état ne consomme pas forcément plus que les états qu'il remplace. Dans l'exemple suivant (figure 7.2), l'état Q_{12} qui remplace les états S_1 et S_2 consomme plus que S_1 mais moins que S_2. La propriété *"énergie consommée par S"* $\leq$ *"énergie consommée par Q"* dépend alors des entrées.



(a) Modèle détaillé, R_1.



(b) Modèle abstrait, R_2.

entrées (trace t)		*Idle*	*Off*	-	*Idle*	*Send*
énergie consommée par R_1	1	9	10	11	19	29
énergie consommée par R_2	5	10	15	20	25	35
conso(R_1) $\leq$ conso(R_2)	true	true	true	true	true	true

(c) Exemple de trace pour laquelle S consomme moins que Q

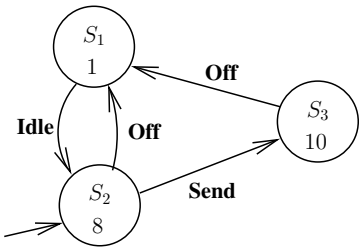
entrées (trace f)		*Idle*	*Idle*	-	*Idle*	*Send*
énergie consommée par R_1	1	9	17	25	33	43
énergie consommée par R_2	5	10	15	20	25	35
conso(R_1) $\leq$ conso(R_2)	true	true	false	false	false	false

(d) Exemple de trace pour laquelle S **ne** consomme **pas** moins que Q

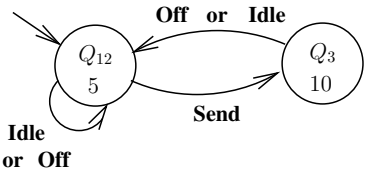
FIG. 7.2 – Abstraction dépendant des entrées.

Pour la trace f, à la fin de l'exécution, l'automate R_1 a consommé plus que R_2. Donc pour cette trace, on ne peut pas dire que R_2 est une abstraction de R_1. Cependant, cette trace n'est pas réaliste : la radio reste dans les états *Idle* et *Send*, elle est donc allumée pendant toute l'exécution. Pour une autre trace (t, tableau 7.2(c)), R_1 consomme moins que R_2. Et c'est ce type de trace où la radio passe peu de temps dans l'état *Idle*, qui est réaliste et donc qui nous intéresse. Notre formalisme doit permettre de considérer des relations d'abstractions qui ne sont valables que pour un ensemble de traces réalistes. Pour proposer une abstraction, il faut également indiquer pour quel ensemble de traces elle est valable. Dans la suite nous appellerons contexte cet ensemble puisqu'il s'agit du contexte dans lequel le composant est utilisé.

Remarquons également qu'il est parfois souhaitable d'accepter des traces où la relation "conso(S) $\leq$ conso(Q)" est momentanément fausse. Sur la figure 7.3, nous avons simplement modifié l'état initial : entre S_1 et S_2, c'est maintenant S_2, l'état le plus cher qui est l'état initial. Pour la trace u, la consommation de S est parfois supérieure à celle de Q, mais ce qui compte c'est qu'à la fin de l'exécution, on ait "conso(S, u) $\leq$ conso(Q, u)". En effet, on a envie que cette trace soit acceptée : elle n'envoie que des signaux *Off*, donc la radio est presque tout le temps éteinte, ce qui est réaliste pour les réseaux de capteurs. Notre formalisme doit permettre de définir des abstractions qui acceptent ce type



(a) Modèle détaillé, R_1.



(b) Modèle abstrait, R_2.

entrées (trace u)		*Off*	*Off*	-	*Off*	*Off*
énergie consommée par R_1	8	9	10	11	12	13
énergie consommée par R_2	5	10	15	20	25	30
conso(R_1) $\leq$ conso(R_2)	false	true	true	true	true	true

(c) Exemple de trace pour laquelle S consomme moins que Q

FIG. 7.3 – Pour la relation d'abstraction, il est souhaitable d'accepter des traces où la relation n'est vraie qu'à la fin de l'exécution.

de contexte.

On note donc $R_1 \preceq_K R_2$, pour "R_2 est une abstraction de R_1 dans le contexte K". Pour valider cette abstraction dans le modèle global il faut que les sorties des autres composants satisfassent le contexte K. Ici, le contexte dépend du protocole MAC. Nous voulons pouvoir écrire $R_1 \parallel \text{MAC} \preceq_{K'} R_2 \parallel \text{MAC}$ où K' est le nouveau contexte qui porte entre autres sur les entrées du protocole MAC.

7.1.2 Un modèle de protocole MAC

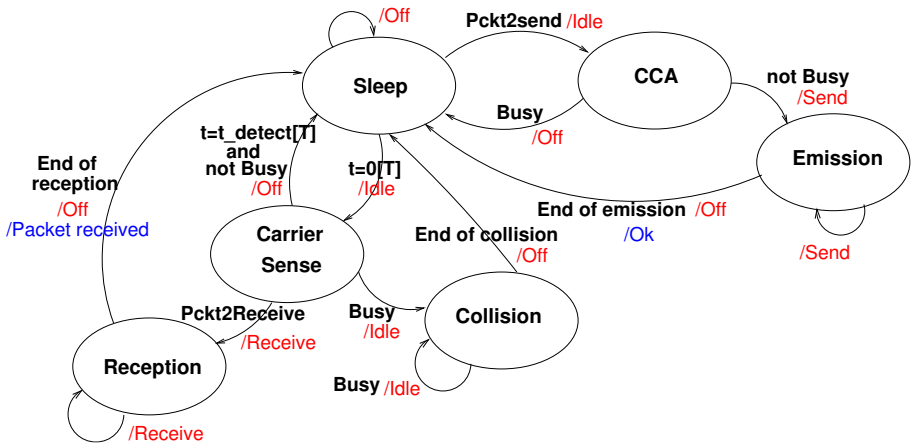


FIG. 7.4 – Protocole MAC

Prenons comme exemple le protocole MAC à échantillonnage de préambule implémenté dans GLONEMO. Ici pour simplifier l'exemple (et le nombre d'entrées) nous nous sommes affranchis des mécanismes d'attente aléatoire et d'acquittement implémentés dans GLONEMO (voir figure 5.4 page 95). Le protocole MAC de notre exemple est décrit figure 7.4. Les sorties en /rouge sur l'automate,

sont les entrées qui pilotent les modèles de radio. Celles en /bleu sont les signaux du protocole MAC destinés au protocole de routage. *Ok* indique au protocole de routage du nœud que le paquet a bien été émis. *Packet received* est un signal qui annonce qu'un paquet vient d'être reçu.

Les entrées du protocole MAC proviennent du protocole de routage et de l'environnement. Le protocole de routage d'un nœud envoie au protocole MAC les messages à émettre. Le signal *Pckt2send* provient donc du routage. Les autres entrées viennent du canal radio que nous avons appelé environnement, il s'agit a priori des nœuds voisins. Le signal *Busy* signifie que le canal radio est occupé. Le signal *Reception* indique un paquet à recevoir.

Le protocole MAC à échantillonnage de préambule sonde le canal périodiquement toutes les T unités de temps. Certaines transitions sont donc conditionnées par le temps. Dans notre formalisme synchrone, le temps est discrétisé : à chaque instant, les processus réagissent. Si un instant correspond à 0.01 seconde, pour modéliser un système qui reste 0.02 secondes dans un même état Q, on crée deux états Q_1 et Q_2 avec une transition de Q_1 à Q_2. Ici, les transitions conditionnées par le temps sont une façon implicite de représenter une succession d'états similaires.

Le formalisme que nous proposons doit fournir un cadre pour faire le produit de composants, il doit offrir un moyen de connecter les sorties du protocole MAC aux entrées des modèles de radio. Puisqu'il s'agit d'automates synchrones, c'est un produit synchrone que nous proposons.

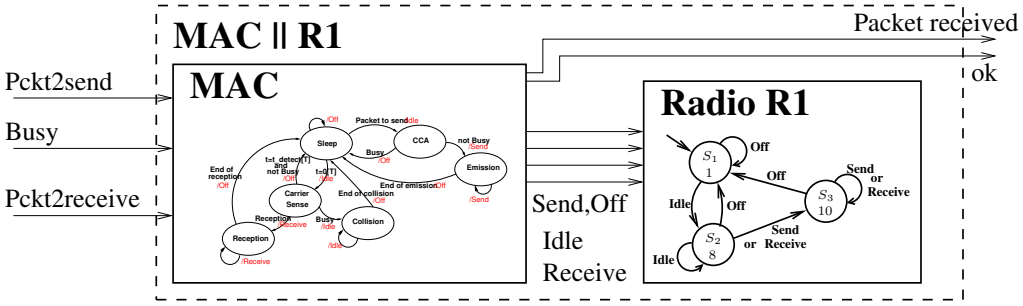


FIG. 7.5 – Schéma de composition du protocole MAC et d'un modèle de radio. On connecte les sorties du MAC au entrées de R_1.

Plus précisément, la relation de composition que nous considérons est la connexion flot-de-donnée des composants. La figure 7.5 décrit la composition MAC $\parallel R_1$, du protocole MAC et du modèle R_1 de la radio. Cette relation de composition consiste à connecter les signaux de sortie d'un composant aux entrées correspondantes (qui ont le même nom) de l'autre composant, et réciproquement. Ici, les sorties *Off*, *Idle*, *Send* et *Receive* du composant MAC sont des entrées du composant R_1, on les connecte. Le composant radio n'a pas de sorties, donc a fortiori pas de sorties vers le composant MAC. Les entrées des deux composants qui ne sont pas des sorties de l'autre restent des entrées pour le composant produit. De même, les sorties des deux composants qui ne sont pas des entrées de l'autre restent des sorties pour le composant produit. Ici, *Pckt2send*, *Busy* et *Pckt2receive* sont des entrées qui ne sont pas connectées par le composant radio, ce sont donc des entrées du composant-produit. De même, *ok* est une sortie du MAC qui n'est pas destinée à la radio, c'est donc une sortie du composant MAC $\parallel R_1$.

Pour les consommations, nous avons vu section 4.2 que pour deux composants qui consomment sur la même batterie, la somme des consommations convient. Ici cependant, le modèle du protocole MAC (figure 7.4) ne fait que piloter les modèles de radio, il ne modélise pas lui-même de consommation. La consommation du produit sera simplement la consommation de la radio.

Enfin, pour propager l'abstraction, il faut s'assurer que les sorties produites par le protocole MAC

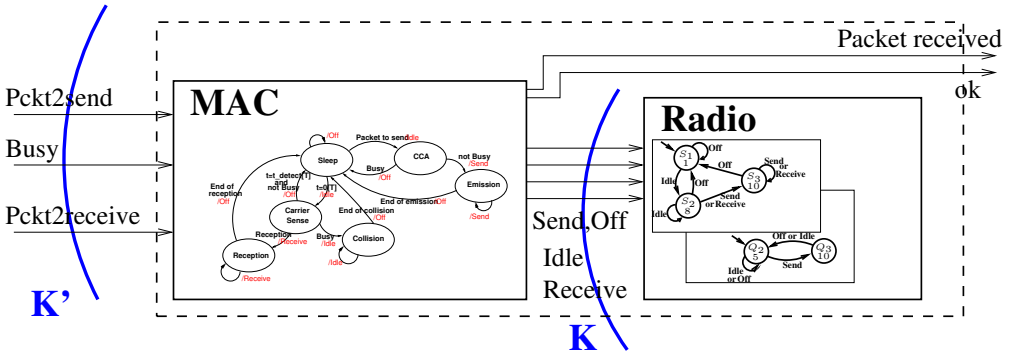


FIG. 7.6 – Nous savons comparer les modèles de radio R_1 et R_2 dans un contexte K : $R_1 \preceq_K R_2$. Pour comparer les composants-produit, un nouveau contexte K' peut intervenir.

satisfont K, le contexte de l'abstraction. Les sorties d'un composant dépendent bien sûr de ses entrées. Pour notre exemple, le fait que le protocole MAC n'envoie pas trop de *Idle* à la radio dépend du protocole mais aussi des entrées qu'il reçoit. On se doute par exemple que si le protocole MAC reçoit de nombreux signaux *Busy*, il produira plus de signaux *Idle*. Il peut donc exister un contexte K' tel que les sorties du protocole MAC satisfassent le contexte K, dans ce cas on veut pouvoir écrire : MAC $\parallel R_1 \preceq_{K'}$ MAC $\parallel R_2$. C'est ce que nous avons représenté figure 7.6. Le nouveau contexte K' porte sur les entrées des composants-produit, ici sur les signaux *Pckt2send*, *Busy* et *Pckt2receive*.

