UNIVERSITE MOHAMMED V-AGDAL
FACULTE DES SCIENCES
Rabat

N° d'ordre 2669

THESE DE DOCTORAT

Présentée par

BEN MAISSA Yann

Discipline : **Sciences de l'Ingénieur**
Spécialité : **Informatique et Télécommunications**

Contribution à la modélisation et à la vérification de réseaux de capteurs sans fil

Soutenue le 21 Septembre 2013

Devant le jury

Président :

Driss ABOUTAJDINE : PES à la Faculté des Sciences de Rabat

Examinateurs :

Fabrice KORDON : PES, Université Pierre et Marie Curie, Paris, France
Laure PETRUCCI : PES, Université de Paris 13, Paris, France
Salma MOULINE : PH, Faculté des Sciences, Rabat
Youssef FAKHRI : PH, Université Ibn Tofaïl, Kénitra
Yann THIERRY-MIEG : MC, Université Pierre et Marie Curie, Paris, France

AVANT-PROPOS

Les travaux présentés dans le mémoire ont été effectués au sein du Laboratoire de Recherche en Informatique et Télécommunications à la Faculté des Sciences de Rabat.

Cette thèse a été préparée en cotutelle internationale avec l'Université Pierre et Marie Curie (Laboratoire d'Informatique de Paris 6/ Equipe MoVe). Cette **cotutelle** internationale constitue une **première** dans le domaine des sciences formelles entre l'Université Mohammed V-Agdal (Rabat) et l'Université Pierre et Marie Curie (Paris). J'espère qu'elle constituera un **point de départ** pour une fructueuse collaboration scientifique entre les deux Universités, et un encouragement pour la coopération entre les deux pays amis : le Maroc et la France.

C'est un honneur pour moi d'avoir eu comme directeurs de recherche les Professeurs Driss ABOUTAJDINE (Faculté des Sciences de Rabat) et Fabrice KORDON (Université Pierre et Marie Curie). Je les remercie infiniment de m'avoir permis de préparer cette thèse dans le cadre d'une **cotutelle** internationale. J'ai bénéficié de leurs précieux conseils, et d'un soutien **sans faille** de leur part, sur les plans académiques et administratifs. Je voudrais particulièrement remercier le Professeur Fabrice KORDON de m'avoir permis d'effectuer des séjours, en France, plus longs que ne le prévoyait la convention de cotutelle. Cela m'a permis d'acquérir des compétences supplémentaires, et de bénéficier, quotidiennement, de ses orientations et remarques.

Je suis honoré que le Professeur Driss ABOUTAJDINE, ait accepté d'être le président du jury de cette thèse de Doctorat.

Je tiens à remercier les membres du jury de thèse pour l'intérêt qu'ils ont porté à ce travail, à sa lecture et au jugement de sa valeur scientifique.

Monsieur Youssef FAKHRI, Professeur habilité à l'Université Ibn Tofaïl (Kénitra) a pris de son temps de repos aoûtien pour juger ce travail.

Je voudrais également exprimer ma profonde gratitude à Madame Salma MOULINE, Professeur habilitée à la Faculté des Sciences de Rabat pour son encadrement. Elle a suivi ma formation tout au long de mon cursus universitaire : mémoire de Licence, Master et Doctorat.

Madame Laure PETRUCCI, Professeur à l'Université Paris Nord (France) a également pris de son temps de repos aoûtien pour juger ce travail.

Ma reconnaissance va aussi au Docteur Yann THIERRY-MIEG. C'est une chance pour moi qu'il ait accepté de co-encadrer mon travail. J'ai beaucoup appris auprès de lui. Je le remercie infiniment pour tout ce qu'il m'a apporté.

Permettez moi, chers Professeurs, d'exprimer ma reconnaissance à l'Agence Universitaire de la Francophonie et au Centre National pour la Recherche Scientifique et Technique. Les bourses d'excellence qu'ils m'ont accordé ont contribué à améliorer mes conditions de séjour à Paris et Rabat.

Enfin, je ne voudrais pas terminer sans exprimer ma gratitude à mes parents, le Professeur Halima GHAZI et le Professeur Abdessalam BEN MAISSA, pour m'avoir soutenu moralement et matériellement tout au long de cette « aventure ». Je leur dédie ce travail.

RÉSUMÉ

Un réseau de capteurs sans fil est constitué d'un ensemble de nœuds autonomes dotés de capteurs, conçus pour collecter des grandeurs physiques et environnementales. Les réseaux de capteurs sans fil ont de nombreuses applications, comme la surveillance de l'environnement ou le monitoring de l'état de santé d'un patient.

Leur conception est complexe, car soumise à plusieurs contraintes : l'énergie limitée, les problèmes de concurrence, l'hétérogénéité des nœuds, et dans certains cas, le caractère critique. C'est pourquoi, il est nécessaire de vérifier et de valider ces systèmes, pour garantir que ces contraintes sont satisfaites.

Plusieurs approches ont été proposées dans ce sens :

– des études de cas qui modélisent de façon *ad hoc* et vérifient formellement un aspect d'un réseau de capteurs sans fil particulier (*e.g.*, un protocole de communication).
– des *model checkers* de programmes dont le but est de trouver des bugs dans l'implémentation des nœuds.
– des langages de modélisation dédiés au domaine (*Domain Specific Modeling Languages* ou DSMLs) qui fournissent les concepts du domaine et supportent l'analyse ou la génération de code.

Cependant, les études de cas nécessitent une phase d'abstraction manuelle (donc potentiellement génératrice d'erreurs) du réseau par un expert en méthodes formelles et en réseaux de capteurs sans fil. Les *model checkers* de programmes interviennent tard dans le cycle de développement, ce qui augmente le coût, en cas de détection d'erreur. Les DSMLs supportent généralement la simulation ou la génération de code. La simulation ne permet pas de vérifier exhaustivement les comportements du système et peut donc omettre des scénarios problématiques. A ce stade, il n'existe donc pas de solution satisfaisante pour modéliser et vérifier un réseau de capteurs sans fil.

Nous proposons `Verification` of Wireless `Sensor` Networks (VeriSensor) un DSML pour les réseaux de capteurs sans fil, supportant la vérification formelle. VeriSensor est destiné à être utilisé par les experts du domaine en leur fournissant des concepts capturant les cas d'utilisation de ces systèmes : la collecte de données et la détection d'évènements. Il peut être transformé automatiquement en une spécification exprimée dans le langage formel des *Instantiable Transition Systems* (ITS). La vérification des propriétés est faite par *model checking* et le résultat est transmis à l'utilisateur.

Mots-clefs : Vérification formelle, modélisation, réseaux de capteurs sans fil, ingénierie dirigée par les modèles, model checking

Abstract

A Wireless Sensor Network (WSN), made of distributed autonomous nodes, is designed to monitor physical or environmental conditions. Their design is complex and must consider multiple constraints. That is why it is necessary to verify these systems, to guarantee that these constraints are satisfied. Several proposals in that direction have emerged in recent years. We can classify them in the following way :

- case studies using formal verification techniques. While they show thepractical and industrial relevance of performing formal analysis on WSN, they use ad-hoc modeling of the system by experts in both WSN and formal verification ;
- domain specific modeling languages (DSMLs) providing concepts of the domain are also used within the context of model-driven engineering (MDE). These specifications can be simulated prior to code generation of the final system. However, simulation is not sufficient to ensure a high confidence in critical systems.
- Program model-checkers are intended to find bugs in implementations. However, these tools detect problems late in the development life-cycle, since an implementation must already be available.

We propose Verification of wireless Sensor Networks (VeriSensor), a DSML for WSN and its mapping to a formal language for verification and analysis. VeriSensor offers “natural” modeling of a WSN to domain experts by providing high-level concepts that capture the main use cases of such systems – periodic data collection, event-detection, etc.

Keywords : Formal verification ; modeling ; wireless sensor networks ; model driven engineering ; model checking

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLES

CHAPITRE 1

INTRODUCTION GÉNÉRALE

1 Contexte de la thèse

Ce premier chapitre situe ce travail à la croisée des domaines des réseaux de capteurs sans fil, de la vérification formelle et de l'ingénierie dirigée par les modèles (IDM). L'objectif de la thèse est d'augmenter la fiabilité des réseaux de capteurs sans fil à l'aide d'une approche de développement IDM, associée à de la vérification formelle.

1.1 Réseaux de capteurs sans fil

Un réseau de capteurs sans fil est constitué de nœuds autonomes, dotés de capteurs, et répartis dans un environnement [YMG08]. Les nœuds surveillent de manière coopérative des données physiques comme la température, le son ou les vibrations. Chaque nœud est composé d'une unité de capture (ensemble de capteurs), d'une unité de traitement (processeur et mémoire), d'une unité de communication (émetteur/récepteur et antenne) et d'une unité d'alimentation (batterie).

1.1.1 Intérêt

L'intérêt des réseaux de capteurs sans fil réside dans une résolution temporelle et spatiale des mesures sans précédent [SOP+04, BHH+10], ainsi qu'une communication à distance des captures. En effet, la fréquence et la densité spatiale élevées (jusqu'à une centaine de noeuds sur une zone de 10 mètres de diamètre [ASSC02, KFV11]), que peuvent atteindre ces systèmes, permettent d'analyser l'environnement de façon très précise. Cela donne la possibilité de détecter de nouveaux phénomènes physiques (*e.g.*, d'infimes vibrations sismiques) que l'on ne pouvait observer dans le passé [EE04, BHH+10].

L'intérêt de ces systèmes leur permet progressivement de s'imposer face aux technologies concurrentes : les *loggers* et les stations de mesure traditionnelles [MCP+02, HSBH+09]. Un *logger* est un appareil qui enregistre des données de l'environnement qui l'entoure. Il nécessite un déplacement physique pour être positionné dans la zone à surveiller et pour récupérer les données enregistrées. Une station de mesure traditionnelle est un appareil (2 m de hauteur en moyenne) équipé d'instruments de mesure (*e.g.*, anémomètre, thermomètre) pour observer les conditions atmosphériques, afin de capter et de transmettre des informations pour l'étude de l'évolution du climat.

L'utilisation de *loggers* induit un retard dans la consultation des données et, dans certaines applications, une perturbation de l'environnement par l'homme. En effet, sachant que les *loggers* n'ont pas de système de communication, il faut que l'expert réseau se déplace jusqu'à la zone surveillée (*e.g.*, jusqu'au domicile d'un patient, pour les réseaux de surveillance corporelle biomédicaux) pour extraire les grandeurs collectées par les appareils, ce qui induit un délai important pour la consultation des données.

Par ailleurs, le déplacement physique de l'expert dans la zone surveillée peut être impossible (*e.g.*, théâtre d'opérations militaires contenant des mines anti-personnel), ou encore fausser les mesures (*e.g.*, perturbation de l'environnement d'une colonie d'oiseaux dans le cas de la surveillance environnementale [And95]).

L'inconvénient des stations de mesure est le nombre de points de mesure limités. En effet, leur grande taille fait qu'on ne peut en placer plusieurs sur une petite surface. Leur coût élevé limite le nombre de stations dont on peut faire l'acquisition. Quelque soit le domaine d'application (*e.g.*, surveillance environnementale, militaire), la densité spatiale de capture peu élevée réduit les données à la disposition de l'expert nécessaires à la compréhension du phénomène physique surveillé.

En conséquence, les réseaux de capteurs sans fil, par leur capacité de transmission quasiment en temps réel, leur faible perturbation de l'environnement, leur faible coût (quelques dollars jusqu'à quelques centaines de dollars [ASSC02, DP10]) et leur petite taille, s'imposent petit à petit.

Les réseaux de capteurs sans fil sont à l'origine d'une *nouvelle révolution industrielle* [Con06, GH+13] : elle consiste à combiner à très grande échelle les données du monde physique (*e.g.*, la température de fonctionnement d'un appareil) à celles du monde des machines (*e.g.*, la vitesse de production de l'appareil) pour une meilleure surveillance et un meilleur contrôle des systèmes industriels. En effet, les valeurs des grandeurs physiques fournies par les réseaux de capteurs sans fil permettent de comprendre les conditions de fonctionnement de ces machines, qui affectent leur performance (*e.g.*, dilatation des composants métalliques à cause de l'augmentation de la température en été), leur durée de vie et leur fiabilité.

1.1.2 Applications

Parmi les types d'applications phares de ces systèmes, il y a :

1. la surveillance biomédicale ;
2. la surveillance sismique ;
3. la surveillance de déchetteries ;
4. la surveillance environnementale.

La surveillance biomédicale est révolutionnée par les réseaux de capteurs sans fil, car ils permettent de surveiller, quasiment en temps-réel, et à moindre coût (moins de personnel soignant nécessaire), les patients, en dehors de l'hôpital. A une époque où la population vieillissante croît de plus en plus dans le monde et où la plupart des pays sont en rigueur budgétaire, les réseaux de capteurs sans fil ont un rôle majeur à jouer dans les réductions de dépense dans le domaine de la santé.

KNOWME [MEL+12], par exemple, est un projet important d'application des réseaux de capteurs sans fil dans la surveillance biomédicale, développé à la *University of Southern California*. Il s'agit d'un système de surveillance du corps

humain qui intègre des capteurs avec un téléphone cellulaire Nokia N95, pour collecter de manière continue et analyser les signaux biométriques d'un patient. L'objectif est d'étudier l'activité physique de sujets obèses.

Dans le domaine de la surveillance sismique, les experts ne sont pas encore capables de prévoir à l'avance l'occurrence de tremblements de terre. Pour aider à cette prévision, Fischer *et al.* [FRZ+12] ont développé un système d'alerte constitué de sismomètres à bas coût, qui utilisent une méthode coopérative d'analyse des signaux pour distinguer les tremblements de terre (avec un magnitude minimale) des autres mouvements du sol.

La surveillance de déchetteries et en particulier l'infiltration de produits toxiques dans le sol est aussi un phénomène toujours mal maîtrisé. Sa compréhension permettrait d'identifier des situations dangereuses avant qu'elles ne touchent la population (*e.g.* infiltration de produits toxiques dans la nappe phréatique). L'idée consiste à placer des micro-capteurs capables de reconnaître les phases dans des mélanges air/eau/sol, et qui traiteraient le signal de manière collaborative pour concentrer la surveillance (*e.g.* déclencher des algorithmes d'analyse de la teneur en produits chimiques) dans les zones à risque. Parmi les initiatives pour la surveillance du déplacement de produits toxiques dans le sol, il y a le projet *Multi-Scale Soil Sensor Network in Support of Groundwater Quality* [HJS+09] [1], initié par le *Center For Embedded Networked Sensing* (*University of California at Los Angeles*), dont le but est d'identifier et d'observer les utilisations potentielles de l'eau recyclée comme substitut ou supplément à la nappe phréatique.

Concernant la surveillance environnementale, on cherche à étudier le comportement de la faune et de la flore dans le temps. Le problème se pose, en particulier, dans des environnements aux conditions extrêmes, où la température, l'humidité et les obstacles physiques rendent l'accès difficile, voire impossible. Un réseau de capteurs sans fil a été déployé sur une île, dans l'état du Maine, pour étudier le comportement et les conditions optimales de reproduction d'une espèce d'oiseaux spécifique [SMP+04].

1.1.3 Complexité et contraintes de conception

Un réseau de capteurs sans fil est un *système embarqué*. Les deux caractéristiques d'un système embarqué sont le fort couplage entre le matériel et le logiciel, et le fait qu'il soit dédié à une tâche donnée [NSL08]. Ainsi, le matériel du processeur de chaque nœud est optimisé pour réaliser des traitements de signaux complexes. Par ailleurs, les composants de chaque nœud sont dédiés à la tâche de collecte des données : les capteurs sont puissants, le processeur également (pour réaliser le maximum de traitement et communiquer un minimum de données afin de prolonger la durée de vie), les émetteur/récepteurs ont un faible débit pour réduire la consommation d'énergie et prolonger la durée de la collecte dans le temps.

Cependant, la conception de réseaux de capteurs sans fil est *complexe*. Cela est dû, notamment, au fait que ce soient des systèmes cyber-physiques [Lee06] (*cyber physical systems*) caractérisés avec une forte hétérogénéité fonctionnelle. Les systèmes cyber-physiques intègrent, à la fois, des processus physiques (*e.g.*, capture) et calculatoires. Cela pose le problème de gérer les boucles de rétroaction où les

1. http ://research.cens.ucla.edu/contam/2011/contam01.pdf

processus physiques (*e.g.*, montée de température) affectent les calculs et vice versa. L'hétérogénéité fonctionnelle des réseaux de capteurs sans fil est due au fait qu'ils réalisent de la capture, du traitement, et de la communication. Chaque fonction utilise des technologies différentes. Il faut aussi coordonner les différentes fonctions, pour réaliser l'objectif de collecte.

Par ailleurs, le problème de la *fiabilité* de ces systèmes est crucial. La fiabilité est la capacité d'un système à remplir ses fonctions sous des conditions données, pendant une période de temps prédéfinie [Ger91]. Les réseaux de capteurs sans fil sont conçus pour fonctionner durant une longue durée (jusqu'à plusieurs années [ASSC02, DP10]) dans des milieux souvent hostiles, sans intervention de l'homme. Il est donc impossible de "redémarrer" le système manuellement comme dans le cas d'un ordinateur de bureau. L'exigence d'autonomie de ces systèmes impose donc de s'assurer de l'absence d'erreurs.

Les réseaux de capteurs sans fil sont, également, soumis à plusieurs *contraintes* :

1. *la durée de vie limitée du réseau*. C'est une des contraintes les plus importantes (plus que la qualité de service [ASSC02, DP10]). En effet, généralement un nœud ne peut être réalimenté en énergie et ne dispose que de petites batteries à cause de la contrainte de taille.
2. *le caractère asynchrone et concurrent*. Cela peut entrainer de l'entrelacement entre actions, ainsi que des conditions de course. Ces dernières se produisent lorsque différents processus accèdent à une donnée partagée, sans synchronisation explicite [NM92].
3. *l'hétérogénéité*. Les réseaux de capteurs sans fil peuvent contenir différents types de nœuds, chacun avec des caractéristiques différentes (capteurs, portée, capacité de la batterie, etc).
4. *les ressources limitées*. Les nœuds ont des capacités de traitement (CPU) et de stockage (mémoires) limitées, à cause des contraintes de taille, de coût et de faible consommation d'énergie. Cela est d'autant plus crucial que les progrès de la miniaturisation (loi de Moore) sont destinés à être utilisés pour réduire la taille et le coût, plutôt que pour augmenter les capacités de traitement et de stockage.
5. *la réactivité*. Un système est réactif lorsque les traitements qu'il réalise sont déclenchés par des stimuli environnementaux. Un réseau de capteurs sans fil est un système réactif. Par conséquent, son comportement est contraint par les fluctuations des grandeurs environnementales.

1.1.4 Besoin d'analyse

Étant donné ces caractéristiques, il est difficile de garantir à l'expert que le réseau de capteurs sans fil fonctionne comme il le souhaite. Il est donc nécessaire d'analyser le réseau de capteurs sans fil pendant la phase de conception pour s'assurer, en amont du processus de développement, qu'il satisfait sa spécification. Ce besoin d'analyse a été identifié, notamment, par Olveczky [OT07] et Tschirner [TXY08]. L'analyse peut se faire par simulation ou par vérification formelle.

La simulation consiste à vérifier qu'un système satisfait une propriété donnée en observant son comportement sur un sous-ensemble de tous les scénarios possibles. L'inconvénient est que rien n'est garanti sur le comportement du système dans les scénarios non observés.

Nous souhaitons vérifier qu'un système satisfait une propriété donnée en observant son comportement sur tous les scénarios possibles. La vérification formelle explore, exhaustivement, l'espace des états du système et fournit donc une *garantie* au concepteur du système, qu'il satisfait la propriété.

1.2 Vérification formelle

La vérification formelle consiste à construire un modèle du système à l'aide d'un formalisme, à exprimer les propriétés qui doivent être satisfaites par le système à l'aide d'un autre formalisme, puis à effectuer la vérification des propriétés par le modèle à l'aide d'outils mathématiques.

Il existe deux approches pour la vérification formelle : la preuve de théorèmes et le *model checking*.

1.2.1 Preuve de théorèmes

La preuve de théorèmes consiste à exprimer le modèle à l'aide d'axiomes, et les propriétés à vérifier sur le modèle sous la forme de théorèmes. Ces derniers sont vérifiés sur le modèle à l'aide d'un prouveur de théorèmes [HVP+05]

Les calculi utilisés pour la preuve de théorèmes ont été introduits par C.A.R. Hoare en 1969 dans [Hoa69] où il raisonne sur la justesse du programme en terme de *pre conditions* et de *post conditions*.

L'avantage de la preuve de théorèmes est qu'elle peut être utilisée dans le cas de systèmes infinis.

L'inconvénient de la preuve de théorèmes est que les outils la supportant ne sont pas totalement automatiques. Ils nécessitent une *grande expertise* et de l'effort de la part de l'utilisateur. C'est le frein le plus important à leur adoption par l'industrie. La preuve de théorèmes est principalement utilisée dans la vérification de systèmes matériels.

Parmi les outils qui supportent la preuve de théorèmes, on trouve Coq [BC04] et PVS [ORR+96]. Coq fournit un langage formel pour écrire des définitions mathématiques, des algorithmes et théorèmes, ainsi qu'un environnement pour le développement semi interactif de preuves de théorèmes. PVS est un environnement interactif pour la construction de spécifications formelles et la vérification par preuve de théorèmes. Sa caractéristique est l'intégration d'un langage de spécification expressif et d'algorithmes de preuve de théorèmes puissants.

Dans l'industrie, la preuve de théorèmes (méthode B [ALN+91]) a été utilisée pour vérifier les composants critiques du système *Météor* équipant la ligne 14 du métro parisien [BBFM99].

1.2.2 *Model checking*

Le *model checking* est une technique pour la vérification automatique de systèmes réactifs à états finis. Elle a été introduite par Clarke *et al.* en 1983 [CES83].

Elle consiste à construire un modèle formel du système sous la forme d'un automate à état finis (par exemple produit à l'aide d'un langage de spécification

dédié comme les réseaux de Petri [Pet80] ou les processus séquentiels communicants [Hoa78]). Ensuite, il faut énoncer les propriétés à vérifier sur le modèle, sous la forme de formules logiques, à l'aide d'un langage de spécification de propriétés.

Après cela, l'espace des états accessibles du modèle est généré. Une procédure de recherche efficace est utilisée pour déterminer s'il satisfait les propriétés ou non.

L'avantage du *model checking* est le caractère automatique de la procédure de vérification et, dans certains cas, de la génération de contre-exemples (lorsque la propriété n'est pas vérifiée). La tâche de l'utilisateur est réduite à la modélisation formelle de son système et à l'énoncé de la propriété, ce qui demande une moindre expertise que pour la preuve de théorèmes.

L'inconvénient du *model checking* est le problème de l'explosion combinatoire de l'espace des états du système, dû à la croissance exponentielle de la taille de l'espace d'états d'un système concurrent en fonction du nombre de processus et du nombre de composants par processus [CKNZ12]. Par exemple, si l'on considère un système composé de n processus, chacun possédant m états, alors la composition asynchrone de ces processus possède $m \times n$ états [CKNZ12].

Dans l'industrie, le caractère automatique du *model checking* en fait une approche de vérification formelle populaire. Par exemple, elle a permis de découvrir plusieurs erreurs sur cinq sous-systèmes critiques du logiciel de contrôle de la sonde *Curiosity* envoyée par la NASA sur la planète Mars. Des parties clefs du code multi-threadé ont été analysées pour la détection de conditions de course ou d'interblocages [Hol13].

Dans nos travaux, nous adoptons le *model checking* comme approche de vérification formelle. Cependant, malgré ses avantages, les fondements mathématiques, qui sous-tendent le *model checking*, restent un frein à son adoption dans l'industrie, car elles rendent difficile la courbe d'apprentissage de cette technique par le concepteur du système. Il nous faut donc trouver un moyen de faciliter l'utilisation du *model checking* par cet expert.

1.3 Ingénierie dirigée par les modèles

L'ingénierie dirigée par les modèles (IDM) est une approche de développement logiciel dont le but est d'élever le niveau d'*abstraction* et d'augmenter le degré d'*automatisation* dans le développement logiciel.

L'idée est de réduire l'activité du concepteur à la construction de modèles métier et d'utiliser des transformations automatiques pour passer d'un modèle à l'autre ou pour générer du code [Ken02]. L'intérêt est de faciliter la génération de code, augmenter la qualité logicielle, réduire les coûts, l'effort de développement, et le temps de commercialisation (*time-to-market*).

L'approche IDM est fortement supportée et encouragée par l'industrie. Par exemple, l'OMG[2] a proposé une instanciation de cette approche sous le nom de Model Driven Architecture ™(MDA) [Poo01]. Elle porte sur la spécification de systèmes et l'interopérabilité basée sur l'utilisation de modèles. En MDA, des modèles indépendants de la plateforme (PIM pour *Platform Independent Models*) sont exprimés, initialement, dans un langage de modélisation indépendant de la

2. l'OMG est un consortium international, à but non lucratif, pour le développement de standards industriels informatiques. Il a comme membres des entreprises telles que AT&T, EADS, Ericsson, Hitachi, ou Lockheed Martin. Adresse internet : http ://www.omg.org/

plateforme (*e.g.* UML). Ce modèle est traduit en un modèle spécifique à la plateforme (PSM pour *Platform Specific Model*) en utilisant des règles de transformation.

1.3.1 Langages de modélisation

Un langage de modélisation permet d'exprimer de l'information ou des systèmes dans une structure définie par un ensemble cohérent de règles. Les règles sont utilisées pour interpréter la sémantique des composants dans la structure [3]. Certains langages de modélisation sont dédiés au domaine métier dans lequel le modèle est décrit et sont donc appelés : langages de modélisation dédiés au domaine (*Domain Specific Modeling Languages* ou DSMLs).

Les DSMLs, auxquels nous nous intéressons, formalisent la structure, le comportement et les exigences d'une application liée à un domaine particulier, comme les réseaux de capteurs sans fil ou les systèmes de guidage des avions. Les DSMLs définissent des abstractions qui permettent aux experts du domaine d'exprimer leurs idées, directement, avec des concepts du domaine métier. Cela aplatit la courbe d'apprentissage et permet à l'expert d'exprimer son modèle de manière déclarative [Fow10].

La description d'un DSML contient une syntaxe abstraite, une syntaxe concrète et une description de la sémantique du langage. La syntaxe abstraite peut être définie, à l'aide d'un méta-modèle qui précise les relations entre les concepts du domaine et les contraintes entre ces concepts.

1.3.2 Transformations

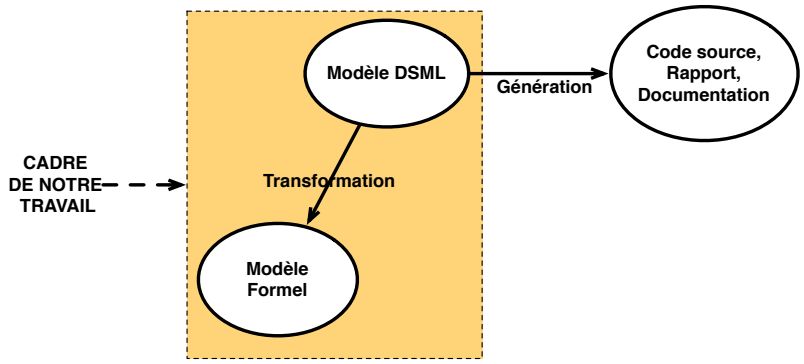


FIGURE 1.1 – Positionnement de notre travail dans l'IDM

Les transformations entre modèles sont réalisées par des moteurs de transformation ou des générateurs de code qui analysent le modèle en entrée et, en respectant des règles de transformation bien définies, synthétisent de manière automatique divers artefacts, comme des modèles ou du code source [Sch06].

Par exemple (voir figure 1.1), un modèle exprimé dans un DSML peut être transformé dans un langage formel. Il est aussi possible de générer du code source ou de la documentation, à partir du DSML.

3. https ://en.wikipedia.org/wiki/Modeling_language

Nous nous intéressons en particulier à la transformation automatique d'un modèle fait à l'aide d'un DSML vers un modèle formel (voir figure 1.1). Cela réduit le travail de l'expert à la spécification de son système et de ce qu'il veut vérifier dessus, sans qu'il ne se préoccupe de la manipulation de la représentation formelle. On résout donc le problème de la difficulté d'appropriation et d'assimilation du *model checking* par les experts des domaines métiers.

2 Objectif de la thèse

L'objectif de la thèse est d'augmenter la fiabilité des réseaux de capteurs sans fil à l'aide d'une approche de développement IDM, associée à de la vérification par *model checking*. Dans notre approche de développement, nous nous appuyons sur les technologies de méta-modélisation qui permettent de réduire l'effort de codage nécessaire (contrairement à l'utilisation directe de langages de programmations généralistes) et sont très bien supportées par des outils (*e.g.*, EMF [Bud04], Atom3 [DLV02], MetaEdit+ [KLR96])

Pour atteindre l'objectif de la thèse, nous proposons de :

(**OBJ$_1$**) Permettre la spécification d'un réseau de capteurs sans fil et des propriétés souhaitées à vérifier dessus, par un expert du domaine.

Pour cela, nous définissons un DSML dédié aux réseaux de capteurs sans fil, nommé `Verification of Wireless Sensor Networks` (VeriSensor). Il fournit des concepts du domaine de connaissance de l'expert tout en cachant les détails des techniques formelles sous-jacentes.

(**OBJ$_2$**) Vérifier par *model checking* que la spécification du réseau de capteurs sans fil satisfait les propriétés.

Dans cette perspective, nous définissons un ensemble de transformations de la spécification VeriSensor vers un modèle formel. A partir du modèle formel, il est possible de vérifier la satisfaction des propriétés.

(**OBJ$_3$**) Garantir la traçabilité des résultats de vérification pour l'expert.

Pour cela, il est nécessaire de définir un ensemble de traductions des résultats de vérification vers un format compréhensible par l'expert. L'objectif est de fournir un retour à l'expert sur la fiabilité de son système. Les verdicts pour chaque propriété doivent être retransmis sous une forme lisible.

3 Plan du mémoire

Le plan du mémoire est structuré de la manière suivante :

Dans le chapitre 2, nous décrivons la problématique en détails. Nous commençons par expliquer pourquoi il existe un besoin d'analyse des réseaux de capteurs sans fil. Ensuite, nous faisons un état de l'art des réseaux de capteurs sans fil à partir duquel nous dégageons les définitions de base du domaine et les principales propriétés que doivent satisfaire ces systèmes. Puis, à partir de notre étude des approches existantes pour la modélisation et la vérification de réseaux de capteurs sans fil, nous justifions la pertinence de notre approche.

Le chapitre 3 présente le langage VeriSensor pour la modélisation des réseaux de capteurs sans fil et illustre sa syntaxe à travers un exemple fil rouge tiré de

la littérature du domaine [OMSJ05]. Le langage est modulaire, hiérarchique et capture les principaux types de comportement des réseaux de capteurs sans fil. Nous détaillons les différents modèles qui composent une spécification VeriSensor. Pour chacun d'entre eux, nous présentons les concepts qui le sous-tendent ainsi que sa syntaxe. Nous présentons également les propriétés supportées par VeriSensor et comment les spécifier.

Dans le chapitre 4, nous détaillons la transformation de VeriSensor vers une spécification formelle à des fins d'analyse par *model checking*. Tout d'abord nous présentons le langage formel cible de la transformation. Ensuite, nous définissons les règles de transformation de VeriSensor vers ce langage et les abstractions qui nous permettent de réduire le risque d'explosion combinatoire du *model checking*. Nous expliquons également comment vérifier les propriétés supportées par VeriSensor sur la spécification formelle. Finalement, nous ajoutons des mesures et des expérimentations sur l'exemple fil rouge afin de valider notre solution.

Le chapitre 5 présente la conclusion générale de notre travail. Il met en regard les objectifs de l'introduction générale avec les contributions détaillées dans le reste de ce manuscrit. Ce chapitre aborde finalement les perspectives à court, moyen et long terme, de notre travail.

4 Publications

Ce travail a donné lieu aux publications suivantes (ordre antichronologique) :

- Y. Ben Maissa, F. Kordon, S. Mouline, Y. Thierry-Mieg : Modeling and Analyzing Wireless Sensor Networks with VeriSensor : an Integrated Workflow, Transactions on Petri Nets and Other Models of Concurrency (ToPNoC), vol. VIII, pp. 24-47 (2013) [BMKMTM13] (`Article de revue`)
- F. Kordon, B. Bérard, Y. Thierry-Mieg, Y. Ben Maissa : Hierarchy is Good For Discrete Time : a Compositional Approach to Discrete Time Verification, Dagstuhl seminar Architecture-Driven Semantic Analysis of Embedded Systemsn^o 12272, Dagstuhl, Allemagne, pp. 38-39 (2012) [KBTMBM12] (`Communication avec actes`).
- Y. Ben Maissa, F. Kordon, S. Mouline, Y. Thierry-Mieg : Modeling and Analyzing Wireless Sensor Networks with VeriSensor, Petri Net and Software Engineering (PNSE), vol. 851, Hambourg, Allemagne, pp. 60-76, (CEUR) (2012) [BMKMTM12](`Communication avec actes`)

Pendant cette thèse, nous avons également publié les travaux suivants (ordre antichronologique) :

- Yann BEN MAISSA et Salma MOULINE : A SysML profile for wireless sensor networks modeling. In The 5th International Symposium on I/V Communications and Mobile Networks (ISIVC'10), Sept. 30 2010-Oct. 2 2010. [BMM10a] (`Communication avec actes`)
- Yann BEN MAISSA et Salma MOULINE : Un langage de modélisation pour réseaux de capteurs sans fil basé sur sysml, In JDTIC'10 , 15-17 Juillet, 2010 [BMM10b]. (`Communication avec actes`)

Positionnement et Problématique

Chapitre 2

1 Introduction

Dans ce chapitre, nous faisons un état de l'art du domaine des réseaux de capteurs sans fil. Nous y présentons les définitions de base concernant la structure du réseau, ses classes de comportement, son hétérogénéité, son déploiement et les propriétés qu'il doit vérifier. Nous nous appuyons sur ces définitions pour exposer dans le chapitre suivant, notre langage VeriSensor.

Ensuite, nous présentons une classification des approches pour la modélisation et l'analyse de ces systèmes en discutant leurs forces et faiblesses.

A partir de cela, nous présentons nos contributions et justifions leur pertinence par rapport aux approches détaillées précédemment.

2 Les réseaux de capteurs sans fil

Il existe plusieurs définitions d'un réseau de capteurs sans fil [ASSC02], [SMZ07], [DP10]. Nous adoptons la définition de Sohraby et al. énoncée dans [SMZ07] pour sa généralité.

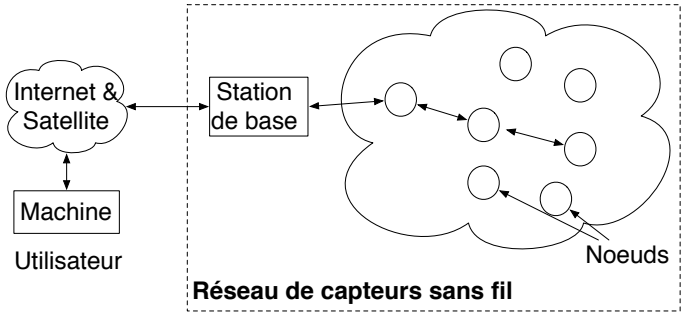


FIGURE 2.1 – Architecture d'un réseau de capteurs sans fil

Définition 2.1. ***Définition d'un réseau de capteurs sans fil*** *[SMZ07]*

Un réseau de capteurs sans fil (voir figure 2.1) est une infrastructure constituée d'éléments de mesure, de traitement et de communication qui permettent à l'administrateur d'observer et de réagir à des événements dans un certain environnement.

Un réseau de capteurs sans fil est constitué de quatre composants :

1. *Un ensemble de nœuds contenant des capteurs.*
2. *Un réseau d'interconnexion sans fil.*
3. *Une station de base pour regrouper l'information et qui agit comme une interface entre les nœuds et l'utilisateur.*
4. *Des ressources de traitement au niveau de la station de base pour gérer la corrélation de données, le data mining, et les requêtes de statut.*

La station de base [1] possède une énergie non limitée et une puissance de calcul importante. Elle connecte les nœuds, à travers Internet ou par satellite, à l'ordinateur de l'utilisateur (voir figure 2.1). Nous nous intéressons uniquement à la station de base, aux nœuds et au réseau d'interconnexion (encadré en pointillés de la figure 2.1) car c'est la partie la plus complexe (communication entre les nœuds) et la plus contrainte (ressources limitées des nœuds, hétérogénéité etc.)

Les quatre composants d'un réseau de capteurs sans fil permettent à un administrateur de l'instrumenter pour capter, traiter ou communiquer les données.

Chaque application a des exigences particulières. La spécificité des réseaux de capteurs sans fil est d'ailleurs leur caractère dédié à l'application (*application-specific*). Cependant, nous avons isolé le noyau d'exigences communes à toutes ces applications.

2.1 Exigences principales des réseaux de capteurs sans fil

Les deux exigences fondamentales des réseaux de capteurs sans fil sont :

Le coût du nœud Un faible coût est exigé pour permettre un déploiement de réseaux à forte densité.

L'autonomie de fonctionnement Le réseau doit fonctionner pendant la durée requise, malgré les pannes de nœuds, et sans possibilité de ré-alimentation en énergie. En effet, généralement les nœuds sont placés dans des environnements hostiles (*e.g.*, à proximité d'un volcan [WALR+06]) où l'homme ne peut accéder.

A partir des exigences de coût et d'autonomie de fonctionnement, on peut déduire les exigences suivantes :

La faible consommation d'énergie Les nœuds contiennent une quantité limitée d'énergie, qui ne peut généralement pas être renouvelée. Par conséquent, la spécificité des réseaux de capteurs sans fil par rapport aux autres types de réseaux réside dans *la priorité* de la contrainte énergétique par rapport aux contraintes de qualité de service (QoS) [ASSC02]. La qualité de service étant la capacité de fournir différentes priorités à différentes applications, utilisateurs, ou flux de données, ou à garantir un certain niveau de performance d'un flux de données [KT11] (*e.g.* taux de pertes des paquets entre la source et le destinataire, débit de transmission).

1. Dans la littérature, il existe certaines applications particulières de réseaux de capteurs sans fil avec plusieurs stations de base. Nous pouvons ramener ces applications à notre définition en considérant qu'il s'agit en fait de plusieurs réseaux de capteurs sans fil, chacun possédant sa propre station de base.

On autorise, par exemple, la réduction du débit de transmission pour prolonger la durée de vie.

Il existe plusieurs définitions de la durée de vie d'un réseau de capteurs sans fil dans la littérature. Par exemple :

- la durée pendant laquelle la proportion des nœuds vivants est au moins égale à v ($v = k/n$ avec n le nombre de nœuds du réseau, $k \in \mathbb{N}$ et $0 < k \leq n$) [CE04, DHHV05, DML02]. La notion de « nœud vivant » n'est pas précisée. On suppose qu'un nœud est vivant s'il possède assez d'énergie pour communiquer (*i.e.*, émettre et recevoir). En effet, si un nœud ne peut communiquer, il est isolé du réseau et n'a plus aucun rôle dans son comportement.
- la durée pendant laquelle une proportion des nœuds au moins égale à c possède un chemin de connexion vers la station de base [CGVC06]. On dit qu'un nœud n_0 dispose d'un chemin de connexion vers la station de base à un instant t, s'il est vivant, non dormant (*i.e.* pas dans l'état *sleep*) et qu'il existe une suite de nœuds $(n_i)_{0 \leq i \leq m}$ vivants et non dormants telle que :
  - il existe un lien de communication entre n_i et n_{i+1} à l'instant t pour tout i tel que $0 \leq i \leq (m-1)$
  - il existe un lien de communication entre n_m et la station de base à l'instant t.

La résistance aux pannes (*fault-tolerance*) La résistance aux pannes d'un système informatique est sa capacité à fournir, par redondance, un service satisfaisant la spécification, malgré les pannes survenues [Lap85].

Dans le cas d'un réseau de capteurs sans fil, la résistance aux pannes désigne la capacité du réseau à fonctionner sans interruption, malgré une panne de nœuds [ASSC02].

L'auto-configuration La capacité des nœuds à détecter la présence d'autres nœuds et à s'organiser en un réseau structuré et fonctionnel sans intervention de l'homme [Cal03].

L'auto-reconfiguration La capacité du réseau à modifier sa configuration pour atteindre un objectif donné. Les principaux objectifs consistent à reconstruire les chemins de connexion entre les nœuds vivants après une panne de noeud ou à réduire la consommation d'énergie d'un groupe de nœuds. [Cal03] L'auto-reconfiguration pour un réseau de capteurs sans fil inclut donc la résistance aux pannes.

Il existe d'autres exigences qui ne sont pas liées directement à l'autonomie de fonctionnement telles que le passage à l'échelle (*scalability*) qui désigne la capacité du réseau à continuer de fonctionner avec les mêmes performances malgré l'ajout d'autres nœuds [Cal03]. Il y a des exigences moins importantes comme les exigences de qualité de service.

Par ailleurs, en fonction de l'application considérée (*e.g.*, surveillance environnementale, détection d'intrusions) s'ajoutent des exigences spécifiques à cette application. En effet, une spécificité des réseaux de capteurs sans fil par rapport aux autres réseaux est que leur conception est déterminée par l'application. Par exemple, pour une application qui exige une durée de vie très importante pour tous les nœuds, on définira une distance moyenne entre deux nœuds faible pour réduire l'énergie nécessaire à l'envoi d'un message.

Les exigences précédentes sont différentes de celles qui pèsent sur les réseaux en général et font la spécificité des réseaux de capteurs sans fil. L'exigence de coût et d'autonomie de fonctionnement sont centrales. Pour y répondre, il faut donc adapter la structure de chaque nœud du réseau, et celle du réseau entier.

2.2 Structure d'un réseau de capteurs sans fil

Dans cette sous-section, nous adoptons une approche ascendante et présentons la structure d'un nœud (ces composants obligatoires et facultatifs), puis celle d'un réseau de capteurs sans fil.

2.2.1 Structure du nœud

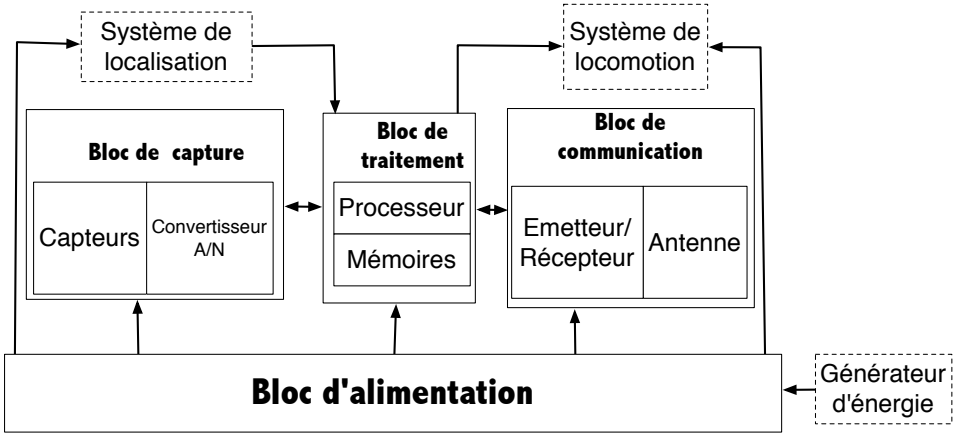


FIGURE 2.2 – Structure d'un nœud

Le nœud (*sensor node*) constitue la « brique de base » du réseau de capteurs sans fil. La définition 2.2 présente sa structure.

Définition 2.2. ***Structure d'un nœud*** *[ASSC02], [SMZ07]*

Un nœud est constitué de composants obligatoires et optionnels.

Les composants obligatoires sont :

1. *Un bloc de capture*
2. *Un bloc de traitement et de stockage temporaire*
3. *Un bloc de communication*
4. *Un bloc d'alimentation*

Les composants optionnels sont :

1. *Un système de localisation*
2. *Un système de locomotion*
3. *Un générateur d'énergie*

La Figure 2.2 représente la structure d'un nœud. Les composants obligatoires sont en trait plein.

Le bloc de capture constitué de capteurs de grandeurs physiques (*e.g.*, accélération, l'humidité, la lumière, la température) et d'un convertisseur analogique/numérique. Un capteur est un appareil qui mesure une grandeur physique et la convertit en un signal lisible par l'utilisateur.

Le bloc de traitement et de stockage temporaire constitué d'un processeur (ou un micro contrôleur) et de mémoires. Ce bloc contient le système d'exploitation et l'application qui s'exécute sur le nœud. La taille de la RAM ne dépasse pas les 10 Ko et celle de la ROM les 100 Ko. On peut trouver en plus de ces deux types de mémoire de la mémoire Flash ou EEPROM.

Le bloc de communication constitué d'un émetteur/récepteur et d'une antenne.

Le bloc d'alimentation constitué généralement d'une batterie.

Les composants optionnels sont (en pointillés dans la figure 2.2) :

Le système de localisation permet de connaître la position du nœud (*e.g.*, GPS).

Le système de locomotion permet au nœud de se déplacer.

Le générateur d'énergie fournit de l'énergie au bloc alimentation (une cellule photovoltaïque, par exemple).

La taille d'un nœud varie approximativement de celle d'une pièce de monnaie, à celle d'une boîte à chaussure [RM04].

Parmi les composants obligatoires, le bloc de capture contient des capteurs qui permet de connecter le réseau au monde physique. Ci-dessous nous présentons les caractéristiques principales d'un capteur.

Définition 2.3. ***Caractéristiques d'un capteur***

Un capteur est un appareil qui mesure une quantité physique et la convertit en un signal lisible par l'utilisateur. Les caractéristiques principales d'un capteur sont :

1. *La grandeur captée (e.g., température).*
2. *La portée de capture : c'est la distance jusqu'à laquelle il est capable de capter la grandeur.*
3. *Le temps de démarrage (startup time) : c'est le temps nécessaire à la stabilisation des mesures du capteur après sa mise sous tension [MCP+02]. Le temps de démarrage peut varier de quelques millisecondes à plusieurs minutes.*

A partir de la portée on détermine la zone de capture (un disque dont le centre est le capteur et le rayon, la portée). L'union des zones de captures des capteurs d'une grandeur g, appartenant à des nœuds non dormants (pas dans l'état *sleep*) constitue la zone de couverture de la grandeur.

2.2.2 Structure du réseau de capteurs sans fil

La structure d'un réseau de capteurs sans fil est déduite de la définition 2.1 que nous avons adoptée.

Définition 2.4. ***Structure d'un réseau de capteurs sans fil***
La structure d'un réseau de capteurs sans fil est composée d'un ensemble de nœuds et d'une station de base.

Dans la littérature, plusieurs structures possibles d'un réseau de capteurs sans fil sont présentées.

- Dans [MCP+02], le réseau est constitué de plusieurs groupes de nœuds (appelés *sensor patch*) reliés par une passerelle (*gateway*) à un réseau appelé réseau de transit. Ce dernier est connecté à une station de base.
- Dans [ASSC02], la structure est plus simple : un ensemble de nœuds connectés à une station de base.
- Dans [BCPR04], le réseau est formé de nœuds directement reliés à un ordinateur portable.

Cependant, dans tous ces exemples on peut se ramener à la définition 2.4. En effet, pour le premier exemple, l'utilisation des passerelles à la place de stations de base est due uniquement au coût moins élevé [MCP+02]. On peut donc considérer que le réseau est en fait un ensemble de réseaux de capteurs sans fil, chacun constitués de nœuds et d'une station de base. Ces réseaux communiquent les données à une station de base de plus forte puissance qui émet par satellite.

Pour le troisième exemple, l'ordinateur portable joue le rôle de station de base, puisqu'il regroupe l'information et agit comme une interface entre les nœuds.

Après avoir décrit la structure du réseau, nous expliquons ce qu'est l'hétérogénéité dans un réseau de capteurs sans fil et son impact sur sa structure et son comportement.

2.3 Hétérogénéité dans les réseaux de capteurs sans fil

L'hétérogénéité dans un réseau de capteurs sans fil consiste à avoir des nœuds dont les caractéristiques structurelles et comportementales sont différentes.

La définition suivante détaille les types d'hétérogénéité dans un réseau de capteurs sans fil.

Définition 2.5. ***Types d'hétérogénéité***
Les types d'hétérogénéité dans un réseau de capteurs sans fil sont :

L'hétérogénéité de capture *les types, ou le nombre de capteurs varie d'un nœud à un autre.*

L'hétérogénéité de traitement *l'application ou les ressources de traitement varient d'un nœud à un autre*

L'hétérogénéité de communication *les protocoles de communication varient d'un nœud à un autre.*

L'hétérogénéité d'énergie *la quantité initiale d'énergie, ou le modèle de consommation d'énergie varie d'un nœud à un autre.*

Les efforts initiaux de recherche se sont uniquement intéressés à des réseaux avec des nœuds aux caractéristiques similaires

L'avantage est que ces réseaux sont résistants aux pannes individuelles de nœuds. Cependant, récemment, l'intérêt s'est porté sur les réseaux hétérogènes qui permettent d'augmenter la durée de vie du réseau, sans en augmenter le coût.

Typiquement, un réseau hétérogène est constitué de deux groupes de nœuds : un premier groupe de nœuds à faible coût qui réalisent la capture de grandeurs ; un deuxième groupe de nœuds avec des capacités de traitement et de communication plus importantes (plus coûteux) traitent la donnée et la transportent. Le partage des tâches réduit le coût de conception et augmente la durée de vie, par rapport à une solution totalement homogène.

Dans ce qui suit, nous expliquons le comportement de chaque composant détaillé dans la section structure et qui participe à réaliser la tâche du réseau.

2.4 Comportement d'un réseau de capteurs sans fil

Dans cette sous-section, nous détaillons le comportement de chacun des différents aspects d'un réseau de capteurs sans fil.

2.4.1 Capture

Chaque capteur est entièrement commandé par l'application.

Nous présentons dans la définition 2.6 le processus suivi par le capteur pour faire l'acquisition des échantillons.

Définition 2.6. ***Étapes du processus de capture***

Le processus de capture suit les trois étapes suivantes :

1. *Si le capteur est éteint, il a attend une durée égale au startup-time avant d'être prêt à fonctionner.*
2. *Si le capteur est allumé, il peut recevoir un ordre de capture de la part de l'application. L'ordre peut :*
   - *Soit commander la capture d'un seul échantillon du capteur.*
   - *Soit commander la capture de n ($n > 1$) échantillons du capteur avec une fréquence d'échantillonnage f.*
3. - *Si l'ordre commande de capter un seul échantillon, alors, le capteur fait l'acquisition d'un échantillon. Ensuite, il le transmet à l'application*
   - *Si l'ordre commande de capter une série de n échantillons avec une fréquence d'échantillonnage f, alors, toutes les $1/f$ secondes, il capte un échantillon et l'envoie à l'application.*

2.4.2 Objectif applicatif d'un réseau de capteurs sans fil

Comme indiqué en sous-section 2.1 (page 12), l'une des spécificités des réseaux de capteurs sans fil est leur caractère dédié à l'application. C'est pourquoi il est difficile

de définir un ou plusieurs comportements applicatifs génériques caractéristiques de ces réseaux.

Cependant, après une étude des applications de réseaux de capteurs sans fil, nous avons identifié deux grandes familles de comportement qui regroupent tous les comportements applicatifs possibles :

la collecte de données (*data collection*) le réseau capte périodiquement des grandeurs environnementales et les envoie à la station de base.

la détection d'événements (*event detection*) le réseau surveille l'environnement à la recherche d'une modification de l'état de l'environnement (événement) et communique, le cas échéant, une alerte à la station de base.

Collecte de données. L'objectif de la collecte de données est de reconstruire les observations d'un phénomène distribué (*e.g.*, la température) avec une forte fidélité spatiale et temporelle. Il s'agit de connaître la distribution dans le temps et dans l'espace d'une ou de plusieurs grandeurs. Parmi les métriques de performance pour cette famille d'applications, on trouve : la précision de la reconstruction du signal, la corrélation entre le signal capté et le signal réel, et la durée de vie du réseau.

Dans la définition 2.7, nous présentons le pattern de comportement que nous avons dégagé pour la collecte de données.

Définition 2.7. ***Etapes de la Collecte de données***

La collecte de données suit les quatre étapes suivantes :

1. *Spécification des tâches :*
   - *Soit la station de base diffuse dans le réseau des requêtes spécifiant les tâches à réaliser par les nœuds.*
   - *Soit ces tâches sont préprogrammées dans les nœuds.*

   Une requête R spécifie une liste de nœuds destinataires, un ensemble de grandeurs physiques à capter et les paramètres suivants :
   - *la période de capture R_c de ces grandeurs (appelée* epoch *[MSFC02])*
   - *la période de traitement des échantillons par l'application R_t ($R_t \geq R_c$)*
   - *le traitement à réaliser*
   - *une période d'envoi des échantillons traités par le nœud R_e ($R_e \geq R_t$)*
   - *l'instant où débute l'exécution de la requête R_D*
   - *l'instant de fin d'exécution de la requête R_F*

   Après l'écoulement de chaque période R_t on traite les échantillons captés pendant cette période. De même, après l'écoulement de chaque période R_e on envoie les échantillons traités pendant cette période.
2. *Chaque nœud exécute les requêtes suivant leurs paramètres.*
3. *Chaque envoi de résultat peut se faire en plusieurs sauts (multi-hop) ou en un seul saut (single-hop). Dans le cas où l'envoi est multi-hop, chaque nœud du chemin peut faire de l'agrégation [KEW02, IGE00] de données entre plusieurs résultats qu'il doit retransmettre. L'agrégation consiste à réaliser un traitement sur les données pour réduire leurs taille (e.g., calculer la moyenne de plusieurs données).*
4. *La station de base reçoit les résultats.*

Les paramètres que nous avons définis dans la première étape permettent d'exprimer la majorité des applications de collecte de données. En fixant par exemple R_c, R_e et R_t à la même valeur, on obtient le cas particulier qui consiste à capter, puis traiter et envoyer immédiatement.

Le but de l'agrégation en étape 3 est de réduire la quantité des données retransmises pour économiser de l'énergie. La perte d'énergie pour l'agrégation des données est largement compensée par le gain lors de la retransmission. En effet, le traitement consomme beaucoup moins d'énergie que la communication. Ainsi, pour 3J d'énergie, on peut soit transmettre 1Kb de données sur une distance de cent mètres, soit exécuter trois millions d'instructions avec un processeur de puissance modeste (100 MIPS/W) [SGAP00].

Les périodes de capture (voir étape 1) pour les applications de collecte périodique sont supérieures à celles des applications de détection d'événements, parce qu'il n'y pas le risque de « manquer » un événement de durée extrêmement courte.

Détection d'événements. La détection d'événements vise à la détection d'un changement d'état de l'environnement. Les exigences pour cette famille d'applications impliquent une architecture de capture et de traitement différente de la collecte de données. Les métriques de performances associées à cette famille d'applications, comprennent les probabilités de détection, de fausses alertes, de classification ou mauvaise classification, la latence de détection et la durée de vie du système.

Dans la définition 2.7, nous présentons le pattern de comportement que nous avons dégagé pour la détection d'événements.

Définition 2.8. ***Etapes de la détection d'événements***

La détection d'événements est constituée des quatre étapes suivantes :

1. *Spécification des tâches :*
   - *Soit la station de base diffuse, dans le réseau, des requêtes spécifiant les tâches à réaliser par les nœuds.*
   - *Soit ces tâches sont préprogrammées dans les nœuds.*

   Une requête R spécifie une liste de nœuds destinataires, une liste de grandeurs à capter et les paramètres suivants :
   - *la période de capture R_c de ces grandeurs (appelée* epoch *[MSFC02])*
   - *la caractérisation de l'événement (condition sur les grandeurs ou sur un vecteur de caractéristiques déduit à partir de ces grandeurs).*
   - *l'instant où débute l'exécution de la requête R_D*
   - *l'instant de fin d'exécution de la requête. requête R_F*
2. *les nœuds exécutent les requêtes : à chaque capture, si la caractérisation de l'événement est satisfaite, une alerte est envoyée à la station de base.*
3. *Dans le cas d'une transmission en plusieurs sauts des alertes, un nœud peut être amené à retransmettre plusieurs alertes. Il peut être amené à faire de la fusion de données : à partir de plusieurs alertes reçues, à décider si un événement s'est réellement produit.*
4. *La station de base reçoit les résultats.*

La notion de vecteur de caractéristiques (*feature vector*) décrite dans l'étape 1 désigne un tuple de valeurs déduites à partir de traitements complexes sur les échantillons captés. Il est utilisé pour statuer sur l'occurrence d'un événement dans les cas de classification et de suivi d'objets.

Les périodes de traitement et d'envoi (étape 1) sont généralement absentes des applications de détection d'événement, car l'alerte doit être envoyée le plus tôt possible après la capture. Une faible latence dans la transmission de l'alerte est exigée pour la détection d'événements, contrairement à la collecte périodique de données.

La fusion de données évoquée (étape 3) permet soit de réduire le taux de fausses alertes en confrontant les signaux d'alerte reçus, soit de déduire l'occurrence d'un événement complexe, à partir de l'occurrence d'événements plus simples (*e.g.*, une explosion se traduit par une température supérieure à 80°C, une luminosité supérieure à 500 Lumen et un niveau sonore supérieur à 60 dB).

Le pattern de comportement provient de la synthèse de plusieurs applications de détection d'événements dans la littérature du domaine.

Vu *et al.* [VBL07] définissent un événement comme un changement de l'état du monde (*a change in real-word state*). Ils caractérisent un événement par une fonction booléenne sur des prédicats de la forme : (grandeur physique $opComp$ Valeur), avec $opComp$ un opérateur de comparaison. Un événement atomique contient un seul prédicat (*e.g.*, temperature>100°C). Un événement composite est une combinaison de plusieurs événements atomiques, *e.g.* :

incendie=((temperature>300°C) AND (fumée>100 mg/L))).

Madden *et al.* [MFHH05] définissent un système distribué de traitement de requêtes où chaque requête est sous la forme :

"SELECT <listeDeGrandeurs>
FROM <listeDeTables>
WHERE <conditionSurLesChampsDesTables>
SAMPLE PERIOD <epoque> FOR <dureeTotale> "

Cela signifie que pendant une durée <dureeTotale>, on va capter à chaque <epoque> (exprimée en secondes) un échantillon des grandeurs spécifiées dans <listeDeGrandeurs>. Si l'échantillon satisfait la condition <conditionSurLesChampsDesTables>, alors il sera envoyé à la station de base.

Les application de détection d'événement basées sur la classification ou le suivi d'objet réalisent des traitements plus complexes sur les échantillons captés pour déduire le vecteur de caractéristiques et décider de l'occurrence de l'événement.

La figure 2.3 contient une comparaison des profils énergétiques pour la collecte de données et la détection d'événements. Chaque ligne correspond à une tâche spécifique que réalise le réseau qui adopte la famille d'applications en question.

L'activité principale dans la détection d'événements est la capture de données (*Sample*/*Sense*). La communication ne se produit qu'après la détection d'un événement. De plus, le réseau n'a pas le temps de faire beaucoup de traitements, sachant que l'événement doit être transmis le plus vite possible à la station de base (*e.g.*, compression).

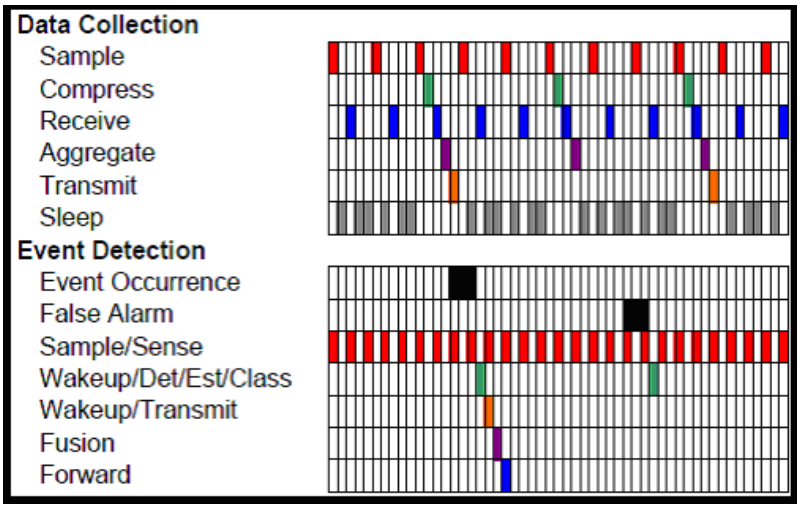


FIGURE 2.3 – Profils énergétiques de la détection d'événements et de la collecte de données [Aro04]

Par contre, pour la collecte de données, la transmission de données est plus importante et plus régulière. Le réseau a aussi le temps de faire plus de traitement puisqu'il n'est pas obligé de transmettre la donnée le plus vite possible.

Dans des situations particulières, la détection d'un événement impliquant des grandeurs $(g_i)_{0\leq i\leq n}$ peut déclencher la capture de grandeurs *différentes* des (g_i). Par exemple, la détection d'une fièvre (grandeur température) chez un patient peut déclencher la capture périodique du rythme cardiaque, ou de l'humidité de la peau.

2.4.3 Communications

La gestion des communications est aussi un aspect du nœud. Il est constitué principalement de deux couches : une couche pour le routage, et une couche pour le contrôle de l'accès au médium (MAC). La couche routage achemine les paquets envoyés par un nœud vers la station de base. La couche MAC permet de partager l'accès des nœuds au canal et garantit que les trames ne contiennent pas d'erreur. Ces deux couches interviennent lors de toutes les phases de fonctionnement du réseau de capteurs sans fil. La définition suivante présente ces phases :

Définition 2.9. ***Phases de fonctionnement d'un réseau de capteurs sans fil***

Nous identifions deux phases de fonctionnement :

1. *L'auto-configuration : le réseau s'organise, à partir de l'instant de déploiement dans l'environnement, en un réseau fonctionnel, sans intervention de l'homme.Cette phase est elle-même constituée de deux phases séquentielles :*
   - *la découverte des voisins.*
   - *la construction des chemins.*
2. *Le fonctionnement nominal : le réseau réalise sa tâche (collecte de données ou détection d'événements) et peut changer à des instants donnés sa configuration*

*(*e.g. *pour remédier à une panne de nœud) Le fonctionnement nominal est donc constitué de deux phases :*

- *l'exécution de la tâche du réseau (collecte ou détection d'événements).*
- *l'auto-reconfiguration : le réseau modifie sa configuration pour atteindre un objectif donné.*

La découverte des voisins consiste pour chaque nœud à déterminer la liste des nœuds qui sont à sa portée de transmission et dont il est à portée de transmission. Chaque nœud prend connaissance de son voisinage immédiat. La couche MAC intervient principalement lors de cette phase

La construction des chemins consiste à déterminer les routes de communication permettant à chaque nœud de communiquer avec des destinataires potentiels (généralement la station de base). La couche routage intervient principalement lors de cette phase.

L'auto-reconfiguration peut intervenir à tout moment du fonctionnement nominal (*e.g.*, suite à une panne de nœud, pour reconstruire les chemins passant par ce nœud).

Nous détaillons dans ce qui suit les mécanismes de fonctionnement du routage et de l'accès au médium.

Routage. Le routage dans un réseau de capteurs sans fil peut être *single-hop* ou *multi-hop* :

- *single-hop* : un nœud transmet ses données directement à la station de base.
- *multi-hop* : Chaque nœud émet ses propres données et joue également le rôle de routeur (route les messages reçus de la part des autres nœuds). Pour remplir ce rôle, la couche routage de chaque nœud du réseau stocke un ensemble de correspondances $\langle destinataire final, chemins \rangle$ (table de routage). Le destinataire final est généralement la station de base[2], mais cela peut aussi être un autre noeud du réseau. Les chemins sont les adresses d'un ou de plusieurs noeuds voisins (deux nœuds n_i et n_j sont voisins si et seulement n_i est à la portée de transmission de n_j et n_j à la portée de n_i). Ces noeuds sont appelés *next-hops*. Tout message reçu par le noeud et destiné au $destinataire final$, est retransmis à un des *next-hops*. Le choix du *next-hop* se fait suivant une politique donnée (*e.g.*, le *next-hop* qui possède le plus d'énergie restante).

Nous détaillons dans la définition suivante, les étapes de la transmission d'un message du nœud source au nœud destinataire final (généralement, la station de base), dans un réseau *multi-hop* :

Définition 2.10. ***Etapes de la transmission dans un réseau multi-hop***
La transmission multi-hop est constituée des cinq étapes suivantes :
1. *Le nœud source n_c construit un message m destiné à un nœud n_{df}.*

2. ou une des stations de base, dans les cas particuliers où il y en a plusieurs

2. *La couche routage de n_c détermine, à l'aide de sa table de routage et du destinataire final du message, les* next-hops *auxquels il faut envoyer le message pour atteindre n_{df}.*
3. *Un des* next-hops *est sélectionné, en fonction d'une politique de choix.*
4. *Le message est transmis au next-hop sélectionné.*
5. *La couche routage du next-hop détermine, à son tour, à partir de l'adresse de destination du message et à l'aide de sa table de routage, les chemins à suivre pour atteindre la destination.*
6. *Les étapes 3 à 5 sont répétées jusqu'à ce que le message soit reçu par le destinataire final (en général, la station de base).*

Dans le cas *multi-hop*, chaque nœud joue donc à la fois le rôle de source de données et de routeur.

L'intérêt du *multi-hop* est de réduire l'énergie consommée par transmission : au lieu d'envoyer le message sur une longue distance (donc une forte consommation d'énergie) à la station de base, il est envoyé à un voisin immédiat du nœud.

Contrôle d'accès au médium. L'objectif des protocoles de couche MAC est d'organiser l'accès au médium partagé de telle sorte à satisfaire les exigences en terme de performance de l'application. Ces protocoles ont donc pour rôle de :

1. gérer l'accès au médium
2. regrouper l'information en trames
3. faire le contrôle d'erreur et réaliser l'adressage.

La couche MAC fait partie de la couche liaison de données dont le but est de cacher le détail du matériel aux protocoles de couche supérieure.

La définition 2.11 présente les deux grandes familles de protocoles MAC :

Définition 2.11. ***Familles de protocoles MAC***

Les deux familles de protocoles MAC sont :

Les protocoles basés sur un programme (*schedule-based protocols*) *Ces protocoles consistent à fixer un programme dans le temps suivant lequel le canal va être alloué à chaque nœud. Chaque nœud attend son tour pour exploiter la ressource canal.*

Les protocoles basés sur la concurrence (*contention-based protocols*) *Ces protocoles laissent les nœuds libres d'accéder au canal quand ils le souhaitent. Cependant, lorsque plusieurs nœuds tentent d'accéder au canal de manière simultanée il se produit une collision et les nœuds doivent faire une autre tentative. Ces protocoles se distinguent par le mécanisme avec lequel ils réduisent la probabilité de collisions tout en minimisant le temps d'écoute du canal (consommateur d'énergie).*

2.4.4 Energie

L'énergie est la contrainte principale des réseaux de capteurs sans fil. La définition suivante présente l'évolution de l'énergie d'un nœud.

Définition 2.12. ***Evolution de l'énergie d'un nœud***
La quantité d'énergie d'un nœud peut décroître à cause des opérations de capture, de traitement ou de communication. Elle peut croître grâce à l'énergie produite par un éventuel générateur d'énergie. La quantité d'énergie ne dépasse jamais une borne supérieure correspondant à la quantité d'énergie maximale stockable dans le bloc d'alimentation.

Une classification des composants d'un nœud en fonction de leur consommation d'énergie a été réalisée : les opérations de communication sont les plus consommatrices, suivies de l'application, puis de la capture [ASSC02]. La tendance actuelle vise à réduire au maximum la taille de la donnée à transmettre par des pré traitements au niveau des nœuds sources et des nœuds intermédiaires (*i.e.*, agrégation, fusion) dans le cas *multi-hop*.

N'importe quel composant du nœud possède plusieurs modes de fonctionnement caractérisés par une puissance énergétique consommée (exprimée usuellement en Watts) différente. La puissance est le taux auquel l'énergie est transférée, utilisée ou transformée. L'énergie (exprimée usuellement en Joules) est obtenue en intégrant une puissance sur un temps de fonctionnement. La définition suivante présente les différents modes de fonctionnement.

Définition 2.13. ***Modes de fonctionnement d'un composant de nœud***
Les modes de fonctionnement de chaque composant du nœud sont typiquement :

1. ***off*** *Le composant est éteint.*
2. ***sleep*** *Le composant est en veille et consomme très peu d'énergie.*
3. ***idle*** *Le composant est allumé mais ne réalise aucune tâche.*
4. ***active*** *Le composant est actif et réalise une tâche (*e.g. *le capteur fait de l'acquisition).*

Les modes de fonctionnement peuvent se décliner en fonction du composant. Par exemple, pour l'émetteur/récepteur le mode *active* contient deux sous-modes : *receiving* (en cours de réception d'un message), *transmitting* (en cours d'émission d'un message). Le mode *transmitting* peut lui même contenir différents modes caractérisés par une puissance de transmission différente. Le mode *idle* pour l'émetteur/réception modélise l'état où l'émetteur/récepteur est allumé mais ne reçoit aucun signal du canal (*idle listening*).

Le fonctionnement du nœud lui même peut être caractérisé par ces modes. En particulier, dans le mode *sleep* il éteint un sous ensemble de ces composants pour consommer moins d'énergie.

Le comportement du réseau varie en fonction de son contexte. En effet, selon son déploiement dans l'environnement (*e.g.* présence d'obstacles, événements fréquents) les interactions entre les nœuds et avec l'environnement changent. Par ailleurs, ce sont les conditions (parfois extrêmes) du déploiement dans l'environnement qui imposent l'exigence centrale d'autonomie de fonctionnement.

2.5 Déploiement d'un réseau de capteurs sans fil

Dans cette sous-section nous expliquons la notion de déploiement d'un réseau de capteurs sans fil dans un environnement à surveiller (*e.g.*, un champ d'opérations militaires). Pour cela, nous définissons la relation du réseau de capteurs sans fil avec son environnement ainsi que sa topologie physique et logique.

La définition 2.14 présente la notion de déploiement d'un réseau de capteurs sans fil qui consiste à placer les nœuds dans l'environnement. Le déploiement est la première étape du fonctionnement du réseau de capteurs sans fil.

Définition 2.14. ***Déploiement d'un réseau de capteurs sans fil***

Le déploiement d'un réseau de capteurs sans fil consiste à affecter à chaque nœud et à la station de base une position dans l'espace.

Le déploiement peut se faire de manière prédéterminée ou aléatoire.

- déploiement prédéterminé : les nœuds sont placés à des positions prédéfinies (*e.g.*, placement individuel de chaque nœud par un être humain ou un robot).
- déploiement aléatoire : les nœuds sont placés à des positions que l'on ne connait pas à l'avance (*e.g.*, largage des nœuds par avion sur la zone à surveiller).

Il est également possible de déployer un réseau de manière incrémentale : plusieurs déploiements d'un sous ensemble des nœuds du réseau à des instants successifs.

2.5.1 Environnement

Un réseau de capteurs sans fil est destiné à fournir des informations à l'expert réseau, sur un environnement donné. La partie de l'environnement qui intéresse l'expert est la zone d'intérêt. La définition 2.15 présente cette notion.

Définition 2.15. ***Zone d'intérêt***

*La zone qui fait l'objet d'une collecte de grandeurs ou d'une surveillance par le réseau est appelée zone d'intérêt. La zone d'intérêt peut être constituée d'un ensemble de points discret ou continu. Dans le premier cas, on parle de cibles d'intérêt (*targets of interest*), dans le second, on parle de région d'intérêt (*region of interest*) [DD09].*

L'environnement détermine en partie le comportement du réseau de capteurs sans fil. En effet, le trafic de données est principalement généré par l'environnement. En particulier, dans le cas de la détection d'événements, le nombre d'alertes transmises

à la station de base dépend du nombre d'événements dans l'environnement. Par ailleurs, les réseaux de capteurs sans fil sont des systèmes dits réactifs. Un système réactif réagit en permanence à son environnement et ne peut être vu de façon pertinente comme faisant du calcul au sens classique [SBB+99].

2.5.2 Topologie

Le terme topologie provient du grec ("$\tau o \pi o \varsigma$" signifiant place, "$\lambda o \gamma o \varsigma$" signifiant étude) et désigne l'étude d'un lieu donné.

Dans le cas des réseaux, la topologie désigne la disposition des éléments (liens de communication, nœuds etc...) d'un réseau informatique. La définition 2.16 présente les types de topologie pour un réseau de capteurs sans fil [EFM06].

Définition 2.16. ***Types de topologie*** *[EFM06]*

Il existe deux types de topologie pour un réseau de capteurs sans fil :

La topologie physique *désigne le positionnement dans l'espace des nœuds et de la station de base. Elle s'intéresse uniquement aux artefacts physiques constituant le réseau.*

La topologie logique *désigne l'ensemble des liens de communication entre les nœuds du réseau, et entre les nœuds et la station de base. Elle s'intéresse au flux de données entre les nœuds et la station de base.*

Il existe un lien de communication asymétrique d'un nœud n_i à un nœud n_j si et seulement si n_j est à la portée de n_i.

Il existe un lien de communication symétrique entre deux nœuds n_i et n_j si et seulement si n_i est à la portée de n_j et n_j est à la portée de n_i.

Un nœud n_j est à la portée d'un nœud n_i si et seulement si il appartient à la sphère de centre n_i et de rayon égal à la portée de n_i.

Les conditions de déploiement des réseaux de capteurs sans fil (*e.g.* obstacles) augmentent le risque d'erreurs. Par ailleurs, l'exigence d'autonomie de fonctionnement impose une très grande fiabilité de ces systèmes. En effet, il n'est pas possible de « redémarrer » un nœud après une panne, comme on le ferait pour un ordinateur personnel. Cependant, pour pouvoir vérifier la fiabilité de ces systèmes, il est nécessaire d'identifier les propriétés qui assurent cette fiabilité.

2.6 Propriétés des réseaux de capteurs sans fil

Les propriétés que doivent satisfaire les réseaux de capteurs sans fil sont liées aux exigences définies en section 2.1 (page 12). Les deux exigences centrales sont le coût et l'autonomie de fonctionnement, à partir desquelles le reste est déduit. Le coût relève de l'aspect commercial plus que de l'aspect technique du réseau, contrairement à l'autonomie de fonctionnement à laquelle nous nous intéressons.

Afin de garantir une autonomie de fonctionnement, la condition nécessaire est que le système possède encore de l'énergie. D'où l'exigence de faible consommation

d'énergie. La propriété que nous dégageons de cette exigence est que la durée de vie du réseau de capteurs sans fil dans le pire des cas doit être supérieure à une borne.

Propriété. (P1) La durée de vie du réseau de capteurs sans fil dans le pire des cas est supérieur à une borne D_{min}

En effet, un des éléments de la spécification d'un réseau de capteurs sans fil est la durée de fonctionnement minimale. Par exemple, dans le cas d'un patient ayant subi une opération, un réseau biomédical doit fonctionner au moins autant de temps que la durée de surveillance post opératoire (*e.g.* quarante huit heures pour une opération de l'appendicite).

Si l'on calcule la durée de vie dans le pire des cas, on fournit une garantie que le système satisfait cet élément de spécification. Cette garantie ne peut être fournie par des techniques de vérification non exhaustives car elles peuvent manquer un scénario rare où le système meurt très vite d'où l'intérêt de notre approche.

Cependant, il existe presque autant de définitions de la durée de vie d'un réseau de capteurs sans fil que d'applications pour réseau de capteurs sans fil.

Nous avons dégagé les définitions qui nous semblent les plus importantes dans la littérature.

Définition 2.17. ***La durée de vie d'un réseau de capteurs sans fil***

Les principales définitions de la durée de vie d'un réseau de capteurs sans fil sont :

1. *la durée pendant laquelle la proportion des nœuds vivants est au moins égale à v ($v = k/n$ avec n le nombre total de nœuds, $k \in \mathbb{N}$ et $0 < k \leq n$) [CE04, DHHV05, DML02].*
2. *la durée pendant laquelle la proportion des nœuds qui disposent d'un chemin de connexion vers la station de base est au moins égale à c ($c = k/n$ avec n le nombre total de nœuds, $k \in \mathbb{N}$ et $0 < k \leq n$) [CGVC06].*
3. *la durée pendant laquelle le taux de messages reçus est supérieur à τ ($0 < \tau \leq 1$) [WXZ+03].*
4. *la durée pendant laquelle le réseau satisfait de manière continue les exigences de l'application [BS02, KAL05].*

La première définition avec $k = n$ est *la plus utilisée*. Autrement dit, la durée jusqu'à ce que le premier nœud meurt. Deng et al. [DHHV05] utilisent cette définition sans préciser ce que veut dire un nœud « vivant ».

On peut émettre l'hypothèse qu'un nœud est vivant, tant qu'il possède encore de l'énergie, ou encore, tant qu'il possède assez d'énergie pour communiquer (émettre ou recevoir un message). En effet, même si un nœud possède assez d'énergie pour capter, ou faire des traitements, si cette énergie n'est pas suffisante pour communiquer ses résultats, alors la station de base (et donc à terme l'utilisateur) ne bénéficie pas de ces résultats.

La deuxième définition s'appuie sur la notion de chemin de connexion. On dit qu'un nœud n_0 dispose d'un chemin de connexion vers la station de base à un instant

t s'il est vivant, non dormant (*i.e.* pas dans l'état *sleep*) et qu'il existe une suite de nœuds $(n_i)_{1 \leq i \leq m}$ vivants et non dormants telle que :

- il existe un lien de communication entre n_i et n_{i+1} à l'instant t pour tout i tel que $0 \leq i \leq (m-1)$
- il existe un lien de communication entre n_m et la station de base à l'instant t.

La troisième définition est souvent utilisée dans le contexte de la détection d'événements où il faut garantir que le taux de messages d'alertes envoyés à la station de base est supérieur à une borne. Cela donne la garantie qu'au moins une proportion des événements détectés seront reçus par l'utilisateur du réseau.

La quatrième définition est correcte mais vague. En effet, vu le caractère dédié à l'application des réseaux de capteurs sans fil, il est légitime de caractériser la durée de vie comme la durée pendant laquelle le réseau satisfait les exigences de l'application. Cependant, ces exigences varient en fonction de l'application et on ne peut donc définir une liste fixe de ces exigences.

Il existe également d'autres définitions pour la durée de vie. Certaines d'entre elles sont liées à la couche physique du réseau et donc ont un niveau d'abstraction très bas qui augmenterait le risque d'explosion combinatoire du *model checking*. Par exemple, la durée de vie peut être définie comme la durée pendant laquelle le rapport signal à interférence plus bruit (SINR) est inférieur à un seuil.

Les définitions 1 à 3 font intervenir des métriques spécifiques. A partir de ces métriques, nous dégageons le reste des propriétés qui doivent être satisfaites par un réseau de capteurs sans fil.

La première propriété que nous dégageons utilise la métrique de la proportion des nœuds vivants évoquée dans la première définition de la durée de vie (voir Définition 2.17).

Propriété. (P2) La proportion des nœuds vivants est supérieure ou égale à v ($v = k/n$ avec n le nombre total de nœuds, $k \in \mathbb{N}$ et $0 < k \leq n$).

La propriété suivante se focalise sur la connectivité des nœuds. En effet, tous les nœuds peuvent être vivants, mais s'ils ne peuvent communiquer leurs résultats à la station de base, le réseau ne peut fonctionner correctement.

Propriété. (P3) La proportion des nœuds qui disposent d'un chemin de connexion vers la station de base est supérieure ou égale à c. ($c = k/n$ avec n le nombre total de nœuds, $0 < k \leq n$).

La propriété suivante se base sur la métrique du taux de messages reçus et elle est souvent utilisée dans les application de détection d'événements, où il faut recevoir un pourcentage d'événements minimal.

Propriété. (P4) le taux de messages reçus par la station de base, contenant les résultats des requêtes dont les identifiants sont ($ReqId_1$, $ReqId_2$, ..., $ReqId_n$), est supérieur ou égal à τ ($0 < \tau \leq 1$).

La propriété (P4) est paramétrée par un ensemble d'identifiants de requêtes. Si ces identifiants représentent des requêtes caractérisant des évènements, alors P4 désigne un taux d'évènements.

La latence est également une métrique importante, en particulier dans les applications critiques de réseaux de capteurs sans fil où le délai de transmission doit

être court. C'est pourquoi, même si elle n'intervient dans aucune des définitions de durée de vie précédentes, nous définissons une propriété basée sur la latence.

Propriété. (P5) La proportion des messages envoyés ayant une latence d'au moins $MaxLatence$ est supérieure ou égale à λ $(0 < \lambda \leq 1)$

En plus de ces cinq propriétés, nous définissons également une propriété (`Deadlock`, P6) qui exprime l'existence d'un interblocage dans le modèle du réseau de capteurs sans fil. C'est une propriété courante et pertinente dans le cas de systèmes concurrents.

Ces six propriétés nous servent de base pour définir l'ensemble des propriétés que l'on peut exprimer en VeriSensor (voir chapitre 3).

2.7 Conclusion sur les réseaux de capteurs sans fil

Un réseau de capteurs sans fil est un système contraint par deux exigences principales : le coût et l'autonomie de fonctionnement (dont découle la contrainte énergétique).

Pour satisfaire ces exigences, ce type de réseau doit avoir une structure et un comportement adaptés. Chaque nœud possède un bloc de capture, de traitement, de communication et d'énergie. Ces blocs représentent les quatre aspects caractéristiques de ce type de réseau. L'aspect applicatif peut être divisé en deux grandes familles : la collecte de données dont l'objectif est la reconstruction de la distribution d'une ou plusieurs grandeurs physiques et la détection d'événements dont l'objectif est la détection d'un changement de l'environnement. La communication des résultats de l'application est l'opération la plus consommatrice en énergie. Elle peut se faire suivant deux modes : *single-hop* (un seul saut) ou *multi-hop* (plusieurs sauts).

Le comportement des quatre aspects est impacté par le déploiement du réseau. Déployer un réseau de capteurs sans fil consiste à affecter une position à chaque nœud et à la station de base. Suivant le déploiement, le réseau peut avoir une topologie différente. La topologie étant la disposition des éléments (liens de communication, nœuds etc...) d'un réseau informatique.

Parmi les propriétés les plus importantes qui augmentent notre confiance en ces systèmes, la durée de vie et un ensemble de propriétés basées sur des métriques caractérisant cette la durée de vie (*e.g.*, proportion de nœuds vivants, proportion de nœuds connectés).

L'étude de l'état de l'art des réseaux de capteurs sans fil nous permet de dégager des notions de base sur lesquels nous nous appuyons pour définir le DSML VeriSensor. Nous nous intéressons maintenant aux approches pour la modélisation et l'analyse de réseaux de capteurs sans fil, dans le but de justifier la pertinence de notre outil VeriSensor.

3 Modélisation et analyse des réseaux de capteurs sans fil

Nous classons les approches pour la modélisation et/ou l'analyse de réseaux de capteurs sans fil en trois catégories :

- La modélisation *ad hoc* et l'analyse, qui consiste à modéliser un aspect (*e.g.*, un protocole de communication particulier) d'un réseau de capteurs sans fil, directement dans un langage formel.
- Les *model checkers* de programmes, qui analysent une implémentation du réseau de capteurs sans fil généralement en *networked embedded systems C* (*nesC*). *nesC* est un langage de programmation orienté composants, dérivé du C, adapté à la programmation de systèmes embarqués en réseau. C'est l'un des plus utilisés pour la programmation de réseaux de capteurs sans fil.
- Les *Frameworks* IDM (Ingénierie Dirigée par les Modèles) constitués d'un langage dédié (*Domain Specific Modeling Language* ou DSML) aux réseaux de capteurs sans fil, d'un générateur de code ou d'un moteur de transformation vers un langage supportant l'analyse.

3.1 Modélisation *ad hoc* et analyse

Dans cette sous-section, nous présentons un ensemble d'études de cas qui réalisent la modélisation et la vérification *ad-hoc* d'un réseau de capteurs sans fil. Elles consistent à modéliser une instance de réseau de capteurs sans fil ou un aspect de ce réseau (*e.g.* un protocole) à l'aide d'un langage formel et à vérifier le modèle formel à l'aide d'un *model checker* supportant le langage utilisé.

L'objectif derrière les études de cas est soit de valider la capacité du *model checker* à modéliser des réseaux de capteurs sans fil, soit de vérifier qu'une instance de réseau de capteurs sans fil satisfait effectivement des propriétés données.

Pour chaque étude de cas, nous détaillons :

- l'objet de la modélisation, qui désigne l'instance de réseau de capteurs sans fil ou de protocole qui est modélisée et vérifiée.
- les spécificités du langage formel utilisé, qui décrivent le langage qui a servi à la modélisation et ses caractéristiques propres.
- les choix de modélisation, qui indiquent les abstractions utilisées dans la modélisation.
- les types de propriétés vérifiées, qui désignent les exigences que l'étude de cas vise à vérifier sur une implémentation. L'objectif de chaque étude de cas est de vérifier un ensemble de propriétés spécifiques sur l'instance de réseau ou de protocole.

3.1.1 Étude de cas en Real-Time Maude

Olveczky et al. [OT07] modélisent, simulent et vérifient en Real-Time Maude un algorithme appelé OGDC (Optimal Geographical Density Control) dont le but est de maintenir une couverture optimale de la zone d'intérêt par des nœuds d'un réseau de capteurs sans fil [ÖM07]. Ils réalisent des simulations Monte-Carlo, puis analysent l'accessibilité en temps borné (*time-bounded accessibility*) et vérifient des propriétés de logique temporelle sur leur modèle.

Objet de la modélisation. L'algorithme OGDC est un algorithme distribué, qui permet à un ensemble de nœuds de couvrir une zone d'intérêt avec un minimum de redondance. La couverture d'un nœud c'est le disque centré en ce nœud, de rayon égal à la portée de capture de son capteur (chaque nœud a un seul capteur). La couverture

du réseau est l'union des couvertures des nœuds. L'objectif d'OGDC est de faire en sorte que la zone d'intérêt soit totalement couverte avec un nombre minimal de nœuds actifs (*i.e.*, allumés) et donc un minimum de recouvrement entre les couvertures des nœuds. OGDC se déroule en plusieurs rounds de durée fixe, contenant chacun deux phases. La première phase est une phase de sélection des nœuds actifs. La deuxième phase est une phase de fonctionnement en régime permanent où les nœuds effectuent leur capture.

Langage utilisé. Real-Time Maude [ÖM07] est un outil et un langage de spécification, qui étend Maude dans le but de supporter la spécification formelle et l'analyse de systèmes temps réel et hybrides. Un système temps réel est un système dont la justesse (*correctness*) globale dépend de la justesse fonctionnelle (les instructions résolvent le problème donné) et de la justesse temporelle (les instructions s'exécutent en dessous d'un délai fixé). Le formalisme de spécification est basé sur les théories de réécriture et il est destiné à spécifier les systèmes temps-réel distribués suivant le paradigme orienté-objet.

Les spécifications Real-time Maude peuvent être analysées par simulation ou par *model checking*. Real-Time Maude permet de :

1. simuler l'exécution du système par réécriture temporelle. Ainsi, un sous-ensemble des comportements possibles est exploré.
2. vérifier le système par *model checking* borné dans le temps de propriétés en Logique Temporelle Linéaire [HR04] (LTL est une logique temporelle, qui permet de définir des formules caractérisant le futur pour un chemin) sur le modèle du système. Cela consiste à envisager tous les comportements possibles jusqu'à une certaine durée. L'espace des états est donc réduit ce qui facilite le *model checking*, mais il n'y a aucune garantie sur la vérification des propriétés au delà.

Choix de modélisation. Toute communication est une diffusion : le message est diffusé à tous les voisins de la source. Une diffusion est modélisée sous la forme d'une suite de transmissions point-à-point (*unicast*). L'abstraction pose problème lorsqu'un nœud diffuseur meurt juste avant la fin des transmissions point-à-point (dans la réalité, ce cas n'arrive jamais, puisque le diffusion est atomique). Le délai est défini comme fonction de la distance uniquement. Un nœud est défini comme une suite d'attributs dont les principaux sont l'énergie restante, la puissance consommée par unité de temps dans chaque mode de fonctionnement du nœud (*idle*, *sleep*), des horloges (utilisées par l'algorithme OGDC).

Types de propriétés vérifiées. Olveczky *et al.* vérifient que le réseau réussit à atteindre la deuxième phase pendant le premier round. C'est-à-dire que l'algorithme de sélection a convergé vers un ensemble fixé de nœuds, permettant au réseau de fonctionner en régime permanent (deuxième phase) avant la fin du round.

Olveczky *et al.* vérifient aussi que toute la zone à surveiller est couverte pendant la deuxième phase du premier round.

Cependant, la vérification réalisée n'est pas exhaustive, et risque d'omettre des scénarios problématiques rares. En effet, l'espace d'états est exploré jusqu'à une profondeur donnée.

3.1.2 Étude de cas en IF

Mounier et al. [MSZ07] modélisent un réseau de capteurs sans fil, qui détecte un nuage de pollution, en utilisant le langage IF [BGO+04]. Ils utilisent le *model checker* Kronos [BDM+98] pour calculer la durée de vie dans le pire des cas du réseau pour deux protocoles de routage différents : l'inondation contrôlée (*controlled flooding*) et l'inondation dirigée (*directed-diffusion*). La description de l'application est très abstraite (simple relai du signal environnemental) et le budget énergétique maximal qu'ils attribuent à un nœud est de 40 unités avec 3 unités par émission, 2 par réception.

Objet de la modélisation. L'objectif est de comparer la consommation énergétique de deux protocoles de routage : inondation dirigée et inondation contrôlée. Pour cela, Mounier et al. insistent sur la nécessité de définir un contexte, avant de pouvoir comparer ces protocoles. Ils modélisent donc deux configurations d'un réseau de capteurs sans fil, chacune avec un protocole de routage différent. Une configuration spécifie plusieurs aspects du réseau de capteurs sans fil : l'environnement, les capteurs, l'application, la couche MAC et la couche routage. Ensuite, ils calculent pour ces deux configurations la durée de vie dans le pire des cas du réseau.

Langage utilisé. IF est un langage destiné à la description de systèmes temporisés asynchrones. Une spécification IF est constituée d'un ensemble de processus, qui s'exécutent en parallèle et communiquent par mémoire partagée ou messages. Un processus est décrit par un automate temporisé avec des variables. Le modèle temporel de IF est celui des *timed automata with urgency* : un attribut est associé à une transition, indiquant sa priorité par rapport à la progression du temps.

Choix de modélisation. Le réseau est modélisé par un processus décrivant l'environnement, un autre pour la topologie, et un pour chaque nœud. Le processus environnement réalise périodiquement les opérations suivantes : envoi d'un stimulus au nœud sélectionné puis sélection aléatoire d'un voisin.

Le processus modélisant la topologie contient une horloge et un tableau de variables. L'horloge est démarrée en même temps que le fonctionnement du réseau. Elle sert à calculer la durée de vie du réseau de capteurs sans fil. Le tableau représente la topologie de communication entre les nœuds : 1 si le nœud n_l représentant la ligne du tableau est voisin avec le nœud n_c représentant la colonne (*i.e.*, il existe un lien de communication entre n_l et n_c), 0 pour l'inverse. Les nœuds sont supposés immobiles donc l'apparition d'un 0 n'est provoquée que par la mort d'un nœud.

Le processus modélisant un nœud envoie une alarme dès qu'il reçoit un stimulus de la part de l'environnement. Il est aussi responsable de diffuser une requête provenant de la station de base.

Types de propriétés vérifiées. Mounier *et al.* calculent la durée de vie dans le pire des cas d'un réseau contenant *6 nœuds*, pour deux configurations (protocole d'inondation dirigée et protocole d'inondation contrôlée) et avec deux critères de mort du réseau : la mort du premier nœud et le partitionnement du réseau. Le réseau est partitionné lorsqu'il existe au moins deux nœuds, qui ne peuvent communiquer (*i.e.*, Il n'y a pas de lien de communication ou de suite de liens de communications, dans le cas multi-hop, entre ces nœuds).

Finalement, un calcul de la durée de vie moyenne du réseau sur une dizaine de scénarios est fait pour montrer la différence avec la durée de vie dans le pire des cas et justifier l'intérêt de la vérification exhaustive.

Néanmoins, le modèle sur lequel les propriétés sont vérifiées reste parfois simpliste. En particulier, le modèle de l'environnement est réduit à un stimulus, et celui de l'application réduite à la retransmission de ce stimulus.

Par ailleurs, les quantités d'énergie modélisées sont relativement faible : 40 unités d'énergie, sachant qu'une réception coûte 2 unités et une émission 3 unités (le coût de capture et de traitement ne sont pas pris en compte). Théoriquement, un nœud peut donc envoyer au maximum 14 messages.

3.1.3 Étude de cas en UPPAAL (1)

Tschirner et al. [TXY08] spécifient un réseau de capteurs sans fil biomédical en utilisant les automates temporisés [AD90] puis vérifient et simulent leur modèle avec UPPAAL [BLL+95]. L'étude de cas se base sur un émetteur/récepteur spécifique (Chipcon CC2420) et la vérification de propriétés qualitatives et quantitatives pour un réseau faisant de la collecte de données.

Objet de la modélisation. L'objet de la modélisation est un réseau de capteurs sans fil biomédical qui fait de la collecte de données, constitué de nœuds équipés d'un émetteur/récepteur spécifique (Chipcon CC2420). Le réseau utilise le protocole de routage de diffusion contrôlée. La diffusion contrôlée consiste pour chaque nœud à rediffuser un message uniquement la première fois où il le reçoit, pour éviter les boucles de rediffusions infinies. Le focus est porté sur la couche MAC et routage. Les collisions sont prises en comptes, ainsi que les acquittements de paquets. L'application est très abstraite : émission périodique de messages de taille fixe. L'environnement n'est pas modélisé. Le réseau est dans une configuration *single-hop* pour la vérification par *model checking*, et il est à la fois *single-hop* et *multi-hop* pour la simulation.

Langage utilisé. UPPAAL est un outil pour la modélisation et la vérification de systèmes temps-réel. Il les modélise comme un réseau d'automates temporisés. UPPAAL est composé de trois parties : un langage de modélisation, un simulateur et un *model checker*. Le langage de modélisation permet de décrire le système sous la forme d'un automate avec des transitions gardées, étendu avec des variables et des horloges. Le simulateur permet d'avoir une idée sur le comportement du système pendant les phases de modélisation ou de conception en examinant un sous-ensemble de tous le scénarios possibles. Il est destiné à être utilisé avant la phase de vérification formelle pour révéler à moindre coût des erreurs du système. Le *model*

checker utilise un sous ensemble de la logique temporelle de temps arborescent *Timed Computation Tree Logic* [ACD93] (TCTL). TCTL prend en compte l'existence de plusieurs chemins pouvant être pris par le système dans le futur et fait intervenir des informations quantitatives sur le temps. Le *model checker* peut vérifier des propriétés d'accessibilité, de sûreté et de vivacité. Une propriété d'accessibilité spécifie qu'il existe au moins un état qui satisfait un prédicat. Une propriété de sûreté garantit qu'un prédicat redouté (*e.g.*, un deadlock) ne se produira jamais. Une propriété de vivacité indique qu'éventuellement un prédicat désiré (*e.g.*, l'acquisition d'un jeton pour communiquer) se produira.

Choix de modélisation. La topologie du réseau est modélisée par une matrice, comme dans [MSZ07]. Une valeur positive dans une case (i, j) de la matrice indique la force du signal entre le nœud i et j, une valeur négative indique l'absence de connectivité. Chaque nœud possède un identifiant unique.

Le réseau de capteurs sans fil est modélisé sous la forme d'un réseau d'automates temporisés étendus où chaque automate représente le comportement d'un nœud.

Pour modéliser la diffusion contrôlée (rediffusion d'un message par un nœud, uniquement la première fois qu'il a reçu ce message), une matrice qui stocke les identifiants des paquets déjà reçus est définie. La même matrice est utilisée pour indiquer si un acquittement a été reçu ou non.

Les collisions sont modélisées à l'aide d'un tableau représentant l'état du signal reçu par chaque nœud. Si le nœud commence une transmission alors qu'un de ses voisins reçoit un signal, l'élément du tableau correspondant est mis à une valeur négative signifiant la corruption du paquet.

Types de propriétés vérifiées. Tschirner *et al.* vérifient trois propriétés de qualité de service sur leur modèle : la connectivité du réseau, le taux de paquets reçus par la station de base et le délai de bout en bout. Le taux de paquets reçus par la station de base est la proportion entre le nombre de paquets envoyés par les nœuds, et le nombre de paquets qu'elle a reçu. Le délai de bout en bout est le temps qui s'écoule entre l'émission d'un paquet par un nœud et la réception de ce paquet par la station de base. Aucune propriété concernant la consommation énergétique du réseau n'est vérifiée.

3.1.4 Étude de cas en UPPAAL (2)

Watteyne et al. [WABU06] utilisent également UPPAAL pour calculer les durées dans le pire des cas des phases d'un protocole MAC temps réel qu'ils ont développé. L'analyse exhaustive est complétée par de la simulation avec OPnet modeler [3] qui permet d'évaluer les performances du protocole dans une liste de scénarios plus détaillés. OPNET modeler est un simulateur de réseau supportant la mobilité, ainsi que les principaux protocoles pour réseaux cellulaires, LAN (Local Area Network), et satellites.

3. http ://www.opnet.com/solutions/network_rd/modeler.html

Objet de la modélisation. Watteyne *et al.* modélisent un réseau de capteurs sans fil *single-hop* constitué de nœuds qui exécutent le protocole MAC temps réel. L'environnement est modélisé et pour chaque nœud seule la couche MAC est modélisée.

Choix de modélisation. Le médium de communication reliant les nœuds, ainsi que l'environnement sont chacun modélisés par un automate. Chaque nœud est modélisé par trois automates représentant les trois phases du protocole MAC exécuté par le nœud.

L'automate environnement est un simple générateur d'alertes à destination des nœuds.

La topologie du réseau est représentée par l'automate médium. L'automate médium est connecté à tous les nœuds. Dès qu'il reçoit un message de la part d'un nœud, il se charge de le retransmettre uniquement à ses voisins.

Types de Propriétés vérifiées. Watteyne *et al.* calculent les temps d'exécution dans le pire des cas des trois phases du protocole MAC ainsi que celui du passage d'une phase à l'autre. Le protocole MAC est temps-réel, donc la justesse globale du protocole dépend de la justesse fonctionnelle (les instructions résolvent le problème donné) et temporelle (les instructions s'exécutent en dessous d'un certain délai). D'où l'intérêt des propriétés vérifiées.

3.1.5 Étude de cas en Hytech

Coleri et al. [EEK02] modélisent et vérifient le système d'exploitation TinyOS (un système d'exploitation open-source orienté composants, pour les réseaux de capteurs sans fil) avec HyTech [HHT97]. Ils étudient également la consommation énergétique d'un seul nœud en analysant une trace d'exécution avec Hytech, puis calculent la durée de vie des nœuds d'un réseau de capteurs sans fil en fonction de leur distance à la station de base, par simulation avec le langage de programmation SHIFT [DGV97].

Objet de la modélisation. Coleri *et al.* modélisent d'abord le système d'exploitation TinyOS en se focalisant sur les interactions concurrentes entre les composants du système. Ensuite, ils modélisent un nœud équipé avec ce système d'exploitation et étudient une trace d'exécution pour ce nœud. Finalement, ils modélisent à des fins de simulation un réseau de capteurs dans une configuration *multi-hop*.

Langage utilisé. HyTech [HHT97] est un outil permettant l'analyse automatique de systèmes embarqués en réalisant la vérification de propriétés de sûreté et de timing. Hytech permet également de générer des traces d'exécution vers un ensemble d'états vérifiant un prédicat d'accessibilité. Le formalisme utilisé est celui des automates hybrides.

Les automates hybrides sont un modèle mathématique pour la description de systèmes dont certaines parties du comportement sont discrètes, d'autres continues. Les transitions de l'automate modélisent les changements d'état discrets, des équations différentielles modélisent les changements continus.

SHIFT [DGV97] est un langage de programmation pour la description de réseaux d'automates hybrides. Il consiste en un ensemble de composants qui peuvent être créés, inter connectés ou détruits, au fur et à mesure de l'évolution du système.

Choix de modélisation. Chaque composant TinyOS est representé par un automate hybride. L'énergie est représentée par une variable continue. Elle est incrémentée brutalement pour la capture, réception ou transmission d'un bit, mais elle est incrémentée de manière continue lorsque le processeur est dans les modes *idle* et *actif*.

Pour modéliser la réception de paquets (à retransmettre) de la part de nœuds voisins, un composant qui génère périodiquement des paquets est connecté aux composants représentant TinyOS.

Pour représenter différentes positions du nœud dans le réseau, la fréquence de génération des paquets est modifiée : typiquement un nœud voisin de la station de base retransmet plus de paquets, qu'un nœud à l'autre extrémité du réseau.

Concernant la modélisation du réseau (uniquement à des fins de simulation), le routage se fait suivant un arbre dont la racine est la station de base. Le *next-hop* d'un nœud est son père. La consommation d'énergie d'un nœud est définie comme une fonction croissante du nombre de fils qu'il possède.

Types de propriétés vérifiées. Coleri et al. vérifient formellement la synchronisation entre l'exécution des instructions du programme et la génération des événements d'un seul nœud.

Coleri et al. étudient la consommation énergétique des nœuds d'un réseau de capteurs sans fil complet par simulation. Ils font varier le taux de génération de paquets destinés à chaque nœud. Pour chaque taux de génération, une trace d'exécution du nœud est calculée avec Hytech, afin d'étudier la consommation d'énergie.

Aucune propriété n'est vérifiée exhaustivement sur tout le réseau de capteurs sans fil.

3.1.6 Étude de cas en AADL

Ghosh *et al.* veulent montrer l'intérêt de l'utilisation d'AADL pour la modélisation de réseaux de capteurs sans fil. Ils modélisent un réseau de capteurs sans fil et utilisent des simulations Monte-Carlo pour calculer le taux moyen de paquets reçus, la latence de bout en bout moyenne et la durée de vie du système.

Objet de la modélisation. Ghosh *et al.* [GPY08] modélisent un réseau de capteurs sans fil *multi-hop* de huit nœuds qui font de la collecte de données. Le protocole de routage utilisé est AODV (Ad hoc On-Demand Distance Vector Routing). Le protocole MAC est *ZigBee*. C'est un protocole hybride : il a des phases où un intervalle de temps est alloué exclusivement à un nœud (*schedule-based*), et des phases où les nœuds peuvent communiquer n'important au risque de provoquer des collisions de messages (*contention-based*).

Langage utilisé. AADL [FGH06] est un langage de description d'architecture (ADL) qui permet de décrire et analyser l'architecture logicielle et matérielle de systèmes embarqués, temps-réel et où la performance est critique. Un système est décrit comme un ensemble de composants logiciels mappés sur un ensemble de composants décrivant la plateforme d'exécution. Chaque composant a un nom et un type représentant son interface. Il existe trois types de composants : *software*, *platform* et *composite*.

Choix de modélisation. Ghosh et al. font uniquement de la simulation, il n'y a donc pas de risque d'explosion combinatoire. Le réseau peut être alors modélisé de manière plus détaillée.

Le réseau est modélisé comme un composite contenant des composants représentant les nœuds. Un nœud n'est pas défini comme composite ce qui entraînerait une décomposition encore plus détaillée. Par contre, on lui associe des informations sous la forme de propriétés AADL concernant sa position, ses composants matériels (mémoire, vitesse du CPU, portée de capture), son environnement (*e.g.*, humidité, température extérieure), son énergie (niveau de la batterie, consommation d'énergie), ses communications (durée de transmission, portée de communication).

Types de propriétés vérifiées. Ghosh et al. réalisent des simulations Monte-Carlo qui leur permettent de calculer les valeurs typiques des métriques suivantes :

- taux de paquets reçus par la station de base en fonction de la température, de l'identifiant du canal de transmission utilisé, et de la puissance de transmission.
- latence de bout en bout moyenne en fonction des canaux de transmission et des puissances de transmission des nœuds.
- Durée de vie du réseau définie comme la durée jusqu'à ce qu'une proportion de nœuds ne peut plus communiquer avec la station de base.

Cependant, ghosh et al. ne vérifient exhaustivement aucune propriété. Par ailleurs, ils ne prennent en compte qu'un type de comportement : la collecte de données.

Après avoir présenté les études de cas qui modélisent et vérifient un réseau de capteurs sans fil de façon *ad hoc*, nous détaillons la deuxième catégorie d'approches : les *model checkers* de programmes qui analysent une implémentation généralement en *nesC*.

3.2 *Model checkers* de programmes

Nous présentons dans cette sous-section les *model checkers* de programmes pour réseaux de capteurs sans fil. Ils interviennent à la fin du cycle de développement logiciel pour vérifier une implémentation de réseau de capteurs sans fil. Pour cela, ils dérivent à partir d'une implémentation une représentation de haut niveau sur laquelle ils vérifient l'absence de bugs. Tous les *model checkers* vérifient le programme à l'exécution, sauf Tos2CProver [BK10] qui effectue la vérification à la compilation.

L'implémentation donnée en entrée aux *model checkers* est toujours écrite en nesC (un langage de programmation orienté composants, dérivé du C, adapté à la programmation de systèmes embarqués en réseau).

Pour chaque outil, nous présentons :

- son architecture, qui est constituée de l'ensemble des modules qui composent l'outil, réalisent chacun une tâche donnée et interagissent éventuellement entre eux. En effet, dans tous les cas, l'outil ne contient pas uniquement le *model checker*, mais aussi des modules qui préparent son entrée (*e.g.*, éditeurs), ou traduisent sa sortie.
- les types de propriétés, qu'il permet de vérifier qui désignent les exigences que l'outil est capable de vérifier sur une implémentation. Chaque outil se focalise sur la vérification d'un ensemble de propriétés données (*e.g.*, propriétés d'accessibilité, absence de deadlocks).

3.2.1 nesC@PAT

nesC@PAT [ZSL+11] est un *model checker* de programmes qui génère automatiquement des modèles à partir de programmes *nesC* et vérifie les propriétés d'accessibilité, de vivacité (*liveness*) et l'absence d'inter-blocages.

Architecture. nesC@PAT est constitué de :

- un éditeur et un parseur
- un générateur de modèle à partir du code *nesC* : il construit en *Labelled Transition Systems* [Kel76] (*LTS*), un modèle du comportement du réseau à partir du code *nesC*. Un *LTS* consiste en un ensemble d'états et de transitions entre ces états. Les transitions sont étiquetées par des actions et un état est désigné comme l'état initial.
- un simulateur
- un *model checker*

La spécificité de nesC@PAT par rapport aux autres *model checkers* de programmes est qu'il permet de vérifier automatiquement une implémentation en *nesC*/*TinyOS* pour le réseau de capteurs sans fil en entier tout en modélisant les interruptions du modèle d'exécution *TinyOS* (*interrupt-driven*).

Types de propriétés vérifiées. nesC@PAT permet de vérifier les propriétés d'accessibilité, de vivacité et d'absence de inter-blocages. Les propriétés d'accessibilité et d'absence de inter-blocages sont vérifiées par exploration exhaustive de l'espace des états à l'aide des algorithmes *Depth first search* (DFS) ou *Breadth first search* (BFS) .

Les propriétés de vivacité sont exprimées en LTL. Puis, la négation de la formule LTL est traduite en un automate de Büchi [RB66]. Un automate de Büchi est un automate non déterministe à états finis qui prend en entrée des mots infinis. Il accepte un mot infini en entrée si et seulement s'il existe une exécution de l'automate qui visite au moins un des états finaux, infiniment souvent. Les composantes fortement connexes acceptantes du produit synchronisé de l'automate et du modèle permettent de produire un contre exemple.

Les propriétés de sûreté ne sont pas vérifiées. Par ailleurs, l'espace d'états croît exponentiellement en fonction du nombre de nœuds. Finalement, une partie des

abstractions est faite manuellement (en particulier l'abstraction des comportement du matériel des capteurs).

3.2.2 tos2cprover

Bucur et al. [BK10] définissent une chaîne d'outils qui fait de l'analyse statique d'implémentations de réseaux de capteurs sans fil à la compilation. L'intérêt par rapport à la détection d'erreurs au moment de l'exécution est qu'on évite les coûts dus au redéploiement de l'application. La chaîne vérifie des propriétés d'accès à la mémoire (*e.g.*, violations d'accès à la mémoire, état des registres) à partir du code implémenté sur un seul nœud. L'élément central de la chaîne est le traducteur Tos2CProver qui traduit un programme écrit en MSP430 embedded C (une extension de C pour les systèmes embarqués pilotés par les interruptions) en ANSI-C. Le *model checker* C Bounded Model Checker [CKL04] (CBMC) est utilisé pour la vérification. C'est un *model checker* pour les programmes ANSI-C sur systèmes embarqués. Les propriétés à vérifier sont exprimées sous la forme d'assertions C.

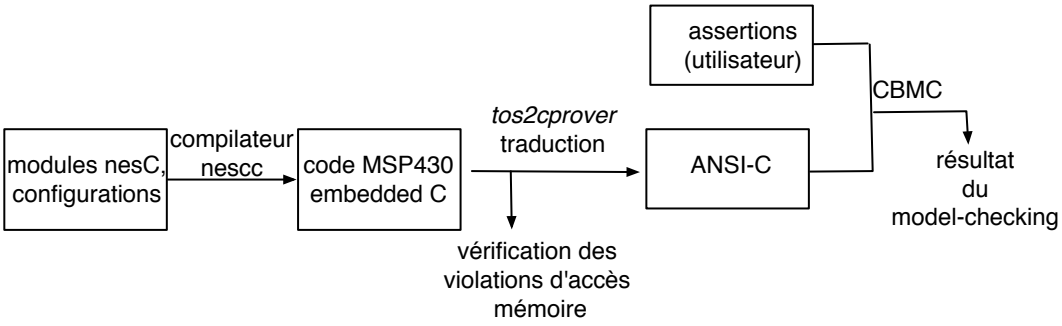


FIGURE 2.4 – La chaîne d'outils pour l'analyse statique de code MSP embedded C

Architecture. La figure 2.4 donne une vue globale de la chaîne d'outils. Elle vérifie un programme en plusieurs étapes :

1. compilation du code nesC en MSP Embedded C
2. traduction par tos2cprover en ANSI-C et vérification de propriétés d'accès mémoire.
3. ajout par l'utilisateur d'assertions à vérifier sur le programme ANSI-C.
4. vérification à l'aide du *model checker* CBMC.

Types de propriétés vérifiées. La chaîne d'outils vérifie des propriétés de sûreté (pour un seul nœud) spécifiées de deux manières :

1. les violations d'accès mémoire non spécifiées par l'utilisateur (vérifiées durant la phase de traduction).
2. les propriétés de sûreté concernant l'état des registres et des périphériques, spécifiées par l'utilisateur sous la forme d'annotations du programme ANSI-C.

Les propriétés liées à l'exécution, comme l'inaccessibilité d'états d'erreurs, ne sont pas supportées.

3.2.3 T-Check

T-check [LR10] est un outil qui réalise des parcours aléatoires de traces d'exécution de programmes nesC ainsi que du *model checking* à *profondeur bornée*, dans le but de vérifier des propriétés de vivacité et de sûreté. Il est construit sur TOSSIM [LLWC03], un simulateur à événements discrets pour réseaux de capteurs sans fil.

Architecture. T-check fonctionne selon deux modes : le mode parcours aléatoire du graphe d'accessibilité (simulation), et le mode *model checker*.

Le mode aléatoire permet à l'utilisateur d'avoir une idée du comportement du système. L'algorithme de parcours aléatoire exécute des traces aléatoires jusqu'à ce qu'une propriété de sûreté soit violée ou jusqu'à ce que l'utilisateur estime que le parcours dure trop longtemps et l'arrête. Ce mode permet de trouver des violations de propriétés de sûreté, ou encore de trouver les transitions critiques conduisant à des violations de propriétés de vivacité.

Le mode *model checker* consiste à exécuter une séquence de transitions aléatoires jusqu'à ce qu'on dépasse la phase d'initialisation du réseau. Ensuite, on effectue le *model checking* explicite pour explorer exhaustivement l'espace d'états jusqu'à une profondeur donnée.

Types de propriétés vérifiées. T-Check permet de vérifier des propriétés de sûreté et de vivacité.

T-Check a la capacité de détecter les erreurs distribuées comme l'incapacité d'un protocole de routage à reconstruire les chemins quand un nœud meurt. Cependant, les propriétés sont exprimées en logique propositionnelle, ce qui empêche la vérification de propriétés temporelles.

T-Check ne détecte pas les erreurs liées aux interruptions.

3.2.4 SLEDE

SLEDE [HR09] est un *framework* qui vérifie automatiquement les protocoles de sécurité pour réseaux de capteurs sans fil. A partir du code nesC, un modèle du protocole est extrait, les propriétés sont vérifiées à l'aide du *model checker* SPIN [Hol03], puis, le cas-échéant un contre-exemple est fourni en nesC.

Architecture. La figure 2.5 donne une vue globale de SLEDE. Le *framework* prend en entrée le code nesC du protocole, la structure d'un message du protocole, la topologie de déploiement, les propriétés à vérifier. Toutes ces entrées sont spécifiées à l'aide d'un langage d'annotations proposé par les contributeurs de SLEDE.

A partir des entrées, un modèle du protocole est généré, ainsi qu'un modèle d'intrusions. Les deux modèles sont fusionnés, puis les propriétés sont vérifiées sur le modèle résultant à l'aide de SPIN. Si la propriété n'est pas vérifiée, le contre exemple produit par SPIN est traduit en nesC.

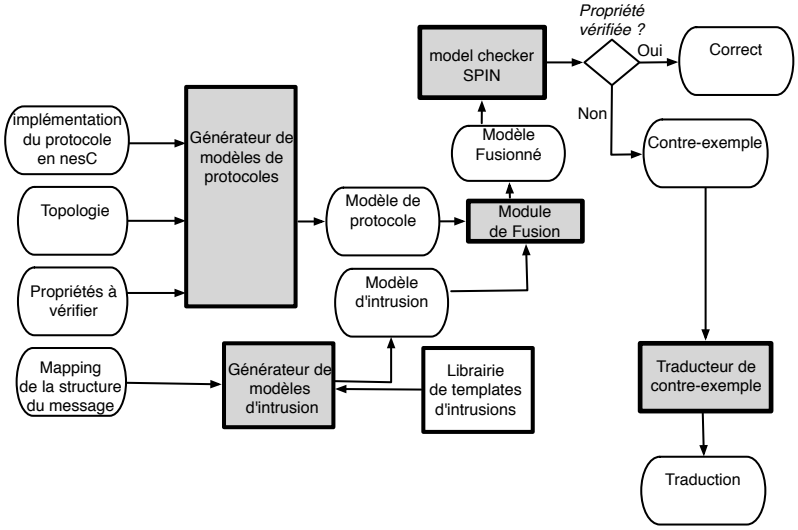


FIGURE 2.5 – Vue globale de SLEDE

Types de propriétés vérifiées. Les propriétés que le protocole vérifie sont décrites sous la forme de commandes et d'événements, à l'aide d'un langage d'annotation proposé par les contributeurs de SLEDE.

Après avoir présenté les *model checkers* de programmes pour réseaux de capteurs sans fil, nous détaillons les *frameworks* IDM (Ingénierie Dirigée par les Modèles) dédiés à la modélisation et à la simulation/ou génération de code de réseaux de capteurs sans fil.

3.3 *Frameworks* IDM supportant l'analyse et/ou la génération de code

L'objectif des *frameworks* IDM que nous présentons est de faciliter la tâche d'implémentation/ou de simulation à un expert réseau. Pour cela, elles fournissent un DSML adapté au domaine de cet expert, ainsi qu'une traduction vers un langage de programmation ou vers un format d'entrée pour un simulateur ou *model checker*.

Pour chaque *framework*, nous présentons :

- son architecture, qui définit l'ensemble des composants du *framework* et les relations entre ces composants.
- le DSML qu'il fournit en entrée à l'utilisateur pour spécifier le système.
- les types de propriétés, qui désignent les exigences que le *framework* permet d'analyser sur le modèle. Chaque *framework* est dédié à l'analyse de propriétés bien déterminées.

3.3.1 Le *framework* VisualSense

VisualSense [BKL+05] est utilisé pour évaluer et visualiser les métriques de performance (*e.g.*, latence, perte de messages) et de consommation d'énergie. C'est un éditeur et simulateur graphique qui permet de construire des spécifications détaillées de protocoles de communication et de modèles de propagation radio. Il est

basé sur Ptolemy II [LJ99] qui est un *framework* pour la modélisation de systèmes embarqués hétérogènes. Sa spécificité est qu'il permet de définir un système de manière hiérarchique en utilisant des modèles de calcul différents (automates, CSP, etc...).

Langage. VisualSense utilise le modèle de calcul à événements discrets orienté acteurs (*discrete event*) fourni par Ptolemy II. Un acteur possède une interface constituée de ports qui sont des points de communication et de paramètres qui sont les variables utilisées lors du fonctionnement de l'acteur. Deux acteurs communiquent entre eux en produisant des événements c'est-à-dire des couples (valeur, timestamp) et en transmettant ces événements à travers des canaux. Un modèle VisualSense est construit en composant de manière hiérarchique les acteurs. Un nœud peut être spécifiée de cette manière.