7.2 Formalisation de la notion d'abstraction

Dans cette section, nous définissons formellement notre notion d'abstraction de modèles à coût. Nous commençons par des définitions générales qui nous permettent de définir les machines de Mealy. Nous enrichissons les machines de Mealy booléennes de coûts. Dans notre cas, les coûts sont des consommations, mais rien n'interdit qu'ils modélisent d'autres coûts, on peut donc imaginer que ce cadre formel soit utile (et utilisé) dans d'autres contextes. Puis nous définissons le produit de telles machines de Mealy. Enfin, nous exprimons notre relation d'abstraction.

7.2.1 Définitions générales et notations

Soit $\mathcal{A}$ l'ensemble des signaux. Les signaux sont des booléens.

Pour un sous-ensemble I de $\mathcal{A}$, on définit $\mathcal{U}(I)$, l'ensemble des monômes sur I ; $\mathcal{U}^*(I)$ l'ensemble des monômes satisfiables sur I ; $\mathcal{U}_c^*(I)$ l'ensemble des monômes complets et satisfiables sur I :

$\mathcal{U}(I) = 2^I \times 2^I$

$\mathcal{U}^*(I) = \{(m^+, m^-) \in \mathcal{U}(I), m^+ \cap m^- = \emptyset\}$

$\mathcal{U}_c^*(I) = \{(m^+, m^-) \in \mathcal{U}(I), m^+ \cap m^- = \emptyset, m^+ \cup m^- = I\}$

Un monôme $m = (m^+, m^-)$ sur I peut s'écrire $x_1.x_2.\cdots.x_m.\overline{y_1}.\overline{y_2}.\cdots.\overline{y_p}$ où $m^+ = \{x_1, x_2, \cdots, x_m\}$ et $m^- = \{y_1, y_2, \cdots, y_m\}$. Un monôme de $\mathcal{U}_c^*(I)$ est une fonction totale de I vers $\mathbb{B}$.

Définition 1 — Trace

Une trace t sur I est une séquence arbitrairement longue de monômes satisfiables et complets, $t \in (\mathcal{U}_c^*(I))^*$. $\mathcal{T}(I)$ dénote l'ensemble des traces possibles sur les variables I.

Définition 2 — Ensemble de traces préfixe-clos

Un ensemble de trace V est clos par préfixe si et seulement si pour toutes les traces t de V, tous les préfixes de t sont aussi dans V.

7.2.2 Modèles à coûts

Nous définissons maintenant les machines de Mealy booléennes à coût, cette définition est simplement une extension des machines de Mealy booléennes [60] auxquelles on a ajouté des coûts sur les états.

Définition 3 — Machine de Mealy booléenne à coût

Une machine de Mealy booléenne à coûts est un n-uplet $M = (S, s_0, I, O, T, \mathcal{L})$ où $I \subseteq \mathcal{A}$, $O \subseteq \mathcal{A}$ sont les ensembles de signaux d'entrées et de sorties ; S est l'ensemble des états ; s_0 est l'état initial ; $T \subseteq S \times \mathcal{B}(I) \times 2^O \times S$ est l'ensemble des transitions ; $\mathcal{L} : Q \longrightarrow \mathcal{D}$ est la fonction de coût qui associe un coût à chaque état. $\mathcal{D}$ est l'algèbre des coûts, par exemple $\mathcal{D}$ peut être $\mathbb{R}$ ou $\mathbb{N}$. Nous notons $\mathcal{M}$, l'ensemble des machines de Mealy booléennes à coût.

$\mathcal{B}(I)$ dénote l'ensemble des formules booléennes à variables dans I. Sans perte de généralité, on peut toujours considérer que les formules booléennes qui décrivent les conditions d'activation des transitions sont réduites à des monômes. Par exemple si l'ensemble des signaux d'entrées est $\{a, b\}$, alors la condition a signifie $a \wedge b \vee a \wedge \overline{b}$, et une transition étiquetée par a/o (une telle transition produit o si a est présent) peut être décomposée en deux transitions (entre les mêmes états) étiquetées par $a \wedge b/o$ et $a \wedge \overline{b}/o$.

2^O dénote l'ensemble des parties de O.

Définition 4 — Run(M, e)

Pour $M = (S, s_0, I, O, T, \mathcal{L}) \in \mathcal{M}$ et $e \in T(I)$, e est une séquence d'entrées de longueur n ($|e| = n$), $e = ((e_i^+, e_i^-))_{0 \leq i < n}$. Nous définissons Run$(M, e)$=$(u_i)_{0 \leq i \leq n}$, la séquence d'états de M telle que $u_0 = s_0$ et $\forall 0 \leq i \leq n$, $\exists o_i^+ \in 2^O$, tel que $(u_i, e_i^+.\overline{e_i^-}, o_i^+, u_{i+1}) \in T$.

Run(M, e) est la séquence des états de M lorsque M est activé par la séquence d'entrées e.

Définition 5 — Cost(M, e)

Pour $M = (S, s_0, I, O, T, \mathcal{L}) \in \mathcal{M}$ et $e \in T(I)$, e est une séquence d'entrées de longueur n ($|e| = n$), $e = ((e_i^+, e_i^-))_{0 \leq i < n}$. Soit $(u_i)_{0 \leq i \leq n}$ = Run(M, e), Nous définissons Cost(M, e) = $\sum_{i=0}^{n} \mathcal{L}(u_i)$.

Cost(M, e) représente le coût de l'exécution de la machine M lorsqu'elle est activée par la séquence d'entrées e.

Définition 6 — Out(M, e)

Pour $M = (S, s_0, I, O, T, \mathcal{L}) \in \mathcal{M}$ et $e \in T(I)$, e est une séquence d'entrées de longueur n ($|e| = n$), $e = ((e_i^+, e_i^-))_{0 \leq i < n}$. Nous définissons Out$(M, e)$= $(o_i^+)_{0 \leq i < n}$. Où $o_i^+ \in 2^O$ et $\forall 0 \leq i < n, (s_i, e_i^+.\overline{e_i^-}, o_i^+, s_{i+1}) \in T$.

Out(M, e) est la suite d'ensembles de sorties produites par la machine M lorsqu'elle est activée par la séquence d'entrées e.

Nous définissons maintenant les opérations de compositions de nos machines. Il s'agit du produit synchrone qui a été introduit au chapitre 4 (section 4.1.4, page 68).

Définition 7 — Produit synchrone de machines de Mealy booléennes à coûts

$\times : \mathcal{M} \times \mathcal{M} \longrightarrow \mathcal{M}$
$(S_1, s_{01}, I_1, O_1, T_1, \mathfrak{L}_1) \times (S_2, s_{02}, I_2, O_2, T_2, \mathfrak{L}_2) = (S_1 \times S_2, (s_{01}, s_{02}), I_1 \cup I_2, O_1 \cup O_2, T', \mathfrak{L}')$
Où, T' est défini par :
$\big((s_1, m_1, o_1, s_1') \in T_1 \wedge (s_2, m_2, o_2, s_2') \in T_2\big) \iff \big(((s_1, s_2), m_1 \wedge m_2, o_1 \cup o_2, (s_1', s_2')) \in T'\big)$
et $\mathfrak{L}'$ est défini par :
$\mathfrak{L}'(s_1, s_2) = (\mathfrak{L}_1(s_1) + \mathfrak{L}_2(s_2))$

Remarque : la fonction de coût $\mathfrak{L}'$ du produit est la somme des fonctions de coût. Cette fonction convient bien pour modéliser le produit de composants qui consomment sur une même batterie. Une autre fonction de combinaison peut parfois être souhaitable. Par soucis de simplicité, nous avons choisi de présenter notre formalisme avec cette fonction de combinaison qui est naturelle. Nous expliquons, section 7.4.1, comment notre formalisme peut facilement considérer d'autres fonction de combinaison.

Comme nous l'avons dit au chapitre 4, le produit synchrone préserve la réactivité et le déterminisme. Notre produit est exactement un produit synchrone de machines de Mealy dans lequel nous prenant en compte les coûts en introduisant une fonction qui permet de composer les coûts. Il préserve donc la réactivité et le déterminisme. Pour définir un langage de programmation déterministe et réactif, il faut que les opérations de composition préservent ces propriétés. C'est donc important pour nous que ce soit le cas ici.

Nous définissons maintenant l'encapsulation. L'opération d'encapsulation est paramétrée par un ensemble de signaux. Intuitivement, il s'agit d'éliminer les transitions non-cohérentes lorsque l'on sait que l'ensemble de ces signaux ne peuvent provenir d'autres composants. Nous avons dessinés deux exemples d'encapsulations, figures 4.12 et 4.13 page 69, où l'ensemble des signaux était $\{a, b\}$.

Définition 8 — Encapsulation

$\setminus : \mathcal{M} \times 2^{\mathcal{A}} \longrightarrow \mathcal{M}$
$(S, s_0, I, O, T, \mathfrak{L}) \setminus \Gamma = (S, s_0, I \setminus \Gamma, O \setminus \Gamma, T', \mathfrak{L})$

Où T' est défini par :
$(s, m, o, s') \in T \wedge m^+ \cap \Gamma \subseteq o \wedge m^- \cap \Gamma \cap o = \emptyset \iff (s, \exists \Gamma.m, o \setminus \Gamma, s') \in T'$

m^+ dénote l'ensemble des variables qui apparaissent positives dans le monôme m (i.e. $m^+ = \{x \in \mathcal{A} \mid (x \wedge m) = m\}$). m^- dénote l'ensemble des variables qui apparaissent négatives dans le monôme m (i.e. $m^- = \{x \in \mathcal{A} \mid (\overline{x} \wedge m) = m\}$).

L'opération d'encapsulation ne préserve pas le déterminisme ni la réactivité. C'est exactement le problème de causalité dont nous parlions au chapitre 4, section 4.1.4, page 68. La figure 4.12 est un exemple où l'encapsulation crée de la non-réactivité et la figure 4.13, un exemple où l'encapsulation crée du non-déterminisme. Nous avons vu, page 69, que ce problème provient des dépendances cycliques instantanées. Afin de ne pas construire de modèles incohérents, nous ferons conjointement les opérations produit et encapsulation, à ce moment-là, il sera possible de déclarer non valide une composition qui crée une dépendance cyclique instantanée. Nous définissons maintenant formellement ce que signifie une dépendance instantanée de variables pour une machine de Mealy booléenne. Nous pourrons alors définir dans quel cas une machine ne contient pas de dépendance cyclique instantanée.

Définition 9 — Dépendance instantanée de données

Pour $M = (S, s_0, I, O, T, \mathcal{L}) \in \mathcal{M}$*,* $o \in O$ *et* $i \in I$*,* o *dépend de* i *dans* M *si et seulement si* $\exists s, s_1, s_2 \in S, \exists t_1, t_2 \in T$ *tel que* $t_1 = (s, (m_1^+, m_1^-), O_1, s_1)$*,* $t_2 = (s, (m_2^+, m_2^-), O_2, s_2)$ *avec* $o \in O_1$*,* $o \notin O_2$ *et*
$\big(i \in m_1^+$ *et* $(m_1^+ \setminus i, m_1^- \cup i) = (m_2^+, m_2^-)$*,*
ou $i \notin m_1^+$ *et* $(m_1^+ \cup i, m_1^- \setminus i) = (m_2^+, m_2^-)\big)$.