VisualSense fournit une librairie d'acteurs représentant des nœuds, des canaux de communication (à portée limitée, avec délai), des antennes et des modèles de terrain.

Il est destiné à la simulation en raison des éléments de modèles extrêmement détaillés et de bas niveau d'abstraction. Par exemple, pour chaque canal, on spécifie la portée, la probabilité de perte de paquets, la vitesse de propagation du signal, etc.

Architecture du *framework*. L'architecture du *framework* est constituée de :

1. un éditeur graphique qui permet de construire les modèles de nœuds en connectant des acteurs (nœuds, canaux de communication) entre eux et en définissant leurs caractéristiques (*e.g*, probabilité de pertes du canal) dans des formulaires.
2. un moteur de simulation à événements discrets. L'interaction entre les acteurs se fait via les événements avec des étiquettes temporelles (*time stamps*). Une file ordonnance dans l'ordre chronologique les événements à traiter.
3. un module de visualisation de la simulation qui permet, par exemple, de voir les nœuds se déplacer et leur portée (représentée par un disque) se réduire avec la perte d'énergie.

Types de propriétés vérifiées. VisualSense permet de visualiser l'évolution de métriques de performance modélisées (*e.g.*, latence, perte de messages) à l'aide d'acteurs dédiés. Par exemple l'acteur plot prend en entrée la métrique et produit le graphique de la métrique en fonction du temps.

3.3.2 Le *framework* Matilda/UML

Matilda/UML [WBSO07] définit un profil UML [RJB99][4] pour la modélisation d'*applications* pour réseaux de capteurs sans fil à architecture BisNet (*Biologically inspired Nets*).

Une machine virtuelle (Matilda) permet de simuler l'exécution du modèle.

4. UML (*Unified Modeling Language*) est un langage de modélisation généraliste standardisé, utilisé dans le domaine de l'ingénierie logicielle. Il inclus un ensemble de notations graphiques pour créer des modèles visuels de systèmes. Un profil UML fournit un mécanisme d'extension pour la personnalisation de modèles UML pour des domaines ou plateformes particuliers. [Alh12]

Langage. Le profil UML permet de modéliser une application pour réseaux de capteurs sans fil sous la forme de diagrammes de classes, de diagrammes de séquence et d'un diagramme d'instance (des représentations graphiques partielles d'un modèle, définis dans le langage UML).

Les diagrammes de classe décrivent la structure de l'application sous la forme d'un ensemble d'agents distribués sur les différents nœuds et qui réalisent des opérations. Chaque nœud contient plusieurs agents. Chaque agent collecte les données acquises par le nœud et en fonction de la situation va décider d'aller transmettre ces données à la station de base. Cette organisation est basée sur l'architecture BisNet (Biologically inspired Network). L'inconvénient est que le profil n'est valable que pour ces architectures.

Les diagrammes de séquence décrivent les opérations réalisées par les agents.

Le diagramme d'instance décrit le déploiement du réseau de capteurs sans fil.

Chaque nœud (caractérisé par un identifiant, une position et une portée de communication) dans le diagramme d'instance est une instance du stéréotype *SensorNode* qui représente un certain modèle de nœud avec ses caractéristiques (nom du modèle, portée de capture). Cela permet de supporter l'hétérogénéité dans le réseau. Cependant les caractéristiques ne sont pas suffisantes (*e.g.*, l'hypothèse qu'un nœud ne possède qu'un seul capteur est faite, ce qui n'est pas toujours vrai).

Deux nœuds du diagramme d'instance peuvent être liés par un lien de communication caractérisé par son taux d'erreur par bits.

Architecture du *framework*. Le *framework* est composé du profil UML et de la machine virtuelle Matilda.

L'architecture de la machine virtuelle Matilda suit le pattern architectural *Pipes and Filters* [BMR+96]. C'est-à-dire qu'elle est décomposée en une suite de blocs encapsulant le traitement (plugins) qui se transmettent les résultats de leurs traitements.

Matilda prend en entrée un modèle du profil UML. La première étape consiste à valider syntaxiquement les diagrammes par rapport au méta-modèle du profil UML. Ensuite, les diagrammes de classe décrivant les types d'agents et les diagrammes de séquence sont transformés en code qui permet de configurer une machine virtuelle. La machine virtuelle permet de construire un simulateur. Le simulateur est configuré par la topologie du réseau représentée dans le diagramme d'instance, puis la simulation est lancée.

Les diagrammes d'instance sont transformés en un fichier de configuration. Le fichier de configuration permet de configurer le simulateur d'applications pour réseaux de capteurs sans fil TOSSIM [LLWC03]. Une fois le simulateur configuré, on déploie le code sur chaque nœud défini dans le simulateur.

Types de propriétés vérifiées. Matilda permet de comparer des métriques de performances et de consommation d'énergie (notamment le taux de faux positifs et de faux négatifs pour les événements) entre des algorithmes utilisant l'architecture BisNet.

3.3.3 Le *framework* Baobab

Baobab [ADBS09] est un *framework* de métamodélisation dédié à la description des applications pour réseaux de capteurs sans fil et à la génération du code nesC correspondant. Une partie de l'équipe de Matilda a contribué à Baobab. Baobab ne supporte pas l'analyse.

Langage. Le GMM (Generic MetaModel) définit les concepts récurrents pour décrire les exigences fonctionnelles et non fonctionnelles d'un réseau de capteurs sans fil. Les exigences fonctionnelles spécifient le comportement ou les fonctions attendues de l'application. Les exigences non-fonctionnelles spécifient les critères suivants lesquels elle doit le réaliser (*i.e.*, performance du système). Dans le GMM ces deux types d'exigences sont séparées.

La description de la structure du réseau est supportée par un ensemble de métaclasses comme les métaclasses *Sensor*, *CommunicationUnit*, *EnergySource*.

Pour décrire le comportement d'un réseau de capteurs sans fil, le langage définit la notion de *Task* ou tâche. Le comportement est décrit sous forme d'une suite de tâches qui se succèdent.

Il existe deux types de *Tasks* : ceux liés aux exigences fonctionnelles et ceux liés aux exigences non fonctionnelles du réseau de capteurs sans fil.

Architecture du *framework*. Baobab est structuré en plusieurs packages. Le premier contient un méta-modèle dit générique (GMM) qui permet de modéliser les concepts que l'on retrouve dans toute application de réseau de capteurs sans fil (*e.g.*, *sensor*, *data transmitter*, *data receiver*). Les autres packages sont destinés à contenir des méta-modèles dédiés à un domaine (DSMM *e.g.* la surveillance d'habitats animaliers) ou dédiés à une plateforme (*i.e.* un triplet constitué d'un langage d'implémentation, d'un système d'exploitation et d'une architecture de l'application) appelé PSMM.

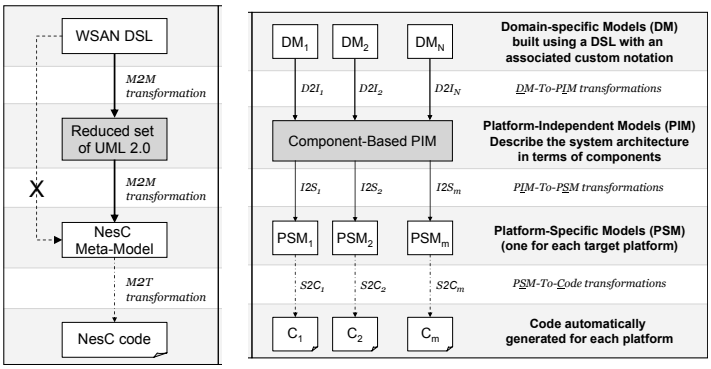
Cette structure reflète l'objectif initial de Baobab : résoudre le conflit entre généralisation et spécialisation dans la conception d'un méta-modèle. En effet, un méta-modèle générique ne peut pas être réutilisé tel quel pour décrire une application spécifique des réseaux de capteurs sans fil. Dans le même temps, un méta-modèle spécifique à une application donnée ne peut pas être réutilisé pour décrire d'autres applications. La solution est de définir un méta-modèle générique et extensible.

Baboab supporte la définition de contraintes en OCL sur les méta-modèles pour que l'extension vers des DSMMs et PSMMs soit faite de manière non ambiguë.

Baobab permet de générer le code nesC pour TinyOS à partir d'un modèle. Les méta-modèles doivent être définis avec EMF (Eclipse Modeling Framework)[5] et utiliser openArchitectureware[6] pour implémenter la transformation du modèle vers le code.

Cependant le support de génération de code de openArchitectureWare n'est plus disponible actuellement. Par ailleurs, il n'existe pas de moyen pour spécifier le déploiement du réseau, ni de syntaxe concrète fournie pour Baobab.

5. http ://www.eclipse.org/modeling/emf/
6. http ://www.openarchitectureware.org/



(a) Métamodèles (b) Modèles

FIGURE 2.6 – Architecture du *framework* MEDWSA [VCLÁ+07]

3.3.4 Le *framework* MEDWSA

MEDWSA [VCLÁ+07] est un *framework* proposant un DSML avec des notations visuelles supportant la génération de code nesC. MEDWSA ne supporte pas l'analyse.

Langage. Le DSML qui fait partie du *framework* permet de décrire l'application pour réseaux de capteurs sans fil sans se préoccuper de l'architecture ou de la plateforme de l'application.

La structure d'une application pour réseaux de capteurs sans fil est décrite par le DSML sous forme de groupes de nœuds (*NodeGroup*) réalisant des tâches similaires. Chaque groupe de nœuds est relié à une *NodeDefinition* représentant les types de ressources utilisées par les nœuds du *NodeGroup*.

Le comportement d'un *NodeGroup* est décrit par une suite de *FunctionalUnit* indiquant les traitements réalisés. Une *FunctionalUnit* est liée aux ressources nécessaires (*e.g.* des capteurs) pour son exécution.

L'ADL décrit la structure du réseau à l'aide de concepts appartenant au diagramme de composants et le comportement à l'aide de concepts appartenant aux diagrammes d'activités et d'états transitions.

L'utilité de l'ADL est uniquement de fournir un niveau de représentation intermédiaire permettant de faciliter la transformation vers le code en la décomposant en deux transformations plus simples.

Architecture du *framework*. La figure 2.6a présente les méta-modèles et les transformations de modèle proposées par le *framework*. La figure 2.6b présente l'instanciation de ces méta-modèles. Selon la figure 2.6a, le *framework* fournit :

- un DSL (Domain specific language).
- un langage de description d'architecture (Architecture Description Language) indépendant de la plateforme. Il est constitué d'un sous ensemble du méta-modèle de UML 2.0.
- Un langage de modélisation dépendant de la plateforme nesC consistant en un méta-modèle basé sur la spécification nesC 1.1.
- Un outil de transformation de modèles du DSL au ADL.

- Un outil de transformation de modèles de l'ADL au méta-modèle nesC.
- Un outil de génération de code nesC à partir du méta-modèle nesC.

3.3.5 Le *framework* CaVi/PRISM/Castalia

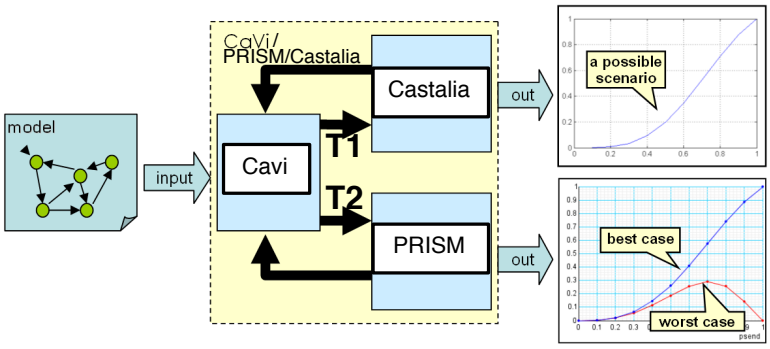


FIGURE 2.7 – Architecture de Cavi/PRISM/Castalia

Cavi/PRISM/Castalia [BFFM08] est un *framework* qui permet de spécifier graphiquement des protocoles de réseaux de capteurs sans fil, de les vérifier et de les simuler. CaVi est une interface graphique commune pour le simulateur Castalia [Bou07]) et le *model checker* probabiliste PRISM [KNP02]). Pour l'instant, le *framework* supporte seulement deux protocoles de routage pour réseaux de capteurs sans fil : le *flooding* et le *gossiping*.

Langage. CaVi offre un langage permettant de modéliser les positions des nœuds et les liens de communication entre eux graphiquement sur un canevas. Pour aider à déterminer la distribution spatiale des nœuds, Cavi permet de visualiser les puissances de réception à un saut (one hop reception strength) dans le meilleur et pire des cas. Il permet de spécifier les protocoles, paramètres et caractéristiques du canal de communication sans fil pour chaque nœud, ou pour le réseau en entier. Les indicateurs de performances liés aux nœuds ou au liens de communication peuvent aussi être spécifiés.

Architecture du *framework*. La figure 2.7 donne une vue globale de l'architecture de CaVi/PRISM/Castalia. Il est constitué de :

- CaVi : l'interface qui permet de modéliser la topologie du réseau graphiquement tout en indiquant les indicateurs de performance pour chaque nœud (*e.g.*, puissance de transmission)
- Une transformation *T2* vers un système à état transitions pour PRISM, et *T1* vers un format d'entrée pour Castalia.
- PRISM : un *model checker* probabiliste pour protocoles de réseaux de capteurs sans fil.
- Castalia : un simulateur de protocoles de réseaux de capteurs sans fil.

L'intérêt de CaVi est d'épargner à celui qui modélise, le calcul complexe des probabilités de pertes sur chaque lien de communication entre deux nœuds. En effet,

à partir de la topologie du réseau spécifiée graphiquement et des indicateurs de performance (*e.g.* force du signal sur un lien de communication), les probabilités de pertes sont calculées automatiquement. Elles sont combinées avec un template du protocole de routage et données en entrée à PRISM.

Types de propriétés vérifiées. Les propriétés vérifiées sont liées aux performances du réseau (principalement la probabilité de réception de messages) et spécifiées en logique temporelle probabiliste. PRISM calcule les probabilités dans le meilleur et le pire des cas de satisfaction de la propriété.

Après avoir étudié les trois types d'approches pour la modélisation et/ou la vérification de réseaux de capteurs sans fil, nous proposons de faire, en section 3.4, une synthèse globale de tous les travaux, suivant un ensemble de critères exhaustifs. Cela nous permettra de positionner notre approche VeriSensor.

3.4 Synthèse comparative des approches

Dans cette sous-section, nous synthétisons les caractéristiques principales des approches que nous avons présentées et discutons leurs forces et faiblesses.

La comparaison des différentes approches se fait sur la base d'un ensemble de critères que nous avons déduits de la littérature et qui sont :

1. La motivation : Quel est l'intérêt de l'approche ?
2. Le langage de modélisation/ou programmation : quel est le langage utilisé pour modéliser/ou programmer le réseau de capteurs sans fil ?
3. La vérification : l'approche supporte-t-elle la vérification formelle ? Si oui, quelles propriétés permet-elle de vérifier ?
4. La simulation : l'approche supporte-t-elle la simulation ?
5. La génération de code : l'approche supporte-t-elle la génération de code ?
6. L'intervention dans le cycle de développement : à quel stade du cycle de développement, l'approche peut-elle être utilisée ?
7. L'expertise requise : quelle est l'expertise (en méthodes formelles ou en programmation) requise de la part de l'utilisateur de l'approche ?
8. Les forces : les meilleurs aspects de l'approche.
9. La modélisation des capteurs : comment les capteurs d'un nœud sont modélisés ?
10. La modélisation de l'application : comment l'application d'un nœud est-elle modélisée ?
11. La modélisation du réseau : comment les communications réseau d'un nœud sont modélisées ?
12. La modélisation de l'énergie : comment la consommation d'énergie est-elle modélisée ?
13. La modélisation du déploiement : comment le déploiement du réseau est-il modélisé ?
14. La modélisation de l'environnement : comment l'environnement est-il modélisé ?
15. Le support de l'hétérogénéité : comment l'approche supporte-t-elle la modélisation de différentes classes de nœuds ?

3.4.1 Discussion

Dans cette sous-section, nous discutons toutes les approches présentées précédemment et expliquons pourquoi aucune d'entre elles n'est totalement satisfaisante. Cela nous permet de positionner VeriSensor dans ce contexte global.

La modélisation *ad hoc* permet de vérifier des propriétés qualitatives et/ou quantitatives sur les réseaux de capteurs sans fil. Le but de toutes les études de cas étudiées est soit de calculer la durée de vie d'une instance de réseau de capteurs sans fil, soit d'évaluer ses performances (voir tableau 2.1, page 50).

Cependant, comme indiqué dans le tableau 2.1 (page 50), le caractère *ad hoc* nécessite une expertise élevée à la fois en méthodes formelles et en réseaux de capteurs sans fil. Une phase d'abstraction manuelle du réseau est nécessaire pour réduire le risque d'explosion combinatoire. Cette phase peut générer un décalage sémantique entre le système et le modèle formel.

Par ailleurs, la modélisation *ad hoc* est difficilement généralisable et le passage à l'échelle en particulier est laborieux. En effet, un réseau de capteurs sans fil peut atteindre des tailles de l'ordre de la centaine voire du millier [ASSC02], avec des topologies complexes et des nœuds de différents types. Une abstraction manuelle d'un tel système est difficile. C'est pourquoi les études de cas ont une portée réduite : modélisation d'un aspect particulier d'un réseau de faible taille. L'aspect détaillé est généralement le réseau (voir tableau 2.2, page 51).

Le *model checking* de programmes résout le problème du caractère manuel de l'abstraction en construisant automatiquement des modèles formels à partir du code. Cela permet de débugger des erreurs de gestion de la mémoire par exemple, mais là encore on rencontre un problème de passage à l'échelle.

C'est pourquoi, certains outils (*e.g. Tos2CProver*) ne vérifient que le comportement d'un seul nœud. Cette difficulté est évoquée par Li et al. [ZSL+11]. Pour illustrer l'efficacité de nesc@PAT, ils modélisent un algorithme simple (Trickle [LCH+11]) avec trois nœuds et obtiennent un espace d'états de l'ordre de 10^7. L'espace d'états croît exponentiellement avec la topologie du réseau et le nombre de nœuds. Li et al. pointent du doigt le problème et évoquent comme perspectives la piste de techniques de réduction d'ordre partiel et de détections de symétries. T-check, lui, essaie de contourner le problème en proposant à l'utilisateur des parcours aléatoires (voir tableau 2.1) du graphes d'états jusqu'à ce qu'une propriété de sûreté soit violée ou jusqu'à ce que l'utilisateur estime que le parcours dure trop longtemps et l'arrête. Cependant, l'utilisateur peut arrêter le parcours avant que l'outil n'ait montré que la propriété est violée.

En outre, le *model checking* de programmes fait intervenir la vérification tard dans le cycle de développement logiciel (voir tableau 2.1, page 50), ce qui augmente les coûts de correction. Hormis Tos2CProver qui réalise l'analyse de programmes à la compilation, les autres *model checkers* de programmes vérifient l'implémentation une fois déployée. Dans le cas où des erreurs sont découvertes, il faut corriger les bugs puis procéder au redéploiement de l'implémentation.

Les *frameworks* contenant des langages dédiés permettent de réduire le décalage entre le domaine de connaissance des experts et l'analyse (ou l'implémentation) des réseaux de capteurs sans fil. Cependant, la plupart d'entre eux (sauf [BFFM08], voir

tableau 2.4, page 53) supportent uniquement la simulation comme analyse, ce qui rend difficile la détection de scénarios problématiques rares.

VisualSense en particulier fournit un niveau de description beaucoup plus détaillé (voir tableau 2.4, page 53) avec des éléments de modèles tels qu'un détecteur de collisions entre paquets, des modèles de propagation d'onde, modèles de gain d'antenne (en dB) et la modélisation explicite des données transmises (*e.g.*, type record, opérations arithmétiques sur les données). Le niveau de description est adapté à la simulation mais pas au *model checking* et risque d'entraîner une explosion combinatoire. Baldwin et al. [BKL+05] se positionnent eux-mêmes uniquement par rapport aux simulateurs réseau (*e.g.*, ns2, omnet++). De plus, VisualSense ne fournit pour l'instant que le type de capteur acoustique.

Le profil UML de Matilda est dédié à l'architecture BisNet ce qui le rend difficilement généralisable. Cavi ne décrit pas l'application et ne supporte que deux protocoles réseaux : *gossiping* et *flooding*.

Hormis [Sad08] et [WBSO07], aucun outil ne fait à la fois de la simulation et de la génération de code. Par ailleurs, aucun des DSML qui supportent la génération de code ne définit des informations à propos de la topologie de déploiement du réseau.

Parmi les *frameworks*, Cavi est le seul à faire de la vérification formelle. Cependant, il modélise uniquement l'aspect réseau et supporte uniquement deux protocoles réseau (*i.e.*, flooding, gossiping).

Nous avons listé les travaux les plus importants, à notre connaissance, pour la modélisation et la vérification de réseaux de capteurs sans fil. Il existe également d'autres travaux de moindre importance selon nous, dont plusieurs ne bénéficient que d'un faible support de développement et de maintenance. Par exemple, [DSS+08] proposent un langage formel de haut niveau pour la modélisation et la vérification de réseaux de capteurs sans fil. Il n'est pas adapté à un expert réseau ne connaissant pas les techniques formelles. SensorML [Bot07] est un langage fournissant un encodage XML pour la description de capteurs et de processus de mesure. Il se focalise sur l'aspect capture et ne fournit pas d'éléments pour décrire les topologie de communications du réseau, ou d'autres aspects en détail. Le langage Lustre [HCRP91] a également été utilisé dans une étude de cas sur les réseaux de capteurs sans fil. Lustre est un langage de *programmation* déclaratif, défini formellement, pour la programmation de systèmes réactifs (Un système réactif réagit en permanence à son environnement et ne peut être vu de façon pertinente comme faisant du calcul au sens classique [SBB+99]). Lustre ne supporte pas la spécification et n'est pas dédié aux réseaux de capteurs sans fil. Il s'intéresse, comme tous les langages de programmation (*e.g.*, nesC) à la description d'une solution et non d'un problème. [GR06] est un prototype basé sur les ontologies qui permet de générer du code OWL. Il se base sur SensorML et est donc focalisé sur la modélisation de systèmes de mesures, au détriment de la description de la topologie et des communications. Le prototype en est à ses tout premiers pas (*i.e.*, proof-of-concept).

Approche	Motivation	Langage	Vérification	Simulation	Gén. de code	Intervention dans le cycle de dév.	Expertise requise	Forces
Etudes de cas								
[OT07]	Vérification d'un protocole	RT-Maude	Durée des phases du protocole (Model-checking borné)	Oui	Non	Modélisation	Elevé (Formel + Réseaux)	Modélisation du déploiement
[MSZ07]	Durée de vie pire-cas	I.F.	Non	Oui	Non	Modélisation	Elevé (Formel + Réseaux)	Modélisation du déploiement
[TXY08]	Réglages de paramètres pour optimiser les perf.	UPPAAL	connectivité, PdR,latence	En UPPAAL	Non	Modélisation	Elevé (Formel + Réseaux)	Modélisation du déploiement
[WABU06]	Vérification protocole MAC temps-réel	UPPAAL	Durée des phases du protocole	En UPPAAL	Non	Modélisation	Elevé (Formel + Réseaux)	Modélisation du déploiement
[EEK02]	Sûreté de TinyOS/ Durée de vie	HyTech	Sûreté	Oui	Non	Modélisation	Elevé (Formel + Réseaux)	Modélisation du déploiement
[GPY08]	Performance et durée de vie du réseau	AADL	Non	Oui	Non	Modélisation	Elevé (Formel + Réseaux)	Modélisation du déploiement
Model checkers de programmes								
nesC@PAT	Vérification d'applications nesC	nesC	Accessibilité/ Vivacité/ Deadlocks. (Problème de passage à l'échelle)	Oui	Non	Déploiement	Moyenne (prog. nesC)	Prise en compte des interruptions en TinyOS
Tos2CProver	Analyse statique nesC	MSP Embedded C	Violation d'accès mémoire /Etats des registres et périphériques	Non	Non	Compilation	Moyenne (MSP Embedded C)	Analyse du programme avant le déploiement
T-check	Vérification d'applications nesC	nesC	Sûreté/Vivacité (Model checking borné, logique propositionnelle)	Oui (parcours aléatoires)	Non	Déploiement	Moyenne (prog. nesC)	Algorithmes distribués
SLEDE	protocoles de sécurité	nesC	sûreté/intrusions dans les protocoles de sécurité	Non	Non	Déploiement	Moyenne (prog. nesC)	Support de l'utilisateur

Table 2.1 – Synthèse des approches pour la modélisation et/ou la vérification de réseaux de capteurs sans fil (Critères 1-7)

Approche	Motivation	Langage	Vérification	Simulation	Gén. de code	Intervention dans le cycle de dév.	Expertise requise	Forces
Frameworks								
VisualSense	Simulation fine de protocoles	VisualSense	Non	Oui	Non	Modélisation	Faible	Modèles fins de propagation d'onde
Matilda/ UML	Simulation de réseaux BisNet	Profil UML	Non	Oui	Oui (nesC)	Modélisation	Faible	Support de l'hétérogénéité
Baobab	Modélisation extensible et précise	Métamodèles uniquement	Non	Non	Oui (nesC)	Modélisation	Faible	Extensibilité
MEDWSA	Modélisation graphique et génération de code	Pas nommé	Non	Non	Oui (nesC)	Modélisation	Faible	Modélisation des traitements
CaVi	Abstraire les calculs d'indicateurs de perf.	Cavi	Calcul de la probabilité de réception (2 protocoles supportés)	Protocoles	Non	Modélisation	Faible	Indicateurs de performance de transmission
VeriSensor	Modélisation multi-aspects modulaire	Verisensor	Oui (durée de vie, performances)	Non	Non	Modélisation	Faible	analyse énergétique/modélisation multi-aspects

Table 2.2 – Synthèse des approches pour la modélisation et/ou la vérification de réseaux de capteurs sans fil (Critères 1-7)

Approche	Capteurs	Application	Réseau	Energie	Déploiement	Environnement	Hétérogénéité
Etudes de cas							
[OT07]	Non	Non	Diffusion d'un message aux voisins	Par mode de fonctionnement	Liste de voisins d'un noeud	Non	Non
[MSZ07]	Un par noeud	Simple relai du stimulus environnemental	Routage/ MAC	coût réception/ émission	Liens de communication	Signal binaire	Non
[TXY08]	Non	Envoi périodique	Routage/ MAC	Non	Liens de communication + force du signal entre noeuds	Non	Non
[WABU06]	Un par noeud	Simple relai du stimulus environnemental	MAC	Non	Oui	Signal binaire	Non
[EEK02]	Non	Composants TinyOS	oui	Evolution continue dans les états et discrète sur les transitions	Seulement pour la simulation	Non	Non
[GPY08]	Non	Non	Routage/MAC	Oui	Oui	Non	Non
Model checkers de programmes							
nesC@PAT	Oui	Oui	Oui	Oui	Oui	Non	Non
Tos2CProver	Oui	Oui	Oui	Non	Non	Non	Non
T-check	Oui	Oui	Oui	Oui	Oui	Non	Non
SLEDE	Non	Non	Oui	Oui	Oui	Non	Non

Table 2.3 – Synthèse des approches pour la modélisation et/ou la vérification de réseaux de capteurs sans fil (Critères 8-15)

Approche	Capteurs	Application	Réseau	Energie	Déploiement	Environnement	Hétérogénéité
Frameworks							
VisualSense	Capteur acoustique supporté	Bas niveau d'abstraction	Adapté (collisions, propagation d'ondes)	Bas niveau d'abstraction	Très fin (modèles de propagation d'ondes)	Signal acoustique seulement	Pas de notion de type de noeud
Matilda/ UML	Max. un par noeud	BisNet (suite d'agents)	Non	Non	Taux d'erreur par lien de communication	Faible	Notion de type de noeud
Baobab	Oui	Oui	Oui	Oui	Faible	Non	Pas de notion de type de noeud
MEDWSA	Oui	Oui	Oui	Non	Oui	Non	Oui
CaVi	Non	Non	Oui	Non	Force du lien de communication/Puissance du signal	Non	Non
Verisensor	Oui. Plusieurs par noeud	Toutes classes d'applications supportées	Oui	Capteurs/ Appli/ Reseau pris en compte	Positions et portées	Scénarios environnementaux réutilisables	Notion de type de noeud

TABLE 2.4 – Synthèse des approches pour la modélisation et/ou la vérification de réseaux de capteurs sans fil (Critères 8-15)

4 Contributions

Sur la base de l'étude du domaine des réseaux de capteurs et des approches pour la modélisation et/ou la vérification de réseaux de capteurs sans fil, nous proposons les contributions suivantes :

- VeriSensor, un DSML pour la modélisation et la vérification par *model checking*. VeriSensor permet de construire une description hiérarchique et compositionnelle des différents aspects d'un réseau de capteurs sans fil (capture, application, réseau, énergie) et supporte la modélisation de l'hétérogénéité.
- Un moteur de transformation d'une spécification VeriSensor vers la spécification formelle cible exprimé dans le langage formel des *Instantiable Transition Systems* (ITS) [TMPHK09b] et en Réseaux de Petri Temporels à des fins de vérification. Les ITS/Réseaux de Petri Temporels nous permettent de décrire la spécification formelle d'une manière compositionnelle et hiérarchique, tout en fournissant un support naturel pour la modélisation de la concurrence et du temps. Le moteur de transformation est basé sur un ensemble d'abstractions automatiques qui permettent de réduire le problème de l'explosion combinatoire.
- Un langage de spécification de propriétés pour l'expert réseau. Les propriétés supportées sont qualitatives et quantitatives. La propriété la plus importante, à savoir le calcul de la durée de vie du réseau, est aussi supportée.
- Un module de traduction des résultats de vérification sous une forme lisible (*e.g.*, graphiques) par l'expert réseau.

VeriSensor permet de couvrir la *chaîne complète* de la spécification du réseau de capteurs sans fil à la production des résultats de vérification pour l'utilisateur. Il se distingue des approches présentées précédemment pour les raisons suivantes.

D'abord, il requiert une expertise plus faible que pour la modélisation *ad hoc* (voir tableau 2.3), en fournissant les concepts du domaine métier de l'expert réseau. Il supporte aussi la modélisation des aspects capture, application, réseau et énergie, contrairement aux études de cas.

Ensuite, contrairement aux *model checkers* de programmes qui détectent les erreurs après l'implémentation, il intervient tôt dans le cycle de développement (voir tableau 2.3) et réduit donc les risques dus à la découverte tardive d'erreurs. Le passage à l'échelle est aussi moins laborieux que pour les *model checkers* de programmes, grâce au niveau d'abstraction plus élevé du modèle.

Finalement, VeriSensor et Cavi sont les seules *frameworks* à permettre de vérifier exhaustivement le modèle du réseau de capteurs sans fil. Cependant, VeriSensor se distingue par le fait qu'il supporte la modélisation de tous les aspects ainsi que l'hétérogénéité. En particulier, plusieurs capteurs par nœud sont supportés, tous les types d'application également, et la possibilité de définir plusieurs scénarios environnementaux est fournie.

5 Conclusion

Dans ce chapitre, nous avons présenté les définitions du domaine des réseaux de capteurs sans fil sur lesquelles nous nous appuyons pour définir VeriSensor.

La structure d'un réseau de capteurs sans fil consiste en un ensemble de nœuds dotés de capteurs, qui communiquent les grandeurs physiques à une station de base possédant de fortes ressources de traitement et de communication.

Un réseau de capteurs sans fil peut se comporter de deux façons différentes : 1) collecter périodiquement les grandeurs physiques et les envoyer à la station de base 2) détecter un certain nombre de changements de l'environnement (événements) et signaler ces changements à la station de base.

Le réseau est déployé sur une zone d'intérêt à surveiller, suivant une topologie physique et logique qui détermine les chemins suivis par l'information dans le réseau.

Pour augmenter la confiance dans ces systèmes, un certain nombre de propriétés que le réseau doit vérifier ont été isolées. La plus importante est le calcul de la durée de vie du réseau à partir de laquelle nous tirons des métriques (*e.g.* connectivité) pour définir d'autres propriétés.

Nous avons aussi présenté les trois catégories d'approches pour la modélisation et l'analyse de réseaux de capteurs sans fil : les études de cas en langage formel, les *model checkers* de programmes, les *Frameworks* IDM pour la génération de code ou la simulation. Nous avons positionné VeriSensor par rapport à ces travaux.

Le langage proposé fournit des concepts du domaine métier facilitant la tâche de modélisation pour l'expert réseau. Tous les aspects du réseau sont supportés, y compris le déploiement. L'outil VeriSensor vérifie les propriétés pendant la phase de modélisation ce qui réduit les risques puisque les erreurs sont détectées en amont du cycle de développement.

VERISENSOR

CHAPITRE 3

1 Introduction

Dans ce chapitre, nous présentons notre DSML Verification of Wireless Sensor Networks (VeriSensor) pour la modélisation et la vérification de réseaux de capteurs sans fil. VeriSensor permet une description hiérarchique et compositionnelle d'un réseau de capteurs sans fil ainsi que la vérification de propriétés qualitatives et quantitatives par *model checking*. Il est caractérisé par :

- la modélisation de tous les aspects du réseau de capteurs sans fil, à savoir, l'environnement, la capture, l'application, le réseau et la consommation d'énergie.
- la *modularité* et la *réutilisabilité* qui résultent de l'encapsulation des aspects dans des modèles séparés et des propriétés à vérifier dans des « tâches de vérification ».
- le support explicite des classes de comportement applicatif d'un réseau de capteurs sans fil, à savoir, la détection d'évènements et la collecte de données.
- le support de l'hétérogénéité, en fournissant des concepts permettant de regrouper les caractéristiques communes entre nœuds.
- la modélisation de l'environnement, en fournissant la possibilité de décrire les scénarios environnementaux auxquels peut être soumis le système.
- le support de plusieurs types de capteurs par nœud.

Nous détaillons l'architecture globale de VeriSensor (section 2) composée d'un ensemble de modèles.

Afin d'illustrer les concepts de VeriSensor détaillés dans les sections 4 à 13, nous présentons un exemple de réseau de capteurs sans fil pour la surveillance biomédicale [OMSJ05] tiré de la littérature du domaine (section 3).

En section 4, nous donnons une vision globale de la syntaxe du DSML sous la forme d'un méta-modèle de haut niveau d'abstraction.

Dans les sections 5 à 13, nous détaillons les modèles évoqués en section 2 en indiquant pour chacun d'entre eux :

- son objectif,
- sa sémantique,
- les abstractions et hypothèses simplificatrices de l'état de l'art (chapitre 2) sur lesquelles il se base,
- sa syntaxe.

La section 14 donne une vue d'ensemble de l'outil implémenté pour supporter le langage VeriSensor.

2 Vue d'ensemble de VeriSensor

Un modèle VeriSensor est composé de la spécification d'un réseau de capteurs sans fil, de son environnement et des propriétés que l'on veut vérifier dessus.

La *spécification du réseau de capteurs sans fil* (voir figure 3.1) est composée de :

1. *la définition de classes de nœuds* : une classe de nœud spécifie les caractéristiques d'un type de nœud. Elle permet de factoriser les caractéristiques communes d'un ensemble de nœuds. Cela est crucial dans le cas de réseaux hétérogènes composés de nœuds de différents types. Le modèle de classe de nœuds contribue à la modularité et à la réutilisabilité dans VeriSensor. Deux nœuds de même classe ont des comportements très proches car toutes les fonctions qu'ils réalisent (i.e., la capture, le traitement, la communication et la gestion de l'énergie) sont similaires. Chaque classe de nœuds spécifie :
   - *l'aspect capture* qui définit le comportement des capteurs à travers leurs caractéristiques techniques (modèle de consommation d'énergie, caractéristiques de fonctionnement). Le modèle de consommation d'énergie spécifie les coûts des opérations de capture. Les caractéristiques de fonctionnement incluent le temps de démarrage du capteur et la grandeur captée.
   - *l'aspect application* qui spécifie le comportement de l'application en terme de commande des capteurs, de gestion des communications avec le réseau et de consommation d'énergie. Cet aspect référence les requêtes de collecte périodique et de détection d'évènements que le noeud doit exécuter.
   - *l'aspect réseau* qui définit la gestion locale des communications du nœud et son modèle de consommation d'énergie. Il décrit avec un haut niveau d'abstraction le comportement des couches réseau du noeud.
   - *La quantité d'énergie initiale* d'une instance de nœud.
2. *la définition du déploiement du système* : la façon dont les nœuds et la station de base sont répartis dans la zone d'intérêt ainsi que les chemins de communications entre ces nœuds et la station de base. Il définit les chemins de routage, et les positions des noeuds dans l'environnement.
3. *la spécification des requêtes* à traiter par le réseau : l'ensemble des requêtes qui seront diffusées dans le réseau et réalisent la collecte périodique ou la détection d'évènements. Les requêtes encapsulent des tâches consistant à capter, traiter et envoyer périodiquement un ensemble de grandeurs physiques. Chaque requête est destinée à un sous ensemble des nœuds du réseau et diffusée par la station de base dans le réseau, à un instant précis.

La spécification de l'environnement définit un ensemble de scénarios environnementaux auxquels le réseau peut être soumis. Un scénario environnemental décrit l'évolution des grandeurs captées par le système dans l'espace et dans le temps.

La spécification des propriétés consiste en un ensemble de tâches qui encapsulent ce que l'utilisateur veut vérifier sur le modèle du réseau de capteurs sans fil. Chaque tâche de vérification définit plusieurs instances du modèle du réseau de capteurs sans fil et de l'environnement. Pour cela, elle instancie les caratéristiques à valeur numérique du modèle du réseau de capteurs sans fil (`startupTime` et `sensingPeriod` sont à valeur numérique mais pas le `nextHop` par exemple) et elle instancie l'environnement par un ou plusieurs scénarios environnementaux.

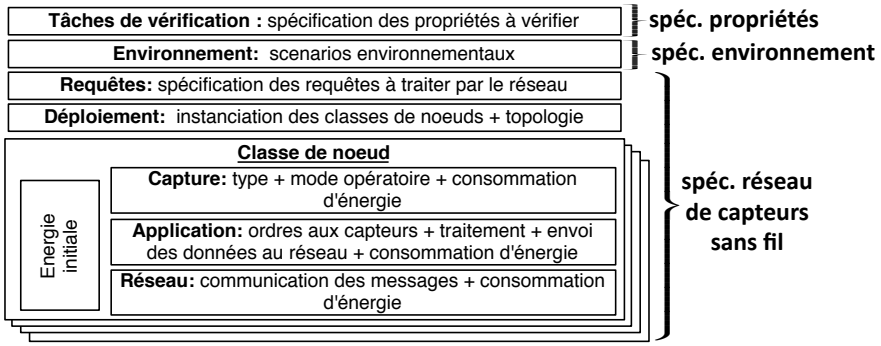


FIGURE 3.1 – Structure d'une spécification VeriSensor

Après avoir donné une vue d'ensemble de VeriSensor, nous présentons un exemple de réseau de capteurs sans fil biomédical tiré de la littérature du domaine qui nous permettra, au cours des sections 4 à 13, d'illustrer chacun des concepts évoqués dans cette section.

3 Exemple fil rouge

Nous présentons dans cette section un exemple de réseau de capteurs sans fil biomédical prototypé par Otto et al. [OMSJ05]. Il s'agit d'un réseau pour la surveillance corporelle (Body Area Network ou BAN). Le contexte du BAN est celui de la surveillance médicale à domicile de patients qui ont besoin d'être monitorés de façon continue sans être à l'hôpital. La surveillance médicale à domicile permet d'éviter une hospitalisation, ce qui est dans l'intérêt du patient et de l'équipe médicale.

Le BAN est composé de : *i)* un ensemble de nœuds capables de capter, traiter et communiquer les signes vitaux à un *Personal Server* ; (voir Figure 3.2) *ii)* un PDA (*Portable Digital Assistant*) qui retransmet les données vitales du patient à un centre médical à travers Internet.

Le BAN collecte les signes vitaux d'un patient qui se remet d'une crise cardiaque. Il vérifie que le patient est bien en train de faire de l'exercice régulièrement, comme cela est recommandé par les médecins. Les réseaux de capteurs sans fil réduisent le degré d'intrusivité dans la vie des patients.

Deux nœuds `Activity1` et `Activity2` captent le niveau d'activité du corps humain (en $Watts.kg^{-1}$) et détectent les périodes d'exercice physique (*i.e.*, activité du corps humain supérieure à 8 $Watts.kg^{-1}$) alors qu'un troisième nœud `ECGTilt1` collecte de manière périodique le rythme cardiaque avec un capteur ECG (Électrocardiogramme) et l'inclinaison du tronc (en termes de valeur absolue de l'angle par rapport à la verticale) avec le capteur *Tilt*. Les coordonnées du PDA `PDA1` et des nœuds `Activity1`, `Activity2` et `ECGTilt1` sont respectivement $\langle 0.1, -0.3, 0.0\rangle$, $\langle -0.1, -0.3, 0.0\rangle$ et $\langle 0.1, 0.4, 0.0\rangle$ (en mètres) dans le repère (x,y,z) sur la figure 3.2. La station de base est à la portée de tous les nœuds, qui communiquent avec elle en *single hop*.

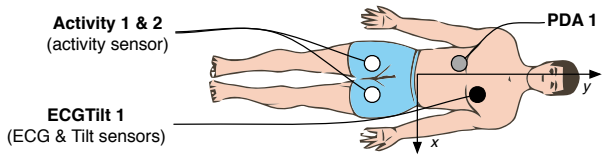


FIGURE 3.2 – Le réseau de capteurs sans fil biomédical

Dans ce qui suit, nous donnons d'abord la vue d'ensemble sur la syntaxe de VeriSensor. Ensuite, nous détaillons chacun des modèles présentés en section 2, en indiquant son objectif, sa sémantique, les hypothèses simplificatrices et abstractions de l'état de l'art (chapitre 2) sur lesquelles il se base, ainsi que sa syntaxe. Nous suivons une approche descendante : des aspects généraux du système vers la description des nœuds.

4 Vue d'ensemble de la syntaxe

Les concepts du DSML VeriSensor sont décrits par un méta-modèle. Un méta-modèle précise les concepts du domaine et les relations entre ces concepts. Nous exprimons le méta-modèle à l'aide d'UML.

La figure 3.3 présente le méta-modèle [1]. La spécification VeriSensor est ainsi composée d'un modèle de requêtes (métaclasse `ModeleRequetes`), d'un modèle de déploiement (`ModeleDeploiement`), de modèles de classes de nœuds (`ModeleClasseDeNoeuds`), d'un modèle d'environnement (`ModeleEnvironnement`) et de l'ensemble des tâches de vérification (`TacheDeVerification`). Chaque modèle de classes de nœuds référence des modèles décrivant les aspects capture, application, et réseau.

La spécification contient également un modèle de définition de types (`DefinitionDeTypes`) qui définit les domaines discrétisés des grandeurs physiques et de l'énergie. La spécification contient aussi des paramètres (`Parameter`) qui peuvent être affectés aux attributs numériques (*e.g.*, la période de capture) des différents modèles.

Le modèle de requêtes contient les spécifications des requêtes (`Requete`) qui doivent être traitées par le réseau de capteurs sans fil.

Le modèle de déploiement contient l'ensemble des instances de nœuds (`InstanceDeNoeud`) caractérisées par leur nom, leur position, et leur prochain saut.

Le modèle de l'environnement contient un ensemble de scénarios environnementaux (`ScenarioEnvironnemental`) auquel le réseau peut être soumis.

Chaque tâche de vérification fait référence à des variables et à des scénarios environnementaux pour réaliser la vérification (section 13).

1. EString désigne le type chaîne de caractères, EBool le type booléen, EInt le type entier, EDouble le type double

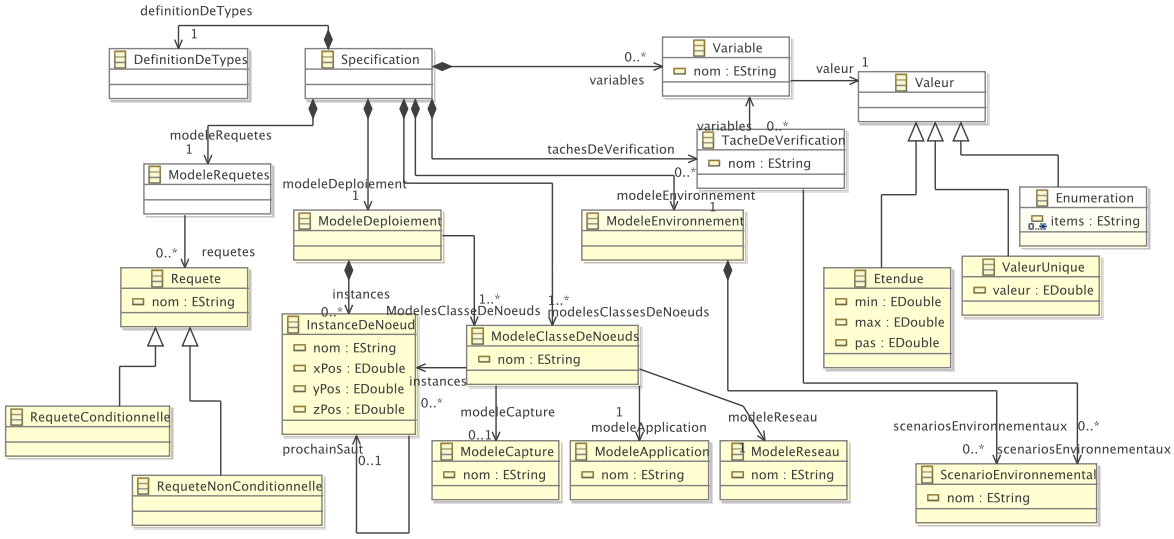


FIGURE 3.3 – Méta-modèle de haut niveau d'abstraction de VeriSensor

5 Modèle de déploiement

Le modèle de déploiement définit comment les nœuds sont répartis dans l'environnement. Il spécifie les instances de nœuds, leurs positions et les chemins de routage.

Le déploiement se base sur la notion d'instance de nœud qui capture un artefact physique (nœud réseau ou station de base). Le concept de classe de nœuds sert à ne définir qu'une fois les caractéristiques d'un nœud (ses capteurs, son application, son réseau et sa consommation énergétique) et il est détaillé dans la définition 3.7.

Définitions : On définit CL l'ensemble des classes de nœuds, ID un ensemble d'identifiants.

La définition suivante présente la notion d'instance de nœud.

Définition 3.1. ***Instance de nœud***

Une instance de nœud est un tuple $\langle id, cl, pos, n_h \rangle$ *où :*

- $id \in ID$ *est son identifiant.*
- $cl \in CL$ *est sa classe de nœuds.*
- $pos \in \mathbb{R}^3$ *est sa position dans l'espace.*
- $n_h \in ID$ *est son* next-hop

pos, la position du nœud, ne varie pas dans le temps. En effet, nous faisons l'hypothèse que tous les nœuds sont immobiles et qu'ils ne possèdent donc pas de système de locomotion (voir la définition 2.2 de l'état de l'art, p. 14).

n_h, le next-hop d'un nœud, est fixé et unique. En effet, nous supposons que les chemins de routage sont déjà construits et nous nous situons donc temporellement après la phase d'auto-configuration (voir définition 2.9 de l'état de l'art, p. 21). Les chemins de routage ne changent pas au cours du temps et il existe un unique chemin de routage pour chaque noeud.

Ces hypothèses sont justifiées par le risque d'explosion combinatoire. En effet, supposons que les chemins ne sont pas construits au moment du déploiement, il faudrait modéliser le processus d'auto-configuration. Celui-ci consiste en l'envoi d'un ensemble de messages de contrôle entre les noeuds pour s'accorder sur les chemins de routage. La prise en compte de cette phase d'envoi de messages suivie de la phase de fonctionnement nominal du réseau augmente l'entrelacement entre les actions et accroît le risque d'explosion combinatoire.

De même, si l'on suppose que les chemins de routage peuvent varier (le plus souvent pour reconfigurer le réseau à la suite de la mort d'un noeud) il faut d'abord modéliser les protocoles de communication de messages de contrôle entre les noeuds. Ces protocoles permettent de prendre connaissance d'une modification de l'état du réseau (*e.g.*, mort d'un noeud) et de s'accorder sur de nouveaux chemins de routage. Pour certains protocoles, les noeuds ont même besoin de recevoir périodiquement des messages de contrôles de la part des voisins signalant qu'ils sont en vie.

Finalement, l'hypothèse de l'unicité du *next-hop* permet d'éviter la prise en compte des algorithmes de répartition de charge. En effet, si plusieurs *next-hops* sont définis, ces algorithmes sélectionnent pour chaque envoi de message, un des *next-hops* comme destinataire. Ces algorithmes modifient également la répartition de charge suite à la mort d'un des *next-hops*.

En nous appuyant sur la définition d'une instance de nœud, nous définissons la notion de déploiement en VeriSensor qui décrit la topologie physique et logique du réseau (voir définition 2.16 de l'état de l'art, p. 26). La topologie physique est modélisée par les positions *pos* des nœuds. La topologie logique (liens de communication) est déduite à partir des positions des nœuds et des portées de leurs classes (voir définition 3.7).

Définition 3.2. ***Déploiement***

Etant donnés CL et ID,

Un Déploiement *est un ensemble N d'instances de nœuds satisfaisant :*

- $\forall (n,m) \in N^2$, $n \neq m$ *ou* $n.id = m.id$
- $\forall n \in N, \exists m \in N$ *tel que* $n.n_h = m.id$
- $\exists! b \in N : b.n_h = b.id$

La première condition impose que deux nœuds différents n'aient pas le même identifiant. La seconde condition spécifie que le *next-hop* d'un nœud doit appartenir au déploiement. La troisième condition indique qu'un déploiement contient une seule station de base qui joue le rôle de puits et ne retransmet aucun résultat. Cette condition se base sur la définition 2.4 de l'état de l'art (p. 16).

La Figure 3.4 présente un exemple de la syntaxe du modèle de déploiement modélisant le BAN. Le nom du système est spécifié (`BAN`). Pour chaque classe de

nœuds on indique la liste de ses instances et leurs attributs. Toutes les distances en VeriSensor sont exprimées en mètres. Par exemple, `ecgtilt1` est une instance de `ECGTilt`, située à la position (0.1,0.4,0.0), dont le prochain saut est `pda1` (la station de base). La station de base n'a pas de *next-hop*. On affecte son identifiant (`pda1`) à son *next-hop*.

```
System BAN {ECGTilt     => ecgtilt1  (x=0.1, y=0.4, z=0.0,  nextHop = pda1);
            Activity1    => activity1 (x=0.1, y=-0.3, z=0.0, nextHop = pda1);
            Activity2    =>  activity2 (x=-0.1, y=-0.3, z=0.0,  nextHop = pda1);
            PDA          => pda1      (x=-0.1, y=0.3, z=0.0, nextHop = pda1);}
```

FIGURE 3.4 – Déploiement du BAN

6 Modèle de l'environnement

Le modèle de l'environnement spécifie la liste de scénarios environnementaux possibles auxquels le réseau peut être soumis. Ces scénarios permettent de tester des propriétés qualitatives du modèle.

Définitions : On définit $\mathcal{G}$, l'ensemble des grandeurs physiques captées par le réseau de capteurs sans fil, et $Dom()$, la fonction qui à une grandeur $g \in \mathcal{G}$ associe l'ensemble de ses valeurs possibles.

6.1 Sémantique

Nous définissons d'abord le concept de scénario environnemental décrivant l'évolution des grandeurs captées par le système dans l'espace et dans le temps.

Définition 3.3. ***Scénario Environnemental***
Etant donnés $\mathcal{G}$, ID, $Dom()$
Un scénario environnemental s est caractérisé par une fonction $evolution()$ qui associe à tout $\langle id, g, t\rangle \in ID \times \mathcal{G} \times \mathbb{N}$ un élément de $Dom(g)$ tel que $evolution(id, g, t)$ désigne la valeur de la grandeur g à l'instant t, à la position du noeud id.

Un environnement VeriSensor est un ensemble de scénarios environnementaux

6.2 Syntaxe

La définition suivante présente la syntaxe d'un scénario environnemental.

Définition 3.4. ***Syntaxe***
Un scénario environnemental s est décrit par :
- *un attribut c indiquant son caractère cyclique,*
- *une durée totale d_{totale} du scénario*

- *un ensemble d'évolutions de grandeurs qui à chaque grandeur* $g \in (G)$ *associent son chronogramme d'évolution à la position de chaque noeud* id *du réseau.*

Le chronogramme d'évolution d'une grandeur $g \in \mathcal{G}$*, à la position d'un noeud* id*, est caractérisé par une suite de couples* $(d_i, V_i)_{1 \leq i \leq n}$ *avec* $(n \in \mathbb{N})$ *tels que :*

- $\forall i \in \mathbb{N} \mid 1 \leq i \leq n \ (d_i, V_i) \in \mathbb{N}^* \times Dom(g)$
- $\sum_{i=1}^{i=n} d_i = d_{totale}$.
- *Si le scénario est acyclique* $(c = false)$*, alors la fonction* $evolution()$ *correspondante est définie par :*
  - $\forall i \mid 2 \leq i \leq n-2$, $evolution(id,g,t) = V_i$ *si et seulement si* $t < \sum_{j=1}^{j=i} d_j$ *et* $\sum_{j=1}^{j=(i-1)} d_j \leq t$.
  - $evolution(id,g,t) = V_1$ *si et seulement si* $t < d_1$
  - $evolution(id,g,t) = V_n$ *si et seulement si* $\sum_{j=1}^{j=(n-1)} d_j \leq t$
- *Si le scénario est cyclique* $(c = true)$*, alors les instants* t *sont interprétés modulo* d_{totale}.

Le chronogramme d'évolution d'une grandeur g à la position d'un noeud id prend la forme d'une fonction en escalier qui vaut V_k pendant une durée d_k. La somme des durées d_k vaut la durée totale du scénario dans le cas acyclique, ou bien la durée totale d'un cycle du scénario, dans le cas cyclique.

Dans le cas acyclique, la valeur de $evolution(id,g,t)$ est égale à la valeur V_k prise par la fonction en escalier décrivant l'évolution de g à la position du noeud id, sur l'intervalle contenant l'instant t. Par ailleurs, l'équivalence : $evolution(id,g,t) = V_n$ si et seulement si $\sum_{j=1}^{j=(n-1)} d_j \leq t$, indique que l'on *prolonge* le chronogramme décrivant l'évolution de g à la position de id par la dernière valeur V_n.

Dans le cas cyclique, la valeur de $evolution(id,g,t)$ à un instant t *du cycle* est égale à la valeur V_k prise par la fonction en escalier décrivant l'évolution de g à la position du noeud id, sur l'intervalle contenant l'instant t.

La figure 3.5 représente un exemple de l'environnement du BAN. Il est constitué de deux scénarios `Resting` (Repos) et `BikeExercising` (exercice sur une bicyclette d'entraînement). Dans chaque scénario sont définies les évolutions des grandeurs captées par les différents capteurs. La définition d'une évolution de grandeur (`DefineEvolution`) référence le type associé à la grandeur physique et le fichier qui décrit cette évolution. Dans la figure 3.5, l'évolution de la grandeur `HeartBeat` dans le scénario `Resting` référence le type `HeartBeat` et le fichier `ConstantHeartBeat.data`.

La Figure 3.6 représente des extraits du fichier `IncreasingHeartBeat.data` décrivant une évolution croissante du rythme cardiaque survenant pendant l'activité physique.

Un fichier `.data` spécifie le chronogramme d'évolution d'une grandeur donnée à la position de chaque noeud du réseau. Ainsi il spécifie pour chaque nœud n du réseau,

```
import NullTilt.data ;
import IncreasingHeartbeat.data ;
import IncreasingActivity.data ;

import LaidBack.data ;
import ConstantHeartBeat.data ;
import NullActivity.data ;

Environment humanBody {
        EnvironmentalScenario  Resting(cyclic) {
                DefineEvolution HeartBeat  = ConstantHeartBeat.data ;
                DefineEvolution Tilt = LaidBack.data ;
                DefineEvolution Activity = NullActivity.data
        }

        EnvironmentalScenario BikeExercising(cyclic) { // on a bike
                DefineEvolution HeartBeat = IncreasingHeartBeat.data ;
                DefineEvolution Tilt = NullTilt.data ;
                DefineEvolution Activity = IncreasingActivity.data
        }
}
```

FIGURE 3.5 – Modèle de l'environnement du BAN

```
ecgtilt1
5, 70
5, 71
8, 73
...
activity1
5, 70
...
activity2
5, 70
...
pda1
5, 70
...
```

FIGURE 3.6 – Fichier IncreasingHeartBeat.data

son identifiant, suivi des tuples $(d_k, V_k)_{0 \leq k \leq m}$ indiquant l'évolution de g à la position de n.

A la position de `ecgtilt1`, le rythme cardiaque vaut 70 pendant les 5 premières unités de temps, puis 71 pendant les 5 suivantes, etc. Pour les autres noeuds, l'évolution du rythme cardiaque est la même, puisque le rythme est le même en tout point du corps humain. Par souci de flexibilité, nous laissons la liberté à l'utilisateur de VeriSensor de déterminer ce que représente une unité de temps. La somme des durées appartenant aux tuples liés au nœud `ecgtilt1` vaut 100 unités de temps, soit la durée totale du scénario. Il en est de même, pour les autres nœuds.

7 Modèle de requêtes

Le concept de requête nous permet de représenter tout type de comportement applicatif : détection d'évènements ou collecte de données (voir définitions 2.8, p.19

et 2.8, p.19 de l'état de l'art). Les requêtes encapsulent des tâches consistant à capter, traiter et envoyer périodiquement un ensemble de grandeurs physiques. Chaque requête est destinée à un sous ensemble des nœuds du réseau et diffusée par la station de base dans le réseau, à un instant précis. Dans le cas où des traitements sont préprogrammés dans un sous ensemble des nœuds du réseau, la station de base diffuse quand même une requête encapsulant ces traitements mais sans comptabiliser l'énergie consommée par cette diffusion (c'est un artefact de modélisation).

Définition : on définit $IDREQ \subset ID$ l'ensemble des identifiants des requêtes et t_{rec} l'instant de réception de la requête par un nœud.

Nous présentons le concept de requête non conditionnelle qui modélise la collecte de données et est inspiré de la définition 2.7 (p.18) de l'état de l'art.