Intuitivement, la sortie o dépend de l'entrée i dans une machine M, si le statut de o peut dépendre de la présence ou de l'absence de i. Ici, avec les notations de la définition, dans l'état s deux transitions t_1 et t_2 sont possibles, t_1 produit o, pas t_2. Au niveau des conditions de ces transitions seul le statut de i change, pour l'une i doit être présent, pour l'autre i doit être absent.

Définition 10 — Pas de dépendances cycliques instantanées

Soient $M_1 = (S_1, s_{01}, I_1, O_1, T_1, \mathcal{L}_1)$ *et* $M_2 = (S_2, s_{02}, I_2, O_2, T_2, \mathcal{L}_2)$ *deux machines de* $\mathcal{M}$*. On dit qu'il n'y a pas de dépendances cycliques instantanées si et seulement si pour tout* $v \in O_1 \cap I_2$*, et pour tout* $u \in O_2 \cap I_1$*, soit* v *ne dépend pas instantanément de* u *dans* M_1*, soit* u *ne dépend pas instantanément de* v *dans* M_2*.*

7.2.3 Composants à coûts

Le problème de causalité apparaît lorsque la composition de composants crée des dépendances cycliques instantannées. Pour déceler les dépendances cycliques instantannées, nous allons faire les opérations produit synchrone et encapsulation en même temps. En effet, de cette façon, le produit correspond à une connexion flot-de-données des composants, et la connexion de composants n'est autorisée que si elle ne crée aucune dépendance cyclique instantannée. Si les opérations produit synchrone et encapsulation sont effectuées séparément, il est plus difficile de déceler les dépendances cycliques instantanées : elles apparaissent lors du produit mais c'est après encapsulation qu'apparaît le non-déterminisme ou la non-réactivité. Nous introduisons donc des composants qui contiennent les machines de Mealy booléennes à coûts introduites précédemment et dont la composition sera une connexion flot-de-données correspondant aux opérations produit synchrone et encapsulation effectuées en même temps.

Les composants que nous présentons contiennent une notion de contrat *Assume/Guarantee*. Intuitivement, les contrats permettent de garantir certaines propriétés (G) sur les sorties dans le cas où les entrées satisfont certaines conditions (A). Les contrats sont utiles pour la programmation par composants [59, 64] parce qu'il fournissent une spécification haut-niveau qui permet d'adopter une méthode progressive de développement. Dans notre contexte d'abstraction modulaire, les contrats peuvent avoir un intérêt supplémentaire : ils peuvent aider à propager les contraintes pour que l'abstraction reste vraie après composition. Autrement dit, voir figure 7.6, dans certains cas ils peuvent aider à trouver K'.

Définition 11 — Composant

Un composant C *est un n-uplet* (I, O, M, A, G) *où :*
- I *et* O *sont les ensembles des signaux d'entrées et de sorties du composant ;*
- $M \in \mathcal{M}$ *est une machine de Mealy booléenne à coût.*
- A *est un ensemble préfixe-clos de traces sur* I
- G *est un ensemble préfixe-clos de traces sur* $O \cup I$

Soit e une trace sur I, si e respecte A alors la trace constituée des entrées e et des sorties de M (Out(M, e)) respecte la garantie G. G est un ensemble de traces sur l'ensemble des sorties et entrées ($O \cup I$) ce qui permet de parler facilement de contraintes entre les entrées et les sorties.

Définition 12 — Extension des définitions précédentes aux composants

Soit $C = (I, O, M, A, G)$ *et* $e \in T(I)$
Run(C*,* e*) = Run(*M*,* e*)*
Out(C*,* e*) = Out(*M*,* e*)*
Cost(C*,* e*) = Cost(*M*,* e*)*

Nous définissons maintenant la connexion flot-de-données de composants. Il s'agit du produit synchrone et de l'encapsulation de machines de Mealy. L'encapsulation est l'opération qui exprime que certains signaux ne peuvent provenir d'autres composants. Ici, puisque l'on fait une connexion flot-de-données, tous les signaux qui sont à la fois des sorties de M_1 et des entrées de M_2 (et réciproquement) ne peuvent provenir d'autres composants. Sur la figure 7.7, il s'agit des signaux $\{a, d, e\}$. Plus généralement, l'encapsulation est faite avec l'ensemble des signaux de $(O_2 \cap I_1) \cup (O_1 \cap I_2)$.

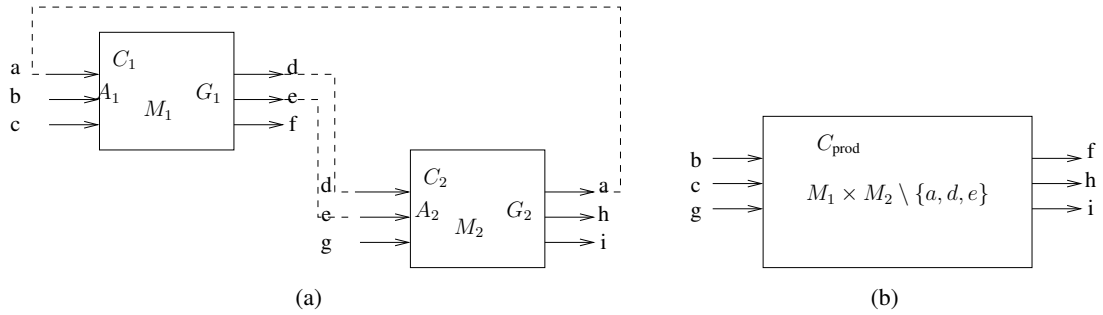


FIG. 7.7 – Connexion flot-de-données de deux composants

Définition 13 — Connexion flot-de-données de composants

Soient $C_1 = (I_1, O_1, M_1, A_1, G_1)$ *et* $C_2 = (I_2, O_2, M_2, A_2, G_2)$ *deux composants.*
La composition de C_1 *et* C_2 *est autorisée, si et seulement s'il n'y a pas de cycles de dépendances instantanées entre les machines* M_1 *et* M_2*.*
Dans ce cas, un composant produit $C = (I, O, M, A, G)$*,* $C = C_1 \parallel C_2$*, est tel que*
– $I = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$
– $O = (O_1 \cup O_2) \setminus (I_1 \cup I_2)$
– $M = (M_1 \times M_2) \setminus ((O_2 \cap I_1) \cup (O_1 \cap I_2))$
– A *est un ensemble de traces préfixe-clos sur* I
– G *est un ensemble de traces préfixe-clos sur* $O \cup I$

Remarque : on dit *un* composant produit parce que plusieurs contrats A/G peuvent être valides pour un même composant. Proposer le meilleur contrat n'est pas forcément une chose facile. La connexion flot-de-données que nous venons de définir consiste seulement à brancher les fils de M_1 et M_2 qui ont les mêmes noms entre eux, comme illustré figure 7.7.

7.2.4 Abstraction

Nous avons maintenant tous les éléments qui nous permettent de définir notre notion d'abstraction. Nous définissons cette notion dans le cas où les deux composants que l'on compare ont les mêmes ensembles d'entrées et de sorties, $I_1 = I_2$ et $O_1 = O_2$. Dans la pratique, les machines que l'on compare n'ont pas forcément les mêmes ensembles d'entrées et de sorties. Il se peut que le modèle abstrait ait

moins d'entrées si par exemple il ne réagit pas à une des entrées du modèle détaillé. Cependant, ça n'est pas une restriction de ne définir l'abstraction que dans ce cas parce que l'on peut toujours se ramener au cas où les deux machines ont les mêmes ensembles d'entrées et de sorties.

En effet, à partir d'une machine définie sur l'ensemble d'entrées I, on peut facilement définir une machine sur $I \cup \{a\}$. Pour ce faire, on transforme chaque transition $(s_1, m, o, s_2) \in T$ en deux transitions $(s_1, m \wedge a, o, s_2)$ et $(s_1, m \wedge \overline{a}, o, s_2)$.

Étendre une machine de $\mathcal{M}$ à un ensemble de sorties plus grand est encore plus simple : si $M = (S, s_0, I, O, T, \mathcal{L})$ ne produit pas b, alors la machine $M = (S, s_0, I, O \cup \{b\}, T, \mathcal{L})$ où l'ensemble des sorties a été étendu ne produit toujours pas b.

Notons $I_{12} = I_1 = I_2$ et $O_{12} = O_1 = O_2$.

Comme nous l'avons montré sur l'exemple, notre définition d'abstraction doit pouvoir prendre en compte un contexte pour lequel l'énergie dépensée par une machine est toujours inférieure à celui dépensée par l'autre. Cette relation n'est pas forcément vraie pour toutes les traces d'entrées. De plus, ce contexte n'est pas forcément préfixe-clos parce que l'on veut autoriser des modèles abstraits qui consomment moins au début mais qui au bout d'un certain temps consomment forcément plus que le modèle détaillé. Dans la définition suivante, ce contexte est noté K.

Définition 14 — Relation d'abstraction avec contexte.

Soit $K \subseteq T(I_{12})$ un ensemble de traces finies sur les signaux I_{12}, et $C_i = (I_{12}, O_{12}, M_i, A_i, G_i)$, $i \in \{1, 2\}$, deux composants. C_2 est une abstraction de C_1 dans le contexte K, notée $C_1 \preceq_K C_2$, si et seulement si,

$$\forall e \in K, \begin{cases} \mathit{Cost}(C_1, e) \leq \mathit{Cost}(C_2, e) \\ \mathit{et} \\ \mathit{Out}(C_1, e) = \mathit{Out}(C_2, e) \end{cases}$$

Nous insistons sur le fait que K n'est pas nécessairement préfixe-clos.

Théorème 1 *Congruence de la relation d'abstraction avec la connexion de composants*

Soient $C_i = (I_{12}, O_{12}, M_i, A_i, G_i)$, $i = 1, 2$ deux composants et K un contexte sur I_{12} tels que $C_1 \preceq_K C_2$. Soit $C = (I, O, M, A, G)$ un troisième composant tel que les compositions de C_1 et C et de C_2 et C sont autorisées (pas de dépendances cycliques instantanées).

Pour tout ensemble de traces K' sur $(I \setminus O_{12}) \cup (I_{12} \setminus O)$ tel que $K' \Longrightarrow K$, c'est-à-dire les composants C_1 et C_2 sont sollicités dans $C \parallel C_1$ et $C \parallel C_2$ par des traces qui satisfont K, on a :

$$C_1 \preceq_K C_2 \Longrightarrow C_1 \parallel C \preceq_{K'} C_2 \parallel C$$

Preuve : pour prouver le théorème 1 il nous faut prouver pour toutes traces de K', la propriété fonctionnelle et la relation concernant les coûts, soit :

$$\forall e \in K', \begin{cases} \text{Out}(C \parallel C_1, e) = \text{Out}(C \parallel C_2, e) \\ \text{and} \\ \text{Cost}(C \parallel C_1, e) \leq \text{Cost}(C \parallel C_2, e) \end{cases}$$

Nous prouvons d'abord la propriété fonctionnelle (1) puis la relation concernant les coûts (2) :

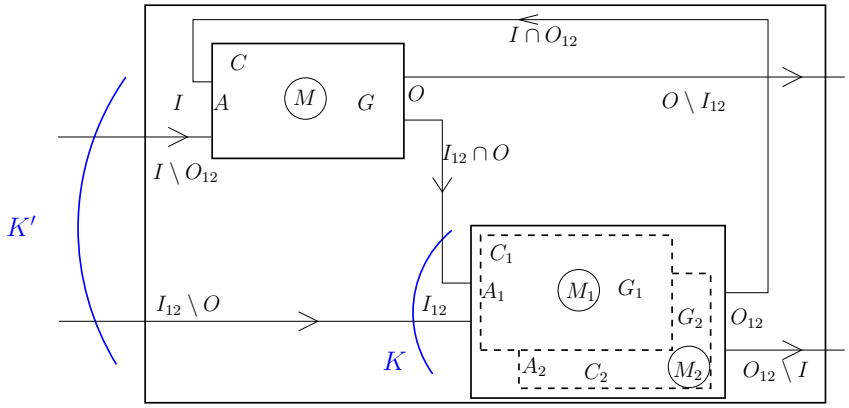


FIG. 7.8 – Connexion de composants avec contextes pour une relation d'abstraction

1. Par hypothèse, il n'y a pas de cycle instantané de données dans les composants $C \parallel C_1$ et $C \parallel C_2$. Donc les machines de Mealy $M \times M_1 \setminus ((O \cap I_{12}) \cup (O_{12} \cap I))$ et $M \times M_2 \setminus ((O \cap I_{12}) \cup (O_{12} \cap I))$ sont déterministes. Comme $K' \Longrightarrow K$, C_1 et C_2 sont sollicités par des entrées qui satisfont K. Donc comme $C_1 \preceq_K C_2$, C_1 et C_2 produisent les mêmes sorties. (En effet, $C_1 \preceq_K C_2 \Longrightarrow \forall e \in K$, Out$(C_1, e) =$ Out(C_2, e))

2. Prouvons maintenant la propriété sur les coûts : $\forall e \in K_P$, Cost$(C \parallel C_1, e) \leq$ Cost$(C \parallel C_2, e)$. Les états de la machine de Mealy $M \times M_1$ sont le produit cartésien d'un état de M et d'un état de M_1. La machine de Mealy $M \times M_1 \setminus ((O \cap I_{12}) \cup (O_{12} \cap I))$ a les mêmes états que $M \times M_1$ (on a seulement enlevé certaines transitions, voir la définition de l'encapsulation, section 7.2.2). Donc un état q^1 de $M \times M_1 \setminus ((O \cap I_{12}) \cup (O_{12} \cap I))$ peut s'écrire $q^1 = (s, s^1)$ où s est un état de M ($s \in S$) et s^1 un état de M_1 ($s^1 \in S_1$). De même, un état q^2 de $M \times M_2 \setminus ((O \cap I_{12}) \cup (O_{12} \cap I))$ peut s'écrire $q^2 = (s', s^2)$ où $s' \in S$ et $s^2 \in S_2$.

   Soit e une trace de K' et $(q_i^1)_{0 \leq i \leq n} =Run(C \parallel C_1, e)$.
   Il est possible d'exprimer le coût de cette exécution :

$$\begin{aligned} \text{Cost}(C \parallel C_1, e) &= \Sigma_{i=0}^{n}(\mathcal{L}_P(q_i^1) \\ &= \Sigma_{i=0}^{n}(\mathcal{L}(s_i) + \mathcal{L}_1(s_i^1)) \\ &= \Sigma_{i=0}^{n}(\mathcal{L}(s_i)) + \Sigma_{i=0}^{n}(\mathcal{L}_1(s_i^1)) \end{aligned}$$

   En effet, lorsque l'on compose deux machines, on somme les coûts.

   De même, Cost$(C \parallel C_2, e) = \Sigma_{i=0}^{n}(\mathcal{L}(s_i)) + \Sigma_{i=0}^{n}(\mathcal{L}_2(s_i^2))$.

   M reçoit les mêmes entrées dans $C \parallel C_1$ ou $C \parallel C_2$. Comme la machine M est déterministe, $\forall 0 \leq i \leq n, s'_i = s_i$, donc, $\Sigma_{i=0}^{n}(\mathcal{L}(s_i)) = \Sigma_{i=0}^{n}(\mathcal{L}(s'_i))$.

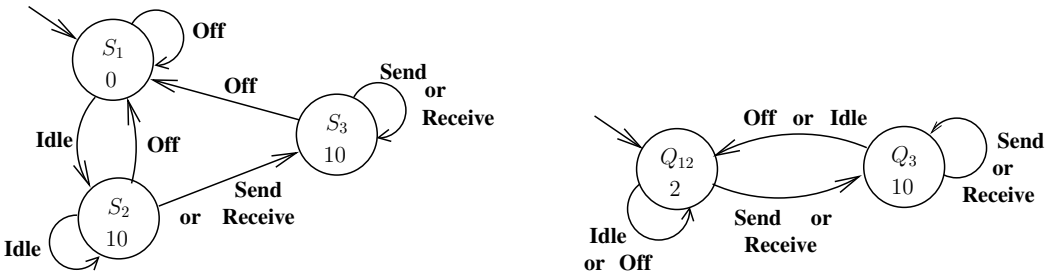
   Les machines M_1 et M_2 sont également activées avec les mêmes entrées ($\forall e \in K'$), de plus ces entrées satisfont le contexte K. Donc $\Sigma_{i=0}^{n}(\mathcal{L}_1(s_i^1)) \leq \Sigma_{i=0}^{n}(\mathcal{L}_2(s_i^2))$.

   Nous pouvons donc conclure que $\Sigma_{i=0}^{n}(\mathcal{L}(s_i)) + \Sigma_{i=0}^{n}(\mathcal{L}_1(s_i^1)) \leq \Sigma_{i=0}^{n}(\mathcal{L}(s_i)) + \Sigma_{i=0}^{n}(\mathcal{L}_2(s_i^2))$. Et enfin, Cost$(C \parallel C_1, e) \leq$ Cost$(C \parallel C_2, e)$.

7.3 Retour sur l'exemple

Nous montrons comment notre formalisme s'applique à l'étude de l'exemple de la section 7.1.

7.3.1 Les modèles de radio



(a) Modèle détaillé, R_1.

(b) Modèle abstrait, R_2.

FIG. 7.9 – Les deux modèles de radio. $R_1 \preceq_K R_2$

Le modèle de radio R_2 représenté sur la figure 7.9(b) est la machine de Mealy booléenne à coût $M_2 = (S, s_0, I, O, T, \mathcal{L})$ où :

- S est l'ensemble des états, $S = \{Q_{12}, Q_3\}$
- L'état initial s_0 est Q_{12}
- L'ensemble des entrées I est {*Idle, Off, Send, Receive* }
- L'ensemble des sorties est vide, $O = \emptyset$
- L'ensemble des transitions T est constitué de trois éléments. Exprimons, par exemple, celle de Q_{12} à Q_3 : $(Q_{12}, \mathit{Send} \vee \mathit{Receive}, \emptyset, Q_3)$
- $\mathcal{L} : S \rightarrow \mathbb{N} = (Q_{12} \mapsto 5, Q_3 \mapsto 10)$

Il est alors possible de créer un composant C_2 à partir de M_2. $C_2 = (I, O, M_2, A, G)$. Pour le contrat A/G, comme souvent, plusieurs contrats sont possibles.

Le contrat le plus faible est celui qui n'impose aucune contrainte. Ce qui revient à ne pas utiliser de contrat. Ici, un contrat n'est pas indispensable parce que M_2 est uniquement un modèle de radio qui ne produit aucune sorties, nous n'avons donc rien besoin de garantir concernant les sorties. Le contrat $A/G = \emptyset/\mathcal{T}(I \cup O)$ est le plus faible possible.

Pour éviter des traces d'entrées ambigües nous aurions pu utiliser un contrat pour interdire les traces qui fournissent à la fois des entrées contradictoires comme par exemple, *Off* et *Receive*. Dans ce cas, le contrat aurait été $A/G = \{t \in \mathcal{T}(I)$tels que, à chaque instant la propriété suivante est vraie : not$\big((\mathit{Off} \wedge \mathit{Receive}) \vee (\mathit{Off} \wedge \mathit{Send}) \vee (\mathit{Idle} \wedge \mathit{Receive}) \vee (\mathit{Idle} \wedge \mathit{Send})\big)\}/\mathcal{T}(I \cup O)$. Ce contrat pourrait servir lors de la connexion pour éviter de brancher ce composant à un protocole MAC qui produirait, par exemple, en même temps *Idle* et *Receive*. Cependant, il n'y a bien sûr pas de contraintes sur les sorties puisqu'il n'y a pas de sorties.

De la même façon, le modèle de radio R_1 de la figure 7.9(a) peut être décrit comme un composant C_1.

Nous souhaitons pouvoir écrire que C_2 est une abstraction de C_1. Comme nous l'avons souligné section 7.1, cette relation n'est vraie que dans un certain contexte. Pour cet exemple simple, le contexte K dans lequel est vrai l'abstraction est simplement une relation arithmétique entre les entrées. Ici la relation qui définit le contexte K est :

$$10 \times \text{Nb } \textit{Idle} + 0 \times \text{Nb } \textit{Off} \leq 2 \times (\text{Nb } \textit{Idle} + \text{Nb } \textit{Off})$$

On a bien,

$$\forall e \in K, \begin{cases} \text{Cost}(C_1, e) \leq \text{Cost}(C_2, e) \\ \text{et} \\ \text{Out}(C_1, e) = \text{Out}(C_2, e) \end{cases}$$

La deuxième partie concernant les sorties est évidente ici puisque les deux composants ne produisent pas de sorties, a fortiori elles sont égales.

K n'est pas préfixe-clos puisque la relation qui le définit dépend du nombre d'occurrences de chacun des signaux reçus. Cette relation peut être fausse puis vraie puis à nouveau fausse au cours du temps suivant les entrées reçues.

Nous cherchons maintenant à propager cette abstraction afin de comparer le système constitué d'un modèle (R_1 ou R_2) et du protocole MAC.

7.3.2 Composition MAC et Radio

Le modèle du protocole MAC de la figure 7.4 peut également être vu comme un composant C. Pour ce composant, on peut proposer un contrat qui garantit que les sorties produites ne contiennent pas à la fois de sorties contradictoires. $A = \mathcal{T}(I_{MAC})$ et $G = \{t \in \mathcal{T}(O_{MAC} \cup I_{MAC})$tels que, à chaque instant la propriété suivante est vraie : not$\big((\textit{Off} \wedge \textit{Receive}) \vee (\textit{Off} \wedge \textit{Send}) \vee (\textit{Idle} \wedge \textit{Receive}) \vee (\textit{Idle} \wedge \textit{Send})\big)\}$. Pour ce contrat, la garantie G est vraie quelles que soient les entrées. Pour le prouver, il suffit de constater qu'aucune transition de l'automate ne produit deux sorties à la fois (parmi les signaux *Off*, *Receive*, *Idle* et *Send*).

Sur la figure 7.6, nous avons représenté la connexion des composants MAC et radio. Il faut proposer un contexte K' qui puisse assurer que le composant radio est sollicité avec des entrées qui satisfont K dans MAC $\parallel$ radio. De cette façon, on pourra écrire MAC $\parallel R_1 \preceq_{K'}$ MAC $\parallel R_2$.

Voici quelques précisions temporelles sur la définition du protocole MAC. Tout d'abord on considère qu'un instant d'exécution de notre modèle correspond à une milli-seconde de temps réel. De plus,

- $T = 1000$, la période de veille dure une seconde[2].
- L'opération *Clear Channel Assessment* dure 10 millisecondes. On passe donc 10 unités de temps dans l'état *CCA*. Pendant ce temps, le composant MAC émet des signaux *Busy*.
- De même pour l'opération *Carrier Sense* : elle dure 10 millisecondes et pendant ce temps le composant MAC émet des signaux *Busy*.
- Nous supposons que la longueur d'émission (et de réception !) d'un paquet est de 700 instants.

Pour proposer un contexte K' réaliste, on peut prendre en compte comment sont produites les entrées du composant MAC. Sur la figure 7.10, nous avons représenté les composants MAC et radio dans un système plus global.