Définition 3.5. ***Requête non conditionnelle VeriSensor***

Une requête non conditionnelle en VeriSensor est un tuple $\langle id, G, T_c, d_t, c_t, d_{totale}, T_t, T_e \rangle$ *avec :*

- $id \in IDREQ$*, l'identifiant de la requête.*
- $G \subset \mathcal{G}$, $G \neq \emptyset$*, l'ensemble des grandeurs reçues via la requête.*
- $T_c \in \mathbb{N}^*$*, la période d'envoi des ordres de capture de ces grandeurs.*
- $T_t \in \mathbb{N}^*$*, la période de traitement des grandeurs de* G $(T_c < T_t)$
- $d_t \in \mathbb{N}$*, la durée de traitement.*
- $c_t \in \mathbb{N}$*, le coût d'un traitement.*
- $T_e \in \mathbb{N}^*$*, la période d'envoi du résultat du traitement* $(T_t < T_e)$
- $d_{totale} \in \mathbb{N}^*$ *la durée totale d'exécution de la requête* ($d_{totale} > T_e$ *et* $d_{totale} > d_t$).

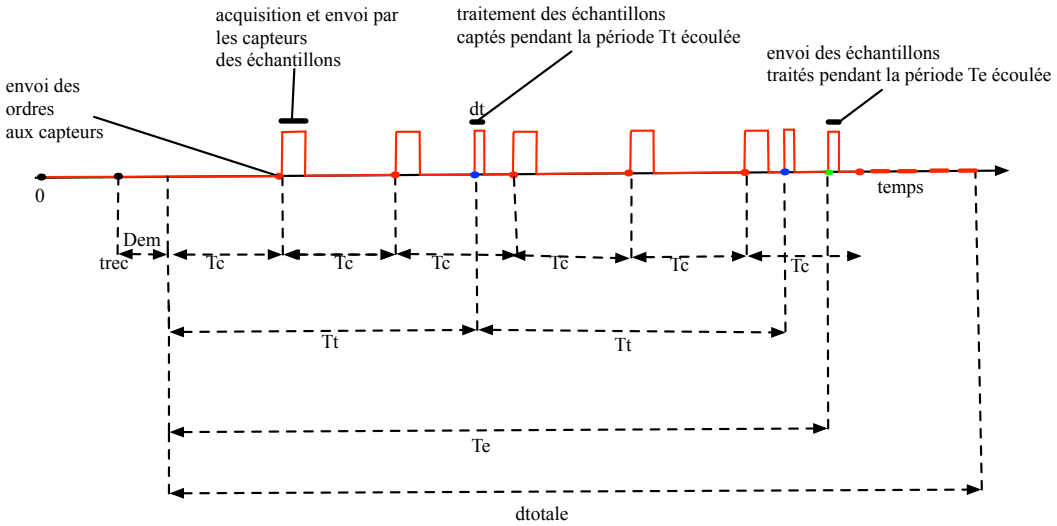


FIGURE 3.7 – Chronogramme pour une requête non conditionnelle

Chronologie des opérations. Le chronogramme en rouge de la figure 3.7 représente la chronologie des opérations réalisées par l'application d'un noeud qui reçoit une requête non conditionnelle. Les pointillés en rouge à la fin du chronogramme indiquent que le reste des opérations peuvent être déduites par le lecteur.

Les opérations sont comme suit :

L'application d'un noeud n reçoit la requête à l'instant t_{rec}. Elle démarre les capteurs des grandeurs $g \in G$, ce qui prend une durée totale de Dem. Elle attend une durée T_c.

Ensuite,

- A chaque période de capture T_c :
  - Elle envoie un ordre d'acquisition d'un échantillon à chaque capteur de n de la grandeur $g \in G$ et elle reçoit les valeurs captées.
  - Après la réception de tous les échantillons, l'application les traite pendant une durée d_t.
  - Si la condition $cond$ est satisfaite, une alerte est envoyée à la station de base.
- A chaque période de traitement T_t (voir les points bleus représentant les instants de traitement), tous les échantillons captés pendant la période T_t écoulée sont traités pendant une durée d_t.
- A chaque période T_e (voir les points verts représentant les instants d'envoi), tous les échantillons traités pendant la période T_e écoulée, sont envoyés à la station de base.

L'application arrête d'exécuter toute opération après l'écoulement d'une durée d_{totale}, comptabilisée à partir de la fin du démarrage de tous les capteurs de $g \in G$.

Attributs de la requête non conditionnelle. T_c permet de déterminer les instants auxquels sont envoyés les ordres aux capteurs des grandeurs de G. Le chronogramme 3.7représente ces instants par des points rouges sur l'axe d'évolution du temps.

T_t permet de déterminer les instants auxquels sont traités les échantillons acquis par les capteurs commandés par la requête. Le chronogramme 3.7 représente ces instants par des points bleus sur l'axe d'évolution du temps. A chacun de ces instants, les échantillons reçus pendant la période T_t écoulée sont traités.

On suppose que d_t, la durée de traitement des échantillons captés, est la même quelque soit l'instant de traitement.Par ailleurs, on fait l'abstraction de ne pas modéliser le contenu du traitement dans la requête, contrairement aux définitions 2.7 et 2.8 de l'état de l'art. On suppose que les valeurs provenant de l'environnement ont déjà subi le traitement (*e.g.* au lieu que ce soit la requête qui calcule une moyenne des températures, la moyenne est produite directement par l'environnement pour le capteur).

c_t est également supposé fixe pour tout traitement. Comme nous l'avons expliqué dans la définition 2.13, p. 24, de l'état de l'art, le fonctionnement des composants se fait selon plusieurs modes chacun caractérisés par une puissance consommée donnée. Sachant que nous discrétisons la consommation d'énergie en associant un coût énergétique à une opération donnée (*e.g.* le traitement) il suffit de s'assurer que ce coût prend en compte l'énergie consommée dans les différents modes de

fonctionnement du processeur. Par ailleurs, la discrétisation de la consommation d'énergie fait que VeriSensor ne prend pas en compte la *notion de nœud dormant* (dans l'état *sleep*).

T_e permet de déterminer les instants auxquels sont envoyés les échantillons traités. Le chronogramme 3.7 représente ces instants par des points verts sur l'axe d'évolution du temps. A chacun de ces instants, les échantillons traités, pendant la période T_e écoulée, sont envoyés à la station de base.

d_{totale} est définie comme la durée entre la fin du démarrage des capteurs et l'arrêt de l'exécution de la requête (voir Chronogramme en figure 3.8).

Nous présentons le concept de requête conditionnelle qui modélise la détection d'évènement et est inspiré de la définition 2.8 (p.19) de l'état de l'art.

Définition 3.6. ***Requête conditionnelle VeriSensor***
Une requête conditionnelle en VeriSensor est un tuple $\langle id, G, T_c, d_t, c_t, d_{totale}, cond \rangle$ *avec :*

- $id \in IDREQ$*, l'identifiant de la requête.*
- $G \subset \mathcal{G}$, $G \neq \emptyset$*, l'ensemble des grandeurs reçues via la requête.*
- $T_c \in \mathbb{N}^*$*, la période d'envoi des ordres de capture de ces grandeurs.*
- $d_t \in \mathbb{N}$*, la durée de traitement.*
- $c_t \in \mathbb{N}$*, le coût d'un traitement.*
- $d_{totale} \in \mathbb{N}^*$ *la durée totale d'exécution de la requête* ($d_{totale} > T_c$ *et* $d_{totale} > d_t$)*.*

Une requête conditionnelle ne spécifie pas de période de traitement, ni d'envoi. Le traitement suit la capture et permet de vérifier si l'évènement s'est produit (*i.e.*, la condition est satisfaite). L'envoi se fait juste après, si l'événement a eu lieu.

En effet, l'application qui exécute une requête conditionnelle est contrainte d'envoyer une alerte, le plus vite possible après la détection d'un évènement (voir définition 2.8, p.19).

Chronologie des opérations. Le chronogramme 3.8 représente la chronologie des opérations réalisées par l'application d'un noeud qui reçoit une requête conditionnelle. Les pointillés en rouge à la fin du chronogramme indiquent que le reste des opérations peuvent être déduites par le lecteur.

Les opérations sont comme suit : l'application reçoit la requête à l'instant t_{rec}, elle démarre les capteurs des grandeurs $g \in G$ ce qui prend une durée totale de Dem, puis elle attend une période égale à T_c.

Ensuite, à chaque période T_c :

- Elle envoie un ordre d'acquisition d'un échantillon à chaque capteur de n de la grandeur $g \in G$.
- Après la réception de tous les échantillons, l'application les traite pendant une durée d_t.
- Si la condition $cond$ est satisfaite, une alerte est envoyée à la station de base.

L'application arrête d'exécuter toute opération après l'écoulement d'une durée d_{totale}, comptabilisée à partir de la fin du démarrage de tous les capteurs de $g \in G$.

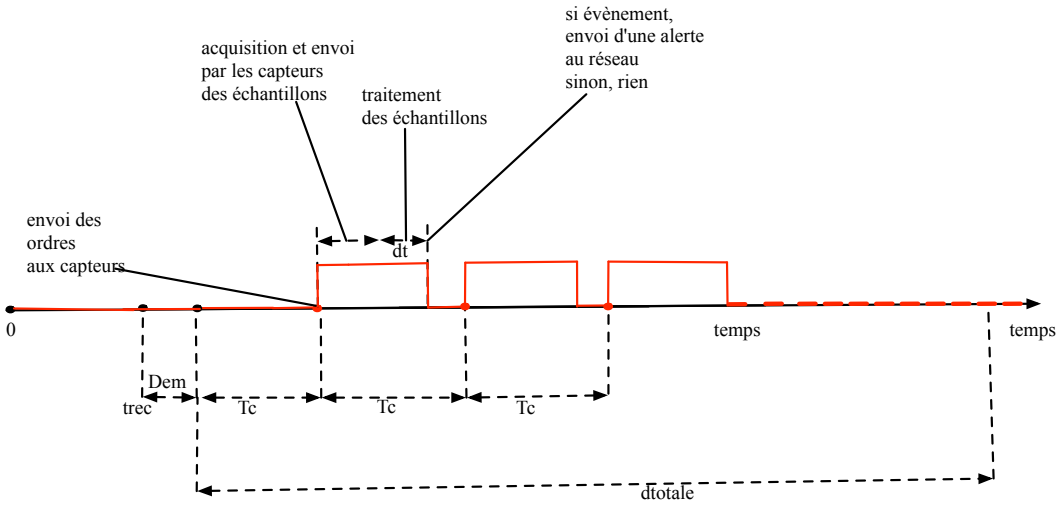


FIGURE 3.8 – Chronogramme pour une requête conditionnelle

Attributs de la requête conditionnelle. T_c permet de déterminer les instants auxquels sont envoyés les ordres aux capteurs des grandeurs de G. Ces instants sont représentés par des points rouges sur l'axe d'évolution du temps.

d_t correspond à la durée de traitement des échantillons reçus par l'application.

On suppose que cette la durée de traitement des échantillons captés est la même peu importe l'instant de capture car le traitement (évaluation de la condition) est identique.Par ailleurs, on fait l'abstraction de ne pas modéliser le contenu du traitement dans la requête, contrairement aux définitions 2.7 et 2.8 de l'état de l'art. On suppose que les valeurs provenant de l'environnement ont déjà subi le traitement (*e.g.* au lieu que ce soit la requête qui calcule une moyenne des températures, la moyenne est produite directement par l'environnement pour le capteur).

c_t est également supposé fixe pour tout traitement. Comme nous l'avons expliqué dans la définition 2.13, p. 24, de l'état de l'art, le fonctionnement des composants se fait selon plusieurs modes chacun caractérisés par une puissance consommée donnée. Sachant que nous discrétisons la consommation d'énergie en associant un coût énergétique à une opération donnée (*e.g.* le traitement) il suffit de s'assurer que ce coût prend en compte l'énergie consommée dans les différents modes de fonctionnement du processeur. Par ailleurs, la discrétisation de la consommation d'énergie fait que VeriSensor ne prend pas en compte la *notion de nœud dormant* (dans l'état *sleep*).

d_{totale} est définie comme la durée entre la fin du démarrage des capteurs et l'arrêt de l'exécution de la requête (voir Chronogramme en figure 3.8).

$cond$ est la condition qui caractérise l'évènement que l'on attend. Si $cond$ est satisfaite par les échantillons captés et traités pendant d_t, alors une alerte est immédiatement envoyée à la station de base. $cond$ peut aussi être une condition sur un vecteur de caractéristiques (*e.g.* pour la classification et le suivi d'objet, voir définition 2.8, p. 19 de l'état de l'art). Dans ce cas, il suffit de définir les éléments du vecteur de caractéristiques comme les grandeurs. Une grandeur représente alors le

résultat d'un prétraitement sur des grandeurs environnementales (*e.g.*, une moyenne ou un maximum de températures).

Remarque : Une requête ne spécifie pas de destinataires. En effet, nous faisons l'abstraction d'indiquer, pour chaque classe de nœud, les requêtes dont les instances de la classe sont destinataires. Il est légitime de supposer que des nœuds de la même classe ont le même comportement applicatif et supportent donc les mêmes requêtes.

Un modèle de requêtes est un ensemble de requêtes ayant des identifiants uniques.

La Figure 3.9 représente le modèle de requêtes dans l'exemple du BAN, qui contient quatre requêtes : `AskHeartBeat`, `AskTilt`, `AskActivity1`, `AskActivity2`. Par exemple, la requête `AskHeartBeat` consiste à capter pendant la durée de la vérification (`duration=infinity`), chaque 13 unités de temps (`sensingPeriod=13`) le rythme cardiaque et à envoyer une alerte s'il est inférieur à 40 (bradycardie).

```
Queries medical {
 condQuery AskHeartBeat (condition = (HeartBeat < 40), duration = infinity,
 sensingPeriod = 13, processingCost = 4 , processingDuration = 2);

 nonCondQuery AskTilt (quantities = [Tilt], duration = infinity,
 sensingPeriod = 8, processingPeriod= 13,
 sendingPeriod = 20, processingCost = 1, processingDuration = 0);

 condQuery AskActivity1 (condition = (Activity > 8), duration = infinity,
 sensingPeriod = 6, processingCost = 4, processingDuration = 2);

 condQuery AskActivity2 (condition = (Activity > 8), duration = infinity,
 sensingPeriod = 9, processingCost = 4, processingDuration = 2);}
```

FIGURE 3.9 – Modèle de requêtes pour le BAN

8 Classe de nœuds

La notion de classe de nœuds permet de factoriser les caractéristiques communes d'un ensemble de nœuds. En effet, comme nous l'avons vu dans la définition 2.5 de l'état de l'art, (p. 16) un réseau de capteurs sans fil peut être constitué de nœuds de différents types, possédant des caractéristiques différentes du point de vue des capteurs, de l'application, du réseau ou de la consommation d'énergie. Un mécanisme d'instanciation permet de définir les nœuds à partir d'une classe de nœuds.

On définit :

- C l'ensemble des capteurs du réseau de capteurs sans fil (voir définition 3.8)
- A l'ensemble des sous-modèles applications du réseau de capteurs sans fil
- R l'ensemble des sous-modèles réseaux du réseau de capteurs sans fil

La définition 3.7 suivante présente le concept de classe de nœuds en VeriSensor.

Définition 3.7. ***Classe de nœuds***
Etant donnés C, A, R,
Une classe de nœuds VeriSensor est un tuple $\langle c, a, r, e \rangle$ *avec*

- $c \subset C$, *le sous-modèle capture de la classe de nœuds.*
- $a \in A$ *le sous-modèle application de la classe de nœuds.*
- $r \in R$ *le sous-modèle réseau de la classe de nœuds.*
- $e \in \mathbb{N}^*$ *la quantité d'énergie initiale de la classe de nœuds.*

Nous définissons le concept de classe de nœuds à partir d'un ensemble d'extensions et d'abstractions faites à partir de l'état de l'art. En effet, dans un même réseau, plusieurs nœuds peuvent être du même modèle commercial (*e.g.* Mica2, IMote). Dans Matilda [WBSO07], la notion de type de nœud est définie comme la spécification d'un modèle commercial et la portée de son capteur. Nous avons étendu cette notion car elle ne nous semble pas pertinente du point de vue comportemental. En effet, la caractéristique matérielle d'un nœud ne détermine pas son comportement : deux nœuds du même modèle commercial peuvent, par exemple, exécuter des applications différentes et avoir donc des comportements différents. Le concept de classe de nœud proposé étend la notion de type de nœud et résout le problème de comportement. Ainsi, deux nœuds de même classe ont des comportements très proches car toutes les fonctions qu'ils réalisent (*i.e.*, la capture, le traitement, la communication et la gestion de l'énergie) sont similaires. Cette décomposition en ces 4 aspects nous paraît la plus pertinente (voir définition 2.2, p. 14, de l'état de l'art). Elle est validée par [ASSC02].

Par ailleurs, nous faisons l'hypothèse par rapport à la définition 2.2 de l'état de l'art p.14 qu'un nœud ne possède pas de générateur d'énergie. En conséquence, la quantité d'énergie du nœud est une fonction décroissante du temps. Cette hypothèse garantit d'une part des traces finies lors du model checking, d'autre part une taille de l'espace d'états générés plus réduite. En effet, si le noeud possède un générateur d'énergie, il peut mourir puis regagner de l'énergie (la limite étant la quantité maximale d'énergie stockable par les batteries).

```
NodeClass ECGTilt {
    sensing => ECGTsensing;
    application => ECGTApplication;
    network => ECGTNetwork;
    initialEnergy = 1000;}
```

FIGURE 3.10 – Modèle de la classe de noeuds ECGTilt du BAN

D'un point de vue génie logiciel, le modèle de classe de nœuds contribue à la *modularité* et à la *réutilisabilité* dans VeriSensor. En effet, comme nous séparons la définition de chaque aspect d'un nœud, chacun peut être référencé par plusieurs classes différentes.

La Figure 3.10 décrit la classe `ECGTilt`. Elle référence les sous-modèles `ECGTsensing` pour la capture, `ECGTApplication` pour l'application, `ECGTNetwork` pour le réseau, et affecte à l'énergie initiale 1000 unités.

9 Sous-modèle capture

Le sous-modèle capture décrit le comportement des capteurs d'une classe de nœud à travers leurs caractéristiques techniques spécifiques.

La définition suivante présente le concept de capteur VeriSensor, qui fait l'acquisition d'une grandeur environnementale et la transmet à l'application.

Définition 3.8. ***Capteur VeriSensor***
Etant donné $\mathcal{G}$,
Un capteur VeriSensor est un tuple $\langle g, d_d, d_c, p_c, c_c \rangle$ avec :
- $g \in \mathcal{G}$*, la grandeur physique captée.*
- d_d*,* $d_c \in \mathbb{N}$*, la durée de démarrage et la durée d'une capture du capteur.*
- $p_c \in \mathbb{R}$*, la portée du capteur.*
- $c_c \in \mathbb{N}$*, le coût d'une capture.*

La durée de démarrage (d_d) correspond à la notion de *startup-time* (voir définition 2.3 de l'état de l'art, p. 15).

La durée de capture (d_c) est la durée nécessaire pour réaliser une acquisition. A l'issue de cette durée, l'échantillon de la grandeur physique est capté. Cette notion est particulièrement utile quand on abstrait un ensemble d'acquisitions par une seule. En effet, en principe, la durée de capture d'un seul échantillon est négligeable par rapport aux autres durées du système (donc approximée par 0). Mais, souvent on abstrait une suite de n d'acquisitions comme une seule. Dans ce cas, la durée de capture est la durée pour capter les n échantillons (non négligeable). Le coût de capture doit aussi comptabiliser ces acquisitions.

La portée p_c du capteur (voir définition 2.3 de l'état de l'art, p. 15), permet de déterminer les surfaces couvertes par les capteurs d'une même grandeur physique. Cette donnée est nécessaire pour la vérification de certaines propriétés.

```
sensing ECGTsensing {
   sensor ECGSensor (                    sensor TiltSensor (
          PhysicalQuantity = Heartbeat,          PhysicalQuantity = Tilt,
          SensingRange = 0.1,                    SensingRange = 0.1,
          StartupTime = 4,                       StartupTime = 0,
          SensingDuration = 1,                   SensingDuration = 2,
          SensingCost = 2);                      SensingCost = 3) }
```

FIGURE 3.11 – le sous modèle capture `ECGTsensing`

La notion de coût de capture c_c provient de la discrétisation de la consommation d'énergie. En effet, en principe un capteur a plusieurs modes de fonctionnement caractérisés par une puissance de consommation donnée (voir définition 2.13 de l'état de l'art, p.24). Sachant que nous discrétisons la consommation d'énergie en associant un coût énergétique à une opération donnée, il suffit de s'assurer que ce coût prend en compte l'énergie consommée dans les différents modes de fonctionnement du capteur.

La Figure 3.11 représente le sous-modèle capture de la classe de nœud `ECGTilt`. Elle est constituée de deux capteurs : `ECGSensor` qui capte le rythme cardiaque et

`TiltSensor` qui capte l'inclinaison. Le capteur `ECGSensor` par exemple, a une portée de 0.1 m, il a besoin de 4 unités de temps pour démarrer, une unité de temps pour faire son acquisition, qui lui coûte 2 unités d'énergie. Par souci de flexibilité, nous laissons la liberté à l'utilisateur de VeriSensor de définir ce que représente pour lui une unité d'énergie.

10 Sous-modèle application

Le sous-modèle application décrit le comportement de l'application : comment elle contrôle les capteurs et communique avec le réseau.

10.1 Sémantique

Définitions : On définit REQ l'ensemble des requêtes du réseau de capteurs sans fil émises par la station de base.

La définition 3.9 suivante présente le concept d'application d'un nœud dans VeriSensor. C'est le concept qui exécute un sous ensemble des requêtes que le nœud reçoit.

Définition 3.9. ***Application de nœud VeriSensor***
Etant donné REQ,
L'application d'un nœud est un sous ensemble de REQ.

L'application d'un nœud est issue de l'abstraction indiquée en section 1.3 de ce chapitre (*i.e.*, spécifier pour chaque classe de nœud, les requêtes dont les instances sont destinataires).

La définition 3.10 suivante présente le concept d'application de la station de base dans VeriSensor qui diffuse à des instants précis dans le réseau les requêtes.

Définition 3.10. ***Application de station de base VeriSensor***
Etant donné REQ,
L'application d'une station de base est un sous ensemble de $REQ \times \mathbb{N}$ avec $\mathbb{N}$ les instants auxquels ces requêtes sont diffusées dans le réseau.

Nous ne modélisons ni l'agrégation des données, ni la fusion de données (voir définition 2.7 et 2.8 de l'état de l'art, p. 18 et p.19 de l'état de l'art), car le niveau d'abstraction est trop bas et cela augmente le risque d'explosion combinatoire. En effet, l'agrégation réalise des opérations arithmétiques sur les données et nécessite une connaissance du contenu de la donnée.

Cependant, nous prenons en compte la réduction de consommation d'énergie due à l'agrégation, en réduisant les coûts d'émission associés aux nœuds retransmetteurs des données. En effet, l'agrégation des données (*e.g.*, calcul d'une moyenne de plusieurs valeurs) par un noeud retransmetteur réduit la quantité de donnée à

communiquer et donc le coût d'émission des messages (voir définition2.7 du chapitre 2 page 18).

10.2 Syntaxe

La Figure 3.12 présente les sous-modèles application `ECGTApplication` de la classe de nœuds `ECGTilt BaseApplication` de la classe de nœud `PDA` qui joue le rôle de station de base. `ECGTApplication` est paramétré par le mot clef `Node` et est destinataire de deux requêtes : `AskHeartBeat` et `AskTilt`. Le sous-modèle application du `PDA` est paramétré par le mot clef `Base` et diffuse les requêtes `AskHeartBeat`, `AskTilt`, `AskActivity1` et `AskActivity2` respectivement aux instants : 4, 10, 14 et 18.

```
// Node ECGTApplication
Application <Node> ECGTApplication {
                Queries = AskHeartBeat, AskTilt ;
}

// Base
Application <Base> BaseApplication {
                BroadcastTime AskHeartBeat=4;
                BroadcastTime AskTilt=10;
                BroadcastTime AskActivity1=14;
                BroadcastTime AskActivity2=18
}
```

FIGURE 3.12 – Le sous-modèle application `ECGTApplication` et `BaseApplication`

11 Sous-modèle réseau

Le réseau gère les communications du nœud avec l'extérieur.

La définition suivante présente le concept de réseau d'une classe de nœud dans VeriSensor.

Définition 3.11. ***Réseau VeriSensor***

Le réseau VeriSensor est un tuple $\langle d_e, p_e, c_e, c_r, \tau \rangle$ *avec :*

- $d_e \in \mathbb{N}$ *la durée d'émission d'un message.*
- $p_e \in \mathbb{R}$ *la portée d'émission.*
- c_e *et* $c_r \in \mathbb{N}$ *le coût de réception et d'émission d'un message.*
- $\tau = 1/p$, $p \in \mathbb{N}$, *est le taux maximal de perte des messages envoyés par l'instance.*

Le concept de réseau est issu d'un ensemble d'hypothèses simplificatrices et d'abstractions que nous avons faites à partir de l'état de l'art. La séparation de la partie réseau et de la partie applicative du nœud est en partie inspirée des modèles TCP/IP et OSI, dans lesquels le rôle d'un terminal est décomposé en une couche applicative et un ensemble de couches gérant les communications (*e.g.*, couche accès réseau, internet, et transport, dans le modèle TCP/IP).

La durée d'émission d'un message (d_e) est fixe quelque soit sa taille. Cette abstraction simplifie le modèle et permet d'obtenir des résultats satisfaisants en fixant, par exemple, une approximation de la durée d'émission à celle du plus long message (ou du plus court, en fonction des cas).

La durée d'émission inclut aussi une approximation du retard possible dû aux retransmissions multiples provoquées par la couche MAC en cas d'erreur. En effet, on ne modélise pas de manière plus détaillée la couche MAC (voir définition 2.11 de l'état de l'art, p. 23) car elle peut impliquer des retransmissions multiples du même message, source d'entrelacement et donc d'explosion combinatoire.

La portée d'émission (p_e) permet de déterminer les nœuds voisins du nœud courant.

τ est un taux maximal, par conséquent le taux réel est choisi de manière indéterministe entre 0 et τ. Cette notion est nécessaire et nous permet de prendre en compte (avec une marge d'erreur) les pertes éventuelles. Nous considérons qu'un message est perdu lorsqu'il n'arrive pas à son destinataire, ou qu'il arrive à son destinataire avec tellement d'erreurs qu'il ne peut pas être corrigé et reconstitué. La prise en compte des pertes de messages dans VeriSensor nous permet de vérifier des propriétés dépendant de la proportion de messages (ou évènements) reçus par la station de base.

Nous supposons que les coûts d'émission c_e et de réception c_r sont fixes quelque soit la taille du message. Là encore, une solution consiste à approximer par excès les coûts d'émission et de réception. On obtient certes une durée de vie d légèrement inférieure à la durée de vie réelle, mais la garantie que le réseau est vivant pendant au moins d unités de temps.

La Figure 3.13 représente le sous-modèle réseau du nœud `ECGTilt1`. Sa portée est de 1 mètre. Les coûts de réception et d'émission sont respectivement de 4 et 5 unités d'énergie. La durée d'une émission est d'1 unité de temps.

```
// Network ECGTNetwork
Network ECGTNetwork {
        Range = 100 ;
        ReceptionCost = 4 ;
        EmissionCost = 5 ;
        EmissionDuration = 1 ;
        MaximumErrorRate = 0.2
 }
```

FIGURE 3.13 – Le sous-modèle réseau du noeud `ECGTilt1`

12 Modèle des types de données

Le modèle des types de données définit les valeurs possibles des grandeurs physiques du système

La définition suivante présente ce modèle.

Définition 3.12. ***Modele des types de données***
Etant donné $\mathcal{G}$
Un modèle de types de données VeriSensor *est caractérisé par la fonction $Dom()$ qui à chaque grandeur $g \in \mathcal{G}$ associe l'ensemble de ses valeurs possibles : un ensemble de valeurs numériques ou un ensemble de chaînes de caractéres.*

La Figure 3.14 représente le modèle des types de données pour le BAN. Un type de données peut consister en une énumeration de chaînes de caractères séparées par des virgules, ou en une étendue de valeurs (flottantes ou entières) spécifiant une valeur minimale, maximale et un pas. Par exemple, le rythme cardiaque (`HeartBeat`) peut prendre une valeur entière comprise entre 0 et 200.

```
type HeartBeat is 0..200 step 1;
type Tilt is 0..180 step 2;
type Activity is 0..15 step 1;
```

FIGURE 3.14 – Définition des types de données

13 Spécification des propriétés à vérifier

Dans cette section, nous indiquons comment se fait la spécification des propriétés à vérifier en VeriSensor. Nous affinons également notre vision des propriétés présentées dans le chapitre 2 (section 2.6).

La spécification des propriétés consiste en un ensemble de tâches de vérification qui encapsulent ce que l'utilisateur veut vérifier sur le modèle du réseau de capteurs sans fil. Chaque tâche de vérification définit plusieurs instances du modèle du réseau de capteurs sans fil et de l'environnement. Pour cela, elle instancie les caratéristiques à valeur numérique du modèle du réseau de capteurs sans fil (`startupTime` et `sensingPeriod` sont à valeur numérique mais pas le `nextHop` par exemple) et elle instancie l'environnement par un ou plusieurs scénarios environnementaux.

Pratiquement, une tâche de vérification spécifie des plages de variation pour un sous ensemble des attributs du réseau de capteurs sans fil et une plage de variation pour l'environnement (*i.e.* un sous ensemble des scénarios environnementaux spécifiés dans le modèle de l'environnement). Le produit cartésien des plages de variation des attributs et de l'environnement représente l'ensemble des instances de la tâche de vérification.

13.1 Sémantique

Dans cette sous-section, nous présentons la sémantique du modèle de spécification de propriétés en VeriSensor, ainsi que la liste des propriétés supportées.

Nous définissons d'abord le concept d'instance de spécification qui représente une spécification, où l'on a affecté une valeur à chaque attribut.

Définition 3.13. ***Instance de spécification***

Une instance de spécification est obtenue en affectant :

- *à chaque attribut numérique une valeur*
- *à l'environnement, un scénario environnemental parmi ceux qui sont spécifiés dans le modèle environnement, ou un scénario environnemental libre.*

Le scénario environnemental libre produit n'importe quelle valeur de grandeur à n'importe quel instant. Il est utilisé pour fournir toutes les situations possibles du point de vue de l'environnement et permet de calculer la durée de vie dans le pire des cas (voir chapitre 4).

Une instance de spécification définit donc un certain réseau de capteurs sans fil avec des caractéristiques données, plongé dans un scénario environnemental spécifique.

A partir du concept d'instance de spécification, nous définissons la notion de tâche de vérification qui consiste à réaliser une action (*e.g.*, tracer l'évolution graphique) sur une métrique (*e.g*, taux de messages reçus par la station de base) pour plusieurs instances de spécification.

Une métrique est une mesure pour évaluer de façon quantitative un aspect du réseau. Par exemple, le taux de messages reçus par la station de base permet d'évaluer de façon quantitative les messages reçus.

Définition 3.14. ***Tâche de vérification***

Soit c_m une métrique, une condition sur des métriques ou un attribut numérique de la spécification.

Une tâche de vérification est un tuple $(Specs, c_m, action)$ *avec*

- $Specs$ *un ensemble d'instances de spécification.*
- c_m *une métrique ou une condition sur plusieurs métriques.*
- $action \in \{verify, plot, maximize, minimize\}$ *une action que l'on veut réaliser sur la métrique pour les instances de spécifications de* $Specs$*.*

c_m désigne :

- une condition sur des métriques (*i.e.*, la propriété que l'on veut vérifier sur les instances de $Specs$), si l'action est $verify$. Dans ce cas, on veut vérifier que la condition est satisfaite par toutes les instances de $Specs$.
- une métrique (*e.g.*, un taux de messages reçus par la station de base) si l'action est $maximize$, $minimize$. Dans ce cas, on veut trouver la valeur optimale de la métrique pour les instances de $Specs$.
- une métrique (*e.g.*, un taux de messages reçus par la station de base), si l'action est $plot$. Dans ce cas, on veut tracer la courbe d'évolution de la métrique en fonction des valeurs d'au plus deux attributs (plot en deux dimensions ou trois dimensions).

La spécification des propriétés à vérifier est constituée d'un ensemble de tâches de vérification.

Toutes les tâches de vérification d'une spécification VeriSensor ne sont pas exécutées. Avant l'exécution de la transformation, l'utilisateur choisit parmi elles, une tâche à effectuer.

13.1.1 Propriétés supportées par VeriSensor

Dans le tableau suivant, nous présentons les métriques m, et les propriétés dont elles sont tirées :

Propriété de l'état de l'art	Métrique
P1	la durée de vie dans le pire des cas.
P2	la proportion de nœuds vivants.
P3	la proportion de nœuds disposant d'un chemin de connexion vers la station de base.
P4	le taux des messages reçus par la station de base, contenant les résultats des requêtes ($ReqId_1, ReqId_2, \ldots, ReqId_n$)
P5	la proportion des messages dont la latence est inférieure à `MaxLatence` (`MaxLatence`>0).
P6	l'existence d'un interblocage (`Deadlock`)
P2 relaxée	la proportion des nœuds vivants pour chaque intervalle de durée Δ_t
P3 relaxée	la proportion de nœuds disposant d'un chemin de connexion vers la station de base pour chaque intervalle de durée Δ_t
P4 relaxée	le taux des messages reçus par la station de base pour chaque intervalle de durée Δ_t, et contenant les résultats des requêtes ($ReqId_1, ReqId_2, \ldots, ReqId_n$)
P5 relaxée	la proportion des messages dont la latence est inférieure à `MaxLatence`, pour chaque intervalle de durée Δ_t.

TABLE 3.1 – Propriétés et métriques

Les **six** premières métriques sont dégagées à partir des propriétés de l'état de l'art (section 2.6) P1 à P6.

Concernant P1, nous adoptons la définition de durée de vie selon laquelle le réseau est mort lorsque le premier nœud meurt. En effet, c'est la plus utilisée dans la littérature [DHHV05] (voir chapitre 2, p.26), donc elle nous permet d'utiliser VeriSensor sur un plus grand nombre d'applications de réseaux de capteurs sans fil. Par ailleurs, nous pouvons exprimer n'importe quelle autre définition de durée de vie de l'état de l'art, car VeriSensor supporte le calcul des métriques qui leur sont associées (voir tableau 3.1).

Nous considérons qu'un nœud est vivant s'il possède encore assez d'énergie pour communiquer (*i.e.*, émettre et recevoir).

Concernant P4, elle prend en paramètres un sous-ensemble des identifiants de requêtes diffusées par la station de base. Si les identifiants en paramètre correspondent à des requêtes conditionnelles, alors P4 désigne un taux d'évènements.

Les **quatre** autres métriques sont tirées de variations (version relaxée) des propriétés P2 à P5 présentées ci-dessous.

L'idée derrière la version relaxée, reprise de Dietrich et al. [DD09], est qu'il n'est pas nécessaire qu'une exigence soit satisfaite à chaque instant t, mais plutôt durant un intervalle de temps de durée Δ_t.

Par exemple, dans les applications de surveillance environnementale, si une proportion de la zone d'intérêt est couvert par les capteurs à un instant donné d'un intervalle I et que le reste de la zone d'intérêt est couvert à un autre instant de I, on peut considérer que toute la zone d'intérêt est couverte sur l'intervalle I.

Les versions relaxées sont importantes car, pour plusieurs application réelles de réseaux de capteurs sans fil, les propriétés non relaxées sont trop fortes : elles ne sont jamais vérifiées.

Nous ne fournissons pas de version relaxée pour P1 car vérifier que la durée de vie pour tout intervalle de temps de durée Δ_t est inférieure à une borne n'a pas de sens.

Nous présentons dans ce qui suit, les versions relaxées des propriétés P2 à P6 :

P2 relaxée exprime le fait que pour tout intervalle de temps de durée Δ_t, la proportion des nœuds qui sont vivants pendant au moins un instant de cet intervalle est supérieure à une borne v. On n'exige plus que le nœud soit vivant pendant l'intervalle complet.

Notations pour la propriété P2 relaxée :
Soit t_i un instant. On note :

- $Vivants(t_i)$ l'ensemble des nœuds vivants du réseau de capteurs sans fil à l'instant t_i.
- $Total$ l'ensemble des nœuds du réseau.
- $Card()$ la fonction qui à un ensemble associe le nombre de ses éléments.

Propriété. (P2 relaxée) Pour tout intervalle $[a,b]$ de durée Δ_t, il existe une suite d'instants $(t_i)_{1\leq i\leq n}$ appartenants à $[a,b]$ telle que : $\frac{Card[\bigcup_{i=1}^{n}(Vivants(t_i)]}{Card(Total)} \geq v$.

P2 relaxée peut être vérifiée, sans que P2 ne soit vérifiée, dans le cas où les nœuds possèdent un générateur d'énergie. Par exemple, il peut exister des intervalles contenant un instant où le nombre de nœuds vivants est nul (*i.e.* P2 non vérifiée), mais où à un instant ultérieur dans cet intervalle, le nombre de nœuds vivants est supérieur à la borne v (gain d'énergie grâce au générateur d'énergie). Cependant, nous faisons l'hypothèse de l'absence de générateur d'énergie (voir section 8) et par conséquent nous ne supportons pas P2 relaxée dans VeriSensor.

La version relaxée de P3 exprime le fait que pour tout intervalle de temps de durée Δt, la proportion des nœuds qui peuvent communiquer avec la station de base est supérieur à une borne c.

Définitions pour la propriété P3 relaxée :
Un nœud n_0 dispose d'un chemin de connexion vers la station de base pour un intervalle de durée Δt, si et seulement si, il existe une suite *croissante* d'instants $(t_i)_{1 \leq i \leq (m+1)}$ et une suite de nœuds $(n_i)_{1 \leq i \leq m}$ vivants tels que :

- pour tout i tel que $0 \leq i \leq (m-1)$, il existe un lien de connexion entre n_i et n_{i+1} à t_{i+1}
- il existe un lien de connexion entre n_m et la station de base à l'instant t_{m+1}.

Propriété. (P3 relaxée) Pour tout intervalle de durée Δt, la proportion de nœuds qui disposent d'un chemin de connexion vers la station de base est supérieure à c.

Propriété. (P4 relaxée) Pour chaque intervalle de durée Δ_t, le taux des messages contenant les résultats des requêtes ($ReqId_1$, $ReqId_2$, ... , $ReqId_n$), reçus par la station de base, doit être supérieure à τ.

Propriété. (P5 relaxée) Pour chaque intervalle de durée Δ_t la proportion de messages ayant une latence inférieure à $MaxLatence$ doit être supérieure à λ.

La version relaxée de P5 est en fait plus forte. En effet, si pour chaque intervalle, la proportion de messages de latence inférieure à $MaxLatence$ est supérieure ou égale à λ, alors, à tout instant, cette proportion est supérieure ou égale à λ.

La version relaxée de P6 est un cas particulier de P4 sachant qu'un message qui contient un évènement est un message particulier.

Propriété. P6 relaxée le taux d'évènements caractérisés par la requête $ReqId$, doit être supérieur à τ ($0 < \tau \leq 1$, pour chaque intervalle de durée Δ_t.

Concernant la propriété P7, il peut exister dans un modèle de réseau de capteurs sans fil des interblocages « attendus » et « inattendus ». Les interblocages attendus se produisent lorsque tous les noeuds manquent d'énergie. Il est normal que l'état de l'espace d'états n'aie pas de successeur. Les interblocages inattendus arrivent alors qu'il existe au moins un noeud qui possède encore de l'énergie. Ce sont ceux que l'on veut détecter. Nous précisons donc qu'on désigne par `Deadlock`, l'existence d'un interblocage inattendu.

P7 permet de révéler de vrais interblocages dans le modèle, ou d'identifier des nœuds cruciaux dont le fonctionnement est requis pour permettre au système de continuer de jouer son rôle.

13.2 Syntaxe

La Figure 3.16 décrit les tâches de vérification indiquées dans la spécification.

Chaque tâche porte un nom qui suit le mot-clef `check`. Ensuite, on indique le nom de l'action, suivi de la condition sur la métrique, suivi des plages de variations des attributs.

Pour référencer les attributs dans une tâche de vérification, on affecte un paramètre à l'attribut dans la spécification du réseau de capteurs sans fil et on y fait référence dans la tâche de vérification. Ainsi, au lieu d'affecter des valeurs

aux attributs numériques (comme dans les listings précédents), on peut affecter des paramètres (définis au préalable) (voir méta-modèle en figure 3.3 et figure 3.15) que l'on veut faire varier dans les tâches de vérification.

Les plages de variation d'un attribut correspondent aux valeurs qu'il peut prendre dans les instances de la spécification.

```
Parameters {
  Energy INIT_ENERGY_HEART = 1000 ;
  Energy INIT_ENERGY_ACTIVITY1 =1000 ;
  Energy INIT_ENERGY_ACTIVITY2 = 1000 ;
  int ECG_SENS_PER =10;
  }
```

FIGURE 3.15 – La déclaration des paramètres de la spécification

Les attributs dont on n'a pas défini de plages de variations prennent les valeurs affectées lors de la déclaration des paramètres qui les référencent (voir figure 3.15) ou, à défaut, la valeur qui leur a été attribuée dans la spécification du réseau de capteurs sans fil.

Pour obtenir les instances de spécification, on affecte à chaque attribut, une valeur de sa plage de variation. Le nombre d'instances de spécification est donc égal à la taille du produit cartésien des plages de variation de tous les attributs numériques et de l'attribut environnement.

```
check plotWorstCaseLifetime {
        Plot WorstCaseLifetime
        with parameterVariations ={
   INIT_ENERGY_HEART = INIT_ENERGY_ACTIVITY1 = INIT_ENERGY_ACTIVITY2 = 10 .. 1000 Step 10
  }
} ForEnvScenarios BikeExercising, Resting

check verifyMessageDeliveryRatio {
        Verify (MessageDeliveryRatio(10) >= 0.5)
        with parameterVariations ={
        INIT_ENERGY_HEART = INIT_ENERGY_ACTIVITY1 = INIT_ENERGY_ACTIVITY2 = 250
   }
        ForEnvScenarios BikeExercising, Resting
}
```

FIGURE 3.16 – Les deux tâches de vérification du BAN

Dans le cas particulier où, les paramètres référençant deux attributs a_1 et a_2 sont liés par une égalité dans la spécification de la tâche de vérification, alors les attributs ont la même valeur pour chaque instance de la spécification.

Par exemple, la première tâche de vérification porte le nom de `plotWorstCaseLifetime` et consiste à tracer le graphique de la durée de vie dans le pire des cas du réseau pour les deux scénarios environnementaux `BikeExercising` et `Resting` en fonction de l'énergie initiale de toutes les instances de nœuds. Les égalités entre les paramètres `INIT_ENERGY_HEART`, `INIT_ENERGY_ACTIVITY1`, `INIT_ENERGY_ACTIVITY2` signifient que leur évolution est liée. La seconde tâche `verifyMessageDeliveryRatio` consiste à vérifier que le taux de messages bien

reçus par la station de base pour chaque 10 unités de temps (métrique relaxée) est supérieur ou égal à 0.5 pour deux scénarios environnementaux.

Pour désigner une métrique relaxée pour une durée égale à celle de l'exécution, on spécifie une durée Δ_t égale à $infinity$.

14 Implémentation

L'environnement de développement VeriSensor est constitué de :

- un éditeur
- un parseur
- un moteur de transformation de VeriSensor vers le langage formel des ITS.
- un module de traduction des résultats de la vérification.

Dans cette section, nous présentons succinctement l'implémentation de l'éditeur et du parseur. Le moteur de transformation et le module de traduction seront détaillés dans le chapitre 4 consacré à la transformation.

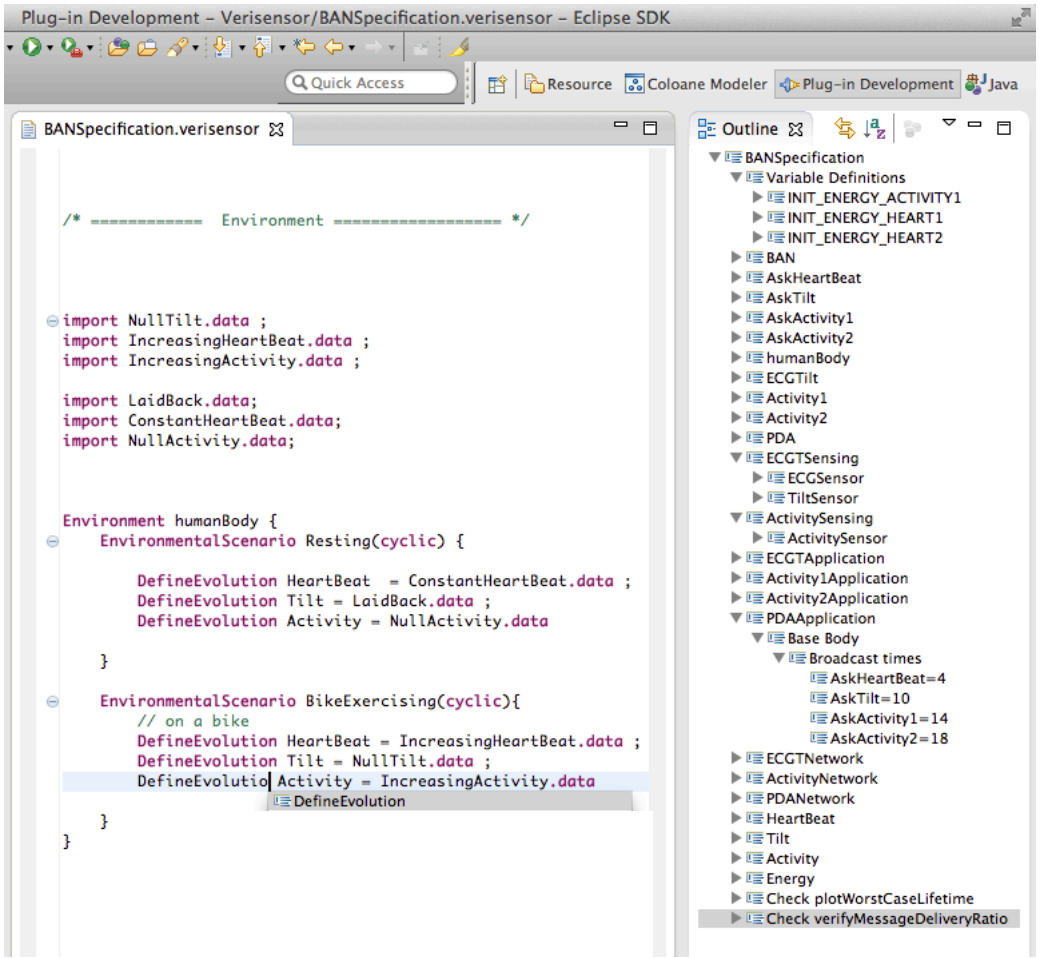


FIGURE 3.17 – Capture d'écran de l'environnement de développement VeriSensor

Le DSML VeriSensor a été implémenté à l'aide du *framework* Xtext [2].

La variante d'EBNF fournie par Xtext permet de décrire la syntaxe concrète d'un DSL ainsi que son mapping vers une représentation en mémoire du modèle sémantique. La grammaire produite précise comment une instance du métamodèle peut être dérivée d'une spécification VeriSensor.

A l'aide de Xtext, nous avons produit un *parseur*, un *méta-modèle Ecore* équivalent à la grammaire (voir le méta-modèle de haut niveau d'abstraction en section 4 et le méta-modèle complet en annexe), et un *environnement de développement* VeriSensor basé sur Eclipse [3] avec des fonctionnalités d'édition avancées.

Au total, le code Java du parseur et l'éditeur contient 83006 lignes (dont 55108 lignes pour l'interface graphique).

L'éditeur de VeriSensor supporte :

- la coloration syntaxique
- la complétion automatique de contenu
- la proposition de *quick fix* en cas d'erreurs
- une vue arborescente du modèle
- l'analyse statique
- la navigation dans le code source
- le *refactoring* après renommage (mise à jour des liens entre éléments de modèle après le renommage d'un élément de modèle)
- le *code folding* (cacher et afficher des sections du fichier édité ce qui permet de visualiser uniquement les sections pertinentes à un moment donné).
- le *hovering* (affichage d'informations sur un élément du modèle en passant la souris au dessus de lui).
- la sérialisation (stockage du modèle sous une représentation interne appropriée, en mémoire).
- le *scoping* (calcul des portées des éléments de modèle)
- le *linking* (les références entre éléments de modèle)

La figure 3.17 est une capture d'écran de l'environnement de développement VeriSensor. Au centre, la spécification du réseau avec coloration syntaxique et proposition de complétion de contenu pour le mot clef *DefineEvolution*. A droite, la vue arborescente (partiellement dépliée) pour le réseau biomédical en VeriSensor. C'est une représentation hiérarchique où un élément de modèle est à un niveau supérieur, s'il contient les autres éléments de modèle. Par exemple, le sous-modèle `PDAApplication` contient un corps de type `Base` spécifiant une liste d'instants de diffusion (*e.g.*, `AskHeartBeat` à l'instant 4).

15 Conclusion

Dans ce chapitre, nous avons présenté notre DSML VeriSensor pour la modélisation et la vérification de réseaux de capteurs sans fil.

Un modèle VeriSensor est composé de la spécification du réseau de capteurs sans fil et de son environnement ainsi que de la spécification des propriétés à vérifier sur le réseau.

2. Xtext est un framework pour le développement de DSLs de programmation et de modélisation (http ://www.eclipse.org/Xtext/)

3. Eclipse est un environnement de développement multi-langages basé sur un système extensible de plugin pour sa personnalisation (http ://www.eclipse.org/)

L'un des concepts VeriSensor les plus importants est celui de modèle de déploiement qui définit comment les nœuds sont répartis dans l'environnement. Pour cela, on y indique les instances de nœuds, leurs positions et les chemins de routage.

Le modèle de l'environnement quant à lui définit une liste de scénarios environnementaux possibles auxquels le réseau peut être soumis. Ces scénarios permettent de tester des propriétés qualitatives et quantitatives du modèle sur des instances de problème données.

Le modèle de requête définit les requêtes diffusées dans le réseau par la station de base. Elles encapsulent des tâches qui captent, traitent et envoient périodiquement un ensemble de grandeurs physiques. Il en existe deux types : conditionnelles (envoi déclenché par la satisfaction d'une condition) et non conditionnelles. Le modèle de classe de nœuds permet de factoriser les caractéristiques communes d'un ensemble de nœuds homogènes. Un mécanisme d'instanciation permet de définir les nœuds à partir d'une classe de nœuds. Le modèle de classe de nœuds décrit les quatre aspects d'un nœud évoqués dans le chapitre 2 : la capture, l'application, le réseau, l'énergie.

Finalement, pour spécifier les propriétés que l'on veut vérifier sur le modèle de réseau de capteurs, on définit un ensemble de tâches de vérification. Chaque tâche de vérification définit plusieurs instances du réseau, de son environnement et réalise une action (tracer un graphique ou verifier une propriété) sur les instances obtenues. Les métriques utilisées se basent sur les propriétés dégagées dans le chapitre 2.

Nous avons implémenté un environnement de développement pour VeriSensor contenant un éditeur avec des fonctionnalités avancées, un parseur, un moteur de transformation et un module de traduction. L'éditeur et le parseur produisent un modèle, fourni en entrée au moteur de transformation qui repose sur les règles définies dans le chapitre suivant.

CHAPITRE 4

VÉRIFICATION ET ANALYSE

1 Introduction

Dans ce chapitre, nous expliquons comment transformer les concepts VeriSensor en une spécification formelle et comment vérifier les propriétés dessus.

Dans la section 2, nous présentons et motivons les choix des langages dans lesquels la spécification formelle est exprimée : les *Instantiable Transition Systems* (ITS) [TMPHK09a] combinés avec les réseaux de Petri temporels.

En section 3, nous expliquons les étapes du processus de transformation d'une spécification VeriSensor. Il se base principalement sur 11 règles de transformation qui à partir d'éléments de la spécification VeriSensor construisent des éléments de la spécification formelle. Il existe deux types de règles : celles qui transforment un modèle ou sous-modèle VeriSensor en ITS, et celles qui permettent d'agencer ces ITS en un nœud, puis en un réseau de capteurs sans fil complet.

En section 4, nous montrons que la transformation conduit à une spécification formelle dont l'espace d'états peut exploser. Nous proposons des abstractions réduisant le risque d'explosion combinatoire tout en préservant la sémantique du modèle.

En section 5, nous expliquons comment se fait le calcul des métriques et la vérification des propriétés sur le modèle formel.

Finalement, en section 6, nous présentons les expérimentations que nous avons faites sur l'exemple fil rouge introduit dans le chapitre précédent.

2 Le modèle formel sous-jacent

Puisque les réseaux de capteurs sans fil sont des systèmes complexes et concurrents, nous avons besoin d'un outil supportant la concurrence, la notion de temps et capable d'atténuer l'explosion combinatoire. VeriSensor met l'accent sur deux aspects des réseaux de capteurs sans fil : les symétries (à l'aide de la notion de classe de nœuds) et la notion de localité dans les composants (grâce à l'architecture modulaire du langage).

Les ITS (*Instantiable Transition Systems*) supportent toutes ces caractéristiques. Initialement dédiés aux systèmes distribués [TMPHK09a][1], ils ont été enrichis pour supporter le temps [TMBK+11]. Les ITS sont supportés par un *model checker*

1. téléchargeable à l'adresse http ://move.lip6.fr/software/DDD/itstools.php

utilisant une librairie de diagrammes de décisions pour gérer la complexité des systèmes de grande taille.

Les ITS fournissent également un moyen de définir une spécification structurée et hiérarchique d'un système ainsi qu'un mécanisme d'instanciation des composants. Cela nous est utile sachant que VeriSensor est un langage hiérarchique. Les ITS ont déjà été utilisés pour analyser les diagrammes d'activité UML à travers une approche de transformation de modèles [TMH08] similaire à celle adoptée dans cette thèse. Les ITS ont démontré leur performance pour ce type de spécifications pendant le *model checking* contest @ Petri nets en 2011 [KLB+12a] et 2012 [KLB+12b].

Le comportement des composants est décrit en utilisant des réseaux de Petri temporels, un formalisme simple mais expressif, adapté à la modélisation des systèmes concurrents, des synchronisations de processus, des ressources partagées et des contraintes de temps [Mur89].

2.1 Instantiable Transition Systems

Les Instantiable Transition Systems, introduits par Thierry-Mieg et al. [TMPHK09b], permettent une modélisation hiérarchique et compositionnelle, grâce à la notion de type et d'instance et l'utilisation du pattern de conception composite au niveau comportemental. Un type a une interface définie comme un ensemble d'étiquettes sur les actions et une définition de son comportement interne . Comme les modèles orientés composants, un ITS *composite* est un type qui contient des *instances* de *types* ITS. Le mécanisme de composition est basé uniquement sur la synchronisation de transitions. Il est bien adapté aux algorithmes de vérification compositionnels.

2.1.1 Type et instance ITS

Nous introduisons la notion de type ITS.

Notations. On note $z.X, z.Y, \ldots$ l'élément X (resp.Y,...) d'un tuple $z = (X, Y, \ldots)$.

Définition 4.1. *[TMPHK09b] Un type ITS est la donnée de 4 éléments (S,A,Locals,Succ) avec*

- *S un ensemble d'états.*
- *A un ensemble de labels d'actions.*
- *$Locals : S \to 2^S$, une fonction qui à un état associe l'ensemble de ses « successeurs » immédiats par des transitions dites « privées ».*
- *$Succ : S \times A \to 2^S$, une fonction qui à un état et à un label d'action associe l'ensemble de « successeurs » de l'état par cette action.*

Une transition est dite *publique* lorsqu'elle est autorisée à être synchronisée avec des transitions d'autres types ITS. Une transition est dite *privée* dans le cas contraire.

Remarque 1. Un type ITS est élémentaire lorsqu'il est défini à partir d'un système à états finis (*e.g.*, réseau de Petri, LTS)

Un type ITS peut être instancié plusieurs fois. Une instance i est associée à un type ITS $type(i)$.

Accessibilité : Soit i une instance de type ITS et s,s' deux états de $type(i).S$. L'état s' est accessible à partir de s s'il existe des états $s_0,...,s_n \in type(i).S$ tels que $s' = s_n$ et $\forall j$ tel que $1 \leq j \leq n, s_j \in type(i).Locals(s_{j-1})$;

Les deux fonctions $Locals$ et $Succ$ sont utilisées pour des buts différents : $Locals$ représente les événements se produisant dans une instance de manière indépendante du reste du système. Par conséquent, elle retourne les états accessibles par l'occurrence d'événements locaux. La fonction $Succ$ produit les successeurs en synchronisant de manière explicite des actions via un label de l'alphabet.

Remarque 2. En pratique, cette définition est enrichie en spécifiant un état initial, et des prédicats basés sur les états, donnant un étiquetage à la Kripke et permettant le *model checking*.

La relation de transition du système peut être définie en termes de synchronisations de sous-systèmes utilisant $Succ$ et des comportements indépendants locaux. Par conséquent, un système complet est défini par une seule instance de type particulier dans un état initial spécifique : le système est auto-contenu et par conséquent l'accessibilité dépend de la définition de $Locals$ et non de $Succs$.

Exemple. La figure 4.1 contient un type ITS *Processus* basé sur un système à états transitions. Le terme *privé* désigne l'ensemble des transitions privées et *public* désigne l'ensemble des transitions publiques.

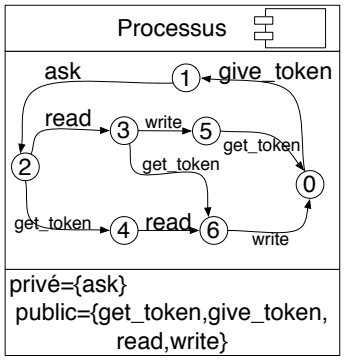


FIGURE 4.1 – Représentation graphique d'un type ITS *Processus* basé sur un LTS

2.1.2 Type ITS composite

Nous introduisons la notion de composite qui permet de construire un type ITS en « synchronisant » plusieurs instances de types ITS existants.

Notations. Soit un tuple $I = i_1,...,i_n$ d'instances ITS, $|I|$ est la taille n de I, S_I est l'ensemble : $type(i_1).S \times ... \times type(i_n).S$. Pour $s \in S_I$ et i une instance, on note $s[i]$ le composant de s qui correspond à i.

Soit un tuple I d'instances ITS et un ensemble Lab de labels, nous définissons inductivement l'ensemble $Stat_C$ d'actions par :

- $\langle call(i,\lambda)\rangle$ un appel de label λ de $type(i)$, qui invoque une transition de i étiquetée par λ,

- $\langle call_{self}(\lambda)\rangle$ un appel à un label λ de Lab, qui invoque une transition du type composite courant, étiquetée par λ. Cela permet de structurer la relation de transition et d'enchaîner les comportements. Les cycles de tels appels sont interdits.
- $\langle\sigma_0;...;\sigma_k\rangle$ une séquence d'actions dans $Stat_C$ (syntaxiquement les actions d'une séquence sont séparées par des point-virgules).

Définition 4.2. *[TMPHK09b] Un composite défini sur l'alphabet Lab est un tuple $C = \langle I, Sync\rangle$ tel que :*

- *I un ensemble fini d'instances ITS. Le type de chaque instance ITS doit exister lors de la définition de I, dans le but d'éviter les définitions de type récursives ou circulaires.*
- *$Sync \subset Lab \times Stat_C$ est un ensemble fini de synchronisations, tel que $t = \langle\lambda,\sigma\rangle \in Sync$, λ est le label de t et σ est son corps.*

Après avoir défini la syntaxe, nous définissons la sémantique du composite (« ssi » est l'abréviation de « si et seulement si »).

Fonction *Next* qui produit l'état successeur par une action : La fonction $Next_I : S_I \times Stat_C \to 2^{S_I}$, utilisée dans la définition 4.3 ci-dessous est définie pour s, $s' \in S_I$ et $\sigma \in Stat_C$ par :

$s' \in Next_I(s,\sigma)$ ssi

$$\begin{cases} \exists s_0...s_{k+1}, s_0 = s, s_{k+1} = s', \forall i \in [1..k], s_{i+1} \in Next_I(s_i,\sigma_i) & \text{si } \sigma = \langle\sigma_0;...;\sigma_k\rangle \\ s'[i] \in type(i).Succ(s[i],\lambda) \wedge \forall j \in I, j \neq i, s'[j] = s[j] & \text{si } \sigma = \langle call(i,\lambda)\rangle \\ \exists t = \langle\lambda,\sigma'\rangle \in Sync, \text{ tel que } s' \in Next_I(s,\sigma') & \text{si } \sigma = \langle call_{self}(\lambda)\rangle \end{cases}$$

Définition 4.3. *(type ITS correspondant à un composite).[TMPHK09b] Soit $C = \langle I, Sync\rangle$ un composite défini sur un alphabet Lab. Le type ITS $\tau_C = \langle S, A, Locals, Succ\rangle$ correspondant à C, est défini par :*

- *$S = S_I$; $A = Lab \setminus \top$;*
- *$Locals : S \to 2^S$ est défini pour $s, s' \in S$ par $s' \in Locals(s)$ ssi*

$$\begin{cases} \exists i \in I, s'[i] \in type(i).Locals(s[i]) \wedge \forall j \in I, j \neq i, s'[j] = s[j] \\ ou \\ \exists t = \langle\top,\sigma\rangle \in Sync, s' \in Next_I(s,\sigma) \end{cases}$$

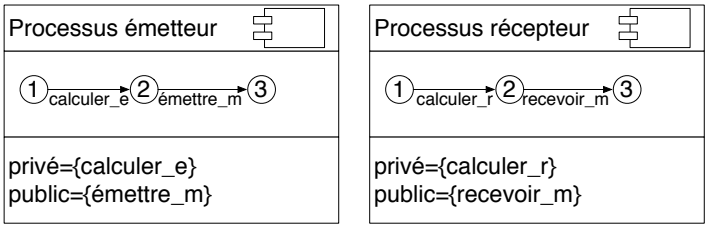
- *$Succ : S \times A \to 2^S$ est défini pour $s, s' \in S, \lambda \in A$ par :*
  $s' \in Succ(s,\lambda)$ ssi il existe $t = \langle\lambda,\sigma\rangle \in Sync, s' \in Next_i(s,\sigma)$.

La définition 4.3 décrit une implémentation du contrat pour le type ITS générique. Il contient soit des instances élémentaires (comme les LTS ou les réseaux de Petri), ou inductivement d'autres instances composites.

Dans cette définition, $Locals(s)$ est défini comme l'ensemble des états résultant de l'action de $Locals$ sur n'importe quelle instance imbriquée, ou sur les états accessibles à partir de s par l'occurrence d'une synchronisation associée au label local $\top$.

L'ensemble des successeurs $Succ(s,\lambda)$ est obtenu en appliquant le label λ, ce qui peut déclencher une séquence d'appels successifs. Cette séquence est franchie atomiquement.

Exemple. La figure 4.2a contient le type ITS *Processus émetteur* qui réalise des calculs (transition *calculer_e*) et émet un message (transition *émettre m*).

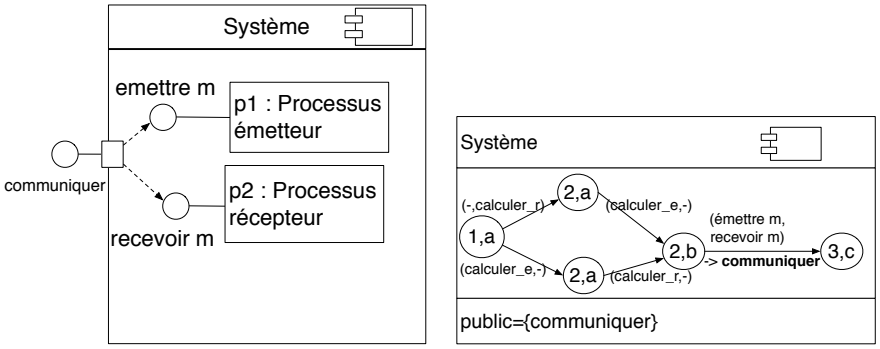


(a) Représentation graphique du type ITS *Processus emetteur*

(b) Représentation graphique du type ITS *Processus récepteur*

FIGURE 4.2 – Les type ITS *Processus Emetteur* et *Processus Recepteur*

La figure 4.2b contient le type ITS *Processus récepteur* qui réalise des calculs (transition *calculer_r*) et reçoit le message (transition *recevoir m*).



(a) Représentation graphique du type ITS composite *Systeme*

(b) Représentation de la forme produit du type ITS composite *Systeme*

FIGURE 4.3 – Le type ITS composite *Systeme*

La figure 4.3 contient deux représentations du type ITS composite *Systeme* : une représentation graphique (figure 4.3a) et une représentation de la forme produit (*i.e.*, le produit synchronisé des processus, voir figure 4.3b). *Systeme* est constitué d'une instance de *Processus émetteur* et d'une instance de *Processus récepteur*. Les transitions *émettre m* et *recevoir m* sont synchronisées. Cette synchronisation est une transition publique du type composite étiquetée par *communiquer*.

Les transitions publiques *émettre m* de l'instance de *Processus émetteur* et *recevoir m* de l'instance de *Processus récepteur* sont synchronisées (voir flèche en pointillés). Cela signifie qu'une instance attend que l'autre soit prête pour commencer la communication (émission et réception). Par ailleurs, cette synchronisation est considérée par le type composite, comme publique et étiquetée par *communiquer*.

La forme produit a la même sémantique que la représentation graphique du composite (voir figure 4.3a).

En pratique, nous utilisons les réseaux de Petri temporels pour définir les types ITS élémentaires.

2.2 Réseaux de Petri

Les réseaux de Petri sont bien adaptés à la description de systèmes concurrents, asynchrones, distribués, parallèles, non déterministes et/ou stochastiques [Mur89]. Ils conviennent donc à la modélisation de réseaux de capteurs sans fil.

Un réseau de Petri est un langage mathématique pour la modélisation de systèmes distribués [Mur89]. Il est représenté par un graphe particulier avec un état initial appelé *marquage initial* M_0. Le graphe est dirigé, valué , biparti, avec deux types de nœuds, appelés places et transitions. Les arcs sont soit orientés d'une place à une transition ou d'une transition à une place. Une place est représentée par un cercle, une transition par un rectangle. Les arcs sont étiquetés par des poids (entiers positifs). Un marquage (état) consiste à affecter à chaque place un entier positif. Graphiquement, un marquage de k dans une place est représenté par k points noirs (jetons) dans la place.

Les places représentent des conditions (ou des états), et les transitions représentent des événements. Une transition possède un nombre de places en entrée et en sortie qui modélisent les pre-conditions et post conditions d'un événement. Un jeton dans une place signifie la validité de la condition associée à la place. k jetons dans une place signifient un état où k ressources sont disponibles.

La définition formelle d'un réseau de Petri est :

Définition 4.4. *[Sil93] Un réseau de Petri est un 4-tuple, PN=*$\langle P, T, Pre, Post\rangle$ *tel que :*

- P *et* T *sont respectivement des ensembles de places et de transitions disjoints, finis et non vides.*
- $Pre : P \times T \longrightarrow \mathbb{N}$ *est le mapping d'incidence arrière ou fonction de pré-incidence.*
- $Post : P \times T \longrightarrow \mathbb{N}$ *est le mapping d'incidence avant ou fonction de post-incidence.*

Un marquage est une fonction $m : P \mapsto \mathbb{N}$ et le réseau est doté d'un marquage initial m_0. La sémantique (appelée aussi dynamique d'exécution) d'un réseau de Petri est donnée sous la forme d'un système de transitions $M_N = (\mathbb{N}^P, m_0, \rightarrow)$ sur T où les configurations sont des marquages et la relation de transition $t \subset \mathbb{N}^P \times T \times \mathbb{N}^P$ est définie par : $m \xrightarrow{t} m'$ si pour tout $p \in P$, $m(p) \geq Pre(p,t)$ et $m'(p) = m(p) - Pre(p,t) + Post(p,t)$.

Un arc est un élément de $P \times T \cup T \times P$. En plus de ces arcs, il est possible d'ajouter à un réseau de Petri deux types d'arcs appelés *reset* et *inhibiteur*, dont la sémantique est différente. Un arc de *reset* vide le contenu de la place dès que la transition est franchie. Un arc inhibiteur empêche le franchissement de la transition si la place n'est pas vide.

2.3 Réseaux de Petri temporels

Plusieurs extensions des réseaux de Petri incluant le temps ont été proposées :

- les réseaux de Petri temporels définissent un intervalle de temps associé à chaque transition (indiquant l'écoulement du temps possible avant le franchissement) pour modéliser les systèmes qui peuvent évoluer de manière non déterministe.

- les réseaux de Petri temporisés définissent une durée associée à chaque transition. Dès que la transition est franchissable, les jetons sont consommés, jusqu'à l'écoulement de la durée, après lequel les jetons sont mis dans les places en sortie.
- les réseaux de Petri stochastiques spécifient une distribution de probabilité associée à chaque transition pour évaluer le comportement du système en régime permanent ou transitoire, où la durée de chaque action est gouvernée par des observations statistiques.

Les réseaux de Petri temporels ont été présentés dans [Mer74] et ajoutent des contraintes temporelles au franchissement des transitions. Un intervalle de temps est associé à chaque transition. Une horloge est associée à chaque transition sensibilisée (*i.e.*, dont chaque place en entrée possède un marquage suffisant. Le terme « validée » est aussi utilisé) et indique le temps écoulé depuis la sensibilisation de la transition. Une transition sensibilisée peut être franchie si la valeur de l'horloge appartient à l'intervalle de la transition. Par ailleurs, le temps ne peut pas progresser si l'écoulement du temps fait que l'on quitte l'intervalle associé à la transition.

Nous nous intéressons en particulier aux *réseaux de Petri temporels discrets*.

2.3.1 Syntaxe

La définition suivante présente la syntaxe d'un réseau de Petri temporel. Elle est tirée de [TMBK+11].

Définition 4.5. *Un réseau de Petri temporel étiqueté N est un tuple $(Pl, Tr, A, enabled, fire, \Lambda, m_0, \alpha, \beta)$ tel que :*

- *Pl un ensemble fini de places ; Tr est un ensemble fini de transitions (avec $Pl \cap Tr = \emptyset$),*
- *A est un ensemble fini (alphabet) d'étiquettes d'actions qui contient une étiquette locale $\top$,*
- *$enabled : \mathbb{N}^{Pl} \times Tr \mapsto \{true, false\}$ est un prédicat de sensibilisation.*
- *$fire : \mathbb{N}^{Pl} \times Tr \mapsto 2^{N^{Pl}}$ est une fonction de franchissement de transitions,*
- *$\Lambda : Tr \mapsto A$ est une fonction d'étiquetage.*
- *$m_0 \in \mathbb{N}^{Pl}$ est le marquage initial du réseau,*
- *$\alpha : Tr \mapsto \mathbb{N}$ et $\beta : Tr \mapsto \mathbb{N} \cup \{\infty\}$ sont les fonctions satisfaisant $\forall t \in Tr$, $\alpha(t) \leq \beta(t)$ appelées respectivement, date de tir statique au plus tôt ($\alpha(t)$) et au plus tard ($\beta(t)$).*

Par exemple, les réseaux de Petri standards sont normalement définis en utilisant les fonctions pre (noté **Pre**) et post (noté **Post**) : $Pl \times Tr \mapsto \mathbb{N}$. Alors, pour un marquage $m \in \mathbb{N}^{Pl}$ et une transition $t \in Tr$, la fonction $enabled()$ est définie par $enabled(m,t)$ si et seulement si $\forall p \in Pl,\ m(p) \geq \mathbf{Pre}(p,t)$ et le franchissement d'une transition par $fire(m,t) = \{m'\}$ avec $\forall p \in Pl,\ m'(p) = m(p) - \mathbf{Pre}(p,t) + \mathbf{Post}(p,t)$. Les arcs inhibiteurs **Inh** et les arcs test **Test** sont définis de manière similaire et ajoutent des conditions de sensibilisation à la transition : $enabled(m,t)$ si et seulement si $\forall p \in Pl, m(p) < \mathbf{Inh}(p,t) \wedge m(p) \geq \mathbf{Test}(p,t)$. Notons que le prédicat de sensibilisation considère uniquement les marquages alors les conditions de timing sont définies séparément. Dans la définition 4.5, les transitions sont étiquetées pour permettre la composition de réseaux de Petri.

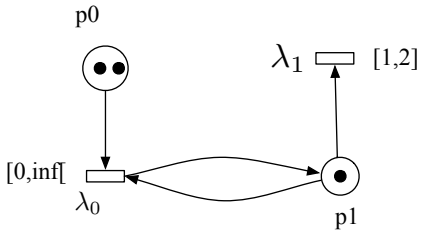


FIGURE 4.4 – Exemple de réseau de pétri temporel

Dans les règles de transformation, pour plus de simplicité et de concision, nous utiliserons la notation Pre et $Post$, et nous associerons à chaque transition t, un intervalle de franchissement $I(t)$ dont la borne inférieure est $\alpha(t)$ et la borne supérieure est $\beta(t)$.

Figure 4.4 représente un exemple de réseau de Petri temporel avec deux transitions étiquetées λ_0 et λ_1.

2.3.2 Sémantique

Nous adoptons une sémantique du réseau de Petri temporel à temps discret pour plusieurs raisons :

- Les techniques sous-jacentes au model-checker ITS-tools [TMBK$^+$11] utilisé y sont adaptées.
- Nous avons analysé notre modèle avec les model-checkers Tina [BRV04] et Roméo [GLM$^+$05] qui adoptent une sémantique à temps dense, mais ils sont plus lents et ne passent pas à l'échelle (au delà de 100 unités d'énergie), par rapport à ITS-tools [TMBK$^+$11]. De manière générale, l'approche en temps discret fournit des performances souvent meilleures que l'approche en temps dense [TMBK$^+$11].

Pour définir la sémantique (appelée aussi dynamique d'exécution) d'un réseau de Petri temporel à temps discret, on a besoin de la notion de système de transitions temporisés à temps discret.

Nous rappelons leur définition, où toutes les actions (appartenant à un ensemble A) sont considérées instantanées et les transitions de délai sont ajoutées, dans un domaine temporel restreint à l'ensemble $\mathbb{N}$ des entiers naturels. Nous considérons uniquement les pas d'une unité de temps (action spéciale $\mathbb{1}$ ci dessous). Avoir une seule opération pour gérer le délai est plus efficace que d'avoir à chercher à chaque pas le délai entier maximal cohérent avec la synchronisation de plusieurs composants dans un ensemble d'états.

Définition 4.6. *[TMBK$^+$11] Soit A un ensemble d'étiquettes d'actions et $\mathbb{1} \notin A$ une action spéciale représentant un délai d'une unité de temps. Un système de transitions temporisé discret sur A est un tuple $\mathcal{T} = \langle S, s_0, A, \rightarrow \rangle$ avec S un ensemble d'états, $s_0 \in S$ est l'état initial, et $\rightarrow \subset S \times (A \uplus \{\mathbb{1}\}) \times S$ est la relation de transitions ($\uplus$ représente l'union disjointe).*

La sémantique d'un réseau de Petri temporel à temps discret est décrite par un Système de transitions temporisés à temps discret. Une valuation v est un élément

de $\mathbb{N}^T$. Par conséquent, pour une transition $t \in T$, $v(t)$ représente la valeur dans $\mathbb{N}$ d'une horloge implicite associée à t.

Définition 4.7. *[TMBK+11] Pour un réseau de Petri temporel* $N = (Pl, Tr, A, enabled, fire, \Lambda, m_0, \alpha, \beta)$ *la sémantique est un système de transitions* $S_N = \langle S, s_0, A \uplus \{\mathbb{1}\}, \rightarrow \rangle$ *où :*

- $S = N^{Pl} \times N^{Tr}$ *est l'ensemble des états. Un élément s de S est une paire* $s = \langle m, v \rangle$*, où m est un marquage de place et v est la valuation de l'horloge de la transition.*
- $s_0 = \langle m_0, \bar{0} \rangle$ *où le tuple* $\bar{0}$ *correspond à la valeur* 0 *pour toutes les horloges de transitions.*
- $\rightarrow \subset S \times (A \uplus \{\mathbb{1}\}) \times S$ *est la relation de la transition définie pour les états* $\langle m, v \rangle$, $\langle m', v' \rangle$ *par :*

  La relation de transition ***discrète*** *:* $\langle m, v \rangle \xrightarrow{a} \langle m', v' \rangle$ *si et seulement s'il y a une transition* $t \in Tr$ *telle que :*

  - $\Lambda(t) = a \wedge enabled(m, t) \wedge v(t) \geq \alpha(t)$
  - $m' \in fire(m, t)$
  - $\forall t' \in Tr, v'(t') = \begin{cases} v(t'), & si\ enabled(m', t') \wedge t \neq t' \\ 0, & sinon \end{cases}$

  La relation de transition ***délai*** *:*

  $\langle m, v \rangle \xrightarrow{\mathbb{1}} \langle m', v' \rangle$ *si et seulement si* $m' = m$ *et pour tout* $t \in Tr$*,*

  - $enabled(m, t) \implies v(t) < \beta(t)$ *(les horloges urgentes empêchent l'écoulement)*
  - $v'(t) = \begin{cases} v(t) + 1 & si\ enabled(m', t) \wedge \beta(t) \neq \infty\ \textit{(écoulement normal)} \\ v(t) + 1 & si\ enabled(m', t) \wedge \beta(t) = \infty \wedge v(t) < \alpha(t)\ \textit{(progresser jusqu'à } \alpha\textit{)} \\ v(t) & sinon \end{cases}$

La sémantique est atomique : concernant les nouvelles transitions sensibilisées, une transition t est exécutée en un seul pas. De plus, un critère (critère de transitions nouvellement sensibilisées) est utilisé pour remettre à 0 les horloges de transitions désensibilisées. Cela garantit qu'une transition désensibilisée a une valeur d'horloge égale à 0.

La règle (progresser jusqu'à α) permet de gérer le problème des date de tir au plus tard (*e.g.*, pour les transitions avec des intervalles de franchissement de $[\alpha(t), \infty[$. Avec cette stratégie, nous ne laissons pas les horloges des transitions correspondantes progresser au delà de leur plus petite valeur significative $\alpha(t)$. Cela permet de représenter correctement le comportement sur un support fini.

2.4 Réseaux de Petri colorés

Les réseaux de Petri colorés [Jen91] ajoutent aux réseaux de Petri, des primitives pour la définition de types de données et la manipulation de valeurs de données. Nous ne les utilisons pas dans la représentation interne du modèle formel mais *uniquement pour représenter* certains des réseaux de Petri génériques que nous produisons par transformation, de manière plus compacte.

Dans un réseau de Petri coloré, un jeton a une valeur appartenant à un type.

- Un type de donnée est associé à une place indiquant qu'elle ne peut contenir que des jetons appartenant à ce type. Un type peut être le produit cartésien de

types existants. Une place peut contenir un multiset de jetons (*i.e.* contenir des jetons de même valeur).
- Un arc entrant d'une transition peut avoir des gardes sur la sensibilisation de la transition. Une transition n'est franchissable que lorsqu'un sous ensemble des jetons dans les places sources satisfait les gardes sur les arcs entrants de la transition.
- Les arcs sortants peuvent contenir des expressions arithmétiques spécifiant comment calculer les valeurs produites par la transition.
- les gardes et les expressions sont exprimées à l'aide de paramètres (ou variables) et sont évaluées en associant la valeur d'un jeton à un paramètre (*binding*).

Avant de définir la syntaxe des réseaux de Petri colorés, nous rappelons qu'un multiensemble m sur un domaine D est une fonction $m : D \rightarrow \mathbb{N}$ dénotant le nombre d'occurrences de $d \in D$ dans le multiensemble m. Les multi ensembles sont dotés des opérations d'addition $+$, de différence $-$ et de comparaison $\leq$ (voir [Jen91] pour plus de détails). Les multiensembles sont notés comme des ensembles avec répétition, et $\emptyset = \{\}$ est le multiensemble vide (fonction constante 0).

2.4.1 Syntaxe

Nous reprenons la définition de Jensen et al. [Jen91, CP92].

Pour donner une définition abstraite des réseaux de Petri colorés, il n'est pas nécessaire de fixer la syntaxe concrète, dans laquelle celui qui modélise écrit les expressions, et nous supposons qu'une telle syntaxe existe (ainsi qu'une sémantique bien définie). Cela permet de parler de manière non ambigüe des notions suivantes :
- Les éléments d'un type T. L'ensemble de tous les éléments dans T est dénoté par le nom de type T. L'ensemble des multiensembles sur T est dénoté par T_{MS}.
- Le type d'une variable v dénoté par $Type(v)$.
- Le type d'un expression $expr$ dénoté par $Var(expr)$. L'ensemble des variables inclus uniquement les variables libres (*i.e.*, celles qui ne sont pas connectés de manière interne à une expression).
- une valuation *(binding)* d'un ensemble de variables $V = \{v_1, ..., v_n\}$ dénoté par $\langle v_1 = c_1, v_2 = c_2, ..., v_n = c_n \rangle$. On exige que chaque $c_i \in Type(v_i)$ pour chaque variable $v_i \in V$.
- la valeur peut être obtenue en évaluant une expression $expr$ dans une valuation b, ce qui est noté $expr\langle b \rangle$. On exige que $Var(expr)$ soit un sous ensemble des variables de b, et que l'évaluation soit réalisée en substituant chaque variable $v_i \in Var(expr)$ avec la valeur $c_i \in Type(v_i)$ déterminée par la valuation.

Pour définir un réseau de Petri coloré, nous définissons $Bool$ comme le type booléen (contenant les éléments $\{false, true\}$ et muni des opérations logiques standard).

Définition 4.8. *[Jen91] Un réseau de Petri coloré est un tuple* $CPN = (\Sigma, P, T, A, N, C, G, E, Init)$ *qui satisfait les conditions suivantes :*
- Σ *est un ensemble fini de types non vides, appelés ensembles de couleurs*
- P *un ensemble fini de places ;*
- T *un ensemble fini de transitions*
- A *est un ensemble fini d'arcs tels que :*
  - $P \cap T = P \cap A = T \cap A = \emptyset$.

- *N est une fonction noeud. Elle est définie de A dans $P \times T \cup T \times P$.*
- *C est une fonction couleur définie de P vers Σ.*
- *G est une fonction garde définie de T vers les expressions telles que :* $\forall t \in T : [Type(G(t)) = Bool \text{ et } Type(Var(G(t)) \subset \Sigma]$
- *E est une fonction étiquetant un arc par une expression, définie de A vers les expressions telle que :* $\forall a \in (P \times T \cup T \times P) : [Type(E(a)) = C(p(a))_{MS} \wedge Type(Var(E(a))) \subset \Sigma]$ *avec $p(a)$ la place de $N(a)$.*
- *$Init()$ est une fonction initialisation définie de P vers les expressions fermées telles que :* $\forall p \in P : [Type(Init(p)) = C(p)_{MS}]$

Nous pouvons interpréter les éléments de la définition de la façon suivante :

- L'ensemble des ensemble de couleurs détermine les types, opérations et fonctions, qui peuvent être utilisées dans le réseau (*i.e.*, expressions d'arcs, gardes, expressions d'initialisation, ensembles de couleurs, etc.).
- Les places, transitions et arcs sont décrites par trois ensembles P, T et A qui doivent être finis et disjoints deux à deux.
- La fonction N associe à chaque arc une paire où le premier élément est le noeud source et le second est le noeud destination. Les noeuds doivent être de types différents. Contrairement aux réseaux de Petri classiques, un réseau de Petri coloré peut avoir plusieurs arcs entre la même paire ordonnée de noeuds.
- La fonction de couleur C associe à chaque place un ensemble de couleurs possibles de jetons $C(p)$. Chaque jeton de p doit avoir une couleur qui appartient au type $C(p)$.
- la fonction G associe à chaque transition t une expression de type booléen (*i.e.*, un prédicat). Par ailleurs, toutes les variables dans $G(t)$ doivent avoir des types appartenant à Σ.
- La fonction E associe à chaque arc une expression qui doit être de type $C(p(a))_{MS}$. Cela signifie que chaque évaluation d'une expression d'arc doit produire un multiensemble sur l'ensemble des couleurs associé à la place correspondante. Une expression peut être de type $C(p(a))$. Dans ce cas, l'expression est évaluée à une couleur dans $C(p(a))$ que l'on peut considérer comme un multiensemble à un élément.
- La fonction d'initialisation $Init$ associe à chaque place p une expression qui doit être de type $C(p)_{MS}$. L'expression ne doit contenir aucune variable. Une expression initiale peut être de type $C(p)$.

Au lieu de définir l'ensemble des arcs de manière explicite A, nous manipulons dans nos règles de transformation uniquement des couples (place,transition) ou (transition, place). Cela ne nous permet pas de définir plusieurs arcs entre une même paire ordonnée de noeuds. Nous ne rencontrons pas cette situation dans les règles de transformation.

2.4.2 Sémantique

Avant de définir la sémantique des réseaux de Petri colorés, nous introduisons les notations et la notion de valuation *(binding)*.

Notations : $Var(t)$ est appelé ensemble des variables de t et $E(x_1, x_2)$ est appelé expression de (x_1, x_2) :

- $\forall t \in T\ Var(t) = \{v \mid v \in Var(G(t)) \vee \exists a \in A(t) :\ v \in Var(E(a))\}$
- $\forall (x_1, x_2) \in (P \times T \cup T \times P) : E(x_1, x_2) = \sum_{a \in A(x_1, x_2)} E(a)$

Nous définissons ce qu'est une valuation. Intuitivement, une valuation, pour une transition t est une substitution qui remplace chaque variable de t avec une couleur. On exige que chaque couleur soit du bon type et que la garde soit évaluée à $true$.

Définition 4.9. *[Jen91] Pour une transition $t \in T$ avec les variables $Var(t) = \{v_1, v_2, ..., v_n\}$ nous définissons le type de valuation* (binding type) *$BT(t)$ comme suit :*

- $BT(t) = Type(v_1) \times Type(v_2) \times ... \times Type(v_n)$

Par ailleurs, nous définissons l'ensemble de toutes les valuations $B(t)$ comme suit :

- $B(t) = \{(c_1, c_2, ..., c_n) \in BT(t) \mid G(t)\langle v_1 = c_1, v_2 = c_2, ..., v_n = c_n\rangle\}$

On dénote les valuations de deux différentes manières : soit sous la forme $\langle v_1 = c_1, v_2 = c_2, ..., v_n = c_n\rangle$, soit sous la forme $(c_1, c_2, ..., c_n)$. Dans les deux cas, cela dénote un élément de $BT(t)$.

A partir des notations précédentes et de la notion de valuation, nous définissons la sémantique d'un réseau de Petri coloré.

Définition 4.10. *[Jen91] Un* step *est un* multiensemble *de valuations* (bindings) *entre une variable et une valeur.*

Un step Y est franchissable à partir d'un marquage M si et seulement si la propriété suivante est satisfaite : $\forall p \in P : \sum_{(t,b) \in Y} E(p,t)\langle b\rangle \leq M(p)$.

Lorsqu'un step Y est franchissable depuis un marquage donné M_1, il peut être franchi, modifiant le marquage M_1 en un marquage M_2, défini par :
$\forall p \in P : M_2(p) = (M_1(p) - \sum_{(t,b) \in Y} E(p,t)\langle b\rangle) + \sum_{(t,b) \in Y} E(t,p)\langle b\rangle$.

On dit que M_2 est directement accessible à partir de M_1 par l'occurrence du step Y. On note cela : $M_1[Y_1 \succ M_2$

Une séquence d'occurrences est une séquence de marquages et de steps *:*

$M_1[Y_1 \succ M_2[Y_2 \succ M_3 ... M_n[Y_n \succ M_{n+1}$

telle que M_{i+1} est directement accessible à partir de M_i pour tout $i \in 1..n$. On dit alors que M_n est accessible à partir de M_1.

La figure 4.5 contient un exemple simple de réseau de Petri coloré. Deux types sont définis : INT (un entier) et BOOL (un booléen). Les valeurs des jetons dans une place sont indiquées dans un encadré adjacent à la place. Les trois variables sont i,v,w. Les types associés aux places sont écrits en italique (*e.g.*, INT). Les variables sur les arcs sont entre parenthèses. Le réseau de Petri fait le produit du numéro i par w.

Remarque : Un réseau de Petri temporel ou coloré peut être considéré comme un ITS=$(S, A, Locals, Succ)$ à condition que :

- S corresponde à l'ensemble des marquages du réseau de Petri
- A soit égal l'ensemble des étiquettes du réseau de Petri
- $Succ()$ associe à un marquage M et à une étiquette e, l'ensemble des marquages successeurs de M par toute transition du réseau de Petri étiquettée par e.
- $Locals()$ associe à un marquage M l'ensemble des marquages successeurs de M par toutes les transitions non étiquetées.

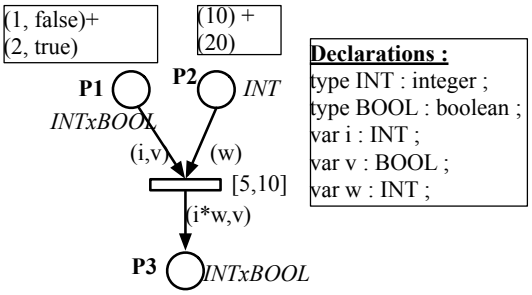


FIGURE 4.5 – Exemple de réseau de Petri temporel coloré

Après avoir présenté les langages dans lesquels nous exprimons le modèle formel, nous présentons la transformation d'une spécification VeriSensor vers ce modèle formel (section 3) , les abstractions que l'on applique au modèle formel pour réduire le risque d'explosion combinatoire (section 4), et finalement comment nous calculons les métriques ainsi que les propriété sur ce modèle formel (section 5)

3 Le Mapping de la spécification VeriSensor vers le modèle formel

Dans cette section, nous expliquons d'abord les étapes du processus de transformation d'une spécification complète VeriSensor, la vérification des propriétés sur le modèle formel et le retour des résultats à l'utilisateur.

Ensuite, nous présentons le processus de transformation de chaque instance de la spécification VeriSensor en un ITS. Nous donnons une vision globale de l'architecture du modèle formel résultant de ce processus ainsi que de la structure d'un noeud dans ce modèle formel. Finalement, nous présentons les règles de transformations qui permettent de produire le modèle formel d'une instance de spécification VeriSensor.

3.1 Processus de transformation d'une spécification complète VeriSensor

La figure 4.6 modélise le processus de transformation d'une spécification complète VeriSensor, la vérification des propriétés sur le modèle formel et le retour des résultats à l'utilisateur, avec un haut niveau d'abstraction.

Ce processus suit 6 étapes :

1. Le choix de la tâche de vérification à exécuter par l'utilisateur (voir figure 4.6)
2. La création des instances de la spécification en fonction des plages de variation des paramètres et de l'environnement.
3. Le mapping de chaque instance de spécification en un modèle formel MF et l'application d'abstractions réduisant la taille du modèle et de l'espace d'états à générer.
4. L'ajout éventuel d'éléments au modèle formel (*e.g.*, compteurs du nombre de messages émis) nécessaire à la vérification de la propriété ou au calcul de la métrique de la tâche de vérification.

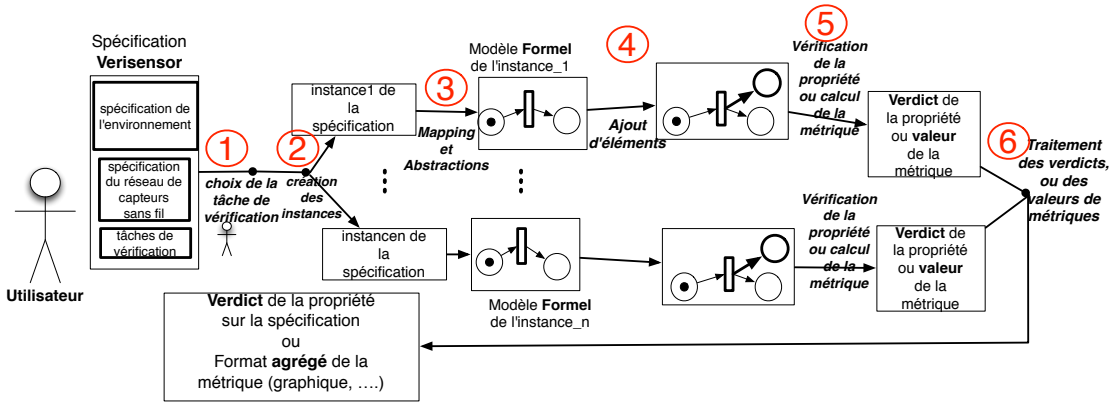


FIGURE 4.6 – Etapes de la transformation et de la vérification de VeriSensor

5. la vérification d'une propriété ou le calcul d'une métrique sur chaque instance de spécification.
6. le traitement de l'ensemble des verdicts, ou des valeurs de métriques
   - Si la tâche consiste à vérifier une propriété, alors le verdict est positif dans le cas où la propriété est vérifiée sur chaque modèle formel MF d'une instance de spécification.
   - Si la tâche consiste à tracer l'évolution d'une métrique, alors sa valeur est calculée pour chaque modèle MF d'une instance de spécification. Le graphique trace l'évolution de la valeur en fonction des paramètres, contraints par leurs plages de variation.
   - Si la tâche consiste à optimiser (maximiser ou minimiser) une métrique, sa valeur est calculée pour chaque modèle MF d'une instance de spécification. Ensuite, la valeur optimale de la métrique (minimale ou maximale) et les valeurs de paramètres pour lesquels elle est atteinte, sont retournés à l'utilisateur.

3.2 Le processus de transformation d'une instance de spécification VeriSensor

Le mapping d'une instance de spécification VeriSensor vers un modèle formel repose sur un ensemble de règles de transformation des éléments de la spécification VeriSensor vers les ITS. Nous ne faisons dans cette section aucune abstraction sur le modèle formel. Les abstractions seront introduites en section 4.

Le modèle formel est structuré en *dimensions* décrivant les aspects orthogonaux du comportement du réseau de capteurs sans fil extrait de la spécification VeriSensor. Le modèle de chaque nœud est composé des dimensions suivantes : réseau, application, capteurs, énergie et environnement local (une projection de l'environnement global de VeriSensor sur la position du nœud). Après, les nœuds sont connectés à travers la table de routage construite à partir du modèle de déploiement.

Chaque dimension a ses propres règles de transformation définies hiérarchiquement, par conséquent bénéficiant des mécanismes des ITS. Le modèle final est obtenu en assemblant et en appliquant ces règles.

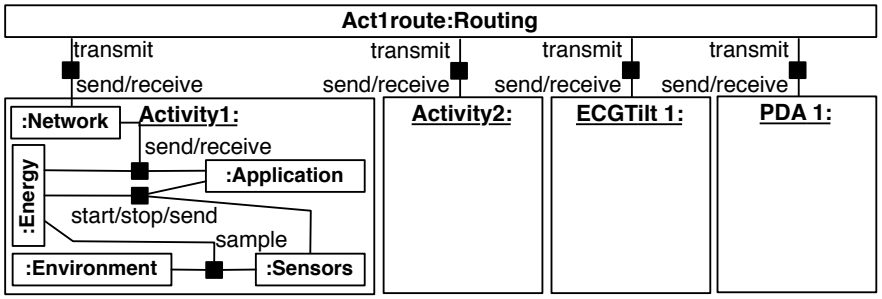


FIGURE 4.7 – Vue informelle de la structure de la spécification du BAN

Chaque dimension respecte une interface fixée qui ne dépend pas du comportement interne. Ce modèle de composition permet de modifier facilement le comportement d'une dimension. Cela est particulièrement utile lors de l'étude de plusieurs scénarios environnementaux ou de plusieurs modèles de consommation de l'énergie.

3.3 Architecture globale du modèle formel

La Figure 4.7 illustre de manière informelle l'architecture globale de la spécification formelle finale pour l'étude de cas du réseau biomédical (voir sous-section 3, chapitre 3), basé sur l'architecture de VeriSensor. Dans ce diagramme, nous distinguons les noms d'instance (suivis par un « : ») des noms de types d'ITS (précédés d'un « : »). Nous ne détaillons que le nœud Activity1 et nous ne montrons que le routage pour l'instance de Activity1.

L'assemblage d'une classe de nœuds est illustré pour Activity1. Nous montrons comment les interfaces des dimensions sont synchronisées entre elles. Par exemple, l'étiquette start représente une synchronisation entre les dimensions application, énergie et capture. De manière similaire, sample synchronise les dimensions capture, énergie et environnement.

La transmission des données à travers le réseau est gérée par un composant ITS dédié (Routing). Il exporte une étiquette par tuple $\langle source, destination, message \rangle$. Son rôle est de transmettre le message à l'instance destinatrice. Chaque nœud a sa propre instance routage ; son ITS va être détaillé plus tard dans la section.

Etant donné que chaque classe de nœud est instanciée suivant le déploiement du modèle, la figure 4.7 montre deux nœuds `Activity`, un nœud `ECGTilt` et un nœud `PDA`.

3.4 Structure d'un nœud dans la spécification formelle

Un nœud contient cinq dimensions :

- la dimension environnements locaux : elle représente la projection de l'évolution des grandeurs physiques sur la position du nœud. Elle est déduite du modèle de l'environnement de VeriSensor ou correspond à un environnement « libre » (voir sous-section 13.1) pour l'analyse dans le pire des cas.
- la dimension capteurs : elle décrit le comportement des capteurs embarqués dans le nœud et incorpore les paramètres (durée de capture, *startup time*, etc.)

fournis par le modèle de capture. Les capteurs font l'acquisition de valeurs de l'environnement et les transmettent à l'application.
- la dimension application : elle décrit le comportement du nœud en réponse aux requêtes et est déduite du modèle de requêtes et de la description de l'application d'un nœud. Elle contrôle les capteurs (`start/stop`) et échange des messages avec le réseau.
- la dimension réseau : elle modélise la couche réseau du nœud. Elle transmet les requêtes et les résultats. Dans les scénarios *multi-hop*, elle gère la retransmission de messages. Elle modélise également les pertes de message du nœud.
- la dimension énergie : elle décrit séparément le modèle de consommation d'énergie du nœud. Elle est construite à partir des informations énergétiques réparties dans la spécification d'une classe de nœuds VeriSensor et est synchronisée avec chaque opération consommatrice d'énergie. Elle est déduite des coûts énergétiques déclarés dans la spécification en entrée.

Dans ce qui suit, nous expliquons les principales règles de transformation permettant de transformer une instance de spécification VeriSensor en réseaux de Petri temporels ou réseaux de Petri temporels colorés.

Remarques :
- Dans tous les réseaux de Petri produits de nos règles de transformation, toutes les places connectées à des arcs inhibiteurs ou reset sont 1-bornées. Par conséquent, les réseaux obtenus ne sont pas Turing complets, et les propriétés d'accessibilité sont décidables. Cela nous permet de réaliser de la vérification formelle poussée de nos modèles.
- La sémantique des réseaux de Petri temporels colorés est obtenue par dépliage de chaque transition t avec un intervalle de franchissement I en plusieurs transitions possédant chacune son horloge et le même intervalle de franchissement I.
- Par précaution, aucune transition synchronisée n'est temporisée, pour éviter les interblocages temporels. Un interblocage temporel se produit lorsque l'horloge associée à une transition t a atteint la date de tir au plus tard, et que la transition t_{sync} avec laquelle elle est synchronisée n'est pas franchissable. En associant à chaque transition synchronisée un intervalle de franchissement de $[0, +\infty[$, nous évitons ce problème.

Pour chaque règle de transformation, on indique son nom, son entrée, sa sortie et éventuellement une représentation graphique du réseau de Petri produit (les transitions publiques ont un contour gras). L'entrée d'une règle de transformation est constituée d'éléments de la spécification VeriSensor (*e.g*, un noeud n ou une classe cl). La sortie est un réseau de Petri (qui correspond à un type ITS, voir la remarque en sous-section 2.4.2).

Hypothèses :
- si $Pre(p,t)$ ou $Post(p,t)$ n'est pas précisé pour une place p, alors sa valeur est nulle.
- si la couleur d'une place n'est pas spécifiée alors elle est destinée à contenir des jetons sans couleurs *(black tokens)*.

- si la valeur de $Init()$ n'est pas précisée pour une place, alors cette place ne contient aucun jeton.
- si un intervalle de franchissement de transition n'est pas spécifié, il est égal à $[0, +\infty[$
- Dans la représentation graphique des réseaux de Petri (temporels ou temporels colorés), on attribue un nom aux transitions non étiquetées (*i.e.* privées) pour faciliter la lecture.
- Soit une transition t d'un réseau temporel coloré. Soient $(v_i)_{0 \leq i \leq n}$ les variables intervenant dans les expressions étiquetant les arcs connectés à une transition t, l'étiquette de t ($\Lambda(t)$) est paramétrée par ces variables (*e.g.*, sendValueToEnv(v)).

Notations :
- Si $Pre(p,t) = 1'(.)$ ou $Post(p,t) = 1'(.)$, on note simplement $Pre(p,t) = 1$ ou $Post(p,t,) = 1$.
- Les commentaires commencent par le signe « // ».
- L'extrémité d'un arc inhibiteur (côté transition) est un cercle. L'extrémité d'un arc reset (côté transition) est un losange noir.
- Si $z = (X,Y,...)$ est un tuple, on note $z.X$, $z.Y$, ... l'élément X ($resp.Y$,...) du tuple. Par exemple, pour tout noeud VeriSensor n, $n.id$ désigne l'identifiant du noeud (voir la définition 3.1, page 61), et $n.cl.c$ désigne la dimension capture de la classe du noeud.
- On indique en gras et entre chevrons (⟨**exemple**⟩) tout nom (ou partie de nom) des éléments de l'ITS en sortie (*e.g.*, étiquette de transition, place) qui dépendent d'éléments de la spécification VeriSensor donnée en entrée. Par exemple, étant donné un noeud n tel que $n.cl = TemperatureClass$, le nom ⟨**n.cl**⟩$Capteur$ donne, après substitution, $TemperatureClassCapteur$. Dans les figures représentant les ITS, nous utilisons une notation en gras, sans les chevrons.

3.5 Architecture de connexion d'un noeud dans la spécification formelle

Nous présentons dans cette sous-section une vue *globale* de l'architecture de connexion entre les ITS composant un noeud. Cette architecture est produite par l'application des règles de transformation pour un noeud. La figure 4.8 présente de manière *informelle* la composition d'ITS modélisant le nœud `ecgtilt1` du réseau biomédical (voir exemple fil rouge, p. 59, chapitre 3).

Chaque boîte blanche représente une instance d'un réseau de Petri construite suivant les règles présentées précédemment. Le modèle est hiérarchique. L'application (fond gris) doit être décomposée à son tour en un dispatcher qui filtre les requêtes et une liste de requêtes. Le réseau est décomposé en un ITS gérant les communications et un ITS loser qui modélise la perte de messages. Comme dans la figure 4.7, les synchronisations locales sont représentées par des carrés noirs, alors que les synchronisations permettant l'interaction avec d'autres nœuds du réseau de capteurs sans fil sont représentées par des carrés blancs. Par souci de lisibilité, nous utilisons une seule synchronisation pour représenter un ensemble de synchronisations similaires. Un nom de synchronisation est connectée à une synchronisation par une flèche en

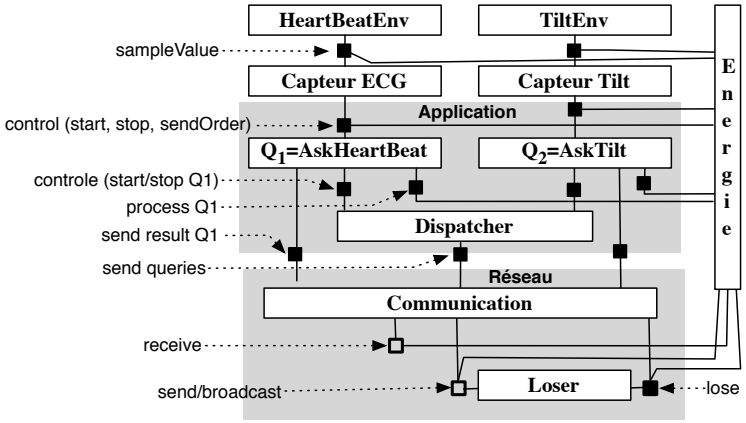


FIGURE 4.8 – Vue informelle et globale de l'ITS du nœud *ecgtilt1*

pointillés. Ces noms sont uniquement utilisés pour expliquer de manière informelle la signification de la synchronisation.

Dans la figure 4.8, `Capteur ECG` par exemple, peut capter une valeur parmi celles que fournit l'environnement. La synchronisation nommée `sampleValue` représente cela.

Q_1 et Q_2 encapsulent les opérations réalisées par les requêtes. Le dispatcher reçoit un identifiant de requête de la part du réseau et s'il correspond à une requête supportée par le nœud déclenche son traitement. Les deux requêtes traitées par ce nœud sont synchronisées avec le dispatcher, les capteurs concernés et le réseau.

Les transitions d'émission de message (send, broadcast) sont synchronisées avec l'ITS `loser` qui en « perd » (synchronisation `lose`) une proportion au plus égale à τ. Le coût énergétique dû aux pertes est comptabilisé grâce à la connexion de la synchronisation *lose* à la dimension énergie.

Toutes les opérations énergivores (*i.e.*, capture, traitement, émission vers l'extérieur et réception) sont synchronisées avec la dimension énergie pour être comptabilisées.

L'interface résultante de cet ITS composite est constituée des services de communication (`send/broadcast` et `receive`). Tous les nœuds ont la même interface.

3.6 Transformation de l'environnement et des capteurs

Dans cette sous-section, nous présentons comment nous transformons les éléments de la spécification VeriSensor en ITS représentant les environnements locaux et les capteurs du noeud.

3.6.1 Transformation de l'environnement en environnement local

Dans cette transformation, nous traduisons l'évolution, en fonction du temps et du nœud, d'une grandeur g, décrite dans un scénario environnemental S, en un type ITS modélisant l'évolution locale de g à la position d'un nœud n_0.

La première étape (sous-section 3.6.1) consiste à projeter l'évolution de g sur la position du nœud n. On obtient un chronogramme sous la forme d'une fonction en escalier représentant l'évolution de la grandeur dans le temps à la position de n_0.

La deuxième étape (sous-section 3.6.1) consiste à construire le réseau de Petri temporel à partir de ce chronogramme.

Projection. Soit un scénario environnemental donné, S et un déploiement, $Depl$.

La projection de l'évolution d'une grandeur g au niveau de la position d'un nœud $n_0 \in Depl$ est $evolution(n_0.id, g)$ (voir définition 3.3, page 63)

Concrètement, la projection d'une grandeur g sur la position d'un nœud n_0 consiste à sélectionner dans le fichier .data (voir sous-section 6.2, chapitre 3), décrivant l'évolution de g, uniquement les couples (d_k, V_k) associés au nœud n_0. On obtient une fonction en escalier représentant l'évolution de la grandeur dans le temps à la position de n_0.

Règle de transformation pour l'environnement. Cette règle décrit la transformation de l'évolution d'une grandeur g projetée sur la position d'un nœud n_0 (*i.e.*, $evolution(id, g) = (d_k, V_k)_{1 \leq k \leq n}$ avec $n \in \mathbb{N}$ et $\sum_{1 \leq k \leq n} d_k = d_{totale}$), vers l'environnement local associé au capteur de la grandeur g appartenant au nœud n_0.

Règle de transformation 1. ***environnement local***

Entrée *: $n_0.id$ (l'identifiant du nœud), g (la grandeur), $(d_k, V_k)_{0 \leq k \leq n}$ (l'évolution de g), c (le booléen indiquant le caractère cyclique du scénario environnemental)*
Sortie *: le Réseau de Petri temporel étiqueté $\langle \mathbf{n_0.id} \rangle \langle \mathbf{g} \rangle Env$ tel que :*
// *******definition de l ensemble des etiquettes*******

- $\forall k$ *tel que,* $0 \leq k \leq n$, $sendToSens \langle \mathbf{V_k} \rangle \in A$

// *******places S_k *******

- $\forall k$, *tel que,* $0 \leq k \leq n$ $\exists S_k \in P$
- $M_0(S_0) = 1$

// *******transitions********

- $\forall k$ *tel que* $,0 \leq k \leq n$, $\exists t \in T$, *tel que* $\Lambda(t) = sendToSens \langle \mathbf{V_k} \rangle$ *et* $Pre(S_k, t) = 1$ *et* $Post(S_k, t) = 1$
- $\forall k$, *tel que,* $1 \leq k \leq n-1$, $\exists t \in T$, *tel que* $Pre(S_k, t) = 1$ *et* $Post(S_{k+1}, t) = 1$ *et* $I(t) = [\langle \mathbf{d_k} \rangle, \langle \mathbf{d_k} \rangle]$

//prise en compte du caractère cyclique

- *si* $c = 1$ *alors* $\exists t \in T$, *tel que* $Pre(S_n, t) = 1$ *et* $I(t) = [\langle \mathbf{d_n} \rangle, \langle \mathbf{d_n} \rangle]$ $Post(S_1, t) = 1$

La figure 4.9 représente l'ITS environnement vu par le capteur d'un nœud. Les deux arcs en pointillés et la transition ϵ_n sont ajoutés si le scénario environnemental est cyclique.

Dans le cas acyclique, sachant que le chronogramme d'évolution est prolongé dans le temps par la dernière valeur V_n (voir chapitre 3, section 6.2), la transition $sendToSens \langle \mathbf{V_n} \rangle$ reste toujours franchissable.

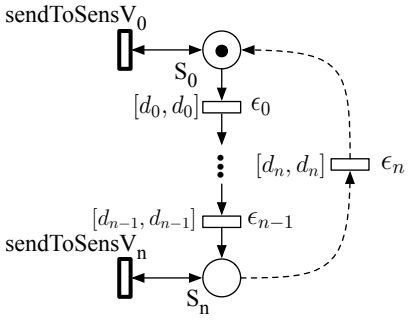
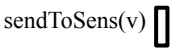


FIGURE 4.9 – Environnement local

Chaque transition publique est étiquetée par une valeur possible de la grandeur physique. La borne temporelle sur les transitions locales (notées ϵ_i) représente l'évolution de $q(t)$ en fonction du temps. L'interface de cet ITS est l'ensemble des valeurs possibles $\mathsf{sendToSensV}_k$ que peut prendre la grandeur g à la position du nœud n.

Concernant la modélisation de l'environnement « libre » (voir sous-section 13.1, chapitre 3) associé à une grandeur g, à la position d'un nœud n, il contient autant de transitions publiques $\mathrm{sendToSensV}_k$ que de valeurs possibles de la grandeur g.

La figure 4.10 modélise l'environnement libre pour une grandeur possédant n valeurs possibles. A un instant donné, n'importe quelle transition est franchissable. Il fournit dont n'importe quelle valeur de grandeur, à tout instant. Il permet de construire un sur-ensemble de tous les comportements possibles, à partir duquel nous obtenons le scénario dans le pire des cas.



Declarations :
type VAL : V0..Vn ;
var v : VAL ;

FIGURE 4.10 – Environnement libre

3.6.2 Règle de transformation pour le capteur

La règle de transformation suivante présente comment, à partir des éléments de la spécification VeriSensor, on construit un type ITS représentant un capteur.

Règle de transformation 2. ***capteur***

Entrée *: c (un capteur VeriSensor),* $(V_k)_{0\leq k\leq m}$ *(les valeurs possibles de la grandeur captée par c), n (le nœud n auquel appartient le capteur)*
Sortie *: le réseau de Petri* ⟨**n**.**cl**⟩ ⟨**c**.**g**⟩*Capteur :*
// etiquettes

- $start \in A$
- $stop \in A$
- $endWait \in A$
- $recvOrderFromApp \in A$
- $recvFromEnv(v) \in A$
- $sendToApp(v) \in A$

// places

- $\{(V_k)_{0 \leq k \leq m}\} \in \Sigma$
- $(started, idle, rdy, waited) \in P^4$
- $(waitedSense, received) \in P^2$
- $off \in P$ *et* $Init(off) = 1$
- $C(waitedSense) = \{(V_k)_{0 \leq k \leq m}\}$
- $C(received) = \{(V_k)_{0 \leq k \leq m}\}$

// transitions de contrôle du capteur

- $\exists t \in T$, *tel que,* $\Lambda(t) = start$ *et* $Pre(off, t) = 1$ *et* $Post(started, t) = 1$
- $\exists t \in T$, *tel que,* $\Lambda(t) = stop$ *et* $Pre(idle, t) = 1$ *et* $Post(off, t) = 1$
- $\exists t \in T$, *tel que,* $Pre(started, t) = 1$ *et* $Post(waited, t) = 1$ *et* $I(t) = [\langle \mathbf{c.d_d} \rangle, \langle \mathbf{c.d_d} \rangle]$.
- $\exists t \in T$, *tel que,* $\Lambda(t) = endWait$ *et* $Pre(waited, t) = 1$ *et* $Post(idle, t) = 1$.
- $\exists t \in T$, *tel que,* $\Lambda(t) = recvOrderFromApp$ *et* $Pre(idle, t) = 1$ *et* $Post(rdy, t) = 1$

// transitions de capture

- $\exists t \in T$, *tel que,* $Pre(rdy, t) = 1$ *et* $Post(waitedSense, t) = 1$ *et* $I(t) = [\langle \mathbf{c.d_c} \rangle, \langle \mathbf{c.d_c} \rangle]$.
- $\exists t \in T$, *tel que,* $\Lambda(t) = recvFromEnv(v)$ *et* $Pre(waitedSense, t) = 1$ *et* $Post(t, received) = 1$ *et* $E(waitedSense, t) = v$ *et* $E(t, received) = v$
- $\exists t \in T$, *tel que,* $\Lambda(t) = sendToApp(v)$ *et* $Pre(received, t) = 1$ *et* $E(received, t) = v$

L'ITS produit de la transformation est représenté en figure 4.11.

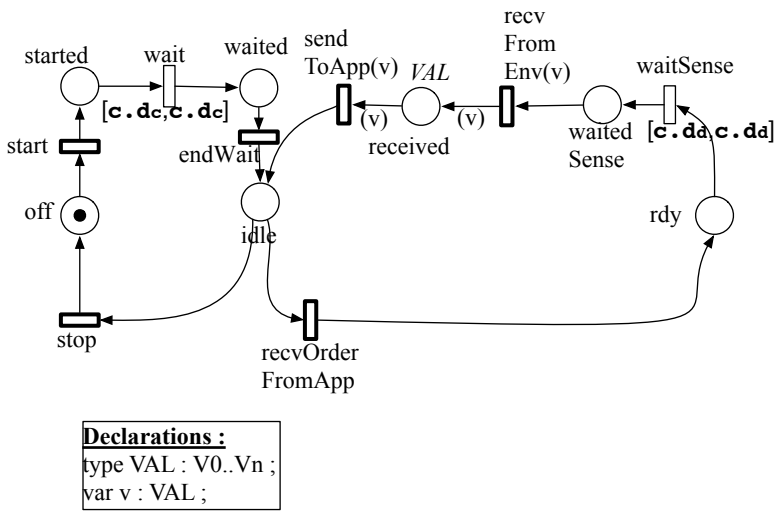


FIGURE 4.11 – réseau de Petri temporel pour le capteur

Le comportement du capteur est paramétré par :

- le nombre de valeurs V_k de la grandeur captée.
- $c.d_d$ le *startup time* du capteur.
- $c.d_c$ la durée d'une capture.

L'interface de cet ITS est composée de commandes de contrôle (start, stop) connectées à l'application et de la transition recvFromEnv(v), connectée à l'environnement local, ainsi que de sendToApp(v) connectée à la dimension application.

La transition *start* représente la réception de l'ordre de démarrage, et $endWait$ notifie l'application comme quoi le capteur est prêt à fonctionner. La transmission d'une valeur V_k de l'environnement au capteur est représentée par une synchronisation entre sendToSensV_k (ITS environnement) et recvFromEnvV_k (résultat du dépliage de recvFromEnv(v) de l'ITS capteur). La transition sendToApp(v) transmet les valeurs captées à la dimension application. Ces définitions du capteur et de l'environnement sont clairement séparées de telle sorte qu'il est possible d'associer la spécification à un environnement arbitraire au lieu d'un scénario fixé. Une telle association est définie dans les tâches de vérification.

3.6.3 Assemblage des ITS capteurs et environnement

La figure 4.12 présente la composition d'ITS entre les environnements locaux du noeud *ecgtilt1* et ses capteurs.

Nous adoptons les notations graphiques suivantes. Chaque boîte blanche représente une instance d'un réseau de Petri, temporel ou coloré, construite suivant les règles présentées précédemment. Le nom de l'instance est indiqué avant « : » et son type est indiqué après. Les synchronisations locales sont représentées par des carrés noirs, alors que les synchronisations publiques (*i.e.*, permettant l'interaction avec d'autres nœuds du réseau de capteurs sans fil) sont représentées par des carrés blancs. Le nom d'une transition publique est indiqué sur le segment connectant l'instance à la synchronisation.

La transmission d'une valeur V_k d'un environnement local, au capteur associé est représentée par une synchronisation entre sendToSensV_k (ITS environnement) et recvFromEnvV_k (résultat du dépliage de recvFromEnv(v) de l'ITS capteur).

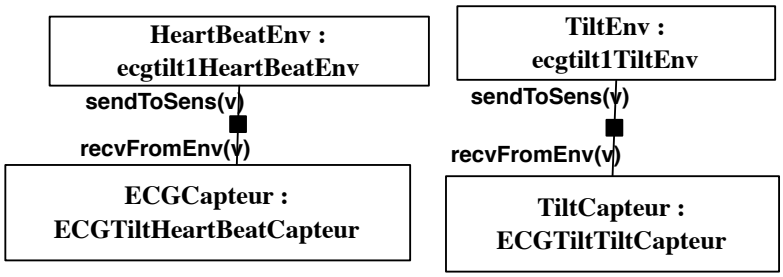


FIGURE 4.12 – Assemblage des ITS capteurs et environnement

3.7 Règles de transformation pour l'application

La transformation du sous-modèle application VeriSensor en une dimension application, consiste en :

- la construction de réseaux de Petri temporels colorés modélisant le traitement réalisé par les requêtes supportées par la classe de nœud
- la construction d'un réseau de Petri temporel coloré appelé dispatcher qui reçoit un identifiant de requête de la part du réseau et s'il correspond à une requête supportée par la classe de nœud déclenche son traitement. Ce réseau de Petri temporel est donc synchronisé avec les réseaux de Petri temporels représentant les traitements des requêtes (voir la construction de l'ITS nœud, dans la règle 10).