Notre abstraction de modèle de radio est valide pour un contexte K de traces qui ne contiennent pas trop de *Idle*. Le protocole MAC émet des signaux *Idle* dans les états *CCA*, *Carrier Sense* et *Collision*. Étant données les constantes internes de temps, on passe, périodiquement, 10 instants dans l'état *Carrier Sense* pour 1000 instants dans l'état *Sleep*. Donc pour ces deux états, l'abstraction ne dépend pas des entrées et elle est respectée puisque l'automate émet 10 *Idle* pour 1000 *Off*. On accède aux états *CCA* et *Collision* lorsque les entrées *Pckt2send* ou *Busy* sont reçues. Donc, les traces du nouveau

[2] ce choix est celui implémenté dans l'application des compteurs d'eau de Coronis Systems, voire section 2.1.3 page 28

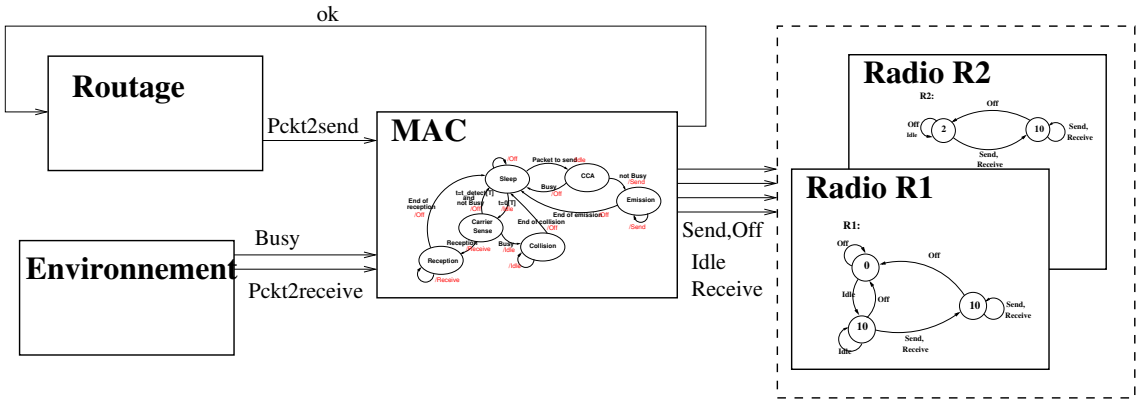


FIG. 7.10 – Schéma de composition des différents modules. Ici, les composants routage et environnement sont très abstraits parce qu'on ne les connaît pas forcément.

contexte K' (K' est un ensemble de trace sur $\{Pckt2send, Busy, Pckt2receive\}$) ne doivent pas contenir trop de signaux *Pckt2send* et *Busy*

Voici comment nous définissons K' :

- La longueur maximale d'une suite consécutive de *Busy* est 700 (qui correspond à la longueur d'un paquet).
- Les fronts montants de signaux *Busy* sont espacés d'au moins 5000 instants.
- Les signaux *Pckt2send* sont espacés d'au moins 5000 instants.

Ce contexte K' est réaliste pour un réseau de capteurs. Premièrement, la durée d'une collision est la durée de transmission d'un paquet. Deuxièmement, on suppose qu'il n'y a pas trop de collision, évidément si le canal était tout le temps brouillé ça ne pourrait pas marcher. Et troisièmement, il ne faut pas qu'il y ait trop de paquets à envoyer.

On peut vérifier que le composant MAC produit bien des sorties qui sont dans K lorsqu'il est sollicité par des traces de K'. Nous avons fait cette vérification à l'aide de REACTIVEML, pour des exemples plus gros, un outil de vérification par modèle pourrait être pratique.

7.3.3 Un contre exemple

Nous insistons sur le fait que le contexte est important pour définir une abstraction. Nous avons défini une abstraction $R_1 \preceq_K R_2$ qui n'est valable que pour le contexte K. K représente un ensemble de traces réalistes pour les réseaux de capteurs pour lesquelles ça a du sens de comparer R_1 et R_2. Il faut bien voir que la relation MAC $\parallel R_1 \preceq$ MAC $\parallel R_2$ n'est pas toujours vraie. Prenons par exemple le protocole MAC de la norme IEEE802.11 (Wi-Fi). Ce protocole dédié aux réseaux d'ordinateurs personnels n'est pas conçu pour économiser l'énergie. Dans ce protocole, la radio n'est jamais éteinte. Dans notre modélisation, le composant qui modélise ce protocole n'envoie jamais de signaux *Off* à la radio, celle-ci est donc toujours en émission, réception ou écoute libre. L'abstraction est donc fausse dans ce cas quelles que soient les entrées du protocole MAC.

7.4 Autres remarques

7.4.1 D'autres fonctions de combinaison des coûts

Dans la définition du produit synchrone de machines de Mealy booléennes à coût, section 7.2.2, nous avons eu besoin de définir une fonction de combinaison des coûts. La fonction que nous avons proposée est la somme. Comme nous l'avions évoqué section 4.2.1 page 74 cette fonction convient bien pour faire le produit de composants qui consomment leurs énergies sur une même batterie, par exemple s'il s'agit de composants d'un même nœud. Par contre, si le produit est celui de deux capteurs du réseau alors la somme de leurs consommations ne correspond à aucune grandeur physique intéressante. Dans ce cas, une autre fonction de combinaison des coûts est nécessaire. La fonction de n-uplisation qui liste l'ensemble des consommations des différents nœuds semble intéressante.

L'intérêt de notre formalisme réside dans le fait que la propriété d'abstraction est préservée par composition. Pour que cette propriété reste vraie, il faut que la fonction de combinaison des coûts dans le produit commute avec la fonction d'intégration des coûts par rapport au temps (voir item 2 de la preuve, section 7.2.4). Pour la fonction d'intégration par rapport au temps, c'est la somme qui convient. Notre preuve fonctionne parce que la fonction somme commute avec elle-même. Notons Σ, la fonction d'intégration des consommations (par abus de notation, $\Sigma_{i=0}^{n} a_i$ correspondrait à l'énergie consommée par la machine a entre les instants O et n). Et soit, $\oplus$ la fonction de composition des coûts pour le produit. Nous avons utilisé pour notre preuve,

$$\Sigma_i(a_i \oplus b_i) = \Sigma_i(a_i) \oplus \Sigma_i(b_i)$$

ce qui est vrai lorsque $\Sigma = \oplus = + =$ somme. C'est également vrai si $\oplus$ est la fonction qui renvoie le n-uplet des consommations, dans ce cas :

$$\Sigma_i(a_i \oplus b_i) = \Sigma_i(a_i, b_i) = (\Sigma_i a_i, \Sigma_i b_i)$$

Mais, il faut être prudent parce que toutes les fonctions ne conviennent pas, par exemple si l'on choisit la fonction *maximum* comme fonction de combinaison des coûts pour le produit ($\oplus$), ça ne marche plus. En effet, le maximum de la somme n'est pas la somme des maxima des sommes.

7.4.2 Propagation automatique des contraintes ?

La méthode que nous avons utilisé pour construire une abstraction entre les composants MAC $\parallel R_1$ et MAC $\parallel R_2$ est la suivante. Nous avions $R_1 \preceq_K R_2$, nous avons proposé un nouveau contexte K'. Nous avons vérifié que pour les entrées de K', le composant R_1 est sollicité avec des entrées qui satisfont K dans MAC $\parallel R_1$. Nous avons donc pu conclure MAC $\parallel R_1 \preceq_{K'}$ MAC $\parallel R_2$.

Il est parfois possible de propager K automatiquement grâce aux contrats. Prenons un exemple, voir figure 7.11. Soient C_1 et C_2 deux composants dont les signaux d'entrées sont a et b. K est l'ensemble de traces sur $\{a, b\}$ défini par : "le nombre de a est plus grand le nombre de b". Supposons, $C_1 \preceq_K C_2$.

On souhaite écrire $C \parallel C_1 \preceq_{K'} C \parallel C_2$. D'après la définition d'un composant, section 7.2.3, $C = (\{x\}, \{a\}, M, A, G)$. C est donc un composant avec une entrée (x) et une sortie (y). On ne connaît pas la machine de Mealy M, par exemple parce que ce composant n'est pas encore complètement implémenté. Toute l'information que l'on a est celle du contrat A/G. Supposons que l'on ait le contrat suivant :

- $A =$ l'ensemble des traces sur $\{x\}$ telles que x n'est émis que les instant impairs.
- $G =$ l'ensemble des traces sur $\{x, a\}$ telles qu'il y a au moins un a tous les 5 x.

Alors un contexte K' qui convient est le suivant :
$K' =$ l'ensemble des traces sur $\{x, b\}$ telles que x n'est émis que les instants impairs et telles qu'il y a au moins 5 fois plus de x que de b.

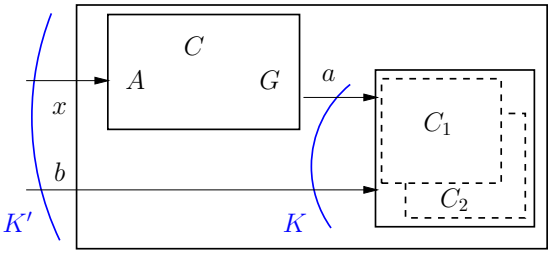


FIG. 7.11 – Illustration de l'exemple : cas simple de propagation du contexte K en K'

Dans le cas général, il est difficile de déduire K' à partir de K et des différents contrats. Dans la méthode que nous proposons, c'est à l'utilisateur de proposer K'. Ce qui est cohérent, puisque comme on l'a vu, c'est parce que l'on a une idée de ce que représente K' dans le modèle que l'on est à même de proposer un contexte K' qui convient pour l'abstraction et qui est réaliste. Ensuite, c'est un outils automatique qui vérifie $K \Longrightarrow K'$ dans le produit des composants. Dans notre exemple, nous avons utilisé pour cette étape un programme simple écrit spécifiquement pour notre problème, mais un programme dédié est facile à concevoir.

7.4.3 L'énergie n'est pas fonctionnelle

Remarquons que dans les modèles de consommation que nous avons introduits section 7.2, le comportement des modèles ne peut pas dépendre de la consommation ni de l'énergie (dépensée ou restante). En effet, les transitions (dans les machines de Mealy booléennes à coûts) ne sont conditionnées que par des combinaisons booléennes de signaux. Les signaux ne pouvant dépendre de l'énergie, les transitions ne dépendent donc pas de la fonction $\mathfrak{L}$. On dit que l'énergie n'est pas fonctionnelle.

Pourtant, pour des systèmes embarqués contraints, il se peut que l'on ait envie (ou besoin) de programmer en utilisant l'information concernant l'énergie restante. Par exemple, on pourrait simplement imaginer qu'un système ait un mode normal et un mode dégradé au cas où l'énergie restante passe sous un certain seuil.

Mais, pour programmer avec des composants, utiliser l'énergie qui est une information globale peut être problématique. Cela nous empêcherait, par exemple, de construire des abstractions modulaires. Sur la figure 7.12, nous avons dessiné deux automates A et B. Si l'on exécute A et B seuls, alors dans les états S_1 et Q_1 la variable E vaut moins de 10, B est bien une abstraction de A. Cependant si on compose respectivement A et B avec une machine M, on ne peut plus rien conclure parce que le comportement de A dépend de la consommation de M.

Programmer en utilisant l'énergie pose donc des problèmes. En fait, la question sous-jacente est : "comment programmer avec de l'énergie réelle ?"[3]. La question que nous nous sommes posé dans notre thèse est "comment modéliser l'énergie ?". Ces deux questions sont finalement assez différentes.

[3]Le terme "énergie réelle" fait référence au temps réel utilisé dans les systèmes embarqués.

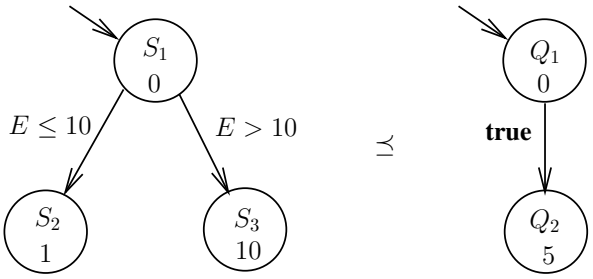


FIG. 7.12 – Exemple d'automates où l'énergie est fonctionnel. La relation d'abstraction n'est plus préservée par composition.

Chapitre 8

Conclusion et perspectives

Sommaire

Les réseaux de capteurs sont de nouveaux systèmes distribués que les dernières avancées technologiques des systèmes embarqués ont rendus possibles. Un frein au déploiement de ces réseaux est le manque d'outils de modélisation. Les nœuds d'un réseau de capteurs étant autonomes en énergie, on souhaite, pour déployer un réseau, certaines garanties sur la durée de vie de celui-ci.