Nous modélisons les traitements effectués par les requêtes dans les nœuds destinataires de ces requêtes. Seuls les identifiants de requêtes sont diffusés dans le réseau et une fois reçus déclenchent les traitements associés. Cette abstraction réduit la quantité de données communiquées dans le réseau, et donc le nombre de synchronisations possibles entre émetteurs et récepteurs.

3.7.1 Règle de transformation pour le dispatcher

La règle de transformation suivante présente comment à partir de l'ensemble des requêtes du réseau de capteurs sans fil et du sous-modèle application d'une classe de nœud VeriSensor on obtient le réseau de Petri temporel coloré dispatcher de cette classe. Le dispatcher filtre les requêtes provenant du réseau. Si la requête reçue est supportée par l'application, il la déclenche.

On note $a.id$ l'ensemble des identifiants des requêtes de l'application a.

Règle de transformation 3. ***dispatcher***

Entrée *: REQ (l'ensemble des requêtes dans le réseau de capteurs), a (le sous-modèle application de la classe de nœuds), cl (la classe de nœuds)*
Sortie *: le réseau de Petri temporel coloré étiqueté* $\langle \mathbf{cl} \rangle Dispatcher$ *:*
// etiquettes
- $A = \{recvFromNet(q), start(q)\}$

// places
- $\{req.id : req \in REQ\} \in \Sigma$
- $received \in P$ *et* $C(received) = \{req.id : req \in REQ\}$

// transitions de réception des identifiants de requêtes
- $\exists t \in T$*, tel que,*
  - $\Lambda(t) = recvFromNet(q)$
  - $Post(Received, t) = 1$
  - $G(t) = q \notin \langle \mathbf{a.id} \rangle$
  - $E(t, Received) = q$
- $\exists t \in T$*, tel que,* $\Lambda(t) = recvFromNet(q)$ *et* $Post(Received, t) = 1$ *et* $G(t) = q \in \langle \mathbf{a.id} \rangle$ *et* $E(t, Received) = q$.

// transitions de démarrage des traitements des requêtes
- $\exists t \in T$*, tel que,* $\Lambda(t) = start(q)$*, et* $Pre(Received, t) = 1$ *et* $E(Received, t) = q$

Le type ITS, produit de la transformation, est représenté en figure 4.13.

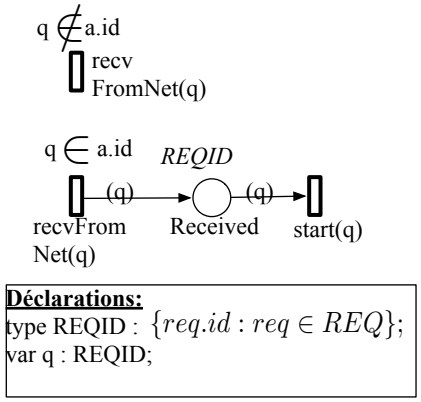


FIGURE 4.13 – Le dispatcher

Le dispatcher reçoit les identifiants de requêtes de la part du réseau et démarre le traitement d'une requête ($start(q)$) si elle est supportée par l'application du nœud (garde $q \in \langle \mathbf{a.id} \rangle$).

3.7.2 Règle de transformation pour une requête non conditionnelle

La règle suivante présente comment construire à partir d'une requête non conditionnelle VeriSensor et des domaines de ses grandeurs, le réseau de Petri temporel coloré qui la modélise.

Nous définissons $R.(g_i)_{0 \leq i \leq p}$ le tuple obtenu à partir de l'ensemble $R.G$ avec R une requête. En effet, nous avons besoin d'ordonner les grandeurs pour modéliser la suite des ordres de démarrage et d'arrêt des capteurs.

Règle de transformation 4. ***requête non conditionnelle***

Entrée *: $nonCondReq$ (la requête), $Dom()$ (la fonction qui fournit les valeurs possibles d'une grandeur)*
Sortie *: le réseau de Petri temporel coloré étiqueté* $\langle$**nonCondReq.id**$\rangle$ *satisfaisant :*
//*****Etiquettes*****

- $start \in A$, $process \in A$
- $sendToNetRes \in A$
- $\forall g \in \langle$**nonCondReq.G**$\rangle$,
  - $sendOrderTo\,\langle$**g**$\rangle \in A$
  - $recv(v\,\langle$**g**$\rangle) \in A$
  - $endWait\,\langle$**nonCondReq.g**$\rangle \in A$
  - $stop\,\langle$**g**$\rangle \in A$
  - $start\,\langle$**g**$\rangle \in A$

//*****places*****

- $\forall g \in \langle$**nonCondReq.G**$\rangle$, $Dom(g) \in \Sigma$
- $initial \in P$ *et* $Init(initial) = 1$.
- $(started, endWaitedStartups, waitingProcDur) \in P^3$
- $(finishedProcdur, waitingSendRes, waitingEndReq) \in P^3$.
- $(waitingProc, waitingSend, waitingSense) \in P^3$.
- $\forall g \in \langle$**nonCondReq.G**$\rangle$,

- $(waitingStart\langle \mathbf{g} \rangle, Started\langle \mathbf{g} \rangle, waitingStop\langle \mathbf{g} \rangle) \in P^3$
- $(waitingSendOrderTo\langle \mathbf{g} \rangle, orderSent\langle \mathbf{g} \rangle) \in P^2$

//****transitions**********

- $\exists t \in T$, *tel que* $\Lambda(t) = start$, *et* $Pre(initial, t) = 1$ *et* $Post(waitingStart\langle \mathbf{g_0} \rangle, t) = 1$
- $\forall g \in \langle \mathbf{nonCondReq.G} \rangle$,
  - // ***démarrage des capteurs
  - $\exists t \in T$, *tel que,* $\Lambda(t) = start\langle \mathbf{g} \rangle$ *et* $Pre(waitingStart\langle \mathbf{g} \rangle, t) = 1$ *et* $Post(Started\langle \mathbf{g} \rangle, t) = 1$
  - // ***capture des échantillons
  - $\exists t \in T$, *tel que,* $\Lambda(t) = sendOrderTo\langle \mathbf{g} \rangle$ *et* $Post(orderSent\langle \mathbf{g} \rangle, t) = 1$ *et* $Pre(waitingSendOrderTo\langle \mathbf{g} \rangle, t)$
  - $\exists t \in T$, *tel que,* $\Lambda(t) = recv(v\langle \mathbf{g} \rangle)$ *et* $Pre(orderSent\langle \mathbf{g} \rangle, t) = 1$
- $\exists t \in T$ *tel que* $\Lambda(t) = waitSense$ *et* $I(t) = [\langle \mathbf{nonCondReq.T_c} \rangle, \langle \mathbf{nonCondReq.T_c} \rangle]$ *et* $Pre(waitingSense, t) = 1$ *et* $Post(waitingSense, t) = 1$ *et* $\forall g \in \langle \mathbf{nonCondReq.G} \rangle Post(waitingSendOrderTo\langle \mathbf{g} \rangle, t)$

// ***transitions construites à partir du tuple $(g_i)_{0 \leq i \leq p}$

- $\forall i \in \mathbb{N}$, *tel que* $0 \leq i \leq (p-1)$,
  - $\exists t \in T$ *tel que* $\Lambda(t) = endWait\langle \mathbf{nonCondReq.g_i} \rangle$ *et* $Pre(started\langle \mathbf{nonCondReq.g_i} \rangle, t) = 1$ *et* $Post(waitingStart\langle \mathbf{g_{i+1}} \rangle, t) = 1$
  - // ***arrêt des capteurs
  - $\exists t \in T$ *tel que* $\Lambda(t) = stop\langle \mathbf{g_i} \rangle$ *et* $Pre(waitingStop\langle \mathbf{g_i} \rangle, t)$ *et* $Post(waitingStop\langle \mathbf{g_{i+1}} \rangle, t)$
- $\exists t \in T$ *tel que* $\Lambda(t) = stop\langle \mathbf{g_p} \rangle$ *et* $Pre(waitingStop\langle \mathbf{g_p} \rangle, t) = 1$
- $\exists t \in T$, *tel que* $\Lambda(t) = endWait\langle \mathbf{g_p} \rangle$ *et* $Post(waitingProc, t) = 1$ *et* $Post(waitingSend, t)$ *et* $Post(waitingEndReq, t)$, *et* $Post(waitingSense, t) = 1$
- $\exists t \in T$ *tel que :*
  - $\Lambda(t) = waitEndReq$ *et* $Pre(waitingEndReq, t) = 1$ *et* $Post(waitingStop\langle \mathbf{g_0} \rangle, t) = 1$ *et* $I(t) = [\langle \mathbf{nonCondReq.d_{totale}} \rangle, \langle \mathbf{nonCondReq.d_{totale}} \rangle]$
  - *il existe un arc de reset entre t et waitingSend, waitingProc, waitingSense.*

  //***traitement
  - $\exists t \in T$, *tel que,* $\Lambda(t) = waitProc$ *et* $Pre(waitingProc, t) = 1$ *et* $Post(waitingProc, t) = 1$ *et* $Post(waitingSendRes, t) = 1$ *et* $I(t) = [\langle \mathbf{nonCondReq.T_t} \rangle, \langle \mathbf{nonCondReq.T_t} \rangle]$.
  - $\exists t \in T$, *tel que,* $\Lambda(t) = waitProcDur$ *et* $Pre(waitingProcDur, t) = 1$ *et* $Post(finishedProcDur, t) = 1$ *et* $I(t) = [\langle \mathbf{nonCondReq.d_t} \rangle, \langle \mathbf{nonCondReq.d_t} \rangle]$.
- $\exists t \in T$, *tel que,* $\Lambda(t) = process$ *et* $Pre(finishedProcDur, t) = 1$

  //***envoi des résultats
  - $\exists t \in T$, *tel que,* $\Lambda(t) = waitSend$ *et* $Pre(waitingSend, t) = 1$ *et* $Post(waitingSend, t) = 1$ *et* $Post(waitingSendRes, t) = 1$ *et* $I(t) = [\langle \mathbf{nonCondReq.T_e} \rangle, \langle \mathbf{nonCondReq.T_e} \rangle]$

– $\exists t \in T$, *tel que,* $\Lambda(t) = sendToNetRes$ *et* $Pre(waitingSendRes, t) = 1$

La figure 4.14 représente le résultat de la transformation. Les noms des places ont été omis par manque d'espace. Les noms des paramètres de la requête sont également abrégés (*e.g.* d_{totale} au lieu du nom qualifié $nonCondReq.d_{totale}$).

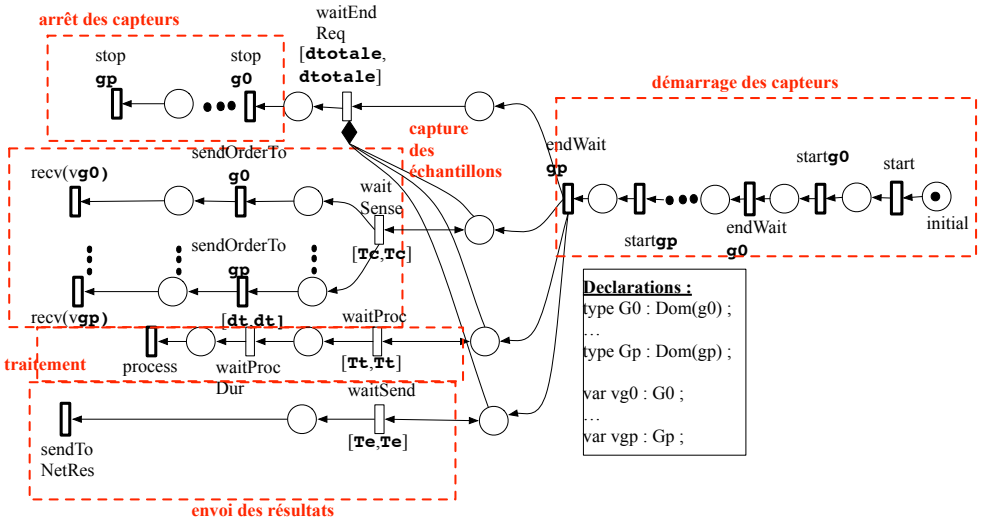


FIGURE 4.14 – ITS pour une requête non conditionnelle

La requête commence par démarrer les capteurs (chaque $start\langle \mathbf{g_i} \rangle$ modélise l'envoi de l'ordre de démarrage et $endWait$ la réception d'une confirmation du démarrage du capteur). A chaque période T_c, les ordres de capture sont envoyés (transition $sendOrderTog_i$) et la requête attend la réception des résultats (transition $recv(vg_i)$). A chaque période T_t, un traitement de durée d_t (transition $waitProcDur$) est réalisé par la requête

Après le démarrage des capteurs, la requête s'exécute pendant une durée d_{totale}, à l'issue de laquelle le traitement, la capture et l'envoi sont stoppés (voir les arcs de *reset* dont l'extrémité est un losange noir), ainsi que les capteurs ($stopg_i$).

Le contenu du traitement est abstrait et la transition $process$ permet de comptabiliser l'énergie d'un traitement en la synchronisant avec la dimension énergie.

Le contenu du message envoyé est également abstrait. La transition $sendToNetRes$ représente l'envoi d'un résultat.

3.7.3 Règle de transformation pour une requête conditionnelle

La règle suivante présente comment construire à partir d'une requête conditionnelle VeriSensor et des domaines de ses grandeurs, le réseau de Petri temporel coloré qui la modélise.

Règle de transformation 5. ***requête conditionnelle***

Entrée *: $condReq$ (la requête), $Dom()$ (la fonction qui fournit les valeurs possibles d'une grandeur)*
Sortie *: le réseau de Petri temporel coloré étiqueté* $\langle$**condReq.id**$\rangle$ *satisfaisant :*
//*****Etiquettes**********

- $start \in A$, $process(v\langle \mathbf{g_0}\rangle, ..., v\langle \mathbf{g_p}\rangle) \in A$
- $sendToNetRes \in A$
- $\forall g \in \langle$**CondReq.G**$\rangle$,
  - $sendOrderTo\langle \mathbf{g}\rangle \in A$
  - $recv(v\langle \mathbf{g}\rangle) \in A$
  - $endWait\langle$**CondReq.g**$\rangle \in A$
  - $stop\langle \mathbf{g}\rangle \in A$
  - $start\langle \mathbf{g}\rangle \in A$

//****places**********

- $initial \in P$, *tel que,* $Init(initial) = 1$.
- $\forall g \in \langle$**condReq.G**$\rangle$, $Dom(\langle \mathbf{g}\rangle) \in \Sigma$ *et le produit cartésien des* $Dom\left\langle (\mathbf{g_i})_{\mathbf{0\leq i\leq p}}\right\rangle \in \Sigma$
- $(started, endWaitedStartups, beforeSend) \in P^3$.
- $waitedProcDur(v\langle \mathbf{g_i}\rangle)_{0\leq i\leq p}) \in P$ *et* $C(waitedProcDur(v\langle \mathbf{g_i}\rangle)_{0\leq i\leq p}))$ *est égal au produit cartésien des* $Dom(g_i)_{0\leq i\leq p}$
- $\forall g \in condReq.G$,
  - $(waitingStart\langle \mathbf{g}\rangle, Started\langle \mathbf{g}\rangle, sentOrder\langle \mathbf{g}\rangle) \in P^3$
  - $(waitingStop\langle \mathbf{g}\rangle, waitingSendOrderTo\langle \mathbf{g}\rangle) \in P^2$
  - $received \in P$ *et* $C(receivedVal\langle \mathbf{g}\rangle) = Dom(g)$

//****transitions**********

//transitions de démarrage et de capture

- $\exists t \in T$, $\Lambda(t) = start$ *et* $Pre(initial, t) = 1$ *et* $Post(waitingStart\langle \mathbf{g_0}\rangle, t) = 1$
- $\exists t \in T$, *tel que* $\Lambda(t) = waitSense$, $I(t) = [\langle$**condReq.T$_c$**$\rangle, \langle$**condReq.T$_c$**$\rangle]$, $Pre(waitingSense, t)$, *et* $\forall g \in G$, $Post(waitingSendOrder\langle \mathbf{g}\rangle, t) = 1$
- $\forall g \in condReq.G$,
  - $\exists t \in T$, *tel que,* $\Lambda(t) = start\langle \mathbf{g}\rangle$ *et* $Pre(waitingStart\langle \mathbf{g}\rangle, t) = 1$ *et* $Post(Started\langle \mathbf{g}\rangle, t) = 1$
  - $\exists t \in T$, *tel que,* $\Lambda(t) = sendOrderTo\langle \mathbf{g}\rangle$ *et* $Pre(waitingSendOrderTo\langle \mathbf{g}\rangle, t) = 1$ *et* $Post(orderSent\langle \mathbf{g}\rangle, t) = 1$
  - $\exists t \in T$, *tel que*
    - $\Lambda(t) = recv(v\langle \mathbf{g}\rangle)$
    - $Pre(orderSent\langle \mathbf{g}\rangle, t) = 1$
    - $Post(receivedVal\langle \mathbf{g}\rangle, t) = 1$
    - $E(t, receivedVal\langle \mathbf{g}\rangle) = v\langle \mathbf{g}\rangle$

// transitions à partir du tuple $(g_i)_{0\leq i\leq p}$

- $\forall i \in \mathbb{N}$, *tel que* $0 \leq i \leq (p-1)$,
  - $\exists t \in T$ *tel que* $\Lambda(t) = endWait\langle$**condReq.g$_i$**$\rangle$ *et* $Pre(started\langle$**condReq.g$_i$**$\rangle, t) = 1$ *et* $Post(waitingStart\langle \mathbf{g_{i+1}}\rangle, t) = 1$
  - //*** arrêt des capteurs
  - $\exists t \in T$ *tel que* $\Lambda(t) = stop\langle \mathbf{g_i}\rangle$ *et* $Pre(waitingStop\langle \mathbf{g_i}\rangle, t)$ *et* $Post(waitingStop\langle \mathbf{g_{i+1}}, \mathbf{t}\rangle$
- $\exists t \in T$ *tel que* $\Lambda(t) = stop\left\langle \mathbf{g_p}\right\rangle$ *et* $Pre(waitingStop\left\langle \mathbf{g_p}\right\rangle, t) = 1$

- $\exists t \in T$, *tel que* $\Lambda(t) = endWait\langle \mathbf{g_p} \rangle$ *et* $Post(waitingSense, t) = 1$ *et* $Post(waitingEndQuery, t) = 1$

//*** traitement et évaluation de la condition

- $\exists t \in T$ *tel que*
  - $\Lambda(t) = waitProcDur((v\langle \mathbf{g_i} \rangle)_{0 \leq i \leq p}))$, $I(t) = [\langle \mathbf{condReq.d_t} \rangle, \langle \mathbf{condReq.d_t} \rangle]$
  - $\forall g \in \langle \mathbf{condReq.G} \rangle$ $Pre(receivedVal\langle \mathbf{g} \rangle, t) = 1$ *et* $E(receivedVal\langle \mathbf{g} \rangle, t) = v\langle \mathbf{g} \rangle$ *et* $E(t, waitedProcDur(v\langle \mathbf{g_i} \rangle)_{0 \leq i \leq p})) = v\langle (\mathbf{g_i})_{\mathbf{0 \leq i \leq p}} \rangle$
- $\exists t \in T$ *tel que* $\Lambda(t) = process(v\langle \mathbf{g_i} \rangle)_{0 \leq i \leq p})$ *et* $Pre(waitedProcDur(v\langle \mathbf{g_i} \rangle)_{0 \leq i \leq p}), t) = 1$ *et* $E(waitedProcDur(v\langle \mathbf{g_i} \rangle)_{0 \leq i \leq p}), t) = v\langle (\mathbf{g_i})_{\mathbf{0 \leq i \leq p}} \rangle$ *et* $G(t) = Not(\langle \mathbf{condReq.cond} \rangle)$
- $\exists t \in T$ *tel que* $\Lambda(t) = process(v\langle \mathbf{g_i} \rangle)_{0 \leq i \leq p})$ *et* $Pre(waitedProcDur(v\langle \mathbf{g_i} \rangle)_{0 \leq i \leq p}), t) = 1$ *et* $E(waitedProcDur(v\langle \mathbf{g_i} \rangle)_{0 \leq i \leq p}), t) = v(g_i)_{0 \leq i \leq p}$ *et* $Post(beforeSend, t) = 1$ *et* $G(t) = \langle \mathbf{condReq.cond} \rangle$

//*** envoi d'une alerte

- $\exists t \in T$ *tel que* $\Lambda(t) = sendToNetRes$ *et* $Pre(beforeSend, t) = 1$

La figure 4.15 représente le résultat de la règle de transformation. Les noms des places ont été omis par manque d'espace. Les noms des paramètres de la requête sont également abrégés (*e.g.* d_{totale} au lieu du nom qualifié $condReq.d_{totale}$). Si les valeurs captées vg_i satisfont la condition caractérisant l'événement, alors la transition $sendToNetRes$ est franchie, communiquant l'alerte (détection de l'événement) au réseau.

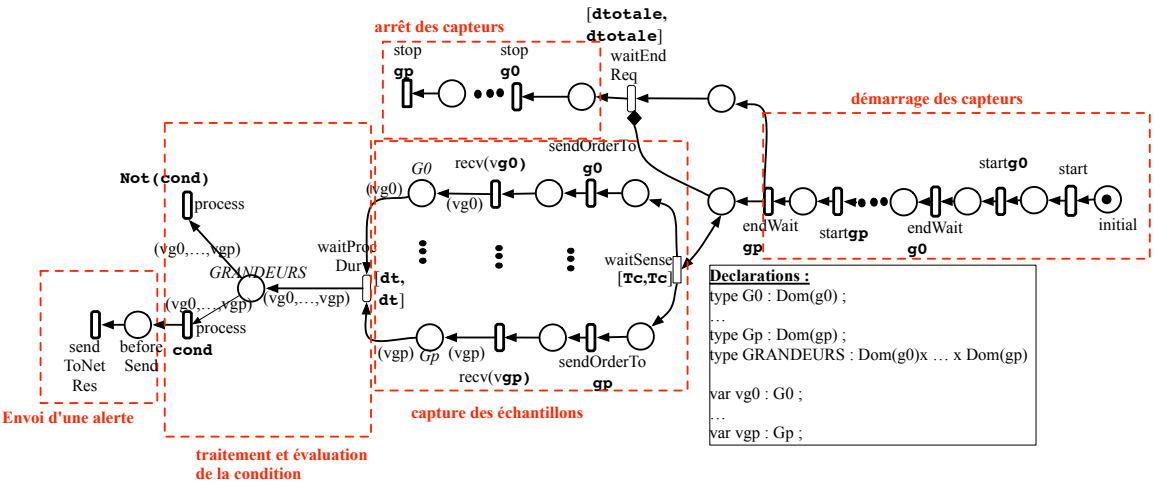


FIGURE 4.15 – ITS pour une requête conditionnelle

3.7.4 Assemblage des ITS capteurs et application

La figure 4.16 présente l'assemblage des ITS représentant les capteurs et l'application, pour le noeud `ecgtilt1`.

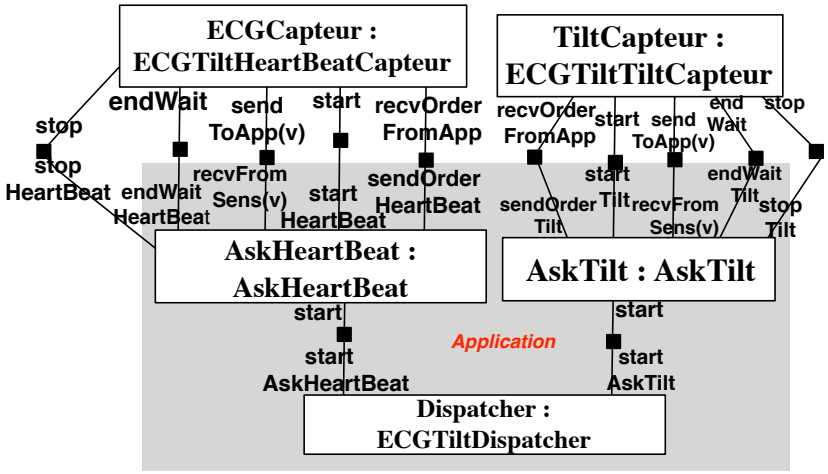


FIGURE 4.16 – Assemblage des ITS capteurs et application

Les requêtes contrôlent le fonctionnement des capteurs qui leurs sont associés :

- démarrage des capteurs (*e.g.*, synchronisation des transitions publiques `startHeartBeat/start`)
- réception des notifications de démarrage (*e.g.*, `endWaitHeartBeat/endWait`)
- arrêt des capteurs (*e.g.*, `stopHeartBeat/stop`)
- envoi des ordres de capture (*e.g.*, `sendOrderHeartBeat/recvOrderFromApp`)
- réception des échantillons (*e.g.*, `sendToApp(v)/recvFromSens(v)`)

Les requêtes sont elles mêmes contrôlées par le dispatcher. Il a pour rôle de démarrer les opérations d'une requête (*e.g.*, synchronisation `startAskHeartBeat/start`) après la réception d'un identifiant de requête supporté par l'application du noeud (*e.g.*, l'application de `ecgtilt1` supporte les requêtes `AskHeartBeat` et `AskTilt`).

3.8 Règle de transformation pour la dimension réseau

La transformation du sous-modèle réseau VeriSensor en une dimension réseau, consiste en :

- la construction du type ITS modélisant les communications
- la construction du type ITS modélisant les pertes de messages du nœud.
- la synchronisation de ces ITS expliquée dans la règle d'assemblage du nœud entier (Règle 10).

3.8.1 Règle de transformation pour les communications du nœud

La règle suivante présente la transformation du sous-modèle réseau d'une classe de nœuds en type ITS communication.

Règle de transformation 6. ***communication***

Entrée *: REQ (l'ensemble des requêtes dans le réseau de capteurs), n (le noeud)*
Sortie *: le réseau de Petri temporel coloré étiqueté $\langle \mathbf{n.cl} \rangle communication$ satisfaisant :*
// *****Etiquettes*****

- $recvFromApp(r) \in A$
- $recvFromPredecessor(r) \in A$
- $recvFromNeighbor(q) \in A$
- $sendToApp(q) \in A$
- $broadcastToNeighbors(q) \in A$
- $recvFromNeighbor(q) \in A$

// *****partie retransmission des résultats*****
//places

- $\{resReq.id : Req \in REQ\} \in \Sigma$.
- $(Received, waited) \in P^2$ *tels que* $C(Received) = \{resReq.id : Req \in REQ\}$ *et* $C(waited) = \{resReq.id : Req \in REQ\}$

//transitions

- $\exists t \in T$, *tel que,* $\Lambda(t) = recvFromApp(r)$, *et,* $Post(Received, t) = 1$ *et* $E(t, Received) = r$
- $\exists t \in T$, *tel que,* $\Lambda(t) = recvFromPredecessor(r)$, *et,* $Post(Received, t) = 1$ *et* $E(t, Received) = r$
- $\exists t \in T$, *tel que,* $\Lambda(t) = wait(r)$, $I(t) = [\langle \mathbf{n.cl.r.d_e} \rangle, \langle \mathbf{n.cl.r.d_e} \rangle]$ *et,* $Pre(Received, t) = 1$, *et* $E(Received, t) = r$ *et* $Post(waited, t) = 1$ *et* $E(t, waited) = r$.
- $\exists t \in T$, *tel que,* $\Lambda(t) = sendToNextHop(r)$, *et,* $Pre(waited, t) = 1$ *et* $E(waited, t) = r$

// *****partie rediffusion des requêtes*****
// places

- $(Received) \in P$ *et* $C(Received) = \{req : req.id \in REQ\}$
- $(Sent, waited) \in P^2$ *et* $C(Sent) = C(waited) = \{req : req.id \in REQ\}$
- $initial \in P$ *et* $C(initial) = \{req : req.id \in REQ\}$

// transitions

- $\exists t \in T$, *tel que,* $\Lambda(t) = recvFromNeighbor(q)$, *et,* $Post(Received, t) = 1$ *et* $E(t, Received) = q$
- $\exists t \in T$, *tel que,* $\Lambda(t) = send(q)ToApp$, *et,* $Pre(Received, t) = 1$, *et* $Post(Sent, t) = 1$ *et* $E(Received, t) = q$ *et* $E(t, Sent) = q$ *et*
- $\exists t \in T$, *tel que,* $\Lambda(t) = wait(q)$, *et,* $Pre(Sent, t) = 1$ *et,* $Post(waited, t) = 1$, *et,* $I(t) = [\langle \mathbf{n.cl.r.d_e} \rangle, \langle \mathbf{n.cl.r.d_e} \rangle]$ *et* $E(SentReqId, t) = q$ *et* $E(t, waited) = q$
- $\exists t \in T$, *tel que,* $\Lambda(t) = broadcast(q)ToNeighbors$, *et,* $Pre(waited, t) = 1$ *et* $E(waited, t) = q$
- $\exists t \in T$, *tel que :*
  - $\Lambda(t) = recvFromNeighbor(q)$
  - *il existe un arc inhibiteur de* $initial$ *à* $recvFromNeighbor(q)$ *étiqueté par l'expression* q

L'ITS communication du nœud a 4 rôles :

1. rediffuser les requêtes aux voisins tout en évitant les boucles infinies de rediffusion.
2. transmettre les requêtes reçues à l'application.
3. retransmettre les résultats de traitement de requêtes provenant des voisins au *next-hop*.
4. transmettre les résultats de l'application au *next-hop*.

La figure 4.17 représente l'ITS communication. La partie supérieure de la figure décrit les rôles 1 et 2 (rediffusion des requêtes, et transmission des requêtes à l'application). La partie inférieure modélise les rôles 3 et 4 (retransmission des résultats des voisins et de l'application).

Concernant la partie supérieure, la place `initial` permet de s'assurer qu'on ne retransmet une requête à l'application et aux voisins que la première fois qu'elle a été reçue. Les autres fois, la requête est reçue (transition `recvFromNeighbor(q)` connectée par un arc inhibiteur à la place `initial`), sans être rediffusée. Nous faisons le choix de cet algorithme de rediffusion contrôlée, parmi les plus utilisés dans les réseaux de capteurs sans fil. La transition `sendToApp(q)` retransmet la requête à l'application.

Concernant la partie inférieure, la transition `recvFromPredecessor(r)` modélise la réception de résultats de la part de prédécesseurs. Les prédécesseurs sont les nœuds dont le nœud courant est le *next-hop*.

Les transitions `wait(q)` et `wait(r)` modélisent l'attente pendant la durée d'émission d'un message (requête ou résultat).

Seuls les identifiants des requêtes et des résultats sont communiqués entre les nœuds. L'identifiant d'un résultat fait simplement référence à la requête dont il est issu (*i.e.*, « Res » suivi de l'identifiant de la requête).

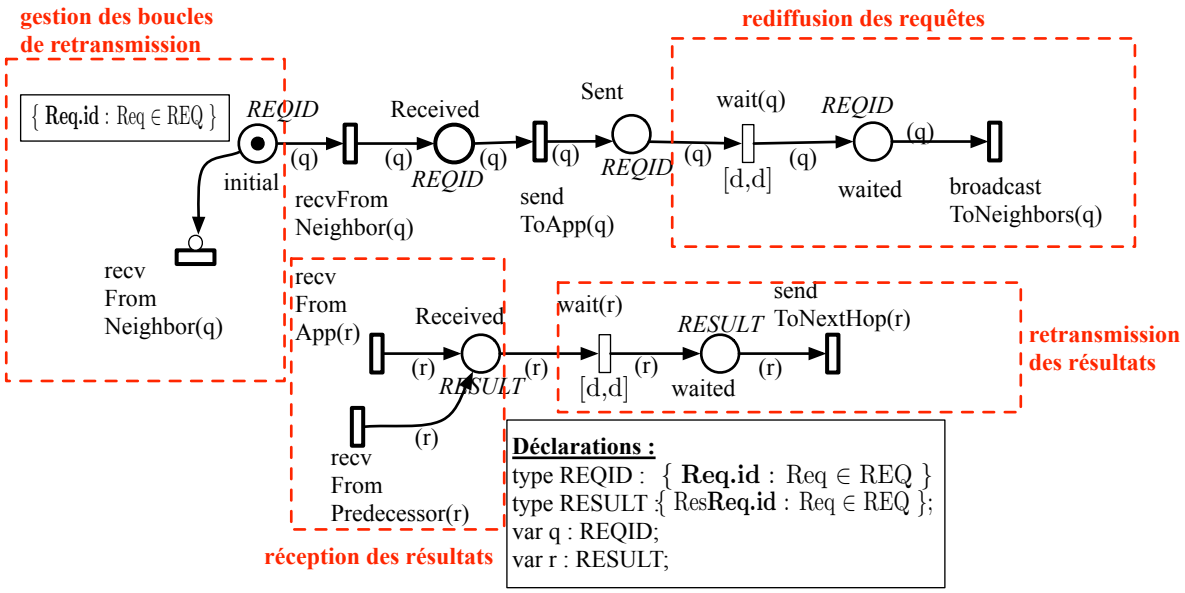


FIGURE 4.17 – ITS communication

3.8.2 Transformation pour les pertes de messages

Nous présentons d'abord la façon avec laquelle nous modélisons les pertes de messages, avant de détailler la règle de transformation produisant l'ITS loser.

Modélisation des pertes de messages. Pour modéliser la perte de messages par un noeud donné, nous définissons un ITS « loser », à part, qui ne retransmet pas une proportion des messages qu'il reçoit. Cet ITS est connecté à l'ITS communication qui émet les messages du noeud au réseau de capteurs sans fil. L'intérêt de cette architecture est de pouvoir modifier le modèle de pertes en altérant le contenu du loser sans modifier son interface et donc, sans aucun impact sur l'assemblage des ITS.

La figure 4.18 modélise sur un exemple simplifié, la façon avec laquelle nous modélisons la perte de messages. Les synchronisations dont les étiquettes sont exportés sont représentés par des carrés blancs. Ces étiquettes apparaissent à proximité de segments à extrêmité circulaire (*lollipop*) qui représentent l'interface des noeuds. Pour rester simple, nous ne modélisons que l'émission d'un seul message « M » (`sendM`).

La transition d'envoi du message de l'ITS communication (`sendM`) est, d'une part, synchronisée à la transition *send* du loser et à la transition de réception du message (`receiveM`) d'un noeud destinataire (en l'occurrence `n2`), d'autre part synchronisée à la transition *lose* du loser (voir la règle 10 d'assemblage d'un nœud).

En fonction du taux de perte de messages du loser, tantôt `send`, tantôt `lose` est franchissable, provoquant respectivement la transmission du message, ou sa perte.

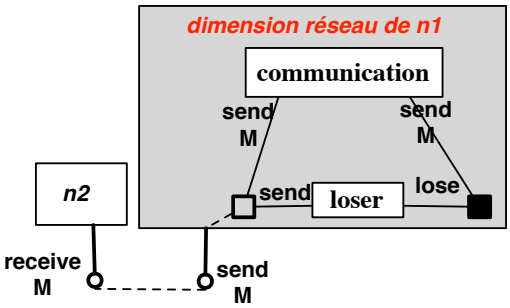


FIGURE 4.18 – Modélisation de la perte des messages

Règle de transformation pour les pertes de messages. La règle suivante présente comment construire le réseau de Petri temporel modélisant la perte de messages d'une instance de nœud. Le taux de perte de messages est défini par $\tau = 1/p$, $p \in \mathbb{N}$ (voir chapitre 3).

Règle de transformation 7. ***pertes de messages***

Entrée *: n (un nœud)*

***Sortie** : le réseau de Petri temporel étiqueté* $\langle \mathbf{n}.\mathbf{cl} \rangle Loser$ *satisfaisant :*
// Etiquettes

- $A = \{send, lose\}$

// places

- $NOK \in P$

// transitions

- $\exists t \in T$, *tel que,* $\Lambda(t) = send$, *et,* $Post(NOK, t) = 1$.
- $\exists t \in T$, *tel que,* $\Lambda(t) = lose$ *et* $Pre(NOK, t) = (1/\langle \mathbf{n}.\mathbf{cl}.\mathbf{r}.\tau \rangle) - 1$.

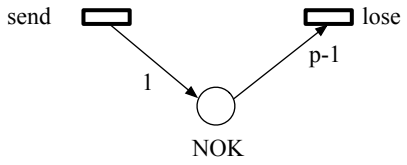


FIGURE 4.19 – Réseau de Petri temporel modélisant un taux de perte d'au plus τ messages

La figure 4.19 représente l'ITS $\langle \mathbf{n}.\mathbf{cl} \rangle Loser$ (n un nœud du déploiement) qui a pour rôle de « perdre » au plus τ message, avec τ l'élément du tuple n représentant le taux de perte maximal.

Les poids et les marquages de l'ITS $\langle \mathbf{n}.\mathbf{cl} \rangle Loser$ font en sorte que pour chaque p messages, $p-1$ sont transmis et au plus 1 est perdu.

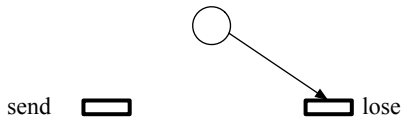


FIGURE 4.20 – Réseau de Petri temporel modélisant un taux de perte de 0 messages

Ce modèle de perte de messages peut être remplacé par un modèle avec aucune perte, en substituant à l'ITS de la figure 4.19, celui de la figure 4.20, sans modifier l'assemblage des ITS du réseau. Dans la figure 4.20, la transition `lose` n'est jamais franchissable et la transition `send` l'est toujours.

3.8.3 Assemblage du réseau et l'application

Le schéma 4.21 présente l'assemblage des ITS représentant l'application et le réseau. Les synchronisations publiques (*i.e.*, l'interface de l'ITS réseau) sont notées par des carrés blancs et connectées à leur étiquette par une flèche en pointillés.

Le réseau est constitué de l'ITS communication et loser.

L'ITS communication transmet les requêtes qu'il reçoit au dispatcher (synchronisation `sendToApp(q)` / `recvFromNet(q)`). Il reçoit les résultats des requêtes de la part de l'application (*e.g.*, `sendToNet ResAskHeartBeat/recvFromApp`

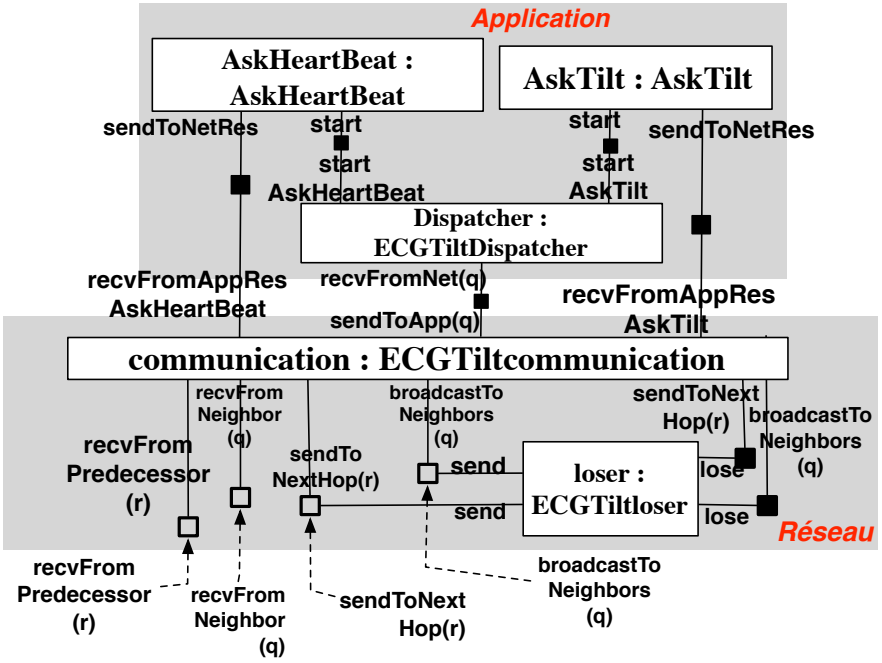


FIGURE 4.21 – Assemblage des ITS réseau et application

`ResAskHeartBeat`). L'ITS communication reçoit également des résultats de requêtes envoyés par les prédécesseurs du noeud (`recvFromPredecessor(r)`) et les retransmet (`sendToNextHop(r)`) à son *next-hop*. Il a aussi pour rôle de recevoir les requêtes diffusées par les voisins du noeud (`recvFromNeighbor(q)`) et des rediffuser en utilisant un algorithme de rediffusion contrôlée (`broadcastToNeighbors(q)`).

Les transitions publiques représentant l'émission d'un message (`broadcastToNeighbors(q)` et `sendToNextHop(r)`) sont synchronisées avec le loser pour modéliser les pertes.

3.9 Règle de transformation pour l'énergie

Cette règle décrit la transformation d'une classe de nœuds VeriSensor vers la dimension énergie.

Règle de transformation 8. ***dimension énergie***

Entrée *: une classe de nœuds cl*
Sortie *: le réseau de Petri temporel étiqueté* $\langle \mathbf{cl} \rangle$*energie satisfaisant :*
//**********etiquettes**********

- $\forall capt \in \langle \mathbf{cl.c} \rangle, decrement \langle \mathbf{capt.c_c} \rangle \in A$
- $decrement \langle \mathbf{cl.r.c_e} \rangle \in A$
- $\forall req \in cl.a, decrement \langle \mathbf{req.c_t} \rangle \in A$
- $noEnergy \in A$

//**********places**********
//energie initiale

- $\exists initial \in P$, *tel que* $M_0(initial) = \langle \mathbf{cl.e} \rangle$

//**********transitions**********

//couts de capture

- $\forall capt \in \langle \mathbf{cl.c} \rangle$, $\exists t \in T$, *tel que* $\Lambda(t) = decrement \langle \mathbf{capt.c_c} \rangle$ *et* $Pre(initial, t) = \langle \mathbf{capt.c_c} \rangle$

//cout d'emission

- $\exists t \in T$, *tel que* $\Lambda(t) = decrement \langle \mathbf{r.c_e} \rangle$ *et* $Pre(initial, t) = \langle \mathbf{r.c_e} \rangle$

//cout de réception

- $\exists t \in T$, *tel que* $\Lambda(t) = decrement \langle \mathbf{r.c_e} \rangle$ *et* $Pre(initial, t) = \langle \mathbf{r.c_r} \rangle$

//couts de traitement des requêtes

- $\forall req \in \langle \mathbf{a} \rangle$, $\exists t$, *tel que* $\Lambda(t) = decrement \langle \mathbf{req.c_t} \rangle$ *et* $Pre(initial, t) = \langle \mathbf{req.c_t} \rangle$
- $\exists t \in T$, *tel que :*
  - $\Lambda(t) = noEnergy$
  - *t est connectée à initial par un arc inhibiteur de valuation* $r.c_e - 1$

Le manque d'énergie (*i.e.*, aucun jeton dans la place) bloque les opérations consommatrices en énergie (capture, accès au réseau, etc.). Quand tous les nœuds ne disposent plus d'énergie, le système ne peut plus continuer de fonctionner. La transition `noEnergy` est un artefact de modélisation qui permet à un nœud de se débarrasser des messages qu'il reçoit (lorsque l'énergie n'est plus suffisante) tout en laissant l'émetteur de ces messages, franchir la synchronisation. Cela est important pour préserver la modularité de la spécification et pour avoir un comportement réaliste sous des conditions dégradées (*e.g.*, certains nœuds sont morts).

Le modèle de consommation de la dimension énergie est utilisé pour déterminer la durée de vie du réseau de capteurs sans fil. Si l'objectif est d'étudier le comportement en régime permanent du système, on utilise un modèle à énergie infinie (suppression des arcs entre l'unique place et les transitions), qui ne consomme pas d'énergie. Puisque la spécification formelle est compositionnelle, d'autres composants et synchronisations du modèle ne sont pas impactées. Cela permet d'analyser des comportements qualitatifs du réseau de capteurs sans fil, sans payer le coût (en terme d'explosion de l'espace des états) dû à la prise en compte de la consommation en énergie des nœuds.

3.10 Transformation pour le routage

Nous présentons d'abord la façon avec laquelle nous modélisons le routage, avant de détailler la règle de transformation produisant l'ITS routage.

Modélisation du routage. On associe à chaque nœud un type ITS qui a pour rôle de router les résultats du nœud vers son *next-hop*. L'intérêt est de pouvoir modifier le *next-hop* du nœud, sans avoir à modifier les synchronisations entre nœuds du réseau.

La figure 4.22 modélise un exemple simple de routage des messages d'un noeud n_1 dans un réseau contenant 3 noeuds (n_1, n_2, n_3) et une station de base (b). Pour simplifier, n_1 ne peut émettre qu'un seul message « M ».

Nous définissons une synchronisation par destinataire possible du message (n_2, n_3, b). Une synchronisation relie la transition d'émission de M à une transition de la table de routage $enable_{n_i}$ et au destinataire n_i. La seule transition de la table de routage qui est franchissable est celle qui correspond au *next-hop* de $n1$ (*i.e.*, $enable_{n_2}$). C'est pourquoi, lorsque n_1 émet M la seule synchronisation active est $enable_{n_2}$ ce qui provoque la réception de M par n_2, le *next-hop* de n_1.

Ce principe d'assemblage permet de supporter le routage multi-chemins (*i.e.*, plusieurs *next-hops* possibles), ainsi que les reconfigurations (*i.e.*, modifications du *next-hop*) dues au déplacement ou à la mort de noeuds.

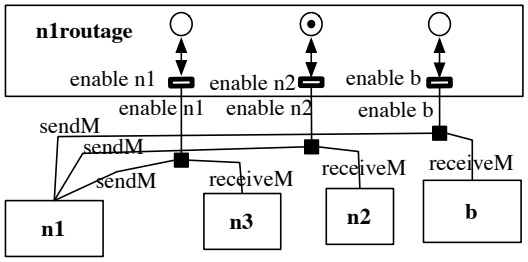


FIGURE 4.22 – Assemblage des ITS pour le routage

Règle de transformation pour le routage. La règle de transformation suivante décrit comment on produit le type ITS décrivant la table de routage à partir d'éléments de la spécification VeriSensor.

Règle de transformation 9. ***table de routage***

Entrée *: Un déploiement N, une instance de nœud* $n_0 \in N$
Sortie *: le réseau de Petri temporel coloré étiqueté* $\langle \mathbf{n_0.id} \rangle Routage$ *satisfaisant :*
//places

- $enable(n) \in A$

//places

- $\Sigma = \{\{n.id \ : \ n \in N\}\}$
- $initial(n) \in P$ *et* $C(initial(n)) = \{n.id \ : \ n \in N\}$
- $Init(initial(n)) = \{n_0.n_h.id\}$

//transitions

- $\exists t \in T$, *tel que* $\Lambda(t) = enable(n)$ *et* $Pre(initial(n), t) = 1$ *et* $Post(initial(n), t) = 1$ *et* $E(t, initial(n)) = n$ *et* $E(initial(n), t) = n$

Le résultat de la règle de transformation est représenté en figure 4.23. Conformément aux notations $\langle \mathbf{n_0.n_h.id} \rangle$ est l'identifiant du *next-hop* du noeud n_0.

Un marquage par un jeton de valeur id signifie que id est le *next-hop* du nœud n_0 (le nœud dont on représente la table de routage).

Le langage VeriSensor impose un seul *next-hop* par noeud (voir la définition 3.2, page 62). Dans le futur, rien n'empêche de supporter le routage multi-chemins.

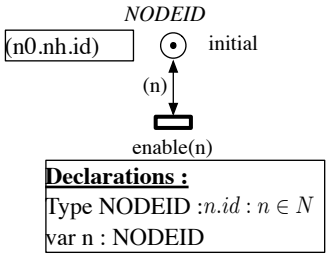


FIGURE 4.23 – le type ITS représentant la table de routage du nœud n_0

Les synchronisations entre le nœud, sa table de routage et les autres nœuds du réseau sont présentées dans la règle de transformation 11 (modèle formel du réseau de capteurs sans fil entier).

3.11 Règle de transformation pour un nœud

Le type ITS représentant un nœud est composé de l'instance de l'ITS communication, du loser, du dispatcher, des requêtes, des capteurs, des environnements locaux et de l'énergie. L'ITS communication est synchronisé avec le dispatcher et les requêtes. Les requêtes sont synchronisées avec les capteurs concernés. Chaque capteur est synchronisé avec l'environnement local modélisant l'évolution de la grandeur du capteur. Les requêtes, les capteurs et l'ITS communication sont synchronisés avec l'énergie pour comptabiliser la consommation d'énergie des traitements, captures et communications du nœud.

Dans cette règle de transformation, on indique en gras et entre crochets ([**exemple**]) tout nom (ou partie de nom) des éléments de l'ITS en sortie (*e.g.*, étiquette de transition, place) qui dépendent d'éléments de la spécification VeriSensor donnée en entrée. Ceci pour éviter la confusion avec la notation entre chevrons des synchronisations.

Règle de transformation 10. ***nœud***

Entrée *: n (un nœud VeriSensor), REQ (l'ensemble des requêtes du modèle de requêtes).*
Sortie *: un ITS composite* [**n**.**id**] *satisfaisant :*
//********instances*********

- $\forall capt \in n.cl.c$,
  - [**capt**.**g**]$Env \in I$ *et* $type$([**capt**.**g**]Env) = [**n**.**id**][**capt**.**g**]Env
  - [**capt**.**g**]$Capteur \in I$ *et* $type$([**capt**.**g**]$Capteur$) = [**n**.**cl**][**capt**.**g**]$Capteur$
- $\forall req \in n.cl.a$, [**req**.**id**] $\in I$ *et* $type$([**req**.**id**]) = [**req**.**id**]
- $dispatcher \in I$, *et* $type(dispatcher)$ = [**n**.**cl**]$dispatcher$
- $communication \in I$, *et* $type(communication)$ = [**n**.**cl**]$communication$
- $loser \in I$, *et* $type(loser)$ = [**n**.**cl**]$loser$
- $energie \in I$ *et* $type(energie)$ = [**n**.**cl**]$energie$

//*******synchronisations******
//capteur, environnement et énergie

- $\forall capt \in n.cl.c,\ \forall Value \in Dom(capt.g)$
  $< \top, < call(sendToSens[\mathbf{Value}], [\mathbf{capt.g}]Env);$
  $call(recvFromEnv[\mathbf{Value}], [\mathbf{capt.g}]Capteur);$
  $call(decrement[\mathbf{capt.c_c}], energie) >>\in Sync$

//requête et énergie

- $\forall req \in a,$
  $< \top, < call(process, [\mathbf{req.id}]); call(decrement[\mathbf{req.c_t}], energie) >>\in Sync$

//capteur et requête

- $\forall req \in a,\ \forall g \in req.G,$
  - $< \top, < call(start[\mathbf{g}], [\mathbf{req.id}]); call(start, [\mathbf{capt.g}]Capteur) >>\in Sync$
  - $< \top, < call(stop[\mathbf{g}], [\mathbf{req.id}]); call(stop, [\mathbf{capt.g}]Capteur) >>\in Sync$
  - $< \top, < call(recv(v[\mathbf{g}]), [\mathbf{req.id}]);$
    $call(sendToApp(v), [\mathbf{capt.g}]Capteur) >>\in Sync$
  - $< \top, < call(sendOrder[\mathbf{g}], [\mathbf{req.id}]);$
    $call(recvOrderFromApp, [\mathbf{capt.g}]Capteur) >>\in Sync$
  - $< \top, < call(endWait[\mathbf{g}], [\mathbf{req.id}]);$
    $call(endWait, [\mathbf{capt.g}]Capteur) >>\in Sync$

//dispatcher et requêtes

- $\forall req \in a < \top, < call(start[\mathbf{req.id}], dispatcher);\ call(start, [\mathbf{req.id}]) >>\in Sync$

//dispatcher et communication

- $< \top, < call(sendToApp(q), communication);$
  $call(recvFromNet(q), dispatcher) >>\in Sync$

//communication et requêtes

- $\forall req \in a < \top, < call(sendToNetRes[\mathbf{req.id}], [\mathbf{req.id}]);$
  $call(recvFromAppRes[\mathbf{req.id}], communication) >>\in Sync$

//****communication, loser et extérieur****

//envoi des résultats

- $< sendToNextHop(r), < call(sendToNextHop(r), communication);$
  $call(decrement[\mathbf{r.c_e}], energie); call(lose, loser) >>\in Sync$
- $< sendToNextHop(r), < call(sendToNextHop(r), communication);$
  $call(decrement[\mathbf{r.c_e}], energie); call(send, loser) >>\in Sync$

//diffusion des requêtes

- $< broadcastToNeighbors(q),$
  $< call(broadcastToNeighbors(q), communication);$
  $call(decrement[\mathbf{q.c_e}], energie); call(send, loser) >>\in Sync$
- $< broadcastToNeighbors(q),$
  $< call(broadcastToNeighbors(q), communication);$
  $call(decrement[\mathbf{r.c_e}], energie); call(lose, loser) >>\in Sync$

//réception des résultats

- $< recvFromPredecessor(r),$
  $< call(recvFromPredecessor(r), communication);$
  $call(decrement[\mathbf{r.c_r}], energie) >>\in Sync$

- $< recvFromPredecessor(r)$,
  $< call(recvFromPredecessor(r), communication)$;
  $call(noEnergy, energie) >>\in Sync$

//réception des requêtes

- $< recvFromNeighbor(q)$,
  $< call(recvFromNeighbor(q), communication)$;
  $call(decrement[\mathbf{r}.\mathbf{c_r}], energie) >>\in Sync$
- $< recvFromNeighbor(q)$,
  $< call(recvFromNeighbor(q), communication)$;
  $call(noEnergy, energie) >>\in Sync$

La figure 4.24 montre l'ITS composite modélisant le nœud *ecgtilt1* du réseau biomédical (voir exemple fil rouge, p. 59, chapitre 3). Par souci de lisibilité, seules les synchronisations pour la grandeur Tilt ne sont pas détaillées (*i.e.*, environnement local, capteur, requête)

L'étiquette d'une synchronisation publique est connectée à la synchronisation par une flèche en pointillés.

`ECGCapteur` par exemple, peut capter une valeur parmi celles que fournit l'environnement local `HeartBeatEnv`. Cela est représenté par la synchronisation entre `sendToSens(v)` et `recvFromEnv(v)`.

Les deux requêtes traitées par ce nœud sont synchronisées avec le dispatcher, les capteurs concernés et le réseau.

Le réseau gère la retransmission des requêtes diffusées et élimine les duplicats en utilisant un algorithme de diffusion contrôlée. Il reçoit les valeurs à envoyer à la station de base de la part de l'application et des autres nœuds. Les requêtes adressées au nœud sont transmises au dispatcher (voir Fig. 4.24).

L'interface du réseau permet de transmettre les requêtes et leurs résultats. Elle définit un ensemble de services :

- diffuser les requêtes (`broadcastToNeighbors(q)`)
- recevoir les requêtes de la part d'un voisin (`recvFromNeighbor(q)`)
- émettre les résultat des requêtes au *next-hop* (`sendToNextHop(r)`)
- recevoir les résultat des requêtes de la part d'un prédécesseur (`recvFromPredecessor(r)`)

Les transitions d'émission de message (`broadcastToNeighbors(q)`, `sendToNextHop(r)`) sont synchronisées avec l'ITS *loser* qui en « perd » (service *lose*) une proportion au plus égale à τ.

Les opérations énergivores (*i.e.* capture et traitement, émission vers l'extérieur et réception) sont synchronisées avec des transitions qui décrémentent l'énergie courante du noeud d'un certain nombre d'unités (*e.g.*, `decrement5`). La transition `noEnergy` est un artefact de modélisation qui permet à un nœud de se débarrasser des messages qu'il reçoit (lorsque l'énergie n'est plus suffisante) tout en laissant l'émetteur de ces messages franchir la synchronisation.

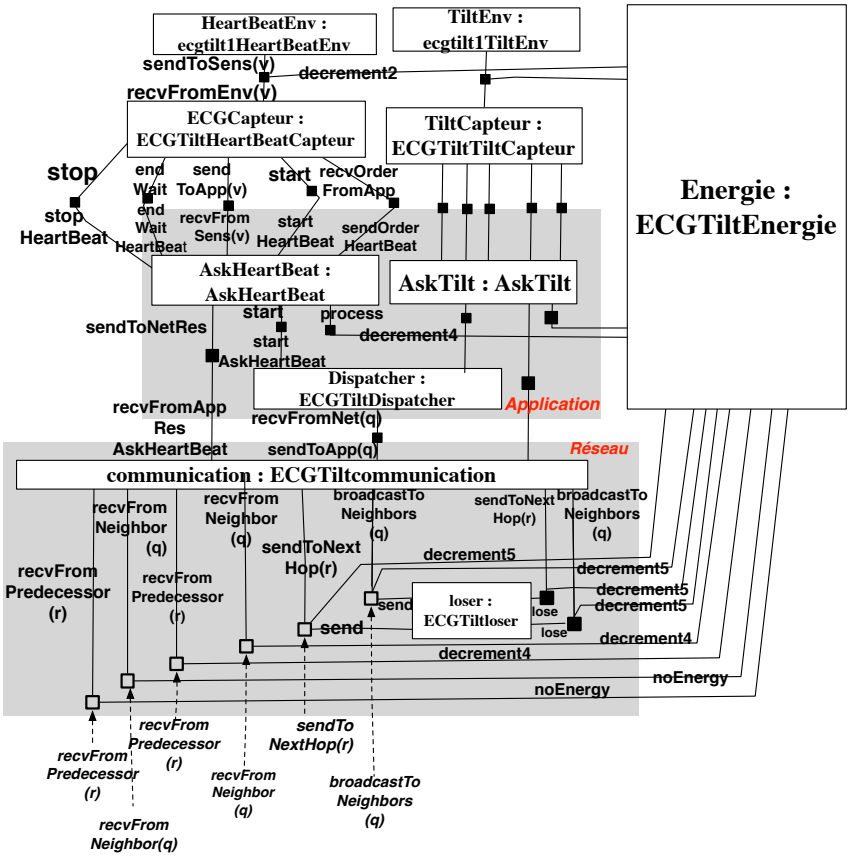


FIGURE 4.24 – Vue globale de l'ITS du nœud *ecgtilt1*

3.12 La construction du type ITS composite représentant le réseau de capteurs sans fil

La construction du type ITS représentant le réseau de capteurs sans fil consiste à synchroniser les interfaces de communication des nœuds entre elles pour l'échange des requêtes et des résultats.

Concernant la diffusion des requêtes dans le réseau, nous créons pour chaque nœud toutes les synchronisations qui lui permettent de recevoir les requêtes d'un de ses voisins et de les diffuser à tous ses voisins. Pour cela, nous utilisons l'algorithme de détermination des voisins énoncé en 3.12.1. Les boucles infinies de rediffusion sont évitées grâce à la structure du réseau de Petri *communication* de chaque nœud (voir règle de transformation 7). La diffusion est implémentée de manière atomique ce qui évite tout effet d'entrelacement (source d'explosion combinatoire en *model checking*). Cette implémentation permet à tous les nœuds destinataires du message diffusé, de le recevoir au même moment, ce qui est plus fidèle à la réalité qu'une implémentation sous la forme de suite d'*unicasts* [OT07].

Par ailleurs, l'abstraction selon laquelle un message émis par un noeud est reçu au même instant par les noeuds voisins nous paraît cohérente avec le niveau de détail que l'on a adopté dans la spécification. En effet, sachant que nous faisons déjà l'hypothèse de ne pas modéliser les détails de transmission de trames de la couche

MAC (nous définissons à la place un taux d'erreurs qui prend en compte l'imperfection du canal de transmission), il nous paraît cohérent de ne pas détailler la couche physique (temps de transmission proportionnel à la distance entre entre les noeuds). C'est une abstraction que nous faisons et qui est commune dans le domaine [MSZ07].

Concernant la transmission et retransmission des résultats aux *next-hops*, on définit un ensemble de synchronisations qui relient un nœud source, un nœud destination et la table de routage. Structurellement, cela permet des communications arbitraires entre les nœuds, mais seules les synchronisations autorisées par la table de routage (découlant d'un marquage de 1 d'une place `enableNodeId`) sont réellement activées. Cela permet de supporter des mises à jour de la table de routage, *sans modification* des nœuds ou des synchronisations entre ces nœuds.

3.12.1 Détermination des voisins d'un nœud

Pour construire le type ITS composite représentant le réseau de capteurs sans fil entier, il faut modéliser la diffusion des requêtes d'un nœud à ses voisins. Pour cela, il faut déterminer pour chaque nœud, l'ensemble de ses voisins.

La définition suivante présente la fonction $V()$ qui à un nœud associe ses nœuds voisins.

Définition 4.11. ***Voisins d'un nœud***

Soit $Depl$ un déploiement VeriSensor
On définit la fonction $V()$, qui à $n_0 \in Depl$ associe l'ensemble de ses voisins, par :

$$V : Depl \rightarrow 2^{Depl}$$
$$n_0 \mapsto \{n \in Depl : distance(n.pos, n_0.pos) \leq minimum(n.cl.p_e, n_0.cl.p_e)\}$$

avec
- $distance()$*, la distance euclidienne entre deux points de* $\mathbb{R}^3$
- $minimum()$*, le minimum de deux réels positifs.*

3.12.2 Règle de transformation pour le réseau de capteurs sans fil

La règle de transformation suivante décrit comment on transforme les éléments de la specification VeriSensor en un type ITS composite représentant le réseau de capteurs sans fil.

Règle de transformation 11. ***réseau de capteurs sans fil***

Entrée *: $Depl$ (Le modèle de déploiement VeriSensor), REQ (l'ensemble des requêtes du modèle de requêtes).*
Sortie *: un ITS composite $ReseauCapteursSansFil$ satisfaisant :*
//*******instances******

- $\forall n \in Depl$, $[\mathbf{n.id}] \in I$ *et* $type([\mathbf{n.id}]) = [\mathbf{n.id}]noeud$
- $\forall n \in Depl$, $[\mathbf{n.id}]Routage \in I$ *et* $type([\mathbf{n.id}]Routage) = [\mathbf{n.id}]Routage$

//*****synchronisations****

//diffusion des requêtes aux voisins

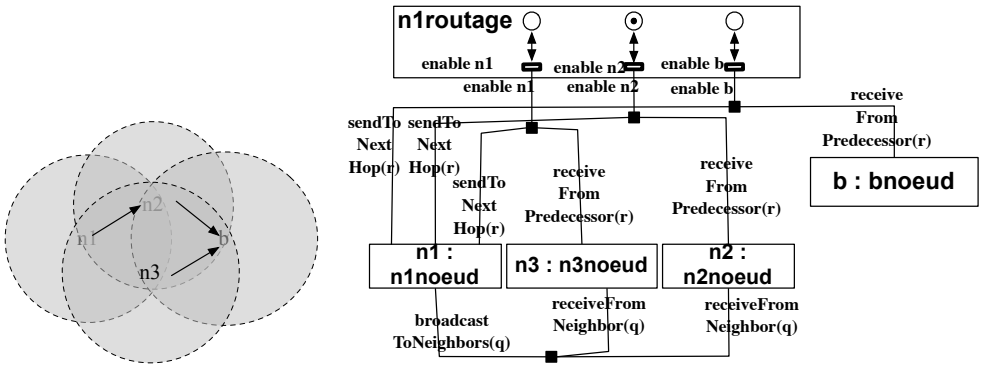
- $\forall n \in Depl$,
  $< \top, < call([\mathbf{n.id}], broadcastToNeighbors(q)); (\sigma_i)_{0 \leq i \leq m} >> \in Sync$, *avec,*
  - $(\sigma_i)_{0 \leq i \leq m}$ *la suite des actions* $call([\mathbf{n_i.id}], receiveFromNeighbor(r))_{0 \leq i \leq m}$
  - $(n_i)_{0 \leq i \leq m}$ *est la suite des nœuds voisins de n*

//envoi du résultat au *next-hop* sauf pour la station de base

- $\forall n \in Depl$ *tel que* $n.n_h \neq n.id$, $\forall n' \neq n$, $< \top, < call([\mathbf{n.id}], sendToNextHop(r));$
  $call([\mathbf{n.id}]Routage, enable[\mathbf{n.id}]);$
  $call([\mathbf{n'.id}], recvFromPredecessor(r)) >> \in Sync$

La figure 4.25a représente un exemple de topologie de réseau de capteurs sans fil. Les portées de communication des nœuds sont indiquées par des disques gris et les chemins de routage par des flèches à trait plein. n_1, n_2, n_3 sont les identifiants des trois noeuds et b est l'identifiant de la station de base.

La figure 4.25b montre uniquement (par souci de lisibilité) l'ensemble des synchronisations permettant à n_1 d'émettre des messages. n_1 peut émettre un résultat à son *next-hop* ou diffuser une requête à ses voisins. Les trois synchronisations liant n_1, la table de routage, et les nœuds n_2, n_3 et b représentent l'émission d'un résultat. Le marquage de la table de routage active la synchronisation entre n_1 et son *next-hop* n_2 (voir les chemins de routage en figure 4.25a) et autorise la transmission du message. Une seule synchronisation entre n_1 et tous ses voisins (voir les portées en figure 4.25b) représente la diffusion atomique des requêtes par n_1.



(a) topologie d'un réseau de capteurs sans fil

(b) synchronisations de n_1 pour l'émission

FIGURE 4.25 – ITS représentant un exemple de réseau de capteurs sans fil

4 Abstractions

La transformation précédente appliquée en tant que telle génère une spécification formelle et un espace d'états qui croissent très vite avec l'augmentation de la taille des domaines des grandeurs physiques. Le risque d'explosion combinatoire est accru.

Cependant, en fonction de la propriété à vérifier, le comportement de plusieurs exécutions est similaire. On conserve donc de la redondance dans l'espace des états.

Nous proposons d'éliminer cette redondance en abstrayant les parties de la spécification formelle qui ne sont pas nécessaires. Le risque d'explosion combinatoire sera réduit.

La dérivation automatique de telles abstractions est importante car : i) elles sont correctes par construction ii) l'utilisation des abstractions n'implique pas une connaissance de l'utilisateur final des techniques sous-jacentes.

Principalement, nous proposons l'abstraction qui consiste à retenir, pour les grandeurs physiques, une valeur symbolique (*i.e.* classe d'équivalence) par étendue produisant le même comportement du système. Pour cela, nous parcourons les structures de contrôle à partir desquelles nous dérivons la liste de ces classes d'équivalence.

Grâce à cette abstraction, on obtient une spécification formelle et un espace d'états qui dépend du nombre de classes d'équivalence des grandeurs. Le risque d'explosion combinatoire est donc réduit.

L'algorithme suivant détermine comment extraire les domaines de valeur symbolique (ou classes d'équivalence) d'une grandeur, à partir des structures de contrôle (*i.e.*, la liste des conditions spécifiées dans les requêtes conditionnelles du modèle).

4.1 Détermination des classes d'équivalence d'une grandeur

Les techniques de vérification sous-jacentes nécessitent des ensembles discrets de valeurs ; les grandeurs physiques continues doivent être discrétisées. Pour cela, il est seulement nécessaire de retenir une valeur symbolique par étendue produisant le même comportement du système (*i.e.*, classe d'équivalence). Par exemple, le comportement d'une requête conditionnelle dépend des valeurs captées, et peut être déduit à partir des seuils intervenant dans les conditions. Dans le réseau biomédical, pour la requête conditionnelle `AskHeartBeat` (voir la figure 3.9, p. 70), le domaine de `HeartBeat` peut être réduit à deux valeurs symboliques : $\hat{S_0}$ (strictement inférieur à 40) et $\hat{S_1}$ (supérieur à 40).

L'algorithme de détermination des classes d'équivalence suppose initialement que tous les domaines de valeurs sont symétriques (*i.e.*, Il existe une seule classe d'équivalence égale à $Dom(g)$). Ensuite, par analyse des structures de contrôle de la spécification VeriSensor, la partition de $Dom(g)$ est raffinée jusqu'à obtenir l'ensemble des classes d'équivalence de la grandeur. Toutes les contraintes sont combinées pour produire un ensemble minimal de valeurs pour chaque domaine. Ces techniques sont dérivées de la détection automatique de symétries [TMDM03] ou de l'évaluation symbolique de trajectoire [ABMS07]. La complexité de ces techniques est très faible, puisqu'elle repose sur la taille de la spécification et non de celle de l'espace des états.

Les définitions suivantes présentent les notions de partition issue d'une comparaison et de raffinement d'une partition.

Définition 4.12. ***Partition issue d'une comparaison***

- *Soit une requête conditionnelle* $condReq$.
- *Soit une des comparaisons* $comp =< g, \diamond, C >$ *sur laquelle porte la condition* $condReq.cond$, *avec :*
  - $g \in condReq.G$,
  - $\diamond \in \{<, >, \leq, \geq, =\}$,
  - $C \in Dom(g)$

La partition de $Dom(g)$ *issue de* $comp$ *est définie par :*
$P_{comp} = \{V_{comp}, Dom(g) \setminus V_{comp}\}$ *avec* $V_{comp} = \{v \in Dom(g) \ : \ v \diamond C\}$

Définition 4.13. ***Raffinement d'une partition***

- *Soit une partition* $P = (P_i)_{0 \leq i \leq n}$ *du domaine d'une grandeur* g.
- *Soit* P_{comp} *la partition issue d'une comparaison* $comp$ *de* g.

Le raffinement de P *par* P_{comp} *est défini comme la partition* $P_{refined}$ *satisfaisant :*

- *si* $P_i \cap V_{comp} = P_i$ *ou* $P_i \cap P \setminus V_{comp} = P_i$ *alors* $P_i \in P_{refined}$
- *sinon* $P_i \cap V_{comp} \in P_{refined}$ *et* $P_i \cap V_{comp} \in P_{refined}$

4.2 Discrétisation

Nous discrétisons le chronogramme (projection) décrivant l'évolution de la grandeur g en fonction du temps à une position de nœud n donnée, en fonction des seuils (constantes) présents dans les comparaisons, faisant intervenir la grandeur. On obtient un chronogramme qui prend ses valeurs dans l'ensemble des classes d'équivalences de g.

Soit la partition P_i contenant les classes d'équivalence d'une grandeur g. Soit $(d_k, V_k)_{1 \leq k \leq m}$ la projection de l'évolution de g à la position d'un nœud n.

La discrétisation consiste à associer à chaque V_k une valeur symbolique $\hat{S}$ associée à l'élément C (classe d'équivalence) de P_i tel que $V_k \in C$.

4.3 Conclusion

Les abstractions que nous définissons réduisent le risque d'explosion combinatoire en ayant un impact significatif sur la taille de l'espace d'états.

En effet, ce dernier croît avec le nombre de seuils de chaque grandeur, alors que, sans abstractions, il croît avec la taille de leurs domaines.

Dans le pire des cas, le nombre de seuils est égal au nombre des valeurs des grandeurs. Cependant, cela implique d'avoir un prédicat par valeur de la grandeur, ce qui est rare (voire inexistant) dans les applications du domaine [ASSC02].

Dans le meilleur des cas, la spécification VeriSensor ne contient aucune structure de contrôle (*i.e.* absence de requêtes conditionnelles) et la taille de l'espace d'états dépend de celle de l'ensemble des grandeurs physiques captées par le réseau.

Dans le cas moyen, la taille de l'espace des états est inférieure à celle de l'espace d'états obtenu sans aucune abstraction.

5 Vérification des propriétés sur le modèle formel

Dans cette section, nous présentons comment nous vérifions les propriétés définies en section 13 du chapitre 3 sur le modèle formel d'une instance de spécification VeriSensor.

La vérification de certaines propriétés, nécessite de modifier le modèle formel (*e.g.* ajouter des compteurs). La vérification de toutes les propriétés nécessite une analyse de l'espace d'états généré par le *model checker* ITS (recherche d'états satisfaisant une condition, analyse des traces)

Le *model checker* supporte le calcul de propriétés d'accessibilité et exprimées en logique du temps arborescent [CES86] (une logique temporelle dont le modèle de temps est arborescent : différents chemins peuvent être pris dans le futur). Le reste relève de notre contribution.

5.1 Durée de vie dans le pire des cas (P1)

Pour calculer la durée de vie du réseau de capteurs sans fil dans le pire des cas, dans un scénario environnemental donné, nous déterminons la trace la plus courte (en terme de transitions modélisant l'écoulement du temps) vers un état de l'espace d'états, où le réseau est mort.

Pour cela, on détermine d'abord l'ensemble des états S_{death} où le réseau est mort (*i.e.*, au moins un nœud est mort, voir définition adoptée en section 13, chapitre 3).

Ensuite, nous examinons les traces conduisant à ces états. Elles sont constituées de deux types de transitions :

- celles qui représentent l'écoulement d'une unité de temps (transitions *elapse*)
- celles qui modélisent une action du réseau de capteurs sans fil.

Nous déterminons parmi ces traces, celle qui contient le moins de transitions *elapse*. Le nombre de ces transitions détermine la durée de vie dans le pire des cas du réseau de capteurs sans fil.

Nous exhibons également le *scénario* qui conduit à cette durée de vie, constitué de la séquence des transitions de la trace minimale.

Pour déterminer si la durée de vie dans le pire des cas est supérieure à une borne D_{min} (P1), il suffit de comparer la valeur de la métrique avec D_{min}.

5.2 Proportion de nœuds vivants (P2)

La proportion de nœuds vivants, dans un état de l'espace d'états, est égale au nombre de nœuds qui ont une énergie inférieure au coût d'une émission ou d'une réception de message (voir définition d'un nœud vivant en section 13, chapitre 3) divisé par le nombre total de nœuds du réseau.

5.3 Proportion de nœuds disposant d'un chemin vers la station de base (P3)

Calcul de la métrique. Pour calculer la proportion des nœuds disposant d'un chemin de connexion vers la station de base, il faut connaître les liens de connexions entre les nœuds (voir définition 2.16, p.26). Pour cela, on utilise l'attribut p_e (portée d'émission) du sous-modèle réseau d'un nœud et la position pos du nœud dans le déploiement.

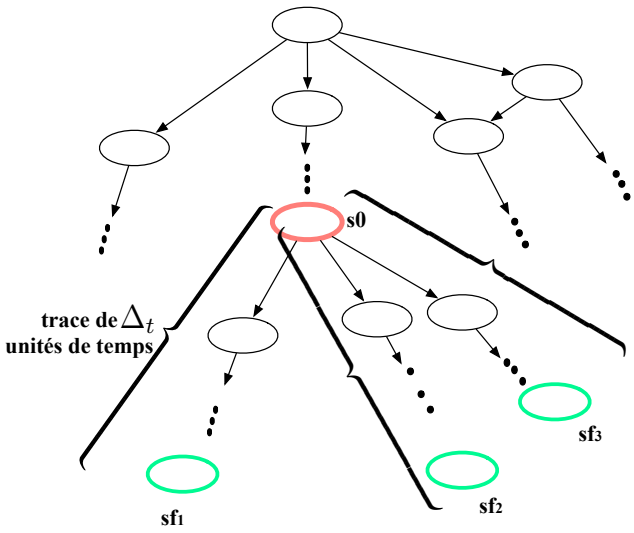


FIGURE 4.26 – Calcul de la propriété relaxée

Vérification de la propriété relaxée. La figure 4.26 représente l'espace d'états généré par le *model checker*. Pour vérifier la version relaxée de la propriété, il faut considérer pour chaque état $s_0 \in S$ l'ensemble des traces partant de s_0 et dont la durée d'exécution est Δ_t (*i.e.*, contenant Δ_t transitions $elapse$). Ce sont les traces allant de s_0 à s_{f_1}, s_{f_2}, s_{f_3} (entre accolades sur la figure 4.26). Pour chacune de ces traces, il est nécessaire de vérifier que la proportion des noeuds disposant d'un chemin de connexion avec la station de base est inférieure à c. Le risque d'explosion combinatoire est très grand.

C'est pourquoi nous proposons de vérifier pour un seul état du système la validité de la propriété relaxée. Nous considérons toutes les exécutions partant de cet état, à un horizon de Δ_t unités de temps.

Un nœud dispose d'un chemin de connexion avec la station de base pendant l'exécution d'une trace, s'il existe une suite d'états $(S_i)_{1\leq i\leq m}$ de la trace tels que S_{i+1} est un successeur (pas forcément immédiat) de S_i, et une suite de nœuds $(n_i)_{1\leq i\leq m}$ vivants tels que :

- pour tout i tel que $1 \leq i \leq (m-1)$, il existe un lien de connexion entre n_i et n_{i+1} dans S_i
- il existe un lien de connexion entre n_m et la station de base dans S_m.

5.4 Taux de messages reçus par la station de base (P4)

Calcul de la métrique. Pour déterminer le taux de messages reçus par la station de base, nous ajoutons au modèle formel un compteur des messages émis par n'importe quel nœud ($cptEmission$), ainsi qu'un compteur des messages reçus par la station de base ($cptReception$). Le quotient $cptReception/cptEmission$ correspond au taux de messages reçus par la station de base.

Vérification de la propriété relaxée. Pour vérifier que, pour un état s_0 de l'espace d'états, pour tout intervalle de temps de durée Δ_t, le taux de messages reçus est supérieur à τ, nous nous assurons que, pour toute trace de Δ_t unités de temps partant de s_0, le taux de messages reçus est supérieur à τ.

D'abord, nous notons $cptReception_s$ le nombre de messages reçus à l'état s et $cptEmission_s$, le nombre de messages émis à l'état s. Le taux de messages reçus sur une trace partant d'un état s_0 à un état s_f est le quotient entre le nombre de messages reçus au cours de cette trace ($cptReception_{s_f} - cptReception_{s_0}$) et le nombre de messages émis au cours de cette trace ($cptEmission_{s_f} - cptEmission_{s_0}$).

5.5 Latence (P5)

Calcul de la métrique. Sachant que dans VeriSensor, les chemins de routage sont fixes et les durées d'émission dépendent du noeud, la latence de bout-en-bout d'un message dépend de la source du message n. La latence Lat_n d'un message envoyé par un nœud n est égale à la somme de la durée d'émission de n (spécifiée dans le sous-modèle réseau de sa classe de nœuds) et de celle des nœuds formant son chemin de routage vers la station de base (à condition que tous ces nœuds soient vivants). Si n ou un des nœuds formant son chemin de routage est mort, alors la latence est égale à $+\infty$.

Pour déterminer le taux de messages émis dont la latence est inférieure à $MaxLatence$, nous associons à chaque nœud n_i, un compteur $cptMessages_{n_i}$ du nombre de messages émis tant qu'il dispose d'un chemin de routage vers la station de base, constitué de noeuds vivants. Nous définissons également un compteur $cptAllMessages$ de tous les messages émis par tous les noeuds.

Le taux de messages émis dont la latence est inférieure à $MaxLatence$ est égal à $cptQuickMessages/cptAllMessages$ avec $cptQuickMessages = \sum_{n_i \in E} cptMessages_{n_i}$ $(E = \{n_i \in Depl \mid Lat_{n_i} \leq MaxLatence\})$

Vérification de la propriété relaxée. Pour vérifier la version relaxée de la propriété dans un état s_0, il suffit de vérifier que pour toute trace de durée Δ_t partant de s_0, le taux de messages émis sur la trace est inférieur à $MaxLatence$.

Le taux de messages émis dont la latence est inférieure à $MaxLatence$ pendant l'exécution d'une trace est égal au quotient entre le nombre de messages émis pendant l'exécution de la trace dont la latence est inférieure à $MaxLatence$ ($cptQuickMessages_{s_f} - cptQuickMessages_{s_0}$) par le nombre de messages émis pendant l'exécution de la trace ($cptAllMessages_{s_f} - cptAllMessages_{s_0}$)

5.6 Proportion des événements reçus par la station de base (P6)

Calcul de la métrique. Pour déterminer la proportion d'événements caractérisés par une requête $ReqId$ reçus par la station de base, nous ajoutons au modèle formel un compteur des événements $ReqId$ émis par n'importe quel nœud ($cptEmissionReqId$), ainsi qu'un compteur des événements $ReqId$ reçus par la station de base ($cptReceptionReqId$). Le quotient $Q = cptReceptionReqId/cptEmissionReqId$ correspond à la proportion d'événements reçus par la station de base.

Vérification de la propriété relaxée. Pour vérifier la version relaxée de la propriété dans un état s_0, il suffit de vérifier que, pour toute trace de Δ_t unités de temps partant de s_0, la proportion d'événements reçus est inférieure à τ.

La proportion d'événements $ReqId$ reçus sur une trace partant d'un état s_0 à un état s_f est le quotient entre le nombre d'événements $ReqId$ reçus pendant l'exécution de la trace ($cptReceptionReqId_{s_f}$-$cptReceptionReqId_{s_0}$) par le nombre d'événements $ReqId$ émis pendant l'exécution ($cptEmissionReqId_{s_f}$-$cptEmissionReqId_{s_0}$).

5.7 Détection des interblocages (P7)

Pour détecter les interblocages inattendus, c'est-à-dire, ceux qui se produisent alors qu'il existe au moins un noeud qui possède de l'énergie nous vérifions la validité de la formule CTL suivante :

$$EF(!EX(TRUE) \bigvee_{i \in Nodes} (energy(i) > Min_i)) \tag{4.1}$$

Cette formule signifie qu'il existe un état qui ne possède pas de successeur, où un des noeuds du réseau est encore vivant. C'est-à-dire qu'il existe un interblocage alors que le réseau a encore de l'énergie pour fonctionner.

- $EF(P)$ signifie qu'il existe un état dans le futur satisfaisant la proposition P.
- $EX(P)$ signifie qu'un des états successeurs satisfait la proposition P. $EX(TRUE)$ exprime donc simplement l'existence d'un état successeur.
- $Nodes$ est l'ensemble des noeuds du réseau.
- Min_i est l'énergie minimale en dessous de laquelle le noeud est considéré comme mort. Pour nous, un noeud est mort lorsqu'il ne peut plus communiquer (*i.e.*, Min_i est égal au coût de réception d'un message par i).
- $energy()$ associe à un noeud son énergie courante.

5.8 Conclusion

Nous avons présenté dans cette section, la procédure de vérification des propriétés définies dans le chapitre 3.

Le calcul de la durée de vie dans le pire des cas est la propriété la plus importante. Elle a été implémentée. Nous permettons également d'exhiber la trace qui conduit à la plus courte durée de vie du réseau. La propriété dérivée de la métrique du taux d'événements reçus, et la détection d'interblocages sont également supportées.

La vérification des versions relaxées est source d'explosion combinatoire. En effet, elle nécessite de vérifier pour chaque état de l'espace d'états que dans un horizon de

Δ_t unités de temps la propriété est satisfaite. Par contre, nous pouvons les vérifier pour un état donné.

Après avoir expliqué le processus de transformation et d'analyse sur lequel se base l'outil VeriSensor, nous présentons les résultats expérimentaux de la vérification que l'on a effectué avec VeriSensor sur le BAN.

6 Etude de cas

Dans cette section, nous présentons les résultats expérimentaux obtenus à l'aide de l'outil VeriSensor sur le BAN (voir section 3, chapitre 3). Le BAN est un réseau biomédical défini par Otto et al. [OMSJ05]. Notre étude de cas complète est disponible dans [BMKMTM13].

Les expérimentations ont été réalisées avec le prototype d'implémentation de l'outil VeriSensor [2]. Il est implémenté en Java. Il offre une interface en Xtext (voir section 14, chapitre 3). Le moteur de transformation, basé sur le méta-modèle de VeriSensor en EMF, construit une représentation ITS suivant les règles et abstractions définies dans les sections 3 et 4. Le moteur de transformation contient 3869 lignes de code. Ensuite, le modèle produit est analysé en utilisant le *model checker* ITS-Tools [TMBK+11].

Nous avons vérifié avec l'outil VeriSensor, les propriétés suivantes sur le BAN :

- (p_1) évaluer le nœud qui limite la durée de vie du système (*i.e.*, meurt le premier) dans le pire des cas.
- (p_2) identifier des scénarios conduisant à des situations indésirables qui doivent être évitées.
- (p_3) vérifier que le système fonctionne correctement en rejouant des situations existantes identifiées par les médecins
- (p_4) comparer des solutions matérielles alternatives suivant leurs caractéristiques (consommation d'énergie des capteurs, durée de traitement, etc...)

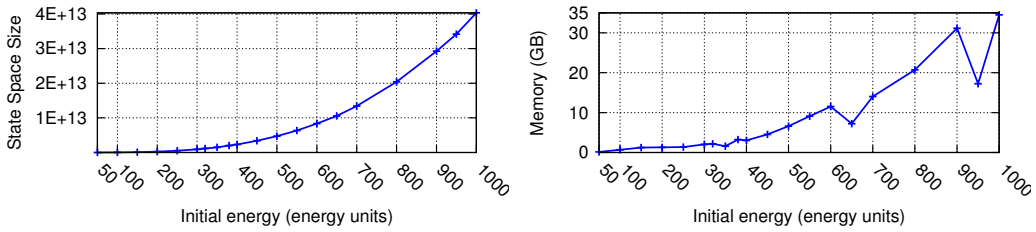
Dans la suite, nous évaluons d'abord les performances de l'outil VeriSensor en terme de consommation de mémoire et de temps : nous montrons sa capacité de calcul de la durée de vie pour des énergies initiales du système très grandes (*i.e.* montée en charge). Ensuite, nous présentons les résultats obtenus par VeriSensor pour chaque propriété.

6.1 Evaluation des performances de l'outil VeriSensor

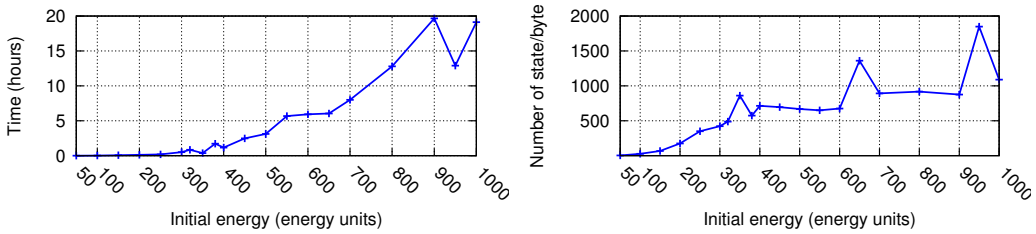
Les expériences ont été faites sur une machine dotée d'un processeur Xeon 64 bits de 2.6 GHz, d'une mémoire de 128 GB de RAM.

Le modèle du BAN est associé avec l'environnement libre qui sur approxime toutes les situations possibles que le BAN peut rencontrer, permettant l'analyse des scénarios du système, dans le pire et le meilleur des cas. La figure 4.27 montre l'évolution du nombre d'états du système, avec l'augmentation de l'énergie, ainsi que du temps et de la mémoire requise pour construire l'espace d'états. Les périodes et les coûts énergétiques sont fixés durant l'expérimentation : la période du capteur activité est de 3 et 1 unités de temps, respectivement, pour les noeuds activity1 et

2. disponible au public à l'adresse : http ://lip6.fr/Yann.Ben-Maissa (dépôt subversion)



(a) Nombre d'états (b) Mémoire utilisée



(c) Temps de calcul (d) Nombre d'états stockés par octet

FIGURE 4.27 – Evolution de la complexité de l'espace d'états quand l'énergie initiale allouée à chaque nœud augmente.

activity2. La période du capteur ECG est d'une unité de temps et celle du capteur Tilt est de 2 unités de temps. Une émission de message pour chaque nœud coûte 5 unités d'énergie pour tous les capteurs, et la réception de messages pour chaque nœud coûte 4 unités d'énergie. La capture coûte 2 unités d'énergie pour tous les capteurs et le traitement des données requiert 3 unités d'énergie pour tous les nœuds).

Comme nous pouvons le voir sur les graphiques, l'espace d'états croît rapidement : jusqu'à 4×10^{13} états. Sa représentation en mémoire, ainsi que son temps de calcul, évoluent de façon plus favorable (graphiques 4.27b et 4.27c) ce qui nous permet de valider le choix des ITS, basés sur les diagrammes de décision, qui ont prouvé leur efficacité pour de tels systèmes [TMPHK09a].

Les graphiques 4.27b et 4.27c montrent l'évolution de la mémoire et du temps nécessaires à la construction de l'espace d'états suivant l'énergie initiale allouée à chaque nœud. Nous pouvons augmenter l'énergie initiale jusqu'à 1000 unités (34.5 Giga octets et 19 heures). Des résultats utiles du point de vue de l'expert, ont été obtenus par [MSZ07] avec des énergies initiales inférieures (40 unités d'énergie au maximum, avec 3 unités par émission de message, 2 par réception de message, voir section 3.1.2, chapitre 2).

A notre connaissance, il s'agit du premier outil permettant de calculer la durée de vie dans le pire des cas d'un réseau de capteurs sans fil pour une énergie initiale de l'ordre du millier.

Notre expérience montre un bon potentiel de passage à l'échelle de notre approche globale. Cela est dû d'une part aux abstractions automatiques que l'on a définies (section 4), d'autre part aux techniques symboliques sous-jacentes (les diagrammes de décision [TMPHK09b]) au *model checker*. Dans le cas du BAN, les abstractions ont un impact très significatif. Elles nous permettent de diviser les tailles des domaines

des grandeurs HeartBeat, Tilt et Activity par 201, 181 et 8, respectivement. En effet, sachant que HeartBeat et Tilt n'interviennent que dans des requêtes non conditionnelles, il suffit de retenir une valeur symbolique pour ces valeurs (on divise les tailles des domaines par leur taille respective). La grandeur activity intervient dans deux requêtes conditionnelles. Par conséquent, on retient 2 valeurs symboliques (*i.e.*, événement, pas d'événement) ce qui correspond à diviser la taille du domaine de la grandeur (16) par 8.

Par ailleurs, nous modélisons de façon plus détaillée que dans [MSZ07] le comportement de l'application et des capteurs, quitte à abstraire un peu plus le comportement des protocoles réseau.

En outre, nous sommes les seuls à prendre en compte la consommation d'énergie de tous les aspects du réseau (*i.e.* capture, traitement, communications) dans notre calcul de la durée de vie dans le pire des cas.

D'un point de vue industriel, il devient faisable de traiter de grandes valeurs sur les serveurs actuels. Nos techniques fournissent plus de données sur le comportement en régime permanent pour l'étude de cas du BAN. La figure 4.27d montre que l'on peut représenter jusqu'à 1850 états par octet. Comme toujours, avec les techniques symboliques, des variations de performances sont observées, qui ne sont pas directement reliées à la croissance de la taille de l'espace d'états. Cette expérience montre la faisabilité de la détection « d'événements rares » (ce qui est difficile avec la simulation uniquement) à l'aide de propriétés d'accessibilité (*e.g.*, p_2). Cependant, si uniquement une réponse Oui/Non pour les propriétés d'accessibilités est fournie dans un temps raisonnable, nous avons mesuré que le calcul d'un contre-exemple demande significativement plus de temps et de mémoire.

6.2 Détermination du nœud limitant la durée de vie (p_1)

Exhiber la consommation d'énergie du réseau de capteurs sans fil dans le pire des cas permet à l'expert d'évaluer une borne inférieure de la durée de vie du système. La figure 4.28 montre l'évolution de la durée de vie dans le pire des cas des nœuds du BAN.

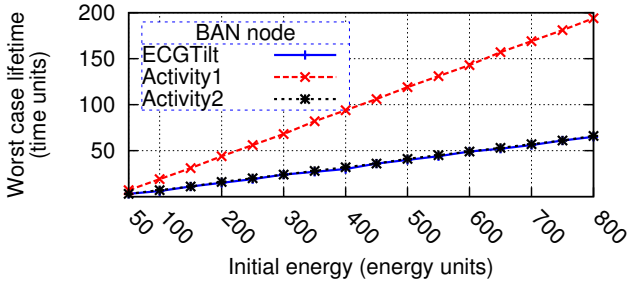


FIGURE 4.28 – Calcul de la durée de vie dans le pire des cas des noeuds du BAN

Pour obtenir cette courbe, nous exécutons la tâche de vérification en figure 4.29. Cette tâche trace le graphique de la durée de vie dans le pire des cas du réseau pour n'importe quel environnement (*i.e.*, l'environnement libre désigné avec le mot clef « Free ») pour différentes unités d'énergie initiale. Il s'agit en principe de la courbe

de ECGTilt (le premier nœud à mourir). Nous faisons également apparaître sur le graphique les durées de vie de Activity1 et Activity2.

```
check plotWorstCaseLifetime {
        Plot WorstCaseLifetime
        with parameterVariations ={
   INIT_ENERGY_HEART = INIT_ENERGY_ACTIVITY1 = INIT_ENERGY_ACTIVITY2 = 50 .. 800 Step 50
  }
} ForEnvScenarios Free
```

FIGURE 4.29 – Calcul de la durée de vie dans le pire des cas

Pour construire ce graphique, nous associons le modèle du BAN à un environnement libre qui autorise n'importe quelle action. Cet environnement consiste en un ensemble de transitions représentant toutes les valeurs possibles des grandeurs physiques, où toutes les transitions sont tout le temps franchissables. Par conséquent, lorsqu'un capteur fait l'acquisition d'un échantillon de l'environnement, il peut théoriquement faire l'acquisition de n'importe quelle valeur de la grandeur.

Nous construisons ensuite un sur-ensemble de tous les comportements possibles, à partir duquel nous obtenons le scénario dans le pire des cas, avant de calculer S_{end}, l'ensemble des états où au moins un nœud ne peut plus communiquer (son énergie est inférieure à Min, la quantité d'énergie minimale pour envoyer un message). La durée de vie est obtenue en comptabilisant le nombre minimal d'occurrences de la transition *elapse* localisée dans la plus petite séquence de transitions conduisant à un état de S_{end} à partir d'un état initial.

Le but n'est pas de fournir l'information quantitative sachant que le nombre initial d'unités d'énergies allouées aux nœuds n'est pas suffisant (La figure 4.28 montre la durée de vie du système en heures [3], alors qu'il faut au moins, plusieurs semaines). Cependant, l'expert peut avoir une idée du composant le plus critique (*i.e.*, celui qui tombe en panne le premier) suivant les divers scénarios. Ce résultat est complémentaire de la simulation qui peut traiter des durées plus longues mais pas de manière exhaustive. Notre vérification exhaustive exhibe le scénario dans le pire des cas, que le système peut avoir. Le calcul de traces minimales est plus coûteux que l'accessibilité, mais il est toujours possible pour les grandes valeurs : pour Activity1 avec 500 unités d'énergie, cela prend moins que 4 heures pour calculer la trace minimale (447 transitions) avec une mémoire de 6.7 Giga octets.

La Figure 4.28 montre que ECGTilt et Activity2 sont les nœuds limitants du système (*i.e.*, ceux qui manquent d'énergie en premier). La mort prématurée de ECGTtilt est problématique parce qu'il surveille un signe vital critique (les battements du coeur). L'expert réseau pourrait augmenter sa durée de vie en fournissant une batterie de capacité plus importante (réduire la période de capture du rythme cardiaque n'est pas une option car nous pouvons ne pas détecter des changements rapides du rythme). La durée de vie faible du noeud Activity2 est moins problématique parce qu'il surveille un paramètre moins critique et qu'il est redondant avec Activity1. Nous montrons dans ce qui suit comment deux conceptions alternatives de Activity peuvent être explorées.

3. Nous n'avons pas fixé une durée exacte pour les unités de temps, mais, considérant nos abstractions, cela doit être en minutes

6.3 Détection les interblocages (p_2)

Avant d'évaluer les propriétés de durée de vie sur le modèle de réseau de capteurs sans fil, nous utilisons un modèle d'énergie infinie et supprimons les compteurs (*e.g.*, le nombre de messages reçus par la station de base) et vérifions que l'espace des états est fini (4.03×10^7 états, calculés en 2.8 secondes, avec 17.1 Mega octets de mémoire). Ce modèle permet d'étudier le comportement en régime permanent (*i.e.*, les traces infinies du système). Prendre en compte la consommation d'énergie pour calculer les durées de vie crée un modèle plus complexe, mais qui n'est pas convenable pour tous les types de propriétés.

Une propriété d'accessibilité typique et intéressante consiste à détecter les interblocages inattendus dans le système (les interblocages attendus étant ceux où tous les nœuds ont consommé leur énergie). Une telle situation peut être détectée avec le tâche de vérification de la figure 4.30. Cette tâche de vérification ordonne au model checker de vérifier l'accessibilité d'états qui n'ont pas de successeurs et où au moins un des noeuds du réseau a encore de l'énergie.

```
check DeadLock {
        Verify Deadlock
        with parameterVariations = {
   INIT_ENERGY_HEART = INIT_ENERGY_ACTIVITY1 = INIT_ENERGY_ACTIVITY2 = 300
  }
} ForEnvScenarios Free
```

FIGURE 4.30 – Vérification de l'absence d'interblocages

La tâche de vérification de la figure 4.30 a été vérifiée sur le BAN avec une configuration initiale de 300 unités d'énergie. Notre prototype a calculé en 44 minutes avec 1.3 GB de mémoire que tous les interblocages correspondent au manque d'énergie du système.

6.4 Vérification des comportements du système dans des situations existantes (p_3)

Pour le BAN, une propriété type consiste à s'assurer que le système ne génère pas de faux négatifs pour des scénarios typiques. Nous définissons un faux négatif comment un événement détecté par un noeud mais qui n'a pas été reçu par la station de base. La tâche de vérification 4.31 s'assure que le taux de détection des événements caractérisés par les requêtes AskActivity1 et AskActivity2 (*i.e.*, les efforts physiques) est égal à 1, pour tout intervalle de durée *infinity* (le mot clef *infinity* désigne la durée totale de l'exécution). Cela est équivalent à vérifier l'absence de faux négatifs.

Un tel calcul est moins complexe en temps et en mémoire que le calcul de la durée de vie dans le pire des cas, sachant que le système est plus contraint. Avec une énergie initiale de 300 unités par nœud, la formule 2 a été calculée en 38 minutes avec 2.4 Giga octets. La formule 3 a été calculée en 84 minutes avec 2.6 Giga octets.

```
check falseNegatives {
        Verify (EventDetectionRatio(AskActivity1,infinity)=1 AND
         EventDetectionRatio(AskActivity2,infinity)=1)
        with parameterVariations = {
   INIT_ENERGY_HEART = INIT_ENERGY_ACTIVITY1 = INIT_ENERGY_ACTIVITY2 = 300
  }
} ForEnvScenarios BikeExercising, Resting
```

FIGURE 4.31 – Vérification de l'absence de faux négatifs

6.5 Comparaison de solutions alternatives (p_4)

Le choix d'un composant donné peut avoir un impact sur la durée de vie du réseau de capteurs sans fil, ou sur une caractéristique importante du BAN. VeriSensor peut être utile pour comparer deux solutions possibles. Pour cela, l'expert peut soit changer les caractéristiques des nœuds à remplacer ou remplacer le nœud par une instance d'une autre classe de nœuds.

Paramètre	**valeur de la configuration *config*$_1$**	**valeur de la configuration *config*$_2$**
fréquence de capture	3 UT	4 UT
durée de capture	0 UT	1 UT
coût de capture	2 UE	3 UE
durée de traitement	0 UT	1 UT
coût de traitement	3 UE	4 UE
durée d'émission	1 UT	2 UT
coût d'émission	5 UE	6 UE
coût de réception	4 UE	5 UE

FIGURE 4.32 – Données pour les deux configurations de Activity1 étudiées, en unités de temps (UT) et d'énergie (UE)

Pour le BAN, nous évaluons l'impact de deux configurations, pour Activity1 sur la durée de vie du système (*e.g.*, quand au moins un nœud ne peut plus communiquer) Toutes les mesures précédentes ont été faites en utilisant la configuration $config_1$ présentée dans la sous-section 6.1. Nous choisissons une autre configuration pour Activity1, $config_2$, dont les caractéristiques sont aussi fournies en figure 4.32. Sachant que ces configurations sont proches l'une de l'autre, il est difficile de prédire laquelle fournit une durée de vie plus longue.

La comparaison des deux configurations est présentée dans le graphique de la figure 4.33. Elle montre que la solution alternative augmente la durée de vie du système.

7 Conclusion

Dans ce chapitre, nous avons présenté en section 2 les langages formels sous-jacents de notre outil VeriSensor. Il s'agit des *Instantiable Transition Systems* et des réseaux de Petri temporels. Les réseaux de Petri temporels sont un langage de modélisation mathématique adapté à la description des systèmes concurrents

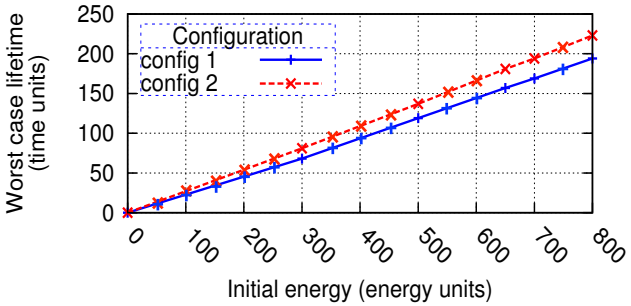


FIGURE 4.33 – Comparaison des durées de vie dans le pire des cas pour activity1

et asynchrones, ce qui est notre cas. Les ITS nous permettent de structurer la spécification formelle de manière hiérarchique et compositionnelle à l'aide de la notion de type et d'instance.

En section 3, nous avons présenté le processus de transformation d'une instance de spécification VeriSensor en un modèle formel. Ce processus se base sur 11 règles. Un premier type de règles dicte comment construire les dimensions d'un nœud (environnements locaux, capteurs, dispatcher et requêtes, réseau, énergie). Le routage est géré par un composant dédié, pour plus de modularité. Un deuxième type de règles permet d'assembler les dimensions pour former un nœud et de connecter les nœuds entre eux en fonction de la topologie pour former un réseau de capteurs sans fil complet.

Nous avons également expliqué en section 4 la nécessité de définir des abstractions pour réduire le risque d'explosion combinatoire. En effet, la taille du modèle formel et de l'espace d'états résultant de l'application « naïve » des règles de transformation est potentiellement grande notamment à cause de la taille des domaines des grandeurs physique. Nous proposons des abstractions qui consistent essentiellement à réduire les domaines de ces grandeurs à des domaines symboliques en analysant les structures de contrôle qui impactent le comportement du système.

En section 5, nous avons expliqué que la vérification des propriétés de VeriSensor se fait, éventuellement en ajoutant des compteurs au modèle formel, mais surtout en analysant l'espace d'états (recherche d'états satisfaisant un prédicat particulier, calcul du temps écoulé sur les traces).

Finalement, en section 6, nous avons validé notre outil VeriSensor en vérifiant un ensemble de propriétés sur le réseau biomédical proposé par Otto et al. [OMSJ05]. VeriSensor a permis de calculer les durées de vie dans le pire des cas des noeuds du système, d'identifier l'existence de deadlocks, de faux négatifs et de choisir entre deux configurations matérielles de noeud. L'outil permet de monter en charge (jusqu'à 800 unités d'énergie pour la trace) et montre un bon potentiel de passage à l'échelle en partie grâce aux abstractions automatiques définies.

CONCLUSION GÉNÉRALE ET PERSPECTIVES

CHAPITRE 5

Dans cette thèse , nous avons présenté notre proposition pour augmenter la fiabilité des réseaux de capteurs sans fil à l'aide d'une approche de développement dirigée par les modèles, associée à de la vérification par *model checking*.

1 Contributions

Notre proposition est constituée des contributions suivantes :

- **VeriSensor**, un DSML dédié aux réseaux de capteurs sans fil, supportant la vérification par *model checking*. Il fournit des concepts adaptés à un expert du domaine pour modéliser son système (**OBJ$_1$**).

  VeriSensor permet de construire une description hiérarchique et compositionnelle des différents aspects d'un réseau de capteurs sans fil (capture, application, réseau, énergie) et supporte la modélisation de l'hétérogénéité (*i.e.*, des types de nœuds différents).

  En effet, une spécification VeriSensor sépare clairement la description de l'environnement, du réseau de capteurs sans fil et des propriétés à vérifier.

  La spécification du réseau de capteurs sans fil est composée d'un ensemble de modèles qui ont des préoccupations différentes : les requêtes, le déploiement (souvent négligé dans les DSMLs existants), les classes de nœuds.

  Le modèle de classes de nœuds permet de regrouper les caractéristiques communes aux nœuds d'un réseau de capteurs sans fil et rend possible le support de l'hétérogénéité dans ces réseaux. Il fait lui-même référence à 3 sous modèles décrivant les différents aspects d'un nœud : la capture, l'application et le réseau.

  Le couplage de VeriSensor et du *model checking* est efficace. En effet, l'architecture de VeriSensor, et en particulier, le regroupement des informations communes dans des classes de nœuds, permet d'exploiter les symétries auxquelles est sensible le *model checker*.

- Un **moteur de transformation** d'une spécification VeriSensor vers la spécification formelle cible, ainsi qu'un **module de traduction** des résultats de la vérification pour l'utilisateur.

  Le moteur de transformation génère un modèle formel sur lequel il est possible de vérifier par *model checking* la satisfaction des propriétés spécifiées par l'expert (**OBJ$_2$**).

Le moteur de transformation produit une spécification exprimée dans le langage formel des *Instantiable Transition Systems* (ITS) [TMPHK09b] et en Réseaux de Petri Temporels, à des fins de vérification. Les ITS/Réseaux de Petri Temporels nous permettent de décrire la spécification formelle d'une manière compositionnelle et hiérarchique, tout en fournissant un support naturel pour la modélisation de la concurrence et du temps. Ils sont supportés par un *model checker* [TMPHK09a], utilisant une librairie de diagrammes de décisions pour gérer la complexité des systèmes de grande taille, qui nous permet de vérifier les propriétés.

Le moteur de transformation est basé sur 11 règles de transformations qui traduisent la spécification VeriSensor en un ensemble d'ITS/réseaux de Petri temporels représentant les 4 dimensions d'un noeud (capture, application, réseau et énergie), puis de combiner ces dimensions en un ITS représentant le noeud. Les noeuds sont combinés pour construire la spécification formelle complète du réseau de capteurs sans fil. Le découplage, entre les différents aspects, augmente la *réutilisabilité* du modèle formel.

Nous proposons un ensemble d'abstractions automatiques qui permettent de réduire considérablement le risque de l'explosion combinatoire. Ces abstractions réduisent la dépendance de la taille de l'espace d'état par rapport à celle des domaines des grandeurs physiques captées par le système en considérant uniquement les parties de ces domaines qui ont un impact sur le comportement du système. Ces abstractions permettent, entre autre, de calculer la durée de vie des réseaux de capteurs sans fil avec une énergie initiale beaucoup plus importante que dans les travaux existants (*e.g.*, [MSZ07]).

Le module de traduction garantit la traçabilité des résultats de vérification pour l'expert (**OBJ$_3$**).

Il lui retourne les résultats de la vérification sous un format lisible par l'homme. Dans le cas d'une propriété à vérifier, nous indiquons à l'utilisateur le verdict (oui/non). Dans le cas d'une métrique à tracer ou à optimiser, nous fournissons le graphique de la métrique ou sa valeur optimale, ainsi que les valeurs des paramètres.

- Un **langage de spécification de propriétés**. Ce langage permet à l'expert de spécifier les propriétés à vérifier sur le réseau de capteurs sans fil (**OBJ$_1$**).

  Les propriétés supportées sont qualitatives et quantitatives. La propriété la plus importante, à savoir, la durée de vie du réseau de capteurs sans fil, est supportée.

  Le langage de spécification de propriétés permet à l'utilisateur de définir un ensemble de tâches de vérification qui réalisent une action (optimiser une métrique, tracer son évolution, ou vérifier une propriété) sur un ensemble d'instances du modèle de réseau de capteurs sans fil. Les propriétés exprimables sont construites à partir de métriques, extraites des définitions de la durée de vie d'un réseau de capteurs sans fil les plus importantes de la littérature.

L'outil VeriSensor permet donc de supporter la *chaîne complète* de la spécification du réseau de capteurs sans fil à la production des résultats de la vérification sous une forme lisible et agrégée à l'expert réseau.

2 Perspectives

Ce travail ouvre différentes perspectives à court, moyen et long terme.

A court terme. Nous souhaitons que VeriSensor supporte une plus grosse partie du *cycle de développement* d'un réseau de capteurs sans fil (pas uniquement la modélisation). Pour cela, nous voulons permettre à l'utilisateur de simuler un modèle VeriSensor et de générer du code à partir de ce modèle.

Le support de la simulation comme approche complémentaire a été notamment suggérée par les relecteurs de nos articles. Notre effort de recherche sera, sans doute, dirigé dans ce sens, d'autant plus que VeriSensor a une sémantique formelle, et qu'il est donc exécutable et peut être simulé. Nous réfléchissons, également, toujours dans la philosophie IDM, à définir des transformations de VeriSensor vers des simulateurs réseau existants (*e.g.* NS-2 [1], TOSSIM [LLWC03])

L'intérêt de la simulation est double. D'une part, elle permet d'étudier l'évolution du réseau dans des *scénarios typiques* auxquels le système peut être soumis (contrairement au scénario du pire des cas). D'autre part, elle fournit la possibilité d'étudier le comportement de *systèmes de plus grande taille* (plus de noeuds).

Concernant le premier intérêt, les scénarios types seraient extraits à partir de traces réelles d'évolution d'une grandeur (*e.g.*, l'évolution des battements cardiaques mesurée sur un patient, pendant une durée donnée).

Concernant le deuxième intérêt, la simulation n'explore pas exhaustivement l'espace d'états et n'est donc pas confrontée au problème d'explosion combinatoire. Nous pouvons donc simuler des réseaux de capteurs sans fil de plus grande taille et, en explorant un sous ensemble des états du système, révéler des erreurs dans ce système.

La génération de code a également été proposée comme piste par les relecteurs de nos articles. En effet, une extension naturelle de notre travail consiste à permettre à l'utilisateur de produire une implémentation (*e.g.*, en *nesC*) en définissant des transformations de VeriSensor vers les *frameworks* existants dédiés à la génération de code comme MEDWSA [VCLÁ+07] et Baobab [ADBS09].

A moyen terme. Nous voulons *optimiser* le *model checking* de la spécification formelle.

A propos des optimisations sur le modèle formel, nous avons remarqué que malgré le fait que nos expérimentations montrent la faisabilité de l'analyse exhaustive sur une spécification VeriSensor, le passage à l'échelle est encore laborieux. En effet, à partir de 7 noeuds, le *model checking* prend plusieurs jours avec une machine Xeon 64 bits de 2.6 GHz avec 128 GB de RAM, et un seul processeur. Il faut donc définir des abstractions supplémentaires.

Par ailleurs, il est possible de réduire le temps de vérification en modifiant la représentation interne du modèle formel ITS (*i.e.*, les diagrammes de décisions). Les diagrammes de décisions sont une structure de données qui permet de représenter le modèle sous une forme compressée. Pour un même modèle formel en ITS/RDP temporels, le temps de vérification peut varier exponentiellement en fonction de

1. http ://www.isi.edu/nsnam/ns/

l'ordre des variables dans le diagramme de décision qui le représente. Nous proposons de définir des heuristiques qui modifient cet ordre de telle sorte à réduire le temps de vérification.

A long terme. Nous souhaitons, d'une part, que VeriSensor supporte la génération native de code (sans passer par des outils de génération de code existants [VCLÁ+07, ADBS09]), d'autre part, tester notre outil dans le *contexte industriel*.

Concernant la génération native de code, il s'agit de définir une transformation de VeriSensor vers un langage de programmation dédié aux réseaux de capteurs sans fil (*e.g.*, nesC). L'intérêt, est d'être indépendant des outils de génération existants. En effet, dans le cas de la génération non native, un changement du langage d'entrée du générateur impose de modifier la transformation de VeriSensor vers ce langage d'entrée. Par ailleurs, VeriSensor est défini à l'aide du *framework* Xtext qui supporte la définition de générateurs et propose même Xtend [2], un langage dédié à cela.

Il serait, également, intéressant de voir l'apport de VeriSensor à l'expert lors de la conception industrielle d'un réseau de capteurs sans fil. En effet, les freins principaux à l'adoption des méthodes formelles dans l'industrie sont la courbe d'apprentissage difficile (modèle mathématique sous-jacent à ces méthodes) et le problème du passage à l'échelle (explosion combinatoire de l'espace des états). Nous pensons avoir apporté une partie de la réponse à ces deux freins. Par ailleurs, rien ne vaut l'utilisation répétée de VeriSensor dans le milieu industriel, pas même le plus élégant des résultats théoriques.

2. Xtend est un langage généraliste à typage statique focalisé sur la génération de code et faisant partie du *framework* Xtext. Adresse internet : www.eclipse.org/xtend/

ANNEXE

CHAPITRE 6

1 Méta-modèles de VeriSensor

Dans cette section, nous présentons les méta-modèles des modèles VeriSensor, à savoir : le modèle de déploiement, de l'environnement, de requêtes, de classe de noeud, de types de données, de définitions de paramètres, de propriétés et des sous-modèles VeriSensor (capture, application, réseau).

1.1 Modèle de déploiement

La Figure 6.1 décrit le méta-modèle du modèle de déploiement. Le modèle de déploiement (méta-classe `DeplModel`) contient un système (méta-classe `System`) qui représente le déploiement du réseau. Il est constitué d'une liste d'instanciations de classe de noeud. Chaque instanciation (méta-classe `ANodeClassInstantiation`) référence une classe de nœuds et contient les instances de cette classe (méta-classe `ANodeInstance`). Une instance est décrite par son identifiant, sa position (coordonnées x,y,z), et son *next-hop* (*i.e.*, une référence à l'instance à laquelle elle retransmet tous ses messages). Un nœud peut ne pas avoir de *next-hop* (*e.g.*, la station de base).

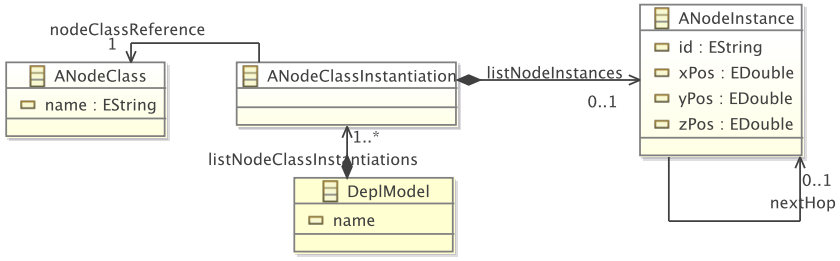


FIGURE 6.1 – Méta-modèle du déploiement

1.2 Modèle de l'environnement

Le modèle de l'environnement définit une liste de scénarios environnementaux (Méta-classe `EnvironmentalScenario`, Figure 6.2) possédant chacun un identifiant. Un scénario environnemental spécifie l'évolution (Méta-classe `QuantityEvolution`) dans l'espace et le temps des grandeurs physiques captées par le système. La

définition d'une évolution de grandeur référence le type associé à la grandeur physique et le fichier CSV (méta-classe `DataFile`) qui décrit cette évolution.

Un scénario environnemental peut être cyclique ou non cyclique (attribut `cyclicity` de la métaclasse `EnvironmentalScenario`).

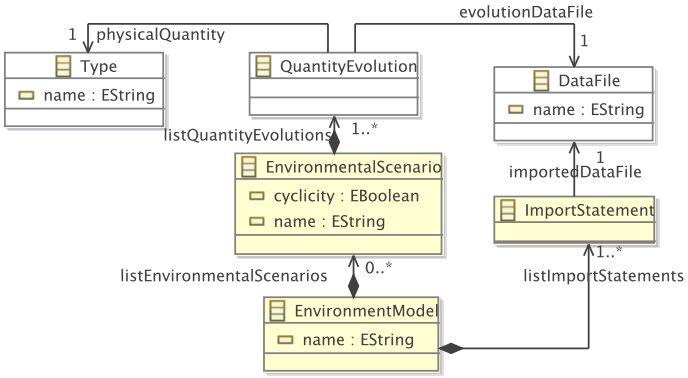


FIGURE 6.2 – Méta-modèle pour l'environnement

1.3 Modèle de requêtes

Le corps d'une requête non conditionnelle (méta-classe `NonCondQueryBody` de la figure 6.3) définit une liste de grandeurs a capter, une période de capture (attribut `sensPer`), une période de traitement (attribut `processPer`), un coût de traitement (attribut hérité `processCost`), une période d'envoi (attribut `sendPer`), une durée totale (attribut `dur`).