8.1 Bilan

8.1.1 Modélisation analytique de protocoles MAC

Les protocoles d'accès au médium (MAC) dédiés aux réseaux de capteurs permettent d'économiser l'énergie dépensée en éteignant la radio le plus souvent possible. Nous avons travaillé avec Abdelmalik Bachir sur l'étude et la conception de protocoles MAC à échantillonnage de préambule. À partir de différents protocoles Nous avons évalué la fiabilité et l'efficacité en énergie de différentes variantes de protocoles à échantillonnage de préambule en considérant un canal radio non fiable. Notre modèle de canal suppose que la probabilité que la transmission d'un bit soit erronée est uniforme et indépendante du temps. L'évaluation de l'énergie dépensée se fait en comptant le temps passé à émettre et à recevoir.

Nous en avons conclu qu'aucun protocole n'était toujours meilleur que les autres. Une méthode d'accès qui change de politique d'émission (*micro-frame* ou *data-frame*) et de réception (persistant ou non persistant) en fonction du taux d'erreur observé sur le canal a été breveté [6]. L'étude comparative a été publiée dans [7].

Une des limitations de ce travail est que le modèle ne prend pas en compte le réseau complet : seule une liaison entre deux nœuds est considérée et au niveau d'un nœud il n'y a que le protocole MAC qui est modélisé. Ça n'est pas le cas de notre modélisation pour la simulation.

8.1.2 Simulation

Nous avons conçu GLONEMO un simulateur de réseaux de capteurs. GLONEMO modélise l'ensemble du réseau, toutes les couches logicielles y sont représentées de la couche applicative au protocole MAC. Des éléments matériels y sont également modélisés afin d'estimer de façon fiable la consommation d'énergie. Nous avons modélisé un réseau dédié à la surveillance d'un environnement dangereux (feux de forêt ou nuage radioactif). Cette étude de cas a montré qu'il était possible de simuler un modèle global et détaillé d'un réseau de plusieurs milliers de nœuds avec GLONEMO. Grâce à la structure modulaire de GLONEMO, il est aisé de changer des composants pour modéliser d'autres systèmes (applications différentes, autres protocoles...). GLONEMO a été présenté dans [78].

Une des nouveautés de GLONEMO par rapport aux simulateurs existants est la prise en compte d'un modèle d'environnement. En effet, dans les réseaux de capteurs, l'environnement est souvent la cause des communication, il nous a donc semblé nécessaire de le prendre en compte dans un modèle global. Nous avons modélisé un environnement non déterministe grâce à LUCKY. Pour valider l'importance du modèle d'environnement dans le comportement des réseaux de capteurs, nous avons comparé les résultats de simulations GLONEMO dans lesquelles seul le modèle d'environnement diffère. Nous en avons conclu qu'il est indispensable de modéliser l'environnement pour obtenir des simulations réalistes de réseaux de capteurs. Cette expérience a été publiée dans [77].

Une spécificité de GLONEMO est que nous l'avons programmé en REACTIVEML. Ce simulateur est donc une étude de cas d'un programme en REACTIVEML. Ça n'est certes pas la contribution majeur de GLONEMO, mais ce travail a fourni à Louis Mandel (concepteur de REACTIVEML) à la fois des retours d'utilisateurs de REACTIVEML et de la visibilité pour ce nouveau langage.

Cette expérience de simulation en REACTIVEML a conforté l'idée que REACTIVEML est un langage qui convient bien à la programmation de simulateurs de réseaux. En fait, GLONEMO est la deuxième expérience de simulateurs en REACTIVEML, un premier simulateur de réseaux mobiles ad hoc avait été écrit pour comparer des protocoles de routage [56].

Dans cette thèse, nous avons expliqué pourquoi REACTIVEML convient pour écrire rapidement des simulateurs en insistant sur le fait que les simulateurs REACTIVEML sont à pas fixes alors que les simulateurs classiques sont des simulateurs à événements discrets. En effet, le modèle réactif synchrone de Boussinot sur lequel est construit REACTIVEML permet d'exécuter tous les processus en parallèle de façon synchrone. Les simulateurs écrits en REACTIVEML sont donc naturellement des simulateurs à pas fixes. Alors que certains pensent que seul un moteur de simulation à événements discrets permet de construire des simulateurs qui passent à l'échelle, nous montrons qu'avec REACTIVEML il est possible de simuler avec un pas fixe plusieurs dizaines de milliers de nœuds.

Nous avons également construit LUSSENSOR, un modèle de réseaux de capteurs en LUSTRE. Ce travail a été publié dans [61]. Ce programme nous a permis de comparer deux simulateurs similaires écrits en LUSTRE et en REACTIVEML. De plus, LUSSENSOR offre des perspectives d'utilisation pour la vérification formelle.

8.1.3 Modélisation formelle

Les simulations ne peuvent offrir que des informations sur le comportement moyen d'un réseau, une campagne de simulations ne permet pas de garantir une propriété du réseau. Au contraire, les techniques de vérification formelles permettent de prouver une propriété de façon exhaustive.

Étude de cas avec IF

Afin de montrer l'intérêt et la faisabilité de la vérification par modèles (*model-checking*) pour les réseaux de capteurs, nous avons effectué une première expérience à l'aide de l'outil de modélisation IF. Nous avons comparé les durées de vie pire-cas d'un réseau pour deux protocoles de routage différents. Pour ce faire, nous avons dû modifier l'outil d'exploration IF pour trouver dans le graphe d'états, le plus court chemin au sens de la durée de vie. Cette expérience est l'une des premièrede vérification par modèle d'une propriété concernant l'énergie pour un réseau de capteurs. Elle nous a permis de montrer que la durée de vie pire-cas d'un réseau est une information que l'on peut difficilement déduire de la durée de vie moyenne. Cette expérience a conduit à une publication : [65]. La vérification exhaustive ne peut se faire que sur des modèles de taille raisonnable, pour cette expérience nous avons donc modélisé un réseau de capteurs de façon assez simplifiée en nous servant de notre expérience acquise grâce au simulateur GLONEMO. Cependant, une des limitations de ce travail est que nous ne sommes pas capable de garantir que nos abstractions sont correctes, c'est-à-dire que la propriété prouvée sur le modèle abstrait est aussi valable sur le modèle détaillé.

Cadre formel définissant une relation d'abstraction pour des modèles de consommation

Pour palier à ce défaut, nous avons défini formellement une notion d'abstraction de modèles à coûts. Pour être utilisable en pratique et s'appliquer à des exemples réels, notre définition d'abstraction permet de comparer des modèles même si la relation d'abstraction n'est vraie que dans un certain contexte. En effet, il faut prendre en compte le mode d'utilisation d'un composant pour être capable de proposer une abstraction qui ne soit pas trop brutale. De plus, afin de construire des abstractions par composants, nous avons proposé un ensemble de conditions qui permettent de propager une abstraction. Intuitivement, si $C_1 \preceq_K C_2$, pour écrire $C_1 \parallel C \preceq_{K'} C_2 \parallel C$, il faut s'assurer que les sous-composants C_1 et C_2 sont bien sollicités par des entrées qui satisfont le contexte K' ($K' \Longrightarrow K$ dans $C_1 \parallel C$ et $C_2 \parallel C$).

Nous avons illustré notre cadre formel définissant une abstraction avec un exemple issu des réseaux de capteurs. Après avoir proposé deux modèles de radio R_1 et R_2 (R_1 étant une abstraction de R_2 dans un contexte K), nous composons ces modèles avec un protocole MAC en faisant en sorte de conserver la relation d'abstraction.

Modélisation de l'énergie

Pour vérifier des propriétés concernant la durée de vie à l'aide de vérification par modèle, nous avons modélisé la consommation d'énergie en utilisant des automates. Dans cette thèse, section 4.2 page 72, nous avons répertorié les différents moyens de modéliser la consommation d'énergie avec des automates en expliquant ce que chacun d'eux modélisent le mieux. Cette contribution est sans doute mineure mais elle peut servir à ceux qui ont besoin de modéliser des dépenses d'énergie.

8.2 Travaux en cours utilisant GLONEMO

Avant de proposer des perspectives possibles à nos travaux, soulignons que GLONEMO est actuellement utilisé dans divers projets.

Matthieu Anne, ingénieur de recherche à France Télécom R&D, a travaillé dans le cadre de NanoSoc, un projet du pôle de compétitivité MINALOGIC, sur un réseau de nez électroniques. Après une phase d'apprentissage, un tel réseau est chargé de détecter des odeurs. Les autres partenaires (le

CEA, Schneider ou encore Alpha Mos) travaillent à la réalisation du nano-capteur lui-même. France Télécom est chargé notamment du traitement local et de la mise en réseau des capteurs. Les capteurs n'étant bien sûr pas disponibles, un simulateur est indispensable pour prévoir le comportement du réseau. Matthieu Anne a utilisé GLONEMO comme outil de simulation qu'il a bien sûr modifié pour ses besoins.

Aujourd'hui, Olivier Bezet a repris GLONEMO dans le cadre du projet RNRT ARESA [1]. Il a amélioré le simulateur en proposant notamment un modèle de propagation plus réaliste. Olivier Bezet et al ont utilisé GLONEMO pour comparer les résultats de simulations effectuées avec des modèles détaillés à celles où une abstraction a été faite. Ils ont d'abord effectué des simulations où l'énergie est modélisée finement (grâce à la modélisation globale et précise de GLONEMO). Puis, ils en ont déduit les coûts moyens d'émission et de réception d'un message. Ces coûts fournissent automatiquement une abstraction en remplaçant le modèle de consommation détaillé par un modèle où les émissions et réceptions ont un coût fixe. Bien sûr, comme tous les éléments ont une influence sur la consommation, si l'on change d'autres hypothèses (comme le modèle de l'environnement) l'abstraction n'est plus valable. Ces travaux seront/ont été soumis à DCOSS 2008 [10].

Un autre projet en cours dans le cadre de ARESA est la modélisation avec GLONEMO d'un réseau de capteurs existant et déjà déployé. L'application choisie est celle des "compteurs d'eau" de Coronis Systems, détaillée section 2.1.1, page 20. Les protocoles implémentés sont les protocoles réels (déjà implémentés dans la version présentée ici) et les valeurs pour les modèles consommation sont fournies par Coronis Systems. Cette expérience va permettre de comparer des données obtenues en simulation à celles obtenues sur le terrain. En effet, cette application est déployée depuis plusieurs années donc les ingénieurs de Coronis Systems ont déjà pu observer le comportement du réseau. Confronter les résultats de simulations à la réalité permettra d'évaluer quelle confiance on peut avoir en notre simulateur. Selon les auteurs de [23], très peu d'expériences de ce type ont été faites. Dans [23], ils comparent pour un réseau très simple les simulations Omnet avec la réalité et concluent qu'il y a des différences importantes. Cette étape semble donc importante pour valider et améliorer GLONEMO. C'est également Olivier Bezet qui travaille actuellement sur ce projet.

Il est arrivé que des chercheurs extérieurs à France Télécom et à Verimag me demandent les sources de GLONEMO. Je n'ai malheureusement pas pu répondre favorablement à ces utilisateurs potentiels parce que le code écrit sous contrat France Télécom n'est pas libre.

8.3 Perspectives

8.3.1 Modélisation analytique de protocoles MAC

Pour analyser la fiabilité de plusieurs variantes d'un protocole MAC (p-MFP, p-DFP, np-MFP, np-DFP) suivant l'état du canal nous avons considéré un modèle simple qui gagnerait à être enrichi. Nous avons plusieurs pistes pour l'enrichir.

Tout d'abord, nous pourrions discerner deux types de transmission en *broadcast* ou en *unicast* comme nous l'avons fait au chapitre 6 : les transmissions en *unicast* étant acquittées alors que celles en *broadcast* ne le sont pas.

Il faudrait prendre en compte les collisions dans la modélisation du canal. En effet, nous avons pris en compte des erreurs de transmission qui ne dépendent pas du temps. Un taux moyen d'erreurs (*bit error rate*) fournit une information sur l'état général du canal, pas sur les collisions. Pour modéliser les collisions, il faudrait prendre en compte des erreurs de transmission qui ont une forte probabilité d'être consécutives. Ou encore, on pourrait considérer des erreurs de transmission qui concernent forcément

plusieurs bit de données consécutifs d'une longueur éventuellement fixe, cette longueur correspondrait à la longueur d'une collision.