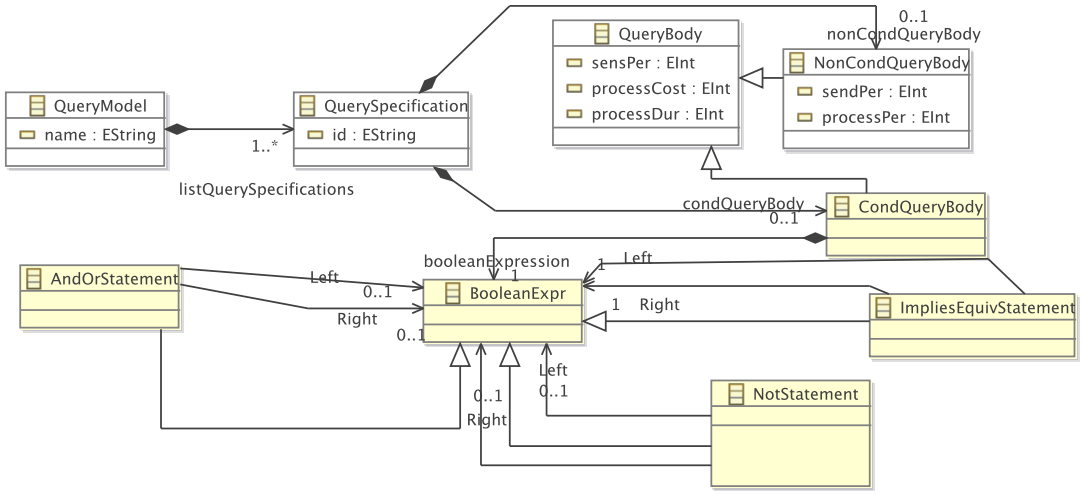


FIGURE 6.3 – Méta-modèle pour les requêtes

Le corps d'une requête conditionnelle (méta-classe `CondQueryBody`) définit une condition booléenne, une période de capture (attribut hérité `sensPer`), un coût de

traitement (attribut hérité `processCost`) et une durée totale. La condition est une expression booléenne sur des comparaisons entre une grandeur physique et une valeur.

1.4 Classe de nœuds

Une classe de nœuds (Méta-classe `ANodeClass`, Figure 6.4) fait référence aux fichiers contenant les descriptions des sous-modèles réseau (Méta-classe `NetworkModel`), capture (Méta-classe `SensingModel`), application (Méta-classe `ApplicationModel`). L'énergie initiale peut contenir une valeur (finie, ou infinie dans le cas de la station de base) ou référencer un `Paramètre` destinée à être réutilisé dans les *checks*).

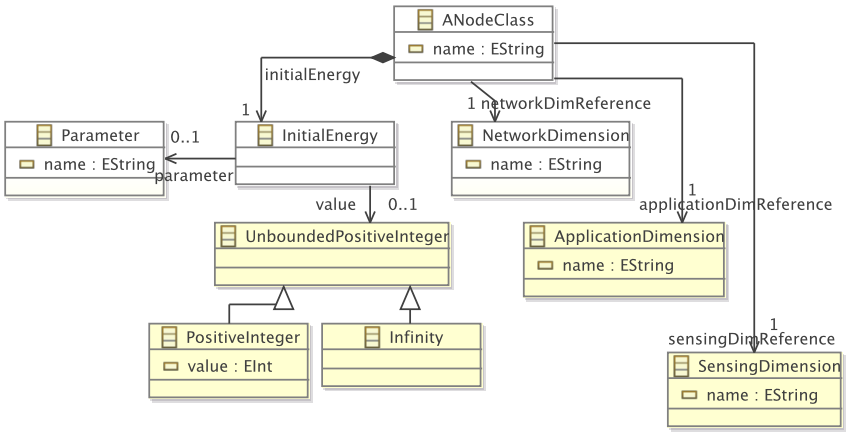


FIGURE 6.4 – Méta-modèle pour la classe de nœud

1.5 Sous-modèle capture

Le sous-modèle capture définit pour chaque capteur (métaclasse `ASensor`, voir Figure 6.5) les caractéristiques suivantes : nom, grandeur captée, *startup time* (`StartupTime`), coût d'une capture (`sensCost`), durée de capture (`sensDur`).

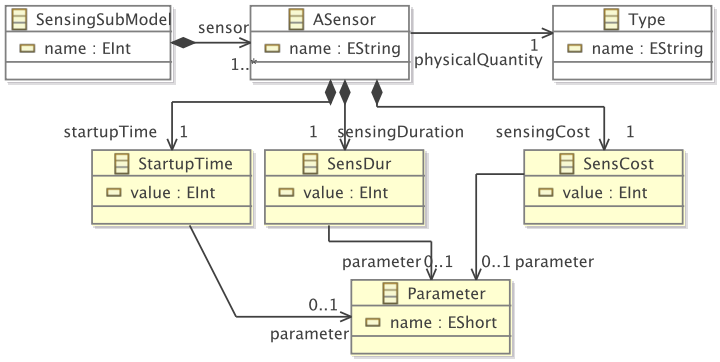


FIGURE 6.5 – Méta-modèle pour le sous-modèle capture

Nous supportons donc la définition de *plusieurs types de capteurs et de plusieurs capteurs* par classe de nœud.

Chacun des attributs du capteur peut contenir une valeur ou faire référence à un paramètre (référence à la méta-classe `Paramètre`) déclarée au début de la spécification.

1.6 Sous-modèle application

Nous avons deux types de sous-modèles application qu'on distingue en paramétrant le sous-modèle application par le mot clef `NODE` ou `BASE` (voir l'énumération `APPLI_TYPE` de la Figure 6.6).

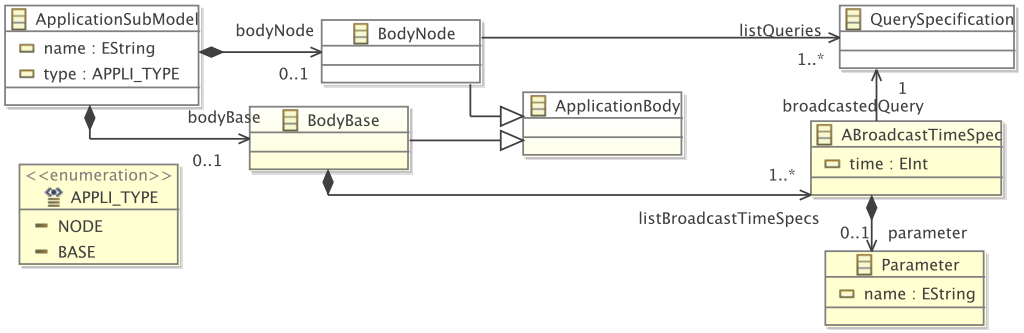


FIGURE 6.6 – Méta-modèle pour le sous-modèle application

Si le sous-modèle application est paramétrée par `NODE`, on y spécifie les identifiants (références `listQueries` aux spécifications `QuerySpecification`) des requêtes dont les instances de la classe référençant le sous-modèle application sont destinataires. En VeriSensor, au lieu qu'une requête spécifie ses nœuds destinataires, ce sont les nœuds (plus précisément les classes de ces nœuds) qui spécifient les requêtes dont ils sont destinataires. En effet, il est logique de supposer que les instances d'une même classe traitent les mêmes requêtes puisque par définition ces instances ont le même comportement applicatif.

Si le sous-modèle application est paramétrée par `BASE`, on y spécifie une liste d'instants de diffusion (méta-classe `ABroadcastTimeSpec`) associés à des requêtes (référence à la méta-classe `QuerySpecification`).

1.7 Sous-modèle réseau

Le coût de réception (voir métaclasse `ReceptionCost`, Figure 6.7) est l'énergie consommée pour recevoir un message. Le coût d'émission (méta-classe `EmissionCost`) est l'énergie consommée pour émettre un message. La durée d'émission (méta-classe `emissionDuration`) est la durée nécessaire pour émettre un message au destinataire. La portée de communication (méta-classe `range`) combinée avec les positions des nœuds spécifiées dans le déploiement, permet de déterminer quels sont nœuds à la portée (*i.e.*, reçoivent une requête rediffusée par un nœud donné).

Chaque attribut du sous-modèle réseau peut contenir une valeur ou faire référence à un paramètre déclaré au début de la spécification.

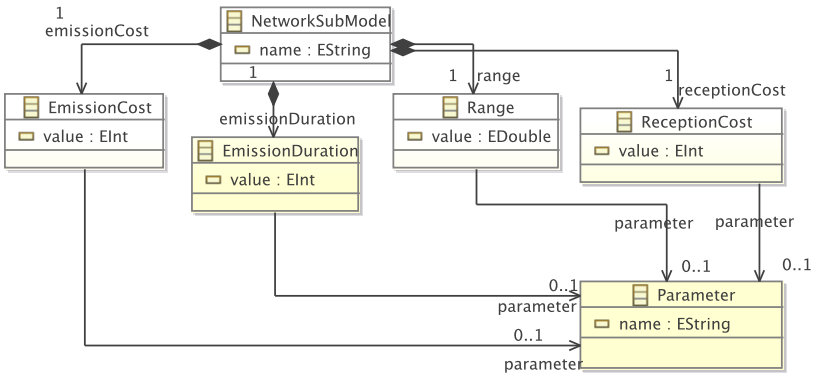


FIGURE 6.7 – Méta-modèle pour le sous-modèle réseau

1.8 Modèle des types de données

La définition des types de données est contenue dans un fichier (voir méta-classe `TypeDefFile`, Figure 6.8). Un type peut être défini par une étendue de valeurs (méta-classe `IntegerRange`, `DoubleRange`), une énumération (méta-classe `Enumeration`) ou un tuple (métaclasse `Record`) constitué de champs de différents types.

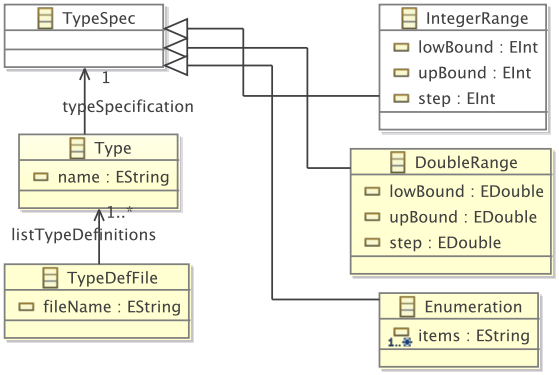


FIGURE 6.8 – Méta-modèle pour la définition des types de données

1.9 Définition de paramètres

L'utilisateur définit un ensemble de paramètres qui permettent de spécifier les domaines de variation des attributs du modèle dans chaque tâche de vérification. Chaque paramètre est affecté à un attribut de la spécification et peut être utilisé dans une tâche de vérification pour définir le domaine de variation de l'attribut correspondant.

Un paramètre fait référence à un type et son domaine de variation est constitué d'une seule valeur (métaclasses `IntValue`, `DoubleValue`, Figure 6.9), d'une étendue de valeurs (métaclasses `IntRange`, `DoubleRange`) ou d'une énumération (métaclasse `Enumeration`).

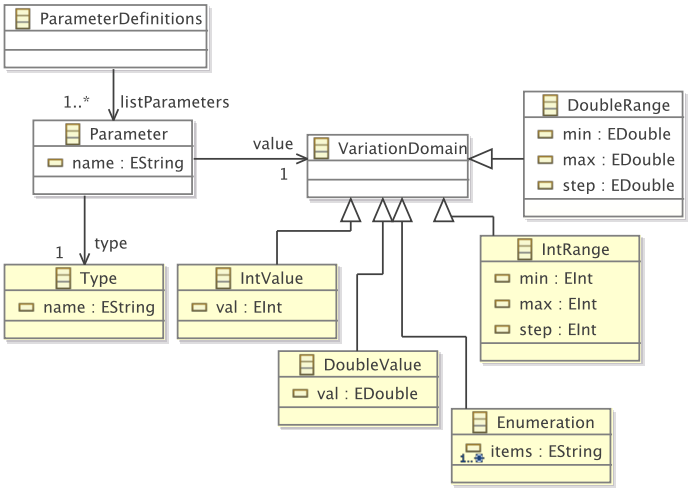


FIGURE 6.9 – Méta-modèle pour la définition des paramètres

1.10 Modèle des propriétés à vérifier

Un *check* effectue une action (tracer un graphique ou vérifier une propriété) sur un argument (une métrique représentée par la métaclasse `Metric` ou une propriété à vérifier représentée par la métaclasse `ImplyEquivStatement`) pour une ou plusieurs valeurs d'un sous ensemble des attributs de la spécification, sur un sous ensemble des scénarios environnementaux (métaclasse `EnvironmentalScenario`) définis dans le modèle d'environnement.

Une métrique peut référencer une grandeur physique, ou une requête (*e.g.*, `EventDeliveryRatio(AskHeartBeat)`), ou contenir une durée. Par exemple, `EventDeliveryRatio(AskHeartBeat) in DeltaT 10 = 1` signifie que tout évènement lié à la requête `AskHeartBeat` doit être détecté avant un délai de 10 unités de temps.

Dans la Figure 6.10, on indique qu'une propriété est constituée de propositions atomiques qui sont des comparaisons entre une métrique et une valeur.

Ces propositions atomiques peuvent être combinées avec (par ordre de précédence) les connecteurs : not, and/or/xor, $\Longrightarrow$, $\Longleftrightarrow$.

Dans chaque *check*, soit on indique le domaine de variation du paramètre (métaclasse `ParameterAssignment` représentant l'affectation d'un intervalle ou d'une valeur à un paramètre) : c'est celui de l'attribut. Soit on n'indique rien, auquel cas le domaine de variation de l'attribut est celui qui a été spécifié lors de la définition des paramètres au début du modèle.

Plusieurs paramètres peuvent appartenir au même `ParameterAssignment` ce qui signifie que leur évolution est liée (*e.g.*, `INIT_ENERGY_HEART = INIT_ENERGY_ACTIVITY1 = INIT_ENERGY_ACTIVITY2 = 250`).

Un *check* consistant à faire un tracé de graphe peut spécifier au maximum deux `ParameterAssignment` (plot en 3 dimensions).

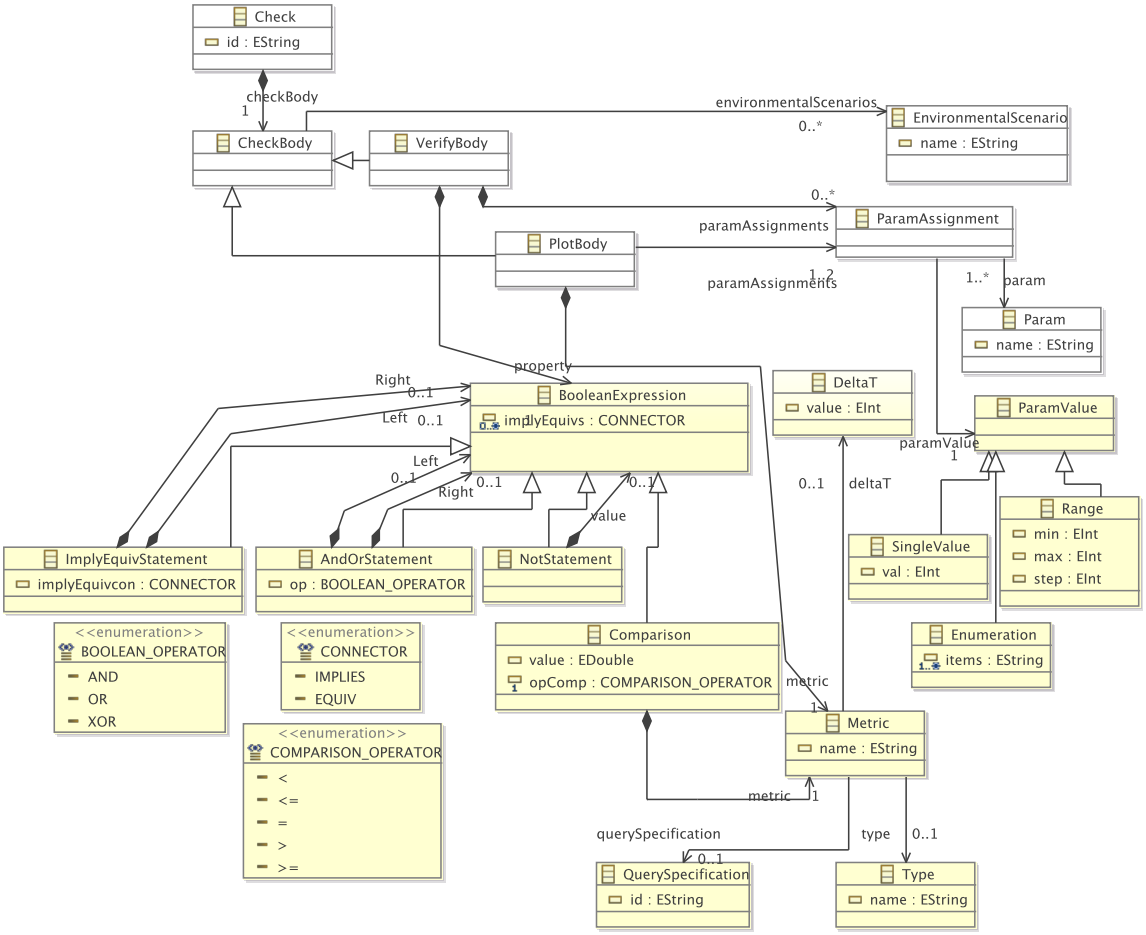


FIGURE 6.10 – Méta-modèle pour la spécification des propriétés

2 Grammaire de VeriSensor

Dans cette section, nous présentons la grammaire Xtext de VeriSensor à partir de laquelle les méta-modèles présentés en section 1 ont été construits.

2.1 Structure de la spécification VeriSensor

La figure 6.11 présente la grammaire de la structure d'une spécification VeriSensor. Elle est constituée de déclarations de paramètres, d'un modèle de déploiement, de requêtes, d'un modèle environnement, d'un ensemble de classes de noeuds, d'un ensemble de sous-modèles captures, application, et réseau, de définitions de types de données et de tâches de vérification (*check*).

```
grammar fr.lip6.move.verisensor.Verisensor with org.eclipse.xtext.common.Terminals

generate verisensor "http://www.lip6.fr/move/verisensor/Verisensor"

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

//extension .verisensor


ASpecification :
        parameterDeclaration=ParametersDeclaration theDeployment=DeplModel
        (theQueries=QueryModel) (environment=Environment)
        (nodeClasses+=ANodeClass)+(typeDefinitions=TypeDefModel)
        (properties=PropertyModel)
                ;


ParametersDeclaration :
'Parameters' '{'
parametersDeclarations+=AParameterDeclaration(';' parametersDeclarations+=AParameterDeclaration)*
'}' ;

AParameterDeclaration:
type=[ATypeDef] name=ID '=' parameterValue=ParameterValue
;

ParameterValue : range=TypeSpec| value=Integer
;
```

FIGURE 6.11 – Structure d'une spécification VeriSensor

2.2 Modèle de déploiement VeriSensor

```
/* ============  Deployment ================== */

  DeplModel :
  'System' systemName=ID '{'
                listNodeClassInstantiations=ListNodeClassInstantiations '}' ;

  ListNodeClassInstantiations :
  listNodeClassInstantiations+=ANodeClassInstantiation
  (';' listNodeClassInstantiations+=ANodeClassInstantiation)* ;

  ANodeClassInstantiation :
  nodeClassReference=[ANodeClass] '=>' listNodeInstances=ListNodeInstances ;

  ListNodeInstances :
  listNodeInstances+=ANodeInstance (',' listNodeInstances+=ANodeInstance)* ;

  ANodeInstance :       name=ID '(''x' '=' xPos=Double ','
   'y' '=' yPos=Double ','
   'z' '=' zPos=Double ','
  'NextHop' '=' (nextHopNode = [ANodeInstance]) ','
  'errorRate' '=' errorRate=Double ')' ;
```

FIGURE 6.12 – Grammaire du modèle de déploiement VeriSensor

La figure 6.12 présente la grammaire du modèle de déploiement VeriSensor qui spécifie l'ensemble des instances de noeuds du système et leurs caractéristiques.

2.3 Modèle de requêtes VeriSensor

La figure 6.13 présente la grammaire du modèle de requêtes de VeriSensor qui spécifie les caractéristiques des requêtes qui vont être diffusées dans le réseau (*e.g.*, type de la requête, période de capture).

```
QueryModel :
(listQuerySpecifications=QuerySpecification)+

QuerySpecification :
queryType=QueryType name=ID queryBody=QueryBody ';' ;

QueryType :
  'CondQuerySpecification' | 'NonCondQuerySpecification' ;

QueryBody :
    CondQueryBody |     NonCondQueryBody ;

CondQueryBody:
   '(' 'Conditions' '=' '[' (condition=Condition) ']' ','
  'Duration' '=' dur=Dur ',' 'SensingPeriod' '=' sensPer=SpecPar ',' 'ProcessingCost' '='
   processCost=SpecParam ','
  'ProcessingDuration' '=' processDur=SpecParam')' ;

SpecPar :
    parameter=[AParameterDeclaration]
  | integer=Integer ;

NonCondQueryBody :
  '(' 'Quantities' '=' '[' (listQuantities+=[ATypeDef])
  (',' listQuantities+=[ATypeDef])*']' ','
  'Duration' '=' dur=Dur ',' 'SensingPeriod' '=' sensPer=SpecPar',' 'ProcessingPeriod' '='
processPer=SpecPar ',' 'SendingPeriod' '=' sendingPer=SpecPar ','
  'ProcessingCost' '=' processCost=SpecPar ',' 'ProcessingDuration' '='
processDur=SpecPar ')' ;

Condition :
  basicComparaisons+=BasicComparaison ('AND' basicComparaisons+=BasicComparaison)* ;

BasicComparaison :
  typeReference=[ATypeDef] opComp=OpComp aValue= AValue ;

AValue :
  Double ;

Quantity :
  ID ;

Dur :
    Integer  | 'Infinity' ;

OpBool:
  'AND' | 'OR' | 'XOR';

OpComp:
  ('<' | '>' | '=' | '>='| '<=' | '!=') ;
```

FIGURE 6.13 – Grammaire du modèle de requête VeriSensor

2.4 Modèle de l'environnement VeriSensor

La figure 6.14 présente la grammaire du modèle de l'environnement VeriSensor qui décrit les scénarios environnementaux auxquels peut être soumis le réseau de capteurs sans fil. Un scénario environnemental consiste à affecter à chaque grandeur une évolution donnée (*i.e.* fichier « .data »)

```
/* ============  Environnement ================== */

Environment:
        (importStatements+=ImportStatement)+
        'Environment' name=ID '{'
        (environmentalScenarios+=EnvironmentalScenario)+
        '}'

;

EnvironmentalScenario:
        'EnvironmentalScenario' name=ID '('cyclicness=Cyclicness ')' '{'

        quantityEvolutions+=QuantityEvolution (';'quantityEvolutions+=QuantityEvolution)*
        '}';

Cyclicness:
        'cyclic'|'acyclic'
;



ImportStatement :
        'import' dataFileName=DataFileName '.data' ';'
;


DataFileName :
        name=ID
;

QuantityEvolution:
        'DefineEvolution' quantity=[ATypeDef] '=' xyztFile= [DataFileName] '.data'
;
```

FIGURE 6.14 – Grammaire du modèle de l'environnement VeriSensor

2.5 Modèle d'une classe de noeuds VeriSensor

La figure 6.15 présente la grammaire du modèle de classe de noeuds VeriSensor qui référence les sous-modèles capture, application, réseau de la classe ainsi que l'énergie initiale des instances.

```
/* ============  NodeClass ================== */

ANodeClass :
'NodeClass' name=ID '{'
'Sensing' '=>' ('Empty'| sensingSubModelReference=[SensingSubModel]) ';'
'Application' '=>' applicationSubModelReference=[ApplicationSubModel] ';'
'Network' '=>' networkSubModelReference=[NetworkSubModel] ';'
'InitialEnergy' '=>' initEnergy=InitEnergy '}' ;

InitEnergy :
   parameter=[AParameterDeclaration]
|  integer=Integer
|       inf='Infinity' ;
```

FIGURE 6.15 – Grammaire du modèle de classe de noeuds VeriSensor

2.6 Sous-modèle capture VeriSensor

La figure 6.16 présente la grammaire du sous-modèle capture de VeriSensor qui spécifie un ensemble de capteurs et leurs caractéristiques de fonctionnement.

```
/* ============  Sensing ================== */
SensingSubModel :
        'Sensing' name=ID '{' listSensors+=ASensor(';' listSensors+=ASensor)*'}' ;

ASensor :
  'Sensor' name=ID '(' 'PhysicalQuantity' '=' typeReference=[ATypeDef] ','
  'StartupTime''=' startup=SpecPar ',' 'SensingCost' '=' sensCost=SpecPar ','
  'SensingDuration' '='sensDur=SpecPar')' ;
```

FIGURE 6.16 – Grammaire du sous-modèle capture VeriSensor

2.7 Sous-modèle application VeriSensor

La figure 6.17 présente la grammaire du sous-modèle application de VeriSensor qui référence les requêtes supportées par l'application (pour un noeud) ou spécifie les instants où chaque requête doit être diffusée (pour la station de base).

```
/* ============  Application ================== */
ApplicationSubModel :
        'Application' '<'applicationType= Application_type'>' name=ID
                '{'
                body=ApplicationBody
                '}' ;

Application_type :
  'Node' // pour un sous modele application d'un noeud
| 'Base'          // pour un sous modele application d'une station de base ;

ApplicationBody :
  bodyNode=QueryReferences
| bodyBase=BroadcastTimes ;

QueryReferences :
  'Queries =' queryReferences+=[QuerySpecification]
  (','queryReferences+=[QuerySpecification])* ';'

  // query_id_reference = un identifiant de requete ;

// la station de base donne le moment de diffusion de chaque requete
BroadcastTimes :
broadcastTimes+=ABroadcastTime (';' broadcastTimes+=ABroadcastTime)* ;

ABroadcastTime :
 'BroadcastTime' queryIdReference=[QuerySpecification] '=' time=Integer ;
```

FIGURE 6.17 – Grammaire du sous-modèle application VeriSensor

2.8 Sous-modèle réseau VeriSensor

La figure 6.18 présente la grammaire du sous-modèle réseau VeriSensor qui spécifie les caractéristiques des opérations de communication de la classe de nœud.

```
/* ============  Network ================== */
NetworkSubModel :
  'Network' name=ID
    '{'
   body=NetworkBody
    '}' ;

NetworkBody :
   'Range' '=' rnge=Double ';'
   'ReceptionCost' '=' recCost=NetworkCost ';'
   'EmissionCost' '=' emCost=NetworkCost ';'
   'EmissionDuration' '=' emDur=SpecPar ;

 NetworkCost :
  parameter=[AParameterDeclaration]
| integer=Integer
| none='None' ;
```

FIGURE 6.18 – Grammaire du sous-modèle réseau VeriSensor

2.9 Modèle de types de données VeriSensor

La figure 6.19 présente la grammaire du modèle de types de données qui définit l'ensemble des valeurs possibles pour les grandeurs physiques du système et l'énergie.

```
/* ============  Types definition file ================== */

TypeDefModel :
        (typeDefs+=ATypeDef)+ ;

ATypeDef :
  'Type' name=ID 'is' typeSpec=TypeSpec ';'  ;

TypeSpec :      integerRange=IntegerRange
  |     doubleRange=DoubleRange
  |     enumeration=Enumeration ;

IntegerRange
  :     lowBound=Integer '..' upBound=Integer ('Step' step=Integer)?    ;


DoubleRange
  :     lowBound=Double '..' upBound=Double 'Step' step=Double ;

Enumeration :
items+=ID (',' items+=ID)* ;
```

FIGURE 6.19 – Grammaire du modèle de types de données VeriSensor

2.10 Métriques VeriSensor

La figure 6.20 présente la grammaire des métriques utilisées pour exprimer les propriétés VeriSensor à vérifier.

```
/* ============  Proprietes specifiees par l utilisateur ================== */
Term returns BooleanExpression :
         '[' (ImpliesEquivProposition)']'
        | (metrics+=Metric) opComp=OpComp (metrics+=Metric|value=Value)
        | 'Deadlock' ;

NotProposition returns BooleanExpression :
        'not'? Term {NotProposition.term=current} ;

AndOrProposition returns BooleanExpression :
        NotProposition  ({AndOrProposition.left=current} OpBool right=NotProposition)* ;

ImpliesEquiv :
        'IMPLIES'|'EQUIV' ;

 ImpliesEquivProposition returns BooleanExpression :
        AndOrProposition ({ImpliesEquivProposition.left=current} ImpliesEquiv
         right=AndOrProposition)* ;

/* reference a un parametre de noeud (process cost, sensing cost, temps) a une metrique*/

 Metric :
    parameter=[AParameterDeclaration]
  | (keywordMetric=KeywordMetric)
  | (compositeMetric=CompositeMetric) ;

OpMultDivide :
    '*'  | '/' ;

OpAddSubstract :
  '+'  |'-' ;

MultDivide :
  multDivide+=Metric (opMultDivide+=OpMultDivide multDivide+=Metric)*  ;

CompositeMetric :
  compositeMetric+=MultDivide (opAddSubstract+=OpAddSubstract compositeMetric+=MultDivide)* ;

KeywordMetric :
  {TrueMetric}  'WorstCaseLifetime'
  | {TrueMetric} 'ProportionOfAliveNodes'
  | {TrueMetric} 'ProportionOfConnectedNodes' 'in' 'DeltaT' deltaT=DeltaT
  | {TrueMetric} 'EventDeliveryRatio' '('queryReference=[QuerySpecification] ')'
  'in' 'DeltaT' deltaT=DeltaT
  | {TrueMetric} 'MaximumLatency'
  | {TrueMetric} 'MessageDeliveryRatio' 'in' 'DeltaT' deltaT=DeltaT
  | {TrueMetric} 'Deadlock';

DeltaT :
        Integer    |'infinity' ;

Value :
    Integer   |  Double ;
```

FIGURE 6.20 – Grammaire des métriques de VeriSensor

2.11 Modèle de propriétés VeriSensor

La figure 6.21 présente la grammaire du modèle de propriétés VeriSensor qui permet d'exprimer la liste des tâches de vérification que l'on veut effectuer sur le modèle du réseau de capteurs sans fil.

```
PropertyModel :
  (checks+=Check)+ ;

Check :
  'check' name=ID '{' checkBody= (OptimizeBody|VerifyBody|PlotBody) '}' ;

PlotBody :
  'Plot' metric=Metric 'with' (parameterDomains=ParameterDomains)
  'ForEnvScenarios' (freeEnvScenario='Free'|(envScenarios+=[EnvironmentalScenario]
  (',' envScenarios+=[EnvironmentalScenario])*)) ;

VerifyBody :
  'Verify' 'Property' proposition=ImpliesEquivProposition 'with'
  (parameterDomains=ParameterDomains)
  'ForEnvScenarios' envScenarios+=[EnvironmentalScenario]
  (',' envScenarios+=[EnvironmentalScenario])* ;

OptimizeBody :
  operationName=('Maximize'|'Minimize') metric=Metric 'with'
  (parameterDomains=ParameterDomains)
  'ForEnvScenarios' (freeEnvScenario='Free'|envScenarios+=[EnvironmentalScenario]
  (',' envScenarios+=[EnvironmentalScenario])*) ;

EnvScenario : ID          ;

ParameterDomain :
 {ParameterDomain} parameters+=Parameter
 (('=' parameters+=Parameter)* '=' interval=Interval) ;

ParameterDomains :
  'parameterDomains' '=' '{'
  parameterDomains+=ParameterDomain
  (';'parameterDomains+=ParameterDomain)* '}' ;

Interval :
  typeSpec=TypeSpec ;
```

FIGURE 6.21 – Grammaire des tâches de vérification de VeriSensor

2.12 Types numériques utilisés dans la grammaire Xtext de VeriSensor

La figure 6.22 présente les règles définissant les types numériques utilisés dans la grammaire Xtext de VeriSensor.

```
/* ============  Types de Base ================== */
        Double returns ecore::EDouble :
                ('-')? INT '.' INT ;

        Integer returns ecore::EInt :
                        ('-')? INT ;
```

FIGURE 6.22 – types numériques de base utilisés dans la grammaire Xtext de VeriSensor

BIBLIOGRAPHIE

[ABMS07] Sara ADAMS, Magnus BJÖRK, Thomas F. MELHAM et Carl-Johan H. SEGER : Automatic abstraction in symbolic trajectory evaluation. *In Formal Methods in Computer-Aided Design*, pages 127–135. IEEE Computer Society, 2007.

[ACD93] Rajeev ALUR, Costas COURCOUBETIS et David DILL : Model-checking in dense real-time. *Inf. Comput.*, 104(1):2–34, mai 1993.

[AD90] Rajeev ALUR et David L. DILL : Automata for modeling real-time systems. *In 17th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 443 de *LNCS*, pages 322–335. Springer, 1990.

[ADBS09] Bahar AKBAL-DELIBAS, Pruet BOONMA et Junichi SUZUKI : Extensible and precise modeling for wireless sensor networks. *In United Information Systems Conference*, pages 551–562, 2009.

[Alh12] S.S. ALHIR : *Guide to Applying the Uml*. Springer Professional Computing. Springer-Verlag New York Incorporated, 2012.

[ALN+91] J-R ABRIAL, Matthew KO LEE, DS NEILSON, PN SCHARBACH et Ib Holm SØRENSEN : The b-method. *In VDM'91 Formal Software Development Methods*, pages 398–405. Springer, 1991.

[And95] J.G.T. ANDERSON : Pilot survey of mid-coast maine seabird colonies : an evaluation of techniques. Rapport technique, State of Maine Dept. of Inland Fisheries and Wildlife, 1995.

[Aro04] Anish K ARORA : *On random event detection with wireless sensor networks*. Thèse de doctorat, The Ohio State University, 2004.

[ASSC02] Ian F. AKYILDIZ, Weilian SU, Yogesh SANKARASUBRAMANIAM et Erdal CAYIRCI : Wireless sensor networks : a survey. *Computer Networks*, 38(4):393–422, 2002.

[BBFM99] Patrick BEHM, Paul BENOIT, Alain FAIVRE et Jean-Marc MEYNADIER : Meteor : A successful application of b in a large project. *In FM'99—Formal Methods*, pages 369–387. Springer, 1999.

[BC04] Yves BERTOT et Pierre CASTÉRAN : *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. springer, 2004.

[BCPR04] Zack BUTLER, Peter CORKE, Ron PETERSON et Daniela RUS : Networked cows : Virtual fences for controlling cows. *In Workshop on Applications of Mobile Embedded Systems 2004*, 2004.

[BDM+98] Marius BOZGA, Conrado DAWS, Oded MALER, Alfredo OLIVERO, Stavros TRIPAKIS et Sergio YOVINE : Kronos : A model-checking tool for real-time systems. *In CAV*, pages 546–550. Springer, 1998.

[BFFM08] A. BOULIS, A. FEHNKER, M. FRUTH et A. MCIVER : Cavi–simulation and model checking for wireless sensor networks. *In Fifth International Conference on Quantitative Evaluation of Systems. QEST'08*, pages 37–38. IEEE, 2008.

[BGO+04] Marius BOZGA, Susanne GRAF, Ileana OBER, Iulian OBER et Joseph SIFAKIS : Tools and Applications : the IF toolset. *In 4th Int. School on Formal Methods for the Design of Computer, Communication and Software Systems : Real Time, SFM-04 :RT*. 2004.

[BHH+10] HR BOGENA, M HERBST, JA HUISMAN, U ROSENBAUM, A WEUTHEN et H VEREECKEN : Potential of wireless sensor networks for measuring soil water content variability. *Vadose Zone Journal*, 9(4):1002–1013, 2010.

[BK10] D. BUCUR et M.Z. KWIATKOWSKA : Software verification for tinyos. *In 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 400–401. ACM, 2010.

[BKL+05] Philip BALDWIN, Sanjeev KOHLI, Edward A. LEE, Xiaojun LIU, Yang ZHAO, Christopher Hylands BROOKS, N. V. KRISHNAN, Stephen NEUENDORFFER, Charlie ZHONG et Rachel ZHOU : Visualsense : Visual modeling for wireless and sensor network systems. Rapport technique, U.C. Berkeley, 2005.

[BLL+95] Johan BENGTSSON, Kim G. LARSEN, Fredrik LARSSON, Paul PETTERSSON et Wang YI : Uppaal — a Tool Suite for Automatic Verification of Real–Time Systems. *In Workshop on Verification and Control of Hybrid Systems III*, volume 1066 de *LNCS*, pages 232–243. Springer, 1995.

[BMKMTM12] Yann BEN MAÏSSA, Fabrice KORDON, Salma MOULINE et Yann THIERRY-MIEG : Modeling and Analyzing Wireless Sensor Networks with VeriSensor. *In Petri Net and Software Engineering (PNSE)*, volume 851, pages 60–76, Hamburg, Germany, juin 2012. CEUR.

[BMKMTM13] Yann BEN MAÏSSA, Fabrice KORDON, Salma MOULINE et Yann THIERRY-MIEG : Modeling and Analyzing Wireless Sensor Networks with VeriSensor : an Integrated Workflow. *Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)*, VIII:24–47, 2013.

[BMM10a] Yann BEN MAISSA et Salma MOULINE : A SysML profile for wireless sensor networks modeling. *In I/V Communications and Mobile Network (ISVC), 2010 5th International Symposium on*, pages 1 – 4, septembre 2010.

[BMM10b] Yann BEN MAISSA et Salma MOULINE : Un langage de modélisation pour réseaux de capteurs sans fil basé sur sysml. *In 2èmes Journées*

Doctorales en Technologies de l'Information et de la Communication, juillet 2010.

[BMR+96] Frank BUSCHMANN, Regine MEUNIER, Hans ROHNERT, Peter SOMMERLAD et Michael STAL : *Pattern-Oriented Software Architecture, Volume 1 : A System of Patterns*. Wiley, Chichester, UK, 1996.

[Bot07] M BOTTS : Ogc implementation specification 07-000 : Opengis sensor model language (sensorml)-open geospatial consortium. Rapport technique, Tech. Rep, 2007.

[Bou07] A. BOULIS : Castalia : revealing pitfalls in designing distributed algorithms in wsn. *In 5th international conference on Embedded networked sensor systems*, pages 407–408. ACM, 2007.

[BRV04] Bernard BERTHOMIEU*, P-O RIBET et François VERNADAT : The tool tina–construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, 42(14):2741–2756, 2004.

[BS02] Douglas M BLOUGH et Paolo SANTI : Investigating upper bounds on network lifetime extension for cell-based energy conservation techniques in stationary ad hoc networks. *In Proceedings of the 8th annual international conference on Mobile computing and networking*, pages 183–192. ACM, 2002.

[Bud04] Frank BUDINSKY : *Eclipse modeling framework : a developer's guide*. Addison-Wesley Professional, 2004.

[Cal03] Edgar H. CALLAWAY : *Wireless Sensor Networks : Architectures and Protocols*. CRC Press, Inc., 2003.

[CE04] Alberto CERPA et Deborah ESTRIN : Ascent : Adaptive self-configuring sensor networks topologies. *mobile computing, IEEE transactions on*, 3(3):272–285, 2004.

[CES83] Edmund Melson CLARKE, E Allen EMERSON et Aravinda Prasad SISTLA : Automatic verification of finite state concurrent system using temporal logic specifications : a practical approach. *In Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 117–126. ACM, 1983.

[CES86] Edmund M. CLARKE, E Allen EMERSON et A Prasad SISTLA : Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[CGVC06] Bogdan CĂRBUNAR, Ananth GRAMA, Jan VITEK et Octavian CĂRBUNAR : Redundancy and coverage detection in sensor networks. *ACM Trans. Sen. Netw.*, 2(1):94–128, février 2006.

[CKL04] Edmund CLARKE, Daniel KROENING et Flavio LERDA : A tool for checking ansi-c programs. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.

[CKNZ12] Edmund M CLARKE, William KLIEBER, Miloš NOVÁČEK et Paolo ZULIANI : Model checking and the state explosion problem. *In Tools for Practical Software Verification*, pages 1–30. Springer, 2012.

[Con06] Rob CONANT : Wireless sensor networks : Driving the new industrial revolution. *Industrial Embedded Systems*, pages 8–11, 2006.

[CP92] Søren CHRISTENSEN et Laure PETRUCCI : *Towards a modular analysis of coloured Petri nets*. Springer, 1992.

[DD09] Isabel DIETRICH et Falko DRESSLER : On the lifetime of wireless sensor networks. *ACM Transactions on Sensor Networks (TOSN)*, 5(1):5, 2009.

[DGV97] Akash DESHPANDE, Aleks GÖLLÜ et Pravin VARAIYA : *SHIFT : A formalism and a programming language for dynamic networks of hybrid automata*. Springer, 1997.

[DHHV05] Jing DENG, Yunghsiang S HAN, Wendi B HEINZELMAN et Pramod K VARSHNEY : Scheduling sleeping nodes in high density cluster-based sensor networks. *Mobile Networks and Applications*, 10(6):825–835, 2005.

[DLV02] Juan DE LARA et Hans VANGHELUWE : Atom3 : A tool for multi-formalism and meta-modelling. *In Fundamental approaches to software engineering*, pages 174–188. Springer, 2002.

[DML02] Enrique J DUARTE-MELO et Mingyan LIU : Analysis of energy consumption and lifetime of heterogeneous wireless sensor networks. *In Global Telecommunications Conference, 2002. GLOBECOM'02. IEEE*, volume 1, pages 21–25. IEEE, 2002.

[DP10] Waltenegus DARGIE et Christian POELLABAUER : *Fundamentals of wireless sensor networks : theory and practice*. Wiley. com, 2010.

[DSS+08] Jin Song DONG, Jing SUN, Jun SUN, Kenji TAGUCHI et Xian ZHANG : Specifying and verifying sensor networks : An experiment of formal methods. *In Formal Methods and Software Engineering*, pages 318–337. Springer, 2008.

[EE04] Jeremy ELSON et Deborah ESTRIN : Sensor networks : a bridge to the physical world. *Wireless sensor networks*, pages 3–20, 2004.

[EEK02] Sinem Coleri ERGEN, Mustafa ERGEN et T. John KOO : Lifetime analysis of a sensor network with hybrid automata modelling. *In ACM international workshop on Wireless sensor networks and applications*, pages 98–104, 2002.

[EFM06] Javier ESPINA, Thomas FALCK et Oliver MÜLHENS : Network topologies, communication protocols, and standards. *In* Guang-Zhong YANG, éditeur : *Body Sensor Networks*, pages 145–182. Springer London, 2006.

[FGH06] Peter H. FEILER, David P. GLUCH et John J. HUDAK : The Architecture Analysis & Design Language (AADL) : An Introduction. Rapport technique, Software Engineering Institute, 2006.

[Fow10] Martin FOWLER : *Domain Specific Languages*. Addison-Wesley Professional, 1st édition, 2010.

[FRZ+12] Joachim FISCHER, Jens-Peter REDLICH, Jochen ZSCHAU, Claus MILKEREIT, Matteo PICOZZI, Kevin FLEMING, Mihal BRUMBULLI, Bjorn LICHTBLAU et Ingmar EVESLAGE : A wireless mesh sensing network for early warning. *Journal of Network and Computer Applications*, 35(2):538 – 547, 2012.

[Ger91] Anne GERACI : *IEEE Standard Computer Dictionary : Compilation of IEEE Standard Computer Glossaries*. IEEE Press, Piscataway, NJ, USA, 1991.

[GH+13] V Cagri GUNGOR, Gerhard P HANCKE *et al.* : *Industrial Wireless Sensor Networks : Applications, Protocols, and Standards*. CRC Press, 2013.

[GLM+05] Guillaume GARDEY, Didier LIME, Morgan MAGNIN *et al.* : Romeo : A tool for analyzing time petri nets. *In Computer Aided Verification*, pages 418–423. Springer, 2005.

[GPY08] Amitabha GHOSH, Luis PEREIRA et Ting YAN : Modeling wireless sensor network architectures using aadl. *In 4th European Congress on Embedded Real Time Software (ERTS)*, 2008.

[GR06] Caleb GOODWIN et David J RUSSOMANNO : An ontology-based sensor network prototype environment. *In Proceedings of the 5th International Conference on Information Processing in Sensor Networks (IPSN 2006), Nashville TN, USA*, 2006.

[HCRP91] Nicholas HALBWACHS, Paul CASPI, Pascal RAYMOND et Daniel PILAUD : The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[HHT97] Thomas A. HENZINGER, Pei H. HO et Howard W. TOI : HYTECH : A Model Checker for Hybrid Systems. *Int. Journal on Software Tools for Technology Transfer*, 1(1-2):110–122, 1997.

[HJS+09] Thomas HARMON, Jenny JAY, Jose SAEZ, William KAISER, Steve MARGULIS, Alexander RAT'KO, Chu-Chin LIN, Jason FISHER, Che-Chuan WU, Christopher BUTLER *et al.* : An overview of cens contaminant transport observation and management research. 2009.

[Hoa69] Charles Antony Richard HOARE : An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa78] Charles Antony Richard HOARE : Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hol03] Gerard HOLZMANN : *Spin model checker, the : primer and reference manual*. Addison-Wesley Professional, 2003.

[Hol13] Gerard J HOLZMANN : Landing a spacecraft on mars. *IEEE software*, 30(2):83–86, 2013.

[HR04] Michael HUTH et Mark RYAN : *Logic in Computer Science : Modelling and reasoning about systems*. Cambridge University Press, 2004.

[HR09] Y. HANNA et H. RAJAN : Slede : Framework for automatic verification of sensor network security protocol implementations. *In 31st International Conference on Software Engineering – Companion Volume*, pages 427–428. IEEE, 2009.

[HSBH+09] Rebecca N HANDCOCK, Dave L SWAIN, Greg J BISHOP-HURLEY, Kym P PATISON, Tim WARK, Philip VALENCIA, Peter CORKE et Christopher J O'NEILL : Monitoring animal behaviour and environmental interactions using wireless sensor networks, gps collars and satellite remote sensing. *Sensors*, 9(5):3586–3603, 2009.

[HVP+05] Jérôme HUGUES, Thomas VERGNAUD, Laurent PAUTET, Yann THIERRY-MIEG, Soheib BAARIR et Fabrice KORDON : On the formal verification of middleware behavioral properties. *Electron. Notes Theor. Comput. Sci.*, 133:139–157, mai 2005.

[IGE00] Chalermek INTANAGONWIWAT, Ramesh GOVINDAN et Deborah ESTRIN : Directed diffusion : a scalable and robust communication paradigm for sensor networks. *In Proceedings of the 6th annual international conference on Mobile computing and networking*, MobiCom '00, pages 56–67, New York, NY, USA, 2000. ACM.

[Jen91] Kurt JENSEN : *Coloured Petri nets : A high level language for system design and analysis*. Springer, 1991.

[KAL05] Santosh KUMAR, Anish ARORA et Ten H LAI : On the lifetime analysis of always-on wireless sensor network applications. *In Mobile Adhoc and Sensor Systems Conference, 2005. IEEE International Conference on*, pages 3–pp. IEEE, 2005.

[KBTMBM12] Fabrice KORDON, Béatrice BÉRARD, Yann THIERRY-MIEG et Yann BEN MAÏSSA : Hierarchy is Good For Discrete Time : a Compositional Approach to Discrete Time Verification. *In Dagstuhl seminar "Architecture-Driven Semantic Analysis of Embedded Systems"*, numéro 12272, pages 38–39, juillet 2012.

[Kel76] Robert M KELLER : Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

[Ken02] Stuart KENT : Model driven engineering. *In Integrated Formal Methods*, pages 286–298. Springer, 2002.

[KEW02] Bhaskar KRISHNAMACHARI, Deborah ESTRIN et Stephen B. WICKER : The impact of data aggregation in wireless sensor networks.

In ICDCSW '02 : Proceedings of the 22nd International Conference on Distributed Computing Systems, pages 575–578, Washington, DC, USA, 2002. IEEE Computer Society.

[KFV11] Raghavendra V KULKARNI, Anna FORSTER et Ganesh Kumar VENAYAGAMOORTHY : Computational intelligence in wireless sensor networks : A survey. *Communications Surveys & Tutorials, IEEE*, 13(1):68–96, 2011.

[KLB+12a] F. KORDON, A. LINARD, D. BUCHS, M. COLANGE, S. EVANGELISTA, K. LAMPKA, N. LOHMANN, E. PAVIOT-ADET, Y. THIERRY-MIEG et H. WIMMEL : Report on the Model Checking Contest at Petri Nets 2011. *ToPNoC*, VI:169–196, 2012.

[KLB+12b] Fabrice KORDON, Alban LINARD, Didier BUCHS, Maximilien COLANGE, Sami EVANGELISTA, Lukasz FRONC, Lom-Messan HILLAH, N. LOHMANN, Emmanuel PAVIOT-ADET, Franck POMMEREAU, C. ROHR, Yann THIERRY-MIEG, Harro WIMMEL et Karsten WOLF : Raw Report on the Model Checking Contest at Petri Nets 2012. Rapport technique, 2012.

[KLR96] Steven KELLY, Kalle LYYTINEN et Matti ROSSI : Metaedit+ a fully configurable multi-user and multi-tool case and came environment. *In Advanced Information Systems Engineering*, pages 1–21. Springer, 1996.

[KNP02] M. KWIATKOWSKA, G. NORMAN et D. PARKER : Prism : Probabilistic symbolic model checker. *Computer Performance Evaluation : Modelling Techniques and Tools*, pages 113–140, 2002.

[KT11] Ajith KUMAR et Sheela Ganesh THORENOOR : Analysis of ip network for different quality of service. *In 2009 International Symposium on Computing, Communication, and Control (ISCCC 2009) Proc. of CSIT*, volume 1, 2011.

[Lap85] Jean-Claude LAPRIE : Dependable computing and fault-tolerance. *Digest of Papers FTCS-15*, pages 2–11, 1985.

[LCH+11] Philip LEVIS, T CLAUSEN, J HUI, O GNAWALI et J KO : The trickle algorithm. *Internet Engineering Task Force, RFC6206*, 2011.

[Lee06] Edward A LEE : Cyber-physical systems-are computing foundations adequate. *In Position Paper for NSF Workshop On Cyber-Physical Systems : Research Motivation, Techniques and Roadmap*, volume 2. Citeseer, 2006.

[LJ99] Edward A LEE et II JOHN : *Overview of the ptolemy project*. Electronics Research Laboratory, College of Engineering, University of California, 1999.

[LLWC03] P. LEVIS, N. LEE, M. WELSH et D. CULLER : Tossim : Accurate and scalable simulation of entire tinyos applications. *In 1st international conference on Embedded networked sensor systems*, pages 126–137. ACM, 2003.

[LR10] Peng LI et John REGEHR : T-check : bug finding for sensor networks. *In 9th ACM/IEEE Int. Conf. on Information Processing in Sensor Networks*, pages 174–185. ACM, 2010.

[MCP+02] Alan MAINWARING, David CULLER, Joseph POLASTRE, Robert SZEWCZYK et John ANDERSON : Wireless sensor networks for habitat monitoring. *In WSNA '02 : Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, New York, NY, USA, 2002. ACM.

[MEL+12] Urbashi MITRA, B Adar EMKEN, Sangwon LEE, Ming LI, Viktor ROZGIC, Gautam THATTE, Harshvardhan VATHSANGAM, D ZOIS, Murali ANNAVARAM, Shrikanth NARAYANAN *et al.* : Knowme : a case study in wireless body area sensor network design. *Communications Magazine, IEEE*, 50(5):116–125, 2012.

[Mer74] Philip Meir MERLIN : *A study of the recoverability of computing systems.* Thèse de doctorat, 1974. AAI7511026.

[MFHH05] Samuel R. MADDEN, Michael J. FRANKLIN, Joseph M. HELLERSTEIN et Wei HONG : Tinydb : an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, mars 2005.

[MSFC02] Samuel MADDEN, Robert SZEWCZYK, Michael J. FRANKLIN et David CULLER : Supporting aggregate queries over ad-hoc wireless sensor networks. *In WMCSA '02 : Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, page 49, Washington, DC, USA, 2002. IEEE Computer Society.

[MSZ07] Laurent MOUNIER, Ludovic SAMPER et Wassim ZNAIDI : Worst-case lifetime computation of a wireless sensor network by model-checking. *In Proceedings of the 4th ACM workshop on Performance evaluation of wireless ad hoc, sensor,and ubiquitous networks*, PE-WASUN '07, pages 1–8, New York, NY, USA, 2007. ACM.

[Mur89] Tadao MURATA : Petri nets : Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[NM92] Robert HB NETZER et Barton P MILLER : What are race conditions ? : Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 1992.

[NSL08] Nicolas NAVET et Francoise SIMONOT-LION : *Automotive Embedded Systems Handbook*. CRC Press, Inc., Boca Raton, FL, USA, 1st édition, 2008.

[ÖM07] P. C. ÖLVECZKY et J. MESEGUER : Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.

[OMSJ05] Chris OTTO, Aleksandar MILENKOVIĆ, Corey SANDERS et Emil JOVANOV : System architecture of a wireless body area sensor

network for ubiquitous health monitoring. *J. Mob. Multimed.*, 1:307–326, 2005.

[ORR+96] Sam OWRE, Sreeranga RAJAN, John M RUSHBY, Natarajan SHANKAR et Mandayam SRIVAS : Pvs : Combining specification, proof checking, and model checking. *In Computer Aided Verification*, pages 411–414. Springer, 1996.

[OT07] Peter Csaba ÖLVECZKY et Stian THORVALDSEN : Formal modeling and analysis of the ogdc wireless sensor network algorithm in real-time maude. *In Proceedings of the 9th IFIP WG 6.1 international conference on Formal methods for open object-based distributed systems*, FMOODS'07, pages 122–140, Berlin, Heidelberg, 2007. Springer-Verlag.

[Pet80] Carl Adam PETRI : Introduction to general net theory. *In Net theory and applications*, pages 1–19. Springer, 1980.

[Poo01] John D POOLE : Model-driven architecture : Vision, standards and emerging technologies. *In Workshop on Metamodeling and Adaptive Object Models, ECOOP*, volume 2001, 2001.

[RB66] J RICHARD BÜCHI : Symposium on decision problems : On a decision method in restricted second order arithmetic. *Studies in Logic and the Foundations of Mathematics*, 44:1–11, 1966.

[RJB99] James RUMBAUGH, Ivar JACOBSON et Grady BOOCH : The unified modeling language reference manual. 1999.

[RM04] Kay RÖMER et Friedemann MATTERN : The design space of wireless sensor networks. *IEEE Wireless Communications*, 11(6):54–61, 2004.

[Sad08] Daniel A. SADILEK : Domain-specific languages for wireless sensor networks. *In Modellierung*, pages 237–241, 2008.

[SBB+99] Philippe SCHNOEBELEN, Béatrice BÉRARD, Michel BIDOIT, François LAROUSSINIE et Antoine PETIT : *Vérification de logiciels : techniques et outils du model-checking*. Vuibert, 1999.

[Sch06] Douglas C SCHMIDT : Model-driven engineering. *IEEE Computer Society*, 39(2):25, 2006.

[SGAP00] Katayoun SOHRABI, Jay GAO, Vishal AILAWADHI et Gregory J POTTIE : Protocols for self-organization of a wireless sensor network. *Personal Communications, IEEE*, 7(5):16–27, 2000.

[Sil93] M SILVA : Introducing petri nets. *In Practice of Petri Nets in manufacturing*, pages 1–62. Springer, 1993.

[SMP+04] Robert SZEWCZYK, Alan MAINWARING, Joseph POLASTRE, John ANDERSON et David CULLER : An analysis of a large scale habitat monitoring application. *In SenSys '04 : Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 214–226, New York, NY, USA, 2004. ACM.

[SMZ07] K. SOHRABY, D. MINOLI et T. ZNATI : *Wireless sensor networks : technology, protocols and applications*. Wiley Interscience, 2007, 2007.

[SOP+04] Robert SZEWCZYK, Eric OSTERWEIL, Joseph POLASTRE, Michael HAMILTON, Alan MAINWARING et Deborah ESTRIN : Habitat monitoring with sensor networks. *Commun. ACM*, 47(6):34–40, juin 2004.

[TMBK+11] Y. THIERRY-MIEG, B. BÉRARD, F. KORDON, D. LIME et O. H. ROUX : Compositional Analysis of Discrete Time Petri nets. *In 1st workshop on Petri Nets Compositions (CompoNet 2011)*, volume 726, pages 17–31. CEUR, 2011.

[TMDM03] Y. THIERRY-MIEG, C. DUTHEILLET et I. MOUNIER : Automatic symmetry detection in well-formed nets. *In Petri Nets*, volume 2679 de *LNCS*, pages 82–101. Springer Verlag, 2003.

[TMH08] Yann THIERRY-MIEG et Lom-Messan HILLAH : UML behavioral consistency checking using Instantiable Petri nets. *Innovations in Systems and Software Engineering*, 4(3):293–300, 2008.

[TMPHK09a] Y. THIERRY-MIEG, D. POITRENAUD, A. HAMEZ et F. KORDON : Hierarchical Set Decision Diagrams and Regular Models. *In TACAS*, volume 5505 de *LNCS*, pages 1–15. Springer, 2009.

[TMPHK09b] Yann THIERRY-MIEG, Denis POITRENAUD, Alexandre HAMEZ et Fabrice KORDON : Hierarchical set decision diagrams and regular models. *In TACAS*, pages 1–15, 2009.

[TXY08] Simon TSCHIRNER, Liang XUEDONG et Wang YI : Model-based validation of qos properties of biomedical sensor networks. *In Proceedings of the 8th ACM international conference on Embedded software*, EMSOFT '08, pages 69–78, New York, NY, USA, 2008. ACM.

[VBL07] Chinh T VU, Raheem A BEYAH et Yingshu LI : Composite event detection in wireless sensor networks. *In Performance, Computing, and Communications Conference, 2007. IPCCC 2007. IEEE Internationa*, pages 264–271. IEEE, 2007.

[VCLÁ+07] Cristina VICENTE-CHICOTE, Fernando LOSILLA, Bárbara ÁLVAREZ, Andrés IBORRA et Pedro SÁNCHEZ : Applying mde to the development of flexible and reusable wireless sensor networks. *Int. J. Cooperative Inf. Syst.*, 16(3/4):393–412, 2007.

[WABU06] Thomas WATTEYNE, Isabelle AUGÉ-BLUM et 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, New York, NY, USA, 2006. ACM.

[WALR+06] Geoffrey WERNER-ALLEN, Konrad LORINCZ, Mario RUIZ, Omar MARCILLO, Jeff JOHNSON, Jonathan LEES et Matt WELSH : Deploying a wireless sensor network on an active volcano. *Internet Computing, IEEE*, 10(2):18–25, 2006.

[WBSO07] Hiroshi WADA, Pruet BOONMA, Junichi SUZUKI et Katsuya OBA : Modeling and executing adaptive sensor network applications with the Matilda UML virtual machine. *In 11th IASTED Int. conf. on Software Engineering and Applications (SEA)*, pages 216–225. ACTA Press, 2007.

[WXZ+03] Xiaorui WANG, Guoliang XING, Yuanfang ZHANG, Chenyang LU, Robert PLESS et Christopher GILL : Integrated coverage and connectivity configuration in wireless sensor networks. *In Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 28–39. ACM, 2003.

[YMG08] Jennifer YICK, Biswanath MUKHERJEE et Dipak GHOSAL : Wireless sensor network survey. *Computer networks*, 52(12):2292–2330, 2008.

[ZSL+11] Manchun ZHENG, Jun SUN, Yang LIU, Jin Song DONG et Yu GU : Towards a model checker for NesC and wireless sensor networks. *In 13th international conference on Formal methods and software engineering*, volume 6991 de *LNCS*, pages 372–387. Springer-Verlag, 2011.