Enfin, pour être encore plus réaliste, notre modèle gagnerait à prendre en compte plusieurs nœuds. Nous n'avons considéré ici qu'un seul lien, il faudrait prendre en compte l'existence de nœuds voisins. Modéliser des erreurs correspondant à des collisions est déjà un moyen de considérer des nœuds voisins. S'il est impossible de modéliser comme nous l'avons fait un réseau complet, cette analyse est peut-être envisageable pour certaines topologies particulières comme un réseau en étoile ou un réseau de seulement trois nœuds.

Cependant, afin d'obtenir des résultats utilisables en pratique on ne peut pas considérer de modèles arbitrairement complexes, c'est pourquoi ce n'est pas la méthode de modélisation que nous avons privilégiée dans cette thèse.

8.3.2 Modélisation formelle

Les techniques de vérification formelle prenant en compte l'énergie sont utiles pour concevoir des réseaux de capteurs, mais ça n'est certainement pas leur seul champ d'applications. En effet, ces techniques semblent également utiles pour d'autres systèmes embarqués contraints en énergie.

Nous avons montré avec IF la faisabilité des techniques de vérification par modèle pour l'étude des réseaux de capteurs. Cette expérience montre qu'il faut trouver des solutions afin de pouvoir étudier des systèmes de plus grande taille.

Cependant, cette expérience est "ad hoc" au sens où les outils que nous avons utilisés ne sont pas dédiés aux réseaux de capteurs. Tout d'abord, il serait souhaitable d'avoir dans le langage de modélisation des primitives plus spécifiques aux réseaux de capteurs. Par exemple, pour les communications, nous avons utilisé les modes de communication disponibles en IF en les paramétrant pour qu'ils modélisent les communications radio. On pourrait imaginer un langage de modélisation formelle qui contienne des primitives de communication correspondant aux communications radio *broadcast* et *unicast*. Ces primitives étant gérées efficacement, la taille du modèle serait réduite.

Un autre point clé à prendre en compte pour la réalisation d'un langage de modélisation dédié à la vérification des réseaux de capteurs est l'énergie. Dans notre expérience en IF, l'énergie est simplement une variable particulière de type entier. Il serait souhaitable d'en avoir une représentation plus efficace. Dans le chapitre 4, nous avons présenté les LPTA (page 76) qui permettent d'avoir une représentation continue de l'énergie. Nous pensons qu'il n'est pas indispensable de pouvoir représenter une énergie continue. Ce qui est important c'est de réduire la taille des modèles et pour cela, avoir une variable pour exprimer l'énergie est une idée intéressante. Ceci permettrait de représenter l'énergie de manière efficace dans le modèle et donc d'avoir un algorithme dédié à la recherche de la durée de vie pire cas.

Un réseau de capteurs est un système complexe pour lequel nous pensons qu'un langage dédié permettrait peut-être de réduire la taille des modèles d'un facteur 2 à 10, mais certainement pas d'un facteur 100. Pour réduire considérablement la taille du modèle, il faut accepter de faire des abstractions quitte à perdre de l'information. C'est pourquoi, dans ce travail, nous nous sommes plutôt intéressés à définir des abstractions au lieu d'un langage dédié. Ces travaux de recherche ont également des perspectives.

Pour pouvoir utiliser notre relation d'abstraction définie au chapitre 7 de façon plus automatique, il nous faut un algorithme pour valider une abstraction entre deux composants. La difficulté pour prouver automatiquement $A \preceq_K B$ réside dans le fait que l'ensemble K n'est pas préfixe clos.

Nous avons défini une relation d'abstraction dans un contexte synchrone qui convient bien à la modélisation des nœuds. Il serait également souhaitable de définir une relation d'abstraction pour un

contexte asynchrone, c'est-à-dire une relation d'abstraction qui soit préservée par une composition asynchrone. Une piste consiste à définir une relation de type bisimulation.

Bibliographie

[1] ARESA Project.
http://aresa-project.insa-lyon.fr/.

[2] Martin Heusse Abdelmalik Bachir, Dominique Barthel and Andrzej Duda. Micro-frame preamble mac for multihop wireless sensor networks. *Proceedings of the IEEE ICC*, Istanbul, Turkey, June 2006.

[3] Ian F. Akyildiz, W. Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks : a survey. *Computer Networks*, 38(4) :393–422, 2002.

[4] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126 :183–235, 1994.

[5] Abdelmalik Bachir. *Optimizing Routing and Channel Access Protocols to Extend the Lifetime of Wireless Sensor Networks*. PhD thesis, INPG, Grenoble, France, January 2007.

[6] Abdelmalik Bachir, Ludovic Samper, and Dominique Barthel. Procédé de communication, stations émettrice et réceptrice et programmes d'ordinateur associés. Brevet numéro : FR2902589 (A1), dec 2007. France Telecom.

[7] Abdelmalik Bachir, Ludovic Samper, Dominique Barthel, Martin Heusse, and Andrzej Duda. Link cost and reliability of frame preamble mac protocols. In *Proceedings of International Workshop on Wireless Ad-Hoc Networks 2006 (IWWAN 2006)*, page 7, New York, United States, June 2006.

[8] Gerd Behrmann. *Data Structures and Algorithms for the Analysis of Real Time Systems*. PhD thesis, Aalborg University Department of Computer Science, 2003.

[9] Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim Guldstrand Larsen, Paul Pettersson, Judi Romijn, and Frits W. Vaandrager. Minimum-cost reachability for priced timed automata. In *HSCC '01 : Proceedings of the 4th International Workshop on Hybrid Systems*, pages 147–161, London, UK, 2001. Springer-Verlag.

[10] Olivier Bezet, Florence Maraninchi, and Laurent Mounier. Abstraction problems in virtual prototyping of wireless sensor networks. In *Proceedings of the 4th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS 2008)*, June 2008. Submitted to publication.

[11] Sébastien Bornot and Joseph Sifakis. An Algebraic Framework for Urgency. *Information and Computation*, 163 :172–202, 2000.

[12] Sébastion Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling urgency in timed systems. In *International Symposium : Compositionality - The Significant Difference, Malente (Holstein, Germany)*, volume 1536 of *LNCS*, 1998.

[13] Frédéric Boussinot and Robert de Simone. The SL synchronous language. *IEEE Transactions on Software Engineering*, 22(4) :256–266, 1996.

[14] Patricia Bouyer and François Laroussinie. Vérification par automates temporisés. In Nicolas Navet, editor, *Systèmes temps-réel 1 : techniques de description et de vérification*, volume I, chapter 4, pages 121–150. Hermès, http ://www.editions-hermes.fr/, 2006.

[15] Marius Bozga. *Vérification symbolique pour les protocoles de communication*. Thèse de doctorat, Université Joseph Fourier, Grenoble, December 1999.

[16] Marius Bozga, Susanne Graf, and Laurent Mounier. IF-2.0 : A Validation Environment for Component-Based Real-Time Systems. In *CAV*, pages 343–348, 2002.

[17] Marius Bozga, Susanne Graf, Laurent Mounier, and Iulian Ober. La boîte à outils IF. In Nicolas Navet, editor, *Systèmes Temps Réel : Techniques de description et de vérification*, volume I, chapter 9, pages 293–326. Hermès, http ://www.editions-hermes.fr/, 2006.

[18] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The IF Toolset. In Flavio Corradinni and Marco Bernanrdo, editors, *Proceedings of SFM'04*, volume 3185 of *LNCS*, Bertinoro (Italy), September 2004. Springer-Verlag.

[19] Lee Breslau, Deborah Estrin, Kevin R. Fall, Sally Floyd, John S. Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in network simulation. *IEEE Computer*, 33(5) :59–67, 2000.

[20] Xinjie Chang. Network simulations with opnet. In *WSC '99 : Proceedings of the 31st conference on Winter simulation*, pages 307–314, New York, NY, USA, 1999. ACM.

[21] Guillaume Chelius, Antoine Fraboulet, and Éric Fleury. Worldsens. http://worldsens.citi.insa-lyon.fr/.

[22] Sinem Coleri, Mustafa Ergen, and T. John Koo. Lifetime analysis of a sensor network with hybrid automata modelling. In *WSNA '02 : Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 98–104. ACM Press, 2002.

[23] Ugo Colesanti, Carlo Crociani, and Andrea Vitaletti. On the accuracy of omnet in the wireless sensor network domain : Simulation vs. testbed. In *Fourth ACM Workshop on Performance Evaluation of Wireless Ad Hoc, Sensor, and Ubiquitous Networks (PE-WASUN)*, Chania, Crete Island, Greece, octobre 2007. ACM.

[24] Victor Delaluz, Mahmut T. Kandemir, Narayanan Vijaykrishnan, Anand Sivasubramaniam, and Mary Jane Irwin. Hardware and software techniques for controlling dram power modes. *IEEE Trans. Computers*, 50(11) :1154–1173, 2001.

[25] Ilker Demirkol, Fatih Alagöz, Hakan Deliç, and Cem Ersoy. Wireless sensor networks for intrusion detection : Packet traffic modeling. *IEEE Communications Letters*, 10(1) :22–24, January 2006.

[26] Ian Downard. Simulating sensor networks in ns-2, 2005.

[27] Esteban Egea-Lopez, Javier Vales-Alonso, Alejandro Martinez-Sala, Pablo Pavon-Mario, and Joan Garcia-Haro. Simulation Scalability Issues in Wireless Sensor Networks. *Communications Magazine, IEEE*, 44(7) :64–73, 2006.

[28] Christian C. Enz, Amre El-Hoiydi, Jean-Dominique Decotignie, and Vincent Peiris. Wisenet : An ultralow-power wireless sensor network solution. *IEEE Computer*, 37(8) :62–70, 2004.

[29] EPFL. Network in A Box, 2004-2006. http://web.archive.org/web/*/http://nab.epfl.ch.

[30] Deborah Estrin, Mark Handley, John Heidemann, Steven McCanne, Ya Xu, and Haobo Yu. Network visualization with nam, the vint network animator. *Computer*, 33(11) :63–68, 2000.

[31] Ansgar Fehnker. *Citius, Vilius, Melius - Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. PhD thesis, KUNijmegen, 2002.

[32] Antoine Fraboulet, Guillaume Chelius, and Eric Fleury. Worldsens : Development and prototyping tools for application specific wireless sensors networks. In *IPSN'07 Track on Sensor Platforms, Tools and Design Methods (SPOTS)*, Cambridge, Massachusetts, USA., April 2007. ACM.

[33] Motorola freescale. Motorola MC13192 Data Sheet, 2005.

[34] Victor S. Frost and Benjamin Melamed. Traffic modeling for telecommunications networks. *IEEE Communications Magazine*, pages 70–81, March 1994.

[35] Richard M. Fujimoto, Kalyan S. Perumalla, Alfred Park, Hao Wu, Mostafa H. Ammar, and George F. Riley. Large-scale network simulation : How big ? how fast ? In *MASCOTS*, pages 116–, 2003.

[36] David Gay, Philip Levis, J. Robert von Behren, Matt Welsh, Eric A. Brewer, and David E. Culler. The nesC language : A holistic approach to networked embedded systems. In *PLDI*, pages 1–11, 2003.

[37] Laure Gonnord. *Accélération abstraite pour l'amélioration de la précision en Analyse des Relations Linéaires*. Thèse de doctorat, Université Joseph Fourier, Grenoble, October 2007.

[38] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991.

[39] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, pages 785–793, September 1992.

[40] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech : A model checker for hybrid systems. In *CAV '97 : Proceedings of the 9th International Conference on Computer Aided Verification*, pages 460–463. Springer-Verlag, 1997.

[41] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2) :98–107, 1989.

[42] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion : a scalable and robust communication paradigm for sensor networks. In *MOBICOM*, pages 56–67, 2000.

[43] Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, John Heidemann, and Fabio Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Transaction on Networking (TON)*, 11(1) :2–16, 2003.

[44] Erwan Jahier and Pascal Raymond. Lucky, A (target) language for describing and simulating stochastic reactive systems. http://www-verimag.imag.fr/SYNCHRONE/index.php?page=lurette/lucky.

[45] Erwan Jahier and Pascal Raymond. The Lucky language Reference Manual. Technical Report TR-2004-6, Verimag Technical Report.

[46] Bertrand Jeannet. *Partitionnement dynamique dans l'analyse de relations linéaires et application à la vérification de programmes synchrones*. Thèse de doctorat, Institut National Polytechnique de Grenoble, September 2000.

[47] Leonard Kleinrock and Fouad A. Tobagi. Packet switching in radio channels : Part i–carrier sense multiple-access modes and their throughput-delay characteristics. In *IEEE Transactions on Communications*, volume 23, pages 1400–1416, december 1975.

[48] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2) :125–143, 1977.

[49] Kim Guldstrand Larsen, Gerd Behrmann, Ed Brinksma, Ansgar Fehnker, Thomas Hune, Paul Pettersson, and Judi Romijn. As cheap as possible : Efficient cost-optimal reachability for priced timed automata. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 493–505. Springer, 2001.

[50] Xavier Leroy. Caml.
http://caml.inria.fr/.

[51] Xavier Leroy. The Objective Caml system release 3.10 Documentation and user's manual. Technical report, INRIA, 2007.

[52] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim : accurate and scalable simulation of entire tinyos applications. In *SenSys '03 : Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137, New York, NY, USA, 2003. ACM Press.

[53] Philip Levis, Sam Madden, David Gay, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. Tinyos : An operating system for sensor networks. In W. Weber, J. Rabaey, and E. Aarts, editors, *Ambient Intelligence*. Springer-Verlag, 2005. © Springer-Verlag.

[54] Louis Mandel and Marc Pouzet. Reactive ML.
http://www.reactiveml.org/.

[55] Louis Mandel. *Conception, Sémantique et Implantation de ReactiveML : un langage à la ML pour la programmation réactive*. PhD thesis, Université Paris 6, 2006.

[56] Louis Mandel and Farid Benbadis. Simulation of mobile ad hoc network protocols in ReactiveML. In *Synchronous Languages, Applications, and Programming (SLAP'05)*, Edinburgh, Scotland, April 2005. ENTCS.

[57] Louis Mandel and Marc Pouzet. ReactiveML, a reactive extension to ML. In *Proceedings of 7th ACM SIGPLAN International conference on Principles and Practice of Declarative Programming (PPDP'05)*, Lisbon, Portugal, July 2005.

[58] Louis Mandel and Marc Pouzet. ReactiveML : un langage fonctionnel pour la programmation réactive. *Technique et Science Informatiques (TSI)*, 2007. Accepted for publication.

[59] Florence Maraninchi and Lionel Morel. Logical-time contracts for reactive embedded components. In *30th EUROMICRO Conference on Component-Based Software Engineering Track, ECBSE'04*, Rennes, France, August 2004.

[60] Florence Maraninchi and Yann Rémond. Argos : an automaton-based synchronous language. *Computer Languages*, 27 :61–92, 2001.

[61] Florence Maraninchi, Ludovic Samper, Kevin Baradon, and Antoine Vasseur. Lustre as a system modeling language : Lussensor, a case-study with sensor networks. In *Proceedings of Model-driven High-level Programming of Embedded Systems (SLA++P'07)*, Braga, Portugal, March 2007.

[62] The Mathworks. Matlab.
http://www.mathworks.com/products/matlab/.

[63] Jan Mikáč. *Raffinements et preuves de systèmes Lustre*. Thèse de doctorat, Institut National Polytechnique de Grenoble, novembre 2005.

[64] Lionel Morel. *Exploitation des structures regulieres et des specifications locales pour le developpement correct de systemes reactifs de grande taille*. PhD thesis, Institut National Polytechnique de Grenoble, march 2005.

[65] Laurent Mounier, Ludovic Samper, and Wassim Znaidi. Worst-case lifetime computation of a wireless sensor network by model-checking. In *Fourth ACM Workshop on Performance Evaluation of Wireless Ad Hoc, Sensor, and Ubiquitous Networks (PE-WASUN)*, Chania, Crete Island, Greece, octobre 2007. ACM.

[66] The Network Simulator - ns-2.
http://www.isi.edu/nsnam/ns/.

[67] Peter C. Ölveczky and Stian Thorvaldsen. Formal modeling and analysis of wireless sensor network algorithms in real-time maude. In *The 14th International Workshop on Parallel and Distributed Real- Time Systems*, 2006.

[68] Peter Csaba Ölveczky and José Meseguer. Semantics and pragmatics of real-time maude. *Higher-Order and Symbolic Computation*, 20(1-2) :161–196, 2007.

[69] Sung Park, Andreas Savvides, and Mani B. Srivastava. Sensorsim : a simulation framework for sensor networks. In *MSWIM '00 : Proceedings of the 3rd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, pages 104–111, New York, NY, USA, 2000. ACM Press.

[70] Simon Perathoner, Ernesto Wandeler, Lothar Thiele, Arne Hamann, Simon Schliecker, Rafik Henia, Razvan Racu, Rolf Ernst, and Michael González Harbour. Influence of different system abstractions on the performance analysis of distributed real-time systems. In *ACM Conference on Embedded Software (EMSOFT)*, Salzburg, Austria, October 2007. ACM Press.

[71] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[72] Jonathan Polley, Dionysys Blazakis, Jonathan McGee, Dan Rusk, and John S. Baras. ATEMU : A Fine-grained Sensor Network Simulator. *Secon*, 2004.

[73] Pascal Raymond, Erwan Jahier, and Yvan Roux. Describing and executing random reactive systems. In *SEFM '06 : Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 216–225, Washington, DC, USA, 2006. IEEE Computer Society.

[74] Pascal Raymond and Yvan Roux. Describing non-deterministic reactive systems by means of regular expressions. In *Synchronous Languages, Applications, and Programming (SLAP'02)*, volume 65.5. Electronic Notes in Theoretical Computer Science, 2002.

[75] George F. Riley. The georgia tech network simulator. In *MoMeTools '03 : Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, pages 5–12, New York, NY, USA, 2003. ACM Press.

[76] Yvan Roux. *Description et simulation de systèmes réactifs non-déterministes*. Thèse de doctorat, Institut National Polytechnique de Grenoble, march 2004.

[77] Ludovic Samper, Florence Maraninchi, Laurent Mounier, Erwan Jahier, and Pascal Raymond. On the importance of modeling the environment when analyzing sensor networks. In *Proceedings of International Workshop on Wireless Ad-Hoc Networks 2006 (IWWAN 2006)*, page 7, New York, United States, June 2006.

[78] Ludovic Samper, Florence Maraninchi, Laurent Mounier, and Louis Mandel. GLONEMO : Global and accurate formal models for the analysis of ad-hoc sensor networks. In *Proceedings of the First*

International Conference on Integrated Internet Ad hoc and Sensor Networks (InterSense'06), Nice, France, May 2006.

[79] Victor Shnayder, Mark Hempstead, Bor rong Chen, Geoff Werner Allen, and Matt Welsh. Simulating the power consumption of large-scale sensor network applications. In *SynSys*, pages 188–200, 2004.

[80] The hybrid simulation programming language. http://www.path.berkeley.edu/shift.

[81] Ahmed Sobeih, Wei-Peng Chen, Jennifer C. Hou, Lu-Chuan Kung, Ning Li, Hyuk Lim, Hung-Ying Tyan, and Honghai Zhang. J-sim : A simulation environment for wireless sensor networks. In *Annual Simulation Symposium*, pages 175–187, 2005.

[82] Avinash Sridharan, Marco Zuniga, and Bhaskar Krishnamachari. Integrating environment simulators with network simulators. Technical report, University of Southern California, 2004.

[83] TinyOS Home Page. http://www.tinyos.net.

[84] Ben L Titzer, Daniel K Lee, and Jens Palsberg. Avrora : Scalable Sensor Network Simulation with Precise Timing. *Proceedings of IPSN*, 2005.

[85] Fouad A. Tobagi and Leonard Kleinrock. Packet switching in radio channels : Part ii–the hidden terminal problem in carrier sense multiple-access and the busy-tone solution. In *IEEE Transactions on Communications*, volume 23, pages 1417–1433, december 1975.

[86] Uppaal home page. http://www.uppaal.com/.

[87] Verimag, équipe DCS. IF. http://www-verimag.imag.fr/~async/IF/.

[88] Thomas Watteyne, Isabelle Augé-Blum, and Stéphane Ubéda. Dual-mode real-time mac protocol for wireless sensor networks : a validation/simulation approach. In *Proceedings of the first international conference on Integrated internet ad hoc and sensor networks (InterSense'06)*, pages 2–8, Nice, France, May 2006. ACM Press.

[89] Wei Ye, John S. Heidemann, and Deborah Estrin. Medium access control with coordinated adaptive sleeping for wireless sensor networks. *IEEE/ACM Transaction on Networking (TON)*, 12(3) :493–506, 2004.

Résumé

Les travaux présentés dans ce document portent sur la modélisation de réseaux de capteurs. Les réseaux de capteurs sont difficiles à concevoir parce qu'ils sont fortement contraints en énergie et que tous les éléments ont potentiellement une influence sur la durée de vie du système. Nous proposons de construire des modèles qui puissent être analysés.

Nous présentons d'abord une méthode de modélisation probabiliste qui nous a permis de comparer les performances en énergie et fiabilité de plusieurs variantes d'un protocole MAC. Cependant, ce modèle n'étant pas global, il ne convient pas pour estimer la durée de vie du système complet.

Nous proposons alors un modèle global et précis d'un réseau de capteurs. Nous avons conçu GLONEMO grâce à des modèles de chacun des composants. Programmé en REACTIVEML, cet outil exécutable permet de simuler des réseaux de grande taille tout en estimant finement la consommation de chacun des nœuds. Pour que les simulations soient réalistes, un composant modélise l'environnement. Même si GLONEMO bénéficie d'une sémantique formellement définie, la taille des modèles exclue la possibilité d'effectuer des analyses exhaustives. Pour cela, il faut des modèles plus abstraits.

Nous proposons donc une modélisation globale mais simplifiée d'un réseau de capteurs. Grâce à la boîte à outils IF, nous calculons une durée de vie pire-cas pour ce système. Cependant, nous ne pouvons rien conclure concernant la durée de vie du système détaillé.

Pour être sûr que les propriétés prouvées sur un modèle abstrait sont vraies sur le modèle détaillé, il faut que les abstractions soient conservatives. De plus, comme nous construisons nos modèles à partir de composants, il faut que la composition préserve la relation d'abstraction. Nous définissons une relation d'abstraction de modèles de consommation qui répond à ces besoins.

Mots clefs : Réseaux de capteurs, modélisation, simulation, vérification formelle, protocole MAC, model-checking, économie d'énergie.

Abstract

This work deals with the modeling of wireless sensor networks. In those systems, the main issue is energy consumption and as each element may have an influence on the lifetime, wireless sensor networks are hard to develop. We propose to build models that can be analyzed.

We first present a probability based modeling method that we used to compare reliability and energy consumption of several MAC protocols. However, this model cannot be used to evaluate the lifetime of the network because it is not global.

Then we propose a global and precise sensor network model. GLONEMO is a component based model. Implemented in REACTIVEML, this executable tool can be used to simulate large scale networks while evaluating precisely energy consumption. In order to have realistic simulations, an environment model is included. Even if GLONEMO semantic is formally-defined, exhaustive analyses are hopeless because of the size of the models. That is why we need abstract models.

We propose a global but simplified sensor network model. Using IF toolbox, we compute the worst-case lifetime of this system. However, we cannot conclude anything about the lifetime of the detailed model.

To be sure that properties we proved on an abstract model are still true on the detailed one, we need conservative abstractions. Moreover, as we build our models using components, composition must preserve the abstraction relation. We define such an abstraction relation for consumption models.

Keywords : Wireless sensor networks, modeling, simulation, formal verification, MAC protocols, model-checking, power-aware systems